

## 6. JavaScript: Introduction to Scripting

*Comment is free, but facts are sacred.*

—C. P. Scott

*The creditor hath a better memory than the debtor.*

—James Howell

*When faced with a decision, I always ask, “What would be the most fun?”*

—Peggy Walker

### Objectives

In this chapter you will:

- Write simple JavaScript programs.
- Use input and output statements.
- Learn basic memory concepts.
- Use arithmetic operators.
- Learn the precedence of arithmetic operators.
- Write decision-making statements to choose among alternative courses of action.
- Use relational and equality operators to compare data items.

### Outline

#### [6.1 Introduction](#)

## [6.2 Your First Script: Displaying a Line of Text with JavaScript in a Web Page](#)

## [6.3 Modifying Your First Script](#)

## [6.4 Obtaining User Input with `prompt` Dialogs](#)

### [6.4.1 Dynamic Welcome Page](#)

### [6.4.2 Adding Integers](#)

## [6.5 Memory Concepts](#)

## [6.6 Arithmetic](#)

## [6.7 Decision Making: Equality and Relational Operators](#)

## [6.8 Web Resources](#)

[Summary](#) | [Self-Review Exercises](#) | [Answers to Self-Review Exercises](#) | [Exercises](#)

### **6.1. Introduction**

In this chapter, we begin our introduction to the **JavaScript<sup>1</sup> scripting language**, which is used to enhance the functionality and appearance of web pages.<sup>2</sup>

---

<sup>1</sup> Many people confuse the scripting language JavaScript with the programming language Java. Java is a full-fledged object-oriented programming language. Java is popular for developing large-scale distributed enterprise applications and web applications. JavaScript is a browser-based scripting language developed by Netscape and implemented in all major browsers.

---

<sup>2</sup> JavaScript was originally created by Netscape. Both Netscape and Microsoft have been instrumental in the standardization of JavaScript by ECMA International—formerly the European Computer Manufacturers' Association—as ECMAScript ([www.ecma-](http://www.ecma-)

[international.org/publications/standards/ECMA-262.htm](http://international.org/publications/standards/ECMA-262.htm)). The latest version of JavaScript is based on ECMAScript 5.

In [Chapters 6–11](#), we present a detailed discussion of JavaScript—the *de facto* standard client-side scripting language for web-based applications due to its highly portable nature. Our treatment of JavaScript serves two purposes—it introduces client-side scripting (used in [Chapters 6–18](#)), which makes web pages more dynamic and interactive, and it provides the programming foundation for the server-side scripting presented later in the book.

Before you can run code examples with JavaScript on your computer, you may need to change your browser's security settings. By default, Internet Explorer 9 *prevents* scripts on your local computer from running, and displays a warning message. To allow scripts to run in files on your computer, select **Internet Options** from the **Tools** menu. Click the **Advanced** tab and scroll down to the **Security** section of the **Settings** list. Check the box labeled **Allow active content to run in files on My Computer**. Click **OK** and restart Internet Explorer. HTML5 documents on your own computer that contain JavaScript code will now run properly. Firefox, Chrome, Opera, Safari (including on the iPhone) and the Android browser have JavaScript enabled by default.

## 6.2. Your First Script: Displaying a Line of Text with JavaScript in a Web Page

We begin with a simple **script** (or **program**) that displays the text "Welcome to JavaScript Programming!" in the HTML5 document. All major web browsers contain **JavaScript interpreters**, which process the commands written in JavaScript. The JavaScript code and its result are shown in [Fig. 6.1](#).

---

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 6.1: welcome.html -->
4 <!-- Displaying a line of text. -->
```

```
5 <html>
6 <head>
7   <meta charset = "utf-8">
8   <title>A First Program in JavaScript</title>
9   <script type = "text/javascript">
10
11     document.writeln(
12       "<h1>Welcome to JavaScript Programming!</h1>" );
13
14   </script>
15 </head><body></body>
16 </html>
```



Fig. 6.1. Displaying a line of text.

Lines 11–12 do the “real work” of the script, namely, displaying the phrase `Welcome to JavaScript Programming!` as an `h1` heading in the web page.

Line 6 starts the `<head>` section of the document. For the moment, the JavaScript code we write will appear in the `<head>` section. The browser interprets the contents of the `<head>` section first, so the JavaScript programs we write there execute *before* the `<body>` of the HTML5 document displays. In later chapters on JavaScript, we illustrate **inline scripting**, in which JavaScript code is written in the `<body>` of an HTML5 document.

### The `script` Element and Commenting Your Scripts

Line 9 uses the `<script>` tag to indicate to the browser that the text which follows is part of a script. The **type** attribute specifies the MIME type of the script as well as the **scripting language** used in the script—in

this case, a text file written in `javascript`. In HTML5, the default MIME type for a `<script>` is `"text/html"`, so you can omit the type attribute from your `<script>` tags. We've introduced this here, because you'll see it in legacy HTML documents with embedded JavaScripts.

## Strings

Lines 11–12 instruct the browser's JavaScript interpreter to perform an **action**, namely, to display in the web page the **string** of characters contained between the **double quotation ( " ) marks** (also called a **string literal**). Individual white-space characters between words in a string are *not* ignored by the browser. However, if consecutive spaces appear in a string, browsers condense them to a single space. Also, browsers ignore *leading white-space characters* (i.e., white space at the beginning of a string).



### SOFTWARE ENGINEERING OBSERVATION 6.1

*Strings in JavaScript can be enclosed in either double quotation marks ( " ) or single quotation marks ( ' ).*

---

## Using the `document` Object

Lines 11–12 use the browser's **document object**, which represents the HTML5 document the browser is currently displaying. This object allows you to specify text to display in the HTML5 document. The browser creates a set of objects that allow you to access and manipulate *every* element of an HTML5 document. In the next several chapters, we overview some of these objects as we discuss the Document Object Model (DOM).

An object resides in the computer's memory and contains information used by the script. The term **object** normally implies that **attributes (data)** and **behaviors (methods)** are associated with the object. The object's methods use the attributes to perform useful actions for the **client of the object** (i.e., the script that calls the methods). A method may

require additional information (**arguments**) to perform its actions; this information is enclosed in parentheses after the name of the method in the script. In lines 11–12, we call the `document` object's **`writeln` method** to write a line of HTML5 markup in the HTML5 document. The parentheses following the method name `writeln` contain the one argument that method `writeln` requires (in this case, the string of HTML5 that the browser is to display). Method `writeln` instructs the browser to write the argument string into the web page for rendering. If the string contains HTML5 elements, the browser interprets these elements and renders them on the screen. In this example, the browser displays the phrase `Welcome to JavaScript Programming!` as an `h1`-level HTML5 heading, because the phrase is enclosed in an `h1` element.

## Statements

The code elements in lines 11–12, including `document.writeln`, its argument in the parentheses (the string) and the **semicolon** (`;`), together are called a **statement**. Every statement ends with a semicolon (also known as the **statement terminator**)—although this practice is *not* required by JavaScript, it's recommended as a way of avoiding subtle problems. Line 14 indicates the end of the script. In line 15, the tags `<body>` and `</body>` specify that this HTML5 document has an empty body.



### GOOD PROGRAMMING PRACTICE 6.1

*Terminate every statement with a semicolon. This notation clarifies where one statement ends and the next statement begins.*



### COMMON PROGRAMMING ERROR 6.1

*Forgetting the ending `</script>` tag for a script may prevent the browser from interpreting the script properly and may prevent the HTML5 document from loading properly.*

---

Open the HTML5 document in your browser. If the script contains no syntax errors, it should produce the output shown in [Fig. 6.1](#).

---



#### COMMON PROGRAMMING ERROR 6.2

JavaScript is case sensitive. *Not using the proper uppercase and lowercase letters is a syntax error. A syntax error occurs when the script interpreter cannot recognize a statement. The interpreter normally issues an error message to help you locate and fix the incorrect statement. Syntax errors are violations of the rules of the programming language. The interpreter notifies you of a syntax error when it attempts to execute the statement containing the error. Each browser has its own way to display JavaScript Errors. For example, Firefox has the Error Console (in its Web Developer menu) and Chrome has the JavaScript console (in its Tools menu). To view script errors in IE9, select **Internet Options...** from the **Tools** menu. In the dialog that appears, select the **Advanced** tab and click the checkbox labeled **Display a notification about every script error** under the **Browsing** category.*

---



#### ERROR-PREVENTION TIP 6.1

*When the interpreter reports a syntax error, sometimes the error is not in the line indicated by the error message. First, check the line for which the error was reported. If that line does not contain errors, check the preceding several lines in the script.*

---

**A Note About** `document.writeln`



In this example, we displayed an `h1` HTML5 element in the web browser by using `document.writeln` to write the element into the web page. For simplicity in [Chapters 6–9](#), we’ll continue to do this as we focus on presenting fundamental JavaScript programming concepts. Typically, you’ll display content by modifying an existing element in a web page—a technique we’ll begin using in [Chapter 10](#).

## A Note About Embedding JavaScript Code into HTML5 Documents

In [Section 4.5](#), we discussed the benefits of placing CSS3 code in external style sheets and linking them to your HTML5 documents. For similar reasons, JavaScript code is typically placed in a separate file, then included in the HTML5 document that uses the script. This makes the code more reusable, because it can be included into any HTML5 document—as is the case with the many JavaScript libraries used in professional web development today. We’ll begin separating both CSS3 and JavaScript into separate files starting in [Chapter 10](#).

## 6.3. Modifying Your First Script

This section continues our introduction to JavaScript programming with two examples that modify the example in [Fig. 6.1](#).

### Displaying a Line of Colored Text

A script can display `Welcome to JavaScript Programming!` in many ways. [Figure 6.2](#) displays the text in magenta, using the CSS `color` property. Most of this example is identical to [Fig. 6.1](#), so we concentrate only on lines 11–13 of [Fig. 6.2](#), which display one line of text in the document. The first statement uses `document` method `write` to display a string. Unlike `writeln`, `write` does not position the output cursor in the HTML5 document at the beginning of the next line after writing its argument. [Note: The output cursor keeps track of where the next character appears in the document’s markup, not where the next character appears in the web page as rendered by the browser.] The next character written in the document appears immediately after the last character written with `write`. Thus, when lines 12–13 execute, the first character written, “W,” appears immediately after the last character displayed with `write` (the



> character inside the right double quote in line 11). Each `write` or `writeln` statement resumes writing characters where the last `write` or `writeln` statement stopped writing characters. So, after a `writeln` statement, the next output appears on the beginning of the next line. Thus, the two statements in lines 11–13 result in one line of HTML5 text. Remember that statements in JavaScript are separated by semicolons ( ; ). Therefore, lines 12–13 represent only one complete statement. JavaScript allows large statements to be split over many lines. The `+` operator (called the “concatenation operator” when used in this manner) in line 12 joins two strings together—it’s explained in more detail later in this chapter.

---

```

1  <!DOCTYPE html>
2
3  <!-- Fig. 6.2: welcome2.html -->
4  <!-- Printing one line with multiple statements. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Printing a Line with Multiple Statements</title>
9      <script type = "text/javascript">
10         <!--
11         document.write( "<h1 style = 'color: magenta'>" );
12         document.write( "Welcome to JavaScript " +
13             "Programming!</h1>" );
14         // -->
15     </script>
16 </head><body></body>
17 </html>

```



Fig. 6.2. Printing one line with separate statements.



#### COMMON PROGRAMMING ERROR 6.3

*Splitting a JavaScript statement in the middle of a string is a syntax error.*

---

The preceding discussion has nothing to do with the actual *rendering* of the HTML5 text. Remember that the browser does *not* create a new line of text unless the browser window is too narrow for the text being rendered or the browser encounters an HTML5 element that explicitly starts a new line—for example, `<p>` to start a new paragraph.



#### COMMON PROGRAMMING ERROR 6.4

*Many people confuse the writing of HTML5 text with the rendering of HTML5 text. Writing HTML5 text creates the HTML5 that will be rendered by the browser for presentation to the user.*

---

## Nesting Quotation Marks

Recall that a string can be delimited by single ( `'` ) or double ( `"` ) quote characters. Within a string, you can't nest quotes of the same type, but you can nest quotes of the other type. A string that's delimited by double quotes, can contain single quotes. Similarly, a string that's delimited by single quotes, can contain nest double quotes. Line 11 nests single quotes inside a double-quoted string to quote the `style` attribute's value in the `h1` element.

## Displaying Text in an Alert Dialog

The first two scripts in this chapter display text in the HTML5 document. Sometimes it's useful to display information in windows called **dialogs**

(or **dialog boxes**) that “pop up” on the screen to grab the user’s attention. Dialogs typically display important messages to users browsing the web page. JavaScript allows you easily to display a dialog box containing a message. The script in [Fig. 6.3](#) displays Welcome to JavaScript Programming! as three lines in a predefined dialog called an **alert dialog**.

---

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 6.3: welcome3.html -->
4 <!-- Alert dialog displaying multiple lines. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Printing Multiple Lines in a Dialog Box</title>
9     <script type = "text/javascript">
10      <!--
11       window.alert( "Welcome to\nJavaScript\nProgramming!" );
12      // -->
13     </script>
14   </head>
15   <body>
16     <p>Click Refresh (or Reload) to run this script again.</p>
17   </body>
18 </html>
```

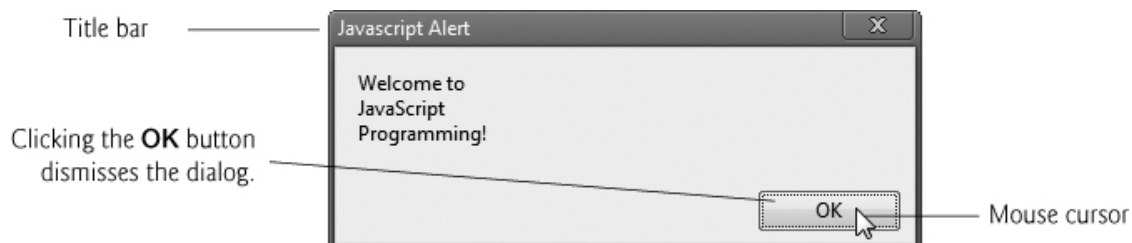


Fig. 6.3. Alert dialog displaying multiple lines.

## The window Object

Line 11 in the script uses the browser's **window** object to display an alert dialog. The argument to the **window** object's **alert** method is the string to display. Executing the preceding statement displays the dialog shown in [Fig. 6.3](#). The **title bar** of this Chrome dialog contains the string **JavaScript Alert** to indicate that the browser is presenting a message to the user. The dialog provides an **OK** button that allows the user to **dismiss** (i.e., **close**) the dialog by clicking the button. To dismiss the dialog, position the **mouse cursor** (also called the **mouse pointer**) over the **OK** button and click the mouse, or simply press the *Enter* key. The contents of the dialog vary by browser. You can refresh the page to run the script again.

## Escape Sequences

The **alert** dialog in this example contains three lines of plain text. Normally, a dialog displays a string's characters exactly as they appear. However, the dialog does not display the characters `\n` (line 11). The **backslash** (`\`) in a string is an **escape character**. It indicates that a "special" character is to be used in the string. When a backslash is encountered in a string, the next character is combined with the backslash to form an **escape sequence**. The escape sequence `\n` is the **newline character**, which causes the **cursor** (i.e., the current screen position indicator) to move to the beginning of the *next* line in the dialog. Some other common JavaScript escape sequences are listed in [Fig. 6.4](#). The `\n` and `\t` escape sequences in the table do not affect HTML5 rendering unless they're in a **pre element** (this element displays the text between its tags in a fixed-width font exactly as it's formatted between the tags, including leading white-space characters and consecutive white-space characters).

Escape sequence	Description
<code>\n</code>	<i>New line</i> —position the screen cursor at the beginning of the next line.
<code>\t</code>	<i>Horizontal tab</i> —move the screen cursor to the next tab stop.
<code>\\</code>	<i>Backslash</i> —used to represent a backslash character in a string.
<code>\"</code>	<i>Double quote</i> —used to represent a double-quote character in a string contained in double quotes. For example, <pre>window.alert( "\"in double quotes\"" );</pre> displays "in double quotes" in an alert dialog.
<code>\'</code>	<i>Single quote</i> —used to represent a single-quote character in a string. For example, <pre>window.alert( "'"in single quotes'" );</pre> displays 'in single quotes' in an alert dialog.

Fig. 6.4. Some common escape sequences.

## 6.4. Obtaining User Input with `prompt` Dialogs

Scripting gives you the ability to generate part or all of a web page's content at the time it's shown to the user. A script can adapt the content based on input from the user or other variables, such as the time of day or the type of browser used by the client. Such web pages are said to be *dynamic*, as opposed to *static*, since their content has the ability to change. The next two subsections use scripts to demonstrate dynamic web pages.

### 6.4.1. Dynamic Welcome Page

Our next script creates a dynamic welcome page that obtains the user's name, then displays it on the page. The script uses another *predefined* dialog box from the `window` object—a **prompt** dialog—which allows the user to enter a value that the script can use. The script asks the user to enter a name, then displays the name in the HTML5 document. [Figure 6.5](#) presents the script and sample output. In later chapters, we'll obtain inputs via GUI components in HTML5 forms, as introduced in [Chapters 2–3](#).]

---

```
1 <!DOCTYPE html>
```

```
2
```

```
3 <!-- Fig. 6.5: welcome4.html -->
```

```
4  <!-- Prompt box used on a welcome screen -->
5  <html>
6  <head>
7    <meta charset = "utf-8">
8    <title>Using Prompt and Alert Boxes</title>
9    <script type = "text/javascript">
10     <!--
11     var name; // string entered by the user
12
13     // read the name from the prompt box as a string
14     name = window.prompt( "Please enter your name" );
15
16     document.writeln( "<h1>Hello " + name +
17       ", welcome to JavaScript programming!</h1>" );
18     // -->
19   </script>
20 </head><body></body>
21 </html>
```

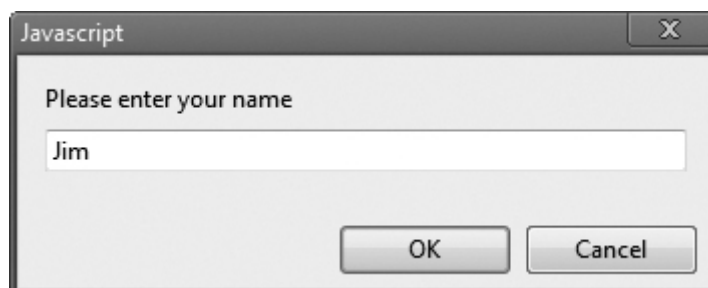


Fig. 6.5. Prompt box used on a welcome screen.

## Declarations, Keywords and Variables

Line 11 is a **declaration** that contains the JavaScript **keyword** `var` . Keywords are words that have special meaning in JavaScript. The keyword `var` at the beginning of the statement indicates that the word `name` is a **variable**. A variable is a location in the computer's memory where a value can be stored for use by a script. All variables have a *name* and *value*, and should be declared with a `var` statement before they're used in a script.

## Identifiers and Case Sensitivity

The name of a variable can be any valid **identifier**. An identifier is a series of characters consisting of letters, digits, underscores ( `_` ) and dollar signs ( `$` ) that does *not* begin with a digit and is *not* a reserved JavaScript keyword. [Note: A complete list of reserved keywords can be found in [Fig. 7.2](#).] Identifiers may *not* contain spaces. Some valid identifiers are `Welcome` , `$value` , `_value` , `m_inputField1` and `button7` . The name `7button` is not a valid identifier, because it begins with a digit, and the name `input field` is not valid, because it contains a space. Remember that JavaScript is **case sensitive**—uppercase and lowercase letters are considered to be different characters, so `name` , `Name` and `NAME` are different identifiers.



### GOOD PROGRAMMING PRACTICE 6.2

*Choosing meaningful variable names helps a script to be “self-documenting” (i.e., easy to understand by simply reading the script).*



### GOOD PROGRAMMING PRACTICE 6.3

*By convention, variable-name identifiers begin with a lower-case first letter. Each subsequent word should begin with a capital first letter. For example, identifier `itemPrice` has a capital *P* in its second word, *Price* .*





#### COMMON PROGRAMMING ERROR 6.5

*Splitting a statement in the middle of an identifier is a syntax error.*

---

Declarations end with a *semicolon* and can be split over several lines with each variable in the declaration separated by a *comma*—known as a **comma-separated list** of variable names. Several variables may be declared either in one or in multiple declarations.

### JavaScript Comments

It's helpful to indicate the purpose of each variable in the script by placing a JavaScript comment at the end of each line in the declaration. In line 11, a **single-line comment** that begins with the characters `//` states the purpose of the variable in the script. This form of comment is called a single-line comment because it terminates at the end of the line in which it appears. A `//` comment can begin at any position in a line of JavaScript code and continues until the end of the line. Comments do not cause the browser to perform any action when the script is interpreted; rather, comments are *ignored* by the JavaScript interpreter.

---



#### GOOD PROGRAMMING PRACTICE 6.4

*Although it's not required, declare each variable on a separate line. This allows for easy insertion of a comment next to each declaration. This is a widely followed professional coding standard.*

---

### Multiline Comments

You can also write **multiline comments**. For example, is a multiline comment spread over several lines. Such comments begin with the delimiter `/*` and end with the delimiter `*/`. All text between the delimiters of the comment is *ignored* by the interpreter.

```
/* This is a multiline  
comment. It can be  
split over many lines. */
```

JavaScript adopted comments delimited with `/*` and `*/` from the C programming language and single-line comments delimited with `//` from the C++ programming language. JavaScript programmers generally prefer C++-style single-line comments over C-style comments. Throughout this book, we use C++-style single-line comments.

### window **Object's** prompt **Method**

Line 13 is a comment indicating the purpose of the statement in the next line. Line 14 calls the `window` object's `prompt` method, which displays the dialog in [Fig. 6.6](#). The dialog allows the user to enter a string representing the user's name.

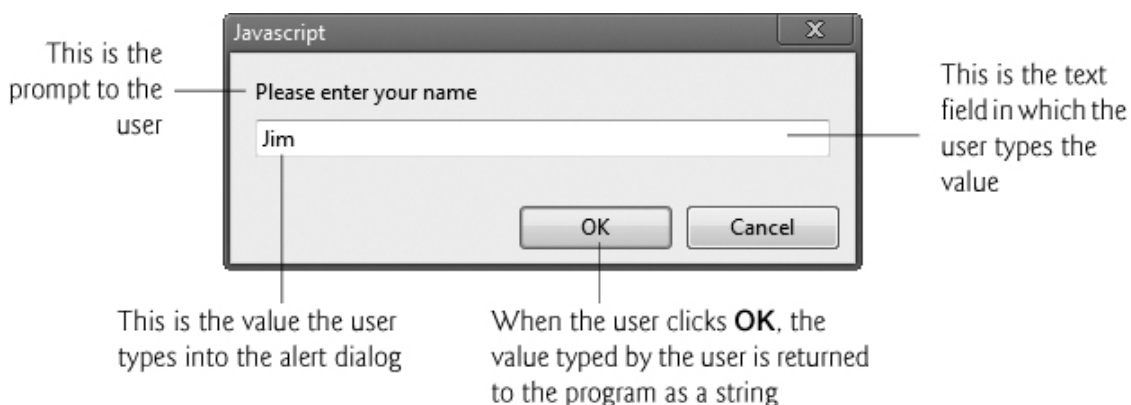


Fig. 6.6. Prompt dialog displayed by the `window` object's `prompt` method.

The argument to `prompt` specifies a message telling the user what to type in the text field. This message is called a *prompt* because it directs the user to take a specific action. An optional second argument, separated from the first by a comma, may specify the default string displayed in the text field; our code does not supply a second argument. In this case, most browsers leave the text field empty, and Internet Explorer displays the

default value `undefined`. The user types characters in the text field, then clicks the **OK** button to submit the string to the script. We normally receive input from a user through a GUI component such as the `prompt` dialog, as in this script, or through an HTML5 form GUI component, as we'll see in later chapters.

The user can type anything in the text field of the `prompt` dialog. For this script, whatever the user enters is considered the name. If the user clicks the **Cancel** button, no string value is sent to the script. Instead, the `prompt` dialog submits the value `null`, a JavaScript keyword signifying that a variable has no value. Note that `null` is not a string literal, but rather a predefined term indicating the absence of value. Writing a `null` value to the document, however, displays the word `null` in the web page.

## Assignment Operator

The statement in line 14 *assigns* the value returned by the `window` object's `prompt` method (a string containing the characters typed by the user—or the default value or `null` if the **Cancel** button is clicked) to variable `name` by using the **assignment operator**, `=`. The statement is read as, “`name` gets the value returned by `window.prompt("Please enter your name")`.” The `=` operator is called a **binary operator** because it has *two operands*—`name` and the result of the expression `window.prompt("Please enter your name")`. This entire statement is called an **assignment** because it assigns a value to a variable. The expression to the right of the assignment operator is always evaluated *first*.



### GOOD PROGRAMMING PRACTICE 6.5

*Place a space on each side of a binary operator. This format makes the operator stand out and makes the script more readable.*

---

## String Concatenation

Lines 16–17 use `document.writeln` to display the new welcome message. The expression inside the parentheses uses the operator `+` to “add” a string (the literal `"<h1>Hello, "`), the variable `name` (the string that the user entered in line 14) and another string (the literal `", welcome to JavaScript programming!</h1>"`). JavaScript has a version of the `+` operator for **string concatenation** that enables a string and a value of another data type (including another string) to be combined. The result of this operation is a new (and normally longer) string. If we assume that `name` contains the string literal `"Jim"`, the expression evaluates as follows: JavaScript determines that the two operands of the first `+` operator (the string `"<h1>Hello, "` and the value of variable `name`) are both strings, then concatenates the two into one string. Next, JavaScript determines that the two operands of the second `+` operator (the result of the first concatenation operation, the string `"<h1>Hello, Jim"`, and the string `", welcome to JavaScript programming!</h1>"`) are both strings and concatenates the two. This results in the string `"<h1>Hello, Jim, welcome to JavaScript programming!</h1>"`. The browser renders this string as part of the HTML5 document. Note that the space between `Hello,` and `Jim` is part of the string `"<h1>Hello, "`.

As you’ll see later, the `+` operator used for string concatenation can convert other variable types to strings if necessary. Because string concatenation occurs between two strings, JavaScript must convert other variable types to strings before it can proceed with the operation. For example, if a variable `age` has an integer value equal to `21`, then the expression `"my age is " + age` evaluates to the string `"my age is 21"`. JavaScript converts the value of `age` to a string and concatenates it with the existing string literal `"my age is "`.

After the browser interprets the `<head>` section of the HTML5 document (which contains the JavaScript), it then interprets the `<body>` of the HTML5 document (which is empty; line 20) and renders the HTML5. The HTML5 page is *not* rendered until the prompt is dismissed because the prompt pauses execution in the `head`, before the `body` is processed. If you reload the page after entering a name, the browser will execute the script again and so you can change the name.

### 6.4.2. Adding Integers

Our next script illustrates another use of `prompt` dialogs to obtain input from the user. [Figure 6.7](#) inputs two *integers* (whole numbers, such as 7, –11, 0 and 31914) typed by a user at the keyboard, computes the sum of the values and displays the result.

Lines 11–15 declare the variables `firstNumber`, `secondNumber`, `number1`, `number2` and `sum`. Single-line comments state the purpose of each of these variables. Line 18 employs a `prompt` dialog to allow the user to enter a string representing the first of the two integers that will be added. The script assigns the first value entered by the user to the variable `firstNumber`. Line 21 displays a `prompt` dialog to obtain the second number to add and assigns this value to the variable `secondNumber`.

---

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 6.7: addition.html -->
4  <!-- Addition script. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>An Addition Program</title>
9      <script type = "text/javascript">
10         <!--
11         var firstNumber; // first string entered by user
12         var secondNumber; // second string entered by user
13         var number1; // first number to add
14         var number2; // second number to add
15         var sum; // sum of number1 and number2
16
17         // read in first number from user as a string
18         firstNumber = window.prompt( "Enter first integer" );
19
20         // read in second number from user as a string
21         secondNumber = window.prompt( "Enter second integer" );
```

```

22
23 // convert numbers from strings to integers
24 number1 = parseInt( firstNumber );
25 number2 = parseInt( secondNumber );
26
27 sum = number1 + number2; // add the numbers
28
29 // display the results
30 document.writeln( "<h1>The sum is " + sum + "</h1>" );
31 // -->
32 </script>
33 </head><body></body>
34 </html>

```

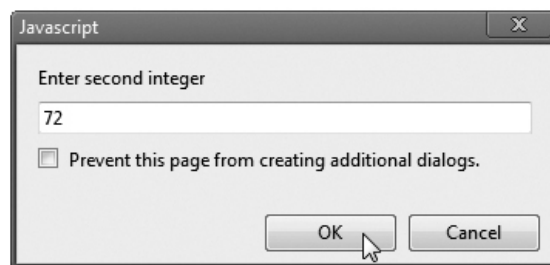
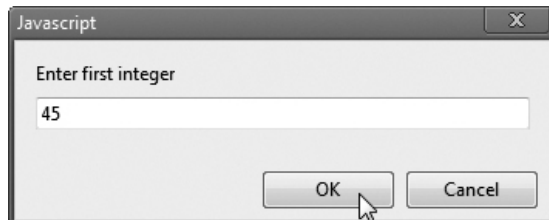


Fig. 6.7. Addition script.

As in the preceding example, the user can type anything in the prompt dialog. For this script, if the user either types a non-integer value or clicks the **Cancel** button, a logic error will occur, and the sum of the two values will appear in the HTML5 document as **NaN** (meaning **not a number**). A

logic error is caused by syntactically correct code that produces an incorrect result. In [Chapter 11](#), we discuss the `Number` object and its methods that can determine whether a value is a number.

Recall that a `prompt` dialog returns to the script as a string the value typed by the user. Lines 24–25 convert the two strings input by the user to integer values that can be used in a calculation. Function `parseInt` converts its string argument to an integer. Line 24 assigns to the variable `number1` the integer that function `parseInt` returns. Similarly, line 25 assigns an integer value to variable `number2`. Any subsequent references to `number1` and `number2` in the script use these integer values. We refer to `parseInt` as a **function** rather than a method because we do not precede the function call with an object name (such as `document` or `window`) and a dot (`.`). The term method means that the function belongs to a particular object. For example, method `writeln` belongs to the `document` object and method `prompt` belongs to the `window` object.

Line 27 calculates the sum of the variables `number1` and `number2` using the **addition operator**, `+`, and assigns the result to variable `sum` by using the assignment operator, `=`. Notice that the `+` operator can perform both addition and string concatenation. In this case, the `+` operator performs addition, because *both* operands contain integers. After line 27 performs this calculation, line 30 uses `document.writeln` to display the result of the addition on the web page.



#### COMMON PROGRAMMING ERROR 6.6

*Confusing the `+` operator used for string concatenation with the `+` operator used for addition often leads to undesired results. For example, if integer variable `y` has the value 5, the expression `"y + 2 = " + y + 2` results in `"y + 2 = 52"`, not `"y + 2 = 7"`, because first the value of `y` (i.e., 5) is concatenated with the string `"y + 2 = "`, then the value 2 is concatenated with the new, larger string `"y + 2 = 5"`. The expression `"y + 2 = " + (y + 2)` produces the string `"y +`*



*2 = 7" because the parentheses ensure that  $y + 2$  is calculated.*

---

## Validating JavaScript

As discussed in the Preface, we validated our code using HTML5, CSS3 and JavaScript validation tools. Browsers are generally forgiving and don't typically display error messages to the user. As a programmer, you should thoroughly test your web pages and validate them. Validation tools report two types of messages—*errors* and *warnings*. Typically, you must resolve errors; otherwise, your web pages probably won't render or execute correctly. Pages with warnings normally render and execute correctly; however, some organizations have strict protocols indicating that all pages must be free of both warnings and errors before they can be posted on a live website.

When you validate this example at [www.javascriptlint.com](http://www.javascriptlint.com), lines 24–25 produce the warning message:

`parseInt` missing radix parameter

Function `parseInt` has an optional second parameter, known as the *radix*, that specifies the base number system that's used to parse the number (e.g., 8 for octal, 10 for decimal and 16 for hexadecimal). The default is base 10, but you can specify any base from 2 to 32. For example, the following statement indicates that `firstNumber` should be treated as a decimal (base 10) integer:

```
number1 = parseInt( firstNumber, 10 );
```

This prevents numbers in other formats like octal (base 8) from being converted to incorrect values.

## 6.5. Memory Concepts

Variable names such as `number1`, `number2` and `sum` actually correspond to **locations** in the computer's memory. Every variable has a **name**, a **type** and a **value**.

In the addition script in [Fig. 6.7](#), when line 24 executes, the string `firstNumber` (previously entered by the user in a `prompt` dialog) is converted to an integer and placed into a memory location to which the name `number1` has been assigned by the interpreter. Suppose the user entered the string `45` as the value for `firstNumber`. The script converts `firstNumber` to an integer, and the computer places the integer value `45` into location `number1`, as shown in [Fig. 6.8](#). Whenever a value is placed in a memory location, the value *replaces* the previous value in that location. The previous value is lost.

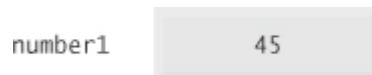


Fig. 6.8. Memory location showing the name and value of variable `number1`.

Suppose that the user enters `72` as the second integer. When line 25 executes, the script converts `secondNumber` to an integer and places that integer value, `72`, into location `number2`; then the memory appears as shown in [Fig. 6.9](#).

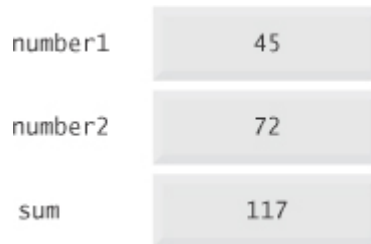


Fig. 6.9. Memory locations after inputting values for variables `number1` and `number2`.

Once the script has obtained values for `number1` and `number2`, it adds the values and places the sum into variable `sum`. The statement

```
sum = number1 + number2;
```

performs the addition and also replaces `sum`'s previous value. After `sum` is calculated, the memory appears as shown in [Fig. 6.10](#). Note that the values of `number1` and `number2` appear exactly as they did before they were used in the calculation of `sum`. These values were used, but not destroyed, when the computer performed the calculation—when a value is read from a memory location, the process is *nondestructive*.



number1	45
number2	72
sum	117

Fig. 6.10. Memory locations after calculating the sum of number1 and number2 .

## Data Types in JavaScript

Unlike its predecessor languages C, C++ and Java, *JavaScript does not require variables to have a declared type before they can be used in a script*. A variable in JavaScript can contain a value of *any* data type, and in many situations JavaScript automatically converts between values of different types for you. For this reason, JavaScript is referred to as a **loosely typed language**. When a variable is declared in JavaScript, but is not given a value, the variable has an **undefined** value. Attempting to use the value of such a variable is normally a logic error.

When variables are declared, they're not assigned values unless you specify them. Assigning the value `null` to a variable indicates that it does *not* contain a value.

## 6.6. Arithmetic

Many scripts perform arithmetic calculations. [Figure 6.11](#) summarizes the **arithmetic operators**. Note the use of various special symbols not used in algebra. The **asterisk ( \* )** indicates multiplication; the **percent sign ( % )** is the **remainder operator**, which will be discussed shortly. The arithmetic operators in [Fig. 6.11](#) are *binary* operators, because each operates on *two* operands. For example, the expression `sum + value` contains the binary operator `+` and the two operands `sum` and `value` .

JavaScript operation	Arithmetic operator	Algebraic expression	JavaScript expression
Addition	+	$f + 7$	<code>f + 7</code>
Subtraction	-	$p - c$	<code>p - c</code>
Multiplication	*	$bm$	<code>b * m</code>
Division	/	$x/y$ or $\frac{x}{y}$ or $x \div y$	<code>x / y</code>
Remainder	%	$r \bmod s$	<code>r % s</code>

Fig. 6.11. Arithmetic operators.

### Remainder Operator, %

JavaScript provides the remainder operator, %, which yields the remainder after division. The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `17 % 5` yields 2 (i.e., 17 divided by 5 is 3, with a remainder of 2), and `7.4 % 3.1` yields 1.2. In later chapters, we consider applications of the remainder operator, such as determining whether one number is a *multiple* of another. *There's no arithmetic operator for exponentiation in JavaScript.* ([Chapter 8](#) shows how to perform exponentiation in JavaScript using the `Math` object's `pow` method.)

Arithmetic expressions in JavaScript must be written in straight-line form to facilitate entering scripts into the computer. Thus, expressions such as “`a` divided by `b`” must be written as `a / b`, so that all constants, variables and operators appear in a *straight line*. The following algebraic notation is generally *not* acceptable to computers:

$$\frac{a}{b}$$

Parentheses are used to *group* expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c` we write:

$$a * (b + c)$$

### Operator Precedence

JavaScript applies the operators in arithmetic expressions in a precise sequence determined by the following **rules of operator precedence**, which are generally the same as those followed in algebra:

1. Multiplication, division and remainder operations are applied *first*. If an expression contains several multiplication, division and remainder operations, operators are applied from *left to right*. Multiplication, division and remainder operations are said to have the *same level of precedence*.
2. Addition and subtraction operations are applied next. If an expression contains several addition and subtraction operations, operators are applied from *left to right*. Addition and subtraction operations have the *same level of precedence*.

The rules of operator precedence enable JavaScript to apply operators in the correct order. When we say that operators are applied from *left to right*, we’re referring to the **associativity** of the operators—the *order* in which operators of equal priority are evaluated. We’ll see that some operators associate from *right to left*. [Figure 6.12](#) summarizes the rules of operator precedence. The table in [Fig. 6.12](#) will be expanded as additional JavaScript operators are introduced. A complete precedence chart is included in Appendix C.

Operator(s)	Operation(s)	Order of evaluation (precedence)
<code>*</code> , <code>/</code> or <code>%</code>	Multiplication Division Remainder	Evaluated first. If there are several such operations, they’re evaluated from left to right.
<code>+</code> or <code>-</code>	Addition Subtraction	Evaluated last. If there are several such operations, they’re evaluated from left to right.

Fig. 6.12. Precedence of arithmetic operators.

Let’s consider several algebraic expressions. Each example lists an algebraic expression and the equivalent JavaScript expression.

The following is an example of an arithmetic mean (average) of five terms:

Algebra:  $m = \frac{a + b + c + d + e}{5}$

Parentheses are required to group the addition operators, because division has higher precedence than addition. The *entire quantity* (  $a + b + c + d + e$  ) is to be divided by 5 . If the parentheses are erroneously omitted, we obtain  $a + b + c + d + e / 5$  , which evaluates as

and would not lead to the correct answer.

The following is an example of the equation of a straight line:

No parentheses are required. The multiplication operator is applied first, because multiplication has a higher precedence than addition. The assignment occurs last, because it has a lower precedence than multiplication and addition.

As in algebra, it's acceptable to use *unnecessary parentheses* in an expression to make the expression clearer. These are also called **redundant parentheses**. For example, the preceding second-degree polynomial might be parenthesized as follows:

$$y = (a * x * x) + (b * x) + c;$$

## 6.7. Decision Making: Equality and Relational Operators

This section introduces a version of JavaScript's **if statement** that allows a script to make a decision based on the truth or falsity of a **condition**. If the condition is met (i.e., the condition is *true*), the statement in the body of the `if` statement is executed. If the condition is *not* met (i.e., the condition is *false*), the statement in the body of the `if` statement is *not* executed. We'll see an example shortly.

Conditions in `if` statements can be formed by using the **equality operators** and **relational operators** summarized in [Fig. 6.13](#). The relational operators all have the *same* level of precedence and associate from left to right. The equality operators both have the same level of precedence, which is lower than the precedence of the relational operators. The equality operators also associate from left to right. Each comparison results in a value of `true` or `false`.

---



#### COMMON PROGRAMMING ERROR 6.7

*Confusing the equality operator, `==`, with the assignment operator, `=`, is a logic error. The equality operator should be read as “is equal to,” and the assignment operator should be read as “gets” or “gets the value of.” Some people prefer to read the equality operator as “double equals” or “equals equals.”*

---

Fig. 6.13. Equality and relational operators.

The script in [Fig. 6.14](#) uses four `if` statements to display a time-sensitive greeting on a welcome page. The script obtains the local time from the user’s computer and converts it from 24-hour clock format (0–23) to a 12-hour clock format (0–11). Using this value, the script displays an appropriate greeting for the current time of day. The script and sample output are shown in [Fig. 6.14](#). Lines 11–13 declare the variables used in the script.



Also note that JavaScript allows you to assign a value to a variable when it's declared.

## Creating and Using a New Date Object

Line 12 sets the variable `now` to a new **Date object**, which contains information about the current local time. In [Section 6.2](#), we introduced the `document` object, which encapsulates data pertaining to the current web page. Here, we use JavaScript's built-in `Date` object to acquire the current local time. We create a new object by using the `new` operator followed by the type of the object, in this case `Date`, and a pair of parentheses. Some objects require that arguments be placed in the parentheses to specify details about the object to be created. In this case, we leave the parentheses empty to create a *default* `Date` object containing information about the current date and time. After line 12 executes, the variable `now` refers to the new `Date` object. We did not need to use the `new` operator when we used the `document` and `window` objects because these objects always are created by the browser. Line 13 sets the variable `hour` to an integer equal to the current hour (in a 24-hour clock format) returned by the `Date` object's `getHours` method. [Chapter 11](#) presents a more detailed discussion of the `Date` object's attributes and methods, and of objects in general. The script uses `window.prompt` to allow the user to enter a name to display as part of the greeting (line 16).

---

```
1 <!DOCTYPE html>
2
3 <!-- Fig. 6.14: welcome5.html -->
4 <!-- Using equality and relational operators. -->
5 <html>
6   <head>
7     <meta charset = "utf-8">
8     <title>Using Relational Operators</title>
9     <script type = "text/javascript">
10       <!--
11       var name; // string entered by the user
12       var now = new Date();    // current date and time
```

```
13     var hour = now.getHours(); // current hour (0-23)
14
15     // read the name from the prompt box as a string
16     name = window.prompt( "Please enter your name" );
17
18     // determine whether it's morning
19     if ( hour < 12 )
20         document.write( "<h1>Good Morning, " );
21
22     // determine whether the time is PM
23     if ( hour >= 12 )
24     {
25         // convert to a 12-hour clock
26         hour = hour - 12;
27
28         // determine whether it is before 6 PM
29         if ( hour < 6 )
30             document.write( "<h1>Good Afternoon, " );
31
32         // determine whether it is after 6 PM
33         if ( hour >= 6 )
34             document.write( "<h1>Good Evening, " );
35     } // end if
36
37     document.writeln( name +
38         ", welcome to JavaScript programming!</h1>" );
39     // -->
40     </script>
41     </head><body></body>
42 </html>
```

---

Fig. 6.14. Using equality and relational operators.

### Decision-Making with the `if` Statement

To display the correct time-sensitive greeting, the script must determine whether the user is visiting the page during the morning, afternoon or evening. The first `if` statement (lines 19–20) compares the value of variable `hour` with `12`. If `hour` is less than `12`, then the user is visiting the page during the morning, and the statement at line 20 outputs the string "Good morning". If this condition is not met, line 20 is not executed. Line 23 determines whether `hour` is greater than or equal to `12`. If `hour` is greater than or equal to `12`, then the user is visiting the page in either the afternoon or the evening. Lines 24–35 execute to determine the appropriate greeting. If `hour` is less than `12`, then the JavaScript interpreter does not execute these lines and continues to line 37.

### Blocks and Decision-Making with Nested `if` Statements

The brace `{` in line 24 begins a **block** of statements (lines 24–35) that are executed together if `hour` is greater than or equal to `12`. Line 26 subtracts `12` from `hour`, converting the current hour from a 24-hour clock format (0–23) to a 12-hour clock format (0–11). The `if` statement (line 29) determines whether `hour` is now less than `6`. If it is, then the time is between noon and 6 PM, and line 30 outputs the beginning of an HTML5 `h1` element (`"<h1>Good Afternoon, "`). If `hour` is greater than or equal to `6`, the time is between 6 PM and midnight, and the script outputs the greeting "Good Evening" (lines 33–34). The brace `}` in line 35 ends the block of statements associated with the `if` statement in line 23. Note that `if` statements can be **nested**—one `if` statement can be placed *inside* another. The `if` statements that determine whether the user is visiting the

page in the afternoon or the evening (lines 29–30 and lines 33–34) execute only if the script has already established that `hour` is greater than or equal to 12 (line 23). If the script has already determined the current time of day to be morning, these additional comparisons are not performed. [Chapter 7](#) discusses blocks and nested `if` statements. Finally, lines 37–38 output the rest of the HTML5 `h1` element (the remaining part of the greeting), which does not depend on the time of day.

---



#### GOOD PROGRAMMING PRACTICE 6.6

*Include comments after the closing curly brace of control statements (such as `if` statements) to indicate where the statements end, as in line 35 of [Fig. 6.14](#).*

---

Note the *indentation* of the `if` statements throughout the script. Such indentation enhances script readability.

---



#### GOOD PROGRAMMING PRACTICE 6.7

*Indent the statement in the body of an `if` statement to make the body of the statement stand out and to enhance script readability.*

---

## The Empty Statement

Note that there's *no* semicolon ( `;` ) at the end of the first line of each `if` statement. Including such a semicolon would result in a *logic error* at execution time. For example,

```
if ( hour < 12 );  
    document.write( "<h1>Good Morning, " );
```

would actually be interpreted by JavaScript erroneously as

```
if ( hour < 12 )  
;  
document.write( "<h1>Good Morning, " );
```

where the semicolon on the line by itself—called the **empty statement**—is the statement to execute if the condition in the `if` statement is true. When the empty statement executes, no task is performed in the script. The script then continues with the next statement, which executes regardless of whether the condition is true or false. In this example, "`<h1>Good Morning,` " would be printed *regardless* of the time of day.



#### ERROR-PREVENTION TIP 6.2

*A lengthy statement may be spread over several lines. If a single statement must be split across lines, choose breaking points that make sense, such as after a comma in a comma-separated list or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines.*

---

### Validating This Example's Script

When you validate this example with [www.javascriptlint.com](http://www.javascriptlint.com), the following warning message is displayed for the `if` statements in lines 19, 29 and 33:

block statement without curly braces

You saw that an `if` statement's body may contain multiple statements in a block that's delimited by curly braces (lines 23–35). The curly braces are not required for an `if` statement that has a one-statement body, such as the ones in lines 19, 29 and 33. Many programmers consider it a good practice to enclose *every* `if` statement's body in curly braces—in fact, many organizations require this. For this reason, the validator issues the preceding warning message. You can eliminate this example's warning

messages by enclosing the `if` statement bodies in curly braces. For example, the `if` at lines 19–20 can be written as:

```
if ( hour < 12 )
{
    document.write( "<h1>Good Morning, " );
}
```

## The Strict Equals ( `===` ) and Strict Does Not Equal ( `!==` ) Operators

As we mentioned in [Section 6.5](#), JavaScript can convert between types for you. This includes cases in which you're comparing values. For example, the comparison `"75" == 75` yields the value `true` because JavaScript converts the string `"75"` to the number `75` before performing the equality ( `==` ) comparison. To prevent implicit conversions in comparisons, which can lead to unexpected results, JavaScript provides the **strict equals** ( `===` ) and **strict does not equal** ( `!==` ) operators. The comparison `"75" === 75` yields the value `false` because one operand is a string and the other is a number. Similarly, `75" !== 75` yields `true` because the operand's types are not equal, therefore the values are not equal. If you do not use these operators when comparing values to `null`, `0`, `true`, `false` or the empty string ( `""` ), [javascriptlint.com](https://jshint.com/)'s JavaScript validator displays warnings of potential implicit conversions.

## Operator Precedence Chart

The chart in [Fig. 6.15](#) shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Note that all of these operators, with the exception of the assignment operator, `=`, associate from left to right. Addition is left associative, so an expression like `x + y + z` is evaluated as if it had been written as `(x + y) + z`. The assignment operator, `=`, associates from right to left, so an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

**GOOD PROGRAMMING PRACTICE 6.8**

*Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operations are performed in the order in which you expect them to be performed. If you're uncertain about the order of evaluation, use parentheses to force the order, exactly as you would do in algebraic expressions. Be sure to observe that some operators, such as assignment ( = ), associate from right to left rather than from left to right.*

---

Fig. 6.15. Precedence and associativity of the operators discussed so far.

## 6.8. Web Resources

[www.deitel.com/javascript](http://www.deitel.com/javascript)

The Deitel JavaScript Resource Center contains links to some of the best JavaScript resources on the web. There you'll find categorized links to JavaScript tools, code generators, forums, books, libraries, frameworks and more. Also check out the tutorials for all skill levels, from introductory to advanced.

## Summary

### Section 6.1 Introduction

- JavaScript ([p. 186](#)) is used to enhance the functionality and appearance of web pages.



## Section 6.2 Your First Script: Displaying a Line of Text with JavaScript in a Web Page

- Often, JavaScripts appear in the `<head>` section of the HTML5 document.
- The browser interprets the contents of the `<head>` section first.
- The `<script>` tag indicates to the browser that the text that follows is part of a script (p. 186). Attribute `type` (p. 187) specifies the MIME type of the scripting language used in the script—such as `text/javascript`.
- A string of characters (p. 187) can be contained between double ( `"` ) quotation marks (p. 187).
- A string (p. 187) is sometimes called a character string, a message or a string literal.
- The browser's `document` object (p. 188) represents the HTML5 document the browser is currently displaying. The `document` object allows a you to specify HTML5 text to display in the document.
- The browser creates a complete set of objects that allow you to access and manipulate every element of an HTML5 document.
- An object (p. 188) resides in the computer's memory and contains information used by the script. The term object normally implies that attributes (data) (p. 188) and behaviors (methods) (p. 188) are associated with the object. The object's methods use the attributes' data to perform useful actions for the client of the object (i.e., the script that calls the methods).
- The `document` object's `writeln` method (p. 188) writes a line of HTML5 text in a document.
- Every statement ends with a semicolon (also known as the statement terminator; p. 188), although this practice is not required by JavaScript.
- JavaScript is case sensitive. Not using the proper uppercase and lowercase letters is a syntax error.

## Section 6.3 Modifying Your First Script

- Sometimes it's useful to display information in windows called dialogs (or dialog boxes; [p. 191](#)) that “pop up” on the screen to grab the user's attention. Dialogs typically display important messages to the user browsing the web page.
- The browser's `window` object ([p. 191](#)) uses method `alert` ([p. 191](#)) to display an alert dialog.
- The escape sequence `\n` is the newline character ([p. 192](#)). It causes the cursor in the HTML5 document to move to the beginning of the next line.

## Section 6.4 Obtaining User Input with `prompt` Dialogs

- Keywords ([p. 193](#)) are words with special meaning in JavaScript.
- The keyword `var` ([p. 193](#)) at the beginning of the statement indicates that the word name is a variable. A variable ([p. 193](#)) is a location in the computer's memory where a value can be stored for use by a script. All variables have a name and value, and should be declared with a `var` statement before they're used in a script.
- The name of a variable can be any valid identifier consisting of letters, digits, underscores ( `_` ) and dollar signs ( `$` ) that does not begin with a digit and is not a reserved JavaScript keyword.
- Declarations end with a semicolon and can be split over several lines with each variable in the declaration separated by a comma—known as a comma-separated list of variable names. Several variables may be declared in one declaration or in multiple declarations.
- It's helpful to indicate the purpose of a variable in the script by placing a JavaScript comment at the end of the variable's declaration. A single-line comment ([p. 194](#)) begins with the characters `//` and terminates at the end of the line. Comments do not cause the browser to perform any action when the script is interpreted; rather, comments are ignored by the JavaScript interpreter.

- Multiline comments begin with the delimiter `/*` and end with the delimiter `*/`. All text between the delimiters of the comment is ignored by the interpreter.
- The `window` object's `prompt` method displays a dialog into which the user can type a value. The first argument is a message (called a prompt) that directs the user to take a specific action. An optional second argument, separated from the first by a comma, may specify the default string to be displayed in the text field.
- A variable is assigned a value with an assignment ([p. 196](#)), using the assignment operator, `=`. The `=` operator is called a binary operator ([p. 196](#)), because it has two operands ([p. 196](#)).
- JavaScript has a version of the `+` operator for string concatenation ([p. 196](#)) that enables a string and a value of another data type (including another string) to be concatenated.

## Section 6.5 Memory Concepts

- Every variable has a name, a type and a value.
- When a value is placed in a memory location, the value replaces the previous value in that location. When a value is read out of a memory location, the process is nondestructive.
- JavaScript does not require variables to have a declared type before they can be used in a script. A variable in JavaScript can contain a value of any data type, and in many situations, JavaScript automatically converts between values of different types for you. For this reason, JavaScript is referred to as a loosely typed language ([p. 200](#)).
- When a variable is declared in JavaScript, but is not given a value, it has an undefined value ([p. 200](#)). Attempting to use the value of such a variable is normally a logic error.
- When variables are declared, they're not assigned default values, unless you specify them. To indicate that a variable does not contain a value, you

can assign the value `null` to it.

## Section 6.6 Arithmetic

- The basic arithmetic operators ( `+` , `-` , `*` , `/` , and `%` ; [p. 200](#) ) are binary operators, because each operates on two operands.
- Parentheses can be used to group expressions in the same manner as in algebraic expressions.
- JavaScript applies the operators in arithmetic expressions in a precise sequence determined by the following rules of operator precedence ([p. 201](#)).
- When we say that operators are applied from left to right, we're referring to the associativity of the operators ([p. 201](#)). Some operators associate from right to left.

## Section 6.7 Decision Making: Equality and Relational Operators

- JavaScript's `if` statement ([p. 202](#)) allows a script to make a decision based on the truth or falsity of a condition. If the condition is met (i.e., the condition is true; [p. 202](#)), the statement in the body of the `if` statement is executed. If the condition is not met (i.e., the condition is false), the statement in the body of the `if` statement is not executed.
- Conditions in `if` statements can be formed by using the equality operators ([p. 202](#)) and relational operators ([p. 202](#)).

## Self-Review Exercises

**6.1** Fill in the blanks in each of the following statements:

- \_\_\_\_\_ begins a single-line comment.
- Every JavaScript statement should end with a(n) \_\_\_\_\_.
- The \_\_\_\_\_ statement is used to make decisions.

- d. The \_\_\_\_\_ object displays alert dialogs and prompt dialogs.
- e. \_\_\_\_\_ words are reserved for use by JavaScript.
- f. Methods \_\_\_\_\_ and \_\_\_\_\_ of the \_\_\_\_\_ object write HTML5 text into an HTML5 document.

**6.2** State whether each of the following is *true* or *false*. If *false*, explain why.

- a. Comments cause the computer to print the text after the `//` on the screen when the script is executed.
- b. JavaScript considers the variables `number` and `Number` to be identical.
- c. The remainder operator (`%`) can be used only with numeric operands.
- d. The arithmetic operators `*`, `/`, `%`, `+` and `-` all have the same level of precedence.
- e. Method `parseInt` converts an integer to a string.

**6.3** Write JavaScript statements to accomplish each of the following tasks:

- a. Declare variables `c`, `thisIsAVariable`, `q76354` and `number`.
- b. Display a dialog asking the user to enter an integer. Show a default value of `0` in the dialog.
- c. Convert a string to an integer, and store the converted value in variable `age`. Assume that the string is stored in `stringValue`.
- d. If the variable `number` is not equal to `7`, display "The variable `number` is not equal to 7" in a message dialog.
- e. Output a line of HTML5 text that will display the message "This is JavaScript" in the HTML5 document.

**6.4** Identify and correct the errors in each of the following statements:

**a.**

```
if ( c < 7 );  
    window.alert( "c is less than 7" );
```

**b.**

```
if ( c => 7 )  
    window.alert( "c is equal to or greater than 7" );
```

**6.5** Write a statement (or comment) to accomplish each of the following tasks:

**a.** State that a script will calculate the product of three integers [*Hint: Use text that helps to document a script.*]

**b.** Declare the variables `x`, `y`, `z` and `result`.

**c.** Declare the variables `xVal`, `yVal` and `zVal`.

**d.** Prompt the user to enter the first value, read the value from the user and store it in the variable `xVal`.

**e.** Prompt the user to enter the second value, read the value from the user and store it in the variable `yVal`.

**f.** Prompt the user to enter the third value, read the value from the user and store it in the variable `zVal`.

**g.** Convert the string `xVal` to an integer, and store the result in the variable `x`.

**h.** Convert the string `yVal` to an integer, and store the result in the variable `y`.

**i.** Convert the string `zVal` to an integer, and store the result in the variable `z`.

- j. Compute the product of the three integers contained in variables `x`, `y` and `z`, and assign the result to the variable `result`.
  - k. Write a line of HTML5 text containing the string "The product is " followed by the value of the variable `result`.
- 6.6 Using the statements you wrote in Exercise 6.5, write a complete script that calculates and prints the product of three integers.

## Answers to Self-Review Exercises

### 6.1

- a. `//`.
- b. Semicolon (`;`).
- c. `if`.
- d. `window`.
- e. Keywords.
- f. `write`, `writeln`, `document`.

### 6.2

- a. False. Comments do not cause any action to be performed when the script is executed. They're used to document scripts and improve their readability.
- b. False. JavaScript is case sensitive, so these variables are distinct.
- c. True.
- d. False. The operators `*`, `/` and `%` are on the same level of precedence, and the operators `+` and `-` are on a lower level of precedence.
- e. False. Function `parseInt` converts a string to an integer value.

## 6.3

- a. `var c, thisIsAVariable, q76354, number;`
- b. `value = window.prompt( "Enter an integer", "0" );`
- c. `var age = parseInt( stringValue );`
- d.  
  
`if ( number != 7 )`  
`window.alert( "The variable number is not equal to 7" );`
- e. `document.writeln( "This is JavaScript" );`

## 6.4

- a. Error: There should not be a semicolon after the right parenthesis of the condition in the `if` statement.

Correction: Remove the semicolon after the right parenthesis. [*Note:* The result of this error is that the output statement is executed whether or not the condition in the `if` statement is true. The semicolon after the right parenthesis is considered an empty statement—a statement that does nothing.]

- b. Error: The relational operator `=>` is incorrect.

Correction: Change `=>` to `>=`.

## 6.5

- a. `// Calculate the product of three integers`
- b. `var x, y, z, result;`
- c. `var xVal, yVal, zVal;`
- d. `xVal = window.prompt( "Enter first integer:" , "0" );`



```
e. yVal = window.prompt( "Enter second integer:" , "0" );

f. zVal = window.prompt( "Enter third integer:" , "0" );

g. x = parseInt( xVal );

h. y = parseInt( yVal );

i. z = parseInt( zVal );

j. result = x * y * z;

k. document.writeln( "<h1>The product is " + result + "</h1>" );
```

**6.6** The script is as follows:

---

```
1 <!DOCTYPE html>
2
3 <!-- Exercise 6.6: product.html -->
4 <html>
5   <head>
6     <meta charset = "utf-8">
7     <title>Product of Three Integers</title>
8     <script type = "text/javascript">
9       <!--
10        // Calculate the product of three integers
11        var x, y, z, result;
12        var xVal, yVal, zVal;
13
14        xVal = window.prompt( "Enter first integer:" );
15        yVal = window.prompt( "Enter second integer:" );
16        zVal = window.prompt( "Enter third integer:" );
17
18        x = parseInt( xVal );
19        y = parseInt( yVal );
20        z = parseInt( zVal );
```

```
21
22     result = x * y * z;
23     document.writeln( "<h1>The product is " + result + "<h1>" );
24     // -->
25     </script>
26 </head><body></body>
27 </html>
```

---

## Exercises

6.7 Fill in the blanks in each of the following statements:

- a. \_\_\_\_\_ are used to document a script and improve its readability.
- b. A dialog capable of receiving input from the user is displayed with method \_\_\_\_\_ of object \_\_\_\_\_.
- c. A JavaScript statement that makes a decision is the \_\_\_\_\_ statement.
- d. Calculations are normally performed by \_\_\_\_\_ operators.
- e. Method \_\_\_\_\_ of object \_\_\_\_\_ displays a dialog with a message to the user.

**6.8** Write JavaScript statements that accomplish each of the following tasks:

- a. Display the message "Enter two numbers" using the window object.
- b. Assign the product of variables `b` and `c` to variable `a`.
- c. State that a script performs a sample payroll calculation.

**6.9** State whether each of the following is *true* or *false*. If *false*, explain why.

- a. JavaScript operators are evaluated from left to right.
- b. The following are all valid variable names: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales$`, `his_$account_total`, `a`, `b$`, `c`, `z`, `z2`.
- c. A valid JavaScript arithmetic expression with no parentheses is evaluated from left to right.
- d. The following are all invalid variable names: `3g`, `87`, `67h2`, `h22`, `2h`.

**6.10** Fill in the blanks in each of the following statements:

- a. What arithmetic operations have the same precedence as multiplication? \_\_\_\_\_.
- b. When parentheses are nested, which ones evaluate first? \_\_\_\_\_.
- c. A location in the computer's memory that may contain different values at various times throughout the execution of a script is called a \_\_\_\_\_.

**6.11** What displays in the alert dialog when each of the given JavaScript statements is performed? Assume that `x = 2` and `y = 3`.

- a. `window.alert( "x = " + x );`
- b. `window.alert( "The value of x + x is " + ( x + x ) );`
- c. `window.alert( "x =" );`

**d.** `window.alert( ( x + y ) + " = " + ( y + x ) );`

**6.12** Which of the following JavaScript statements contain variables whose values are changed?

**a.** `p = i + j + k + 7;`

**b.** `window.alert( "variables whose values are destroyed" );`

**c.** `window.alert( "a = 5" );`

**d.** `stringValue = window.prompt( "Enter string:" );`

**6.13** Given  $y = ax^3 + 7$ , which of the following are correct JavaScript statements for this equation?

**a.** `y = a * x * x * x + 7;`

**b.** `y = a * x * x * (x + 7);`

**c.** `y = (a * x) * x * (x + 7);`

**d.** `y = (a * x) * x * x + 7;`

**e.** `y = a * (x * x * x) + 7;`

**f.** `y = a * x * (x * x + 7);`

**6.14** State the order of evaluation of the operators in each of the following JavaScript statements, and show the value of `x` after each statement is performed.

**a.** `x = 7 + 3 * 6 / 2 - 1;`

**b.** `x = 2 % 2 + 2 * 2 - 2 / 2;`

**c.** `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

- 6.15** Write a script that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Write the script using the following methods:
- Using one `document.writeln` statement.
  - Using four `document.write` statements.
- 6.16** Write a script that asks the user to enter two numbers, obtains the two numbers from the user and outputs text that displays the sum, product, difference and quotient of the two numbers. Use the techniques shown in [Fig. 6.7](#).
- 6.17** Write a script that asks the user to enter two integers, obtains the numbers from the user and outputs text that displays the larger number followed by the words “ is larger ” in an alert dialog. If the numbers are equal, output HTML5 text that displays the message “ These numbers are equal .” Use the techniques shown in [Fig. 6.14](#).
- 6.18** Write a script that takes three integers from the user and displays the sum, average, product, smallest and largest of the numbers in an alert dialog.
- 6.19** Write a script that gets from the user the radius of a circle and outputs HTML5 text that displays the circle’s diameter, circumference and area. Use the constant value 3.14159 for  $\pi$ . Use the GUI techniques shown in [Fig. 6.7](#). [Note: You may also use the predefined constant `Math.PI` for the value of  $\pi$ . This constant is more precise than the value 3.14159. The `Math` object is defined by JavaScript and provides many common mathematical capabilities.] Use the following formulas ( $r$  is the radius):  $diameter = 2r$ ,  $circumference = 2\pi r$ ,  $area = \pi r^2$ .
- 6.20** Write a script that reads five integers and determines and outputs markup that displays the largest and smallest integers in the group. Use only the scripting techniques you learned in this chapter.
- 6.21** Write a script that reads an integer and determines and outputs HTML5 text that displays whether it’s odd or even. [Hint: Use the remainder oper-

ator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of zero when divided by 2.]

**6.22** Write a script that reads in two integers and determines and outputs HTML5 text that displays whether the first is a multiple of the second. [Hint: Use the remainder operator.]

**6.23** Write a script that inputs three numbers and determines and outputs markup that displays the number of negative numbers, positive numbers and zeros input.

**6.24** Write a script that calculates the squares and cubes of the numbers from 0 to 5 and outputs HTML5 text that displays the resulting values in an HTML5 table format, as show below. [Note: This script does not require any input from the user.]

[Support](#)   [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) [PRIVACY POLICY](#)