7

JavaScript: Introduction to Scripting

Objectives

- To be able to write simple JavaScript programs.
- To be able to use input and output statements.
- To understand basic memory concepts.
- To be able to use arithmetic operators.
- To understand the precedence of arithmetic operators.
- To be able to write decision-making statements.
- To be able to use relational and equality operators.

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

Equality, in a social sense, may be divided into that of condition and that of rights.
James Fenimore Cooper

Outline

7.1 Introduction

In the first six chapters, we introduced the Internet and World Wide Web, Internet Explorer 5.5, Adobe Photoshop Elements, XHTML and Cascading Style Sheets (CSS). In this chapter, we begin our introduction to the *JavaScript*[1] *scripting language,* which facilitates a disciplined approach to designing computer programs that enhance the functionality and appearance of Web pages.

In Chapters 7–12, we present a detailed discussion of JavaScript—the *de facto* client-side scripting language for Web-based applications. These chapters provide the programming foundation for both client-side scripting (Chapters 7–20) and server-side scripting (Chapters 25–31). Our treatment of JavaScript (Chapters 7–12) serves two purposes—it introduces client-side scripting, which makes Web pages more dynamic and interactive, and it provides the foundation for the more complex server-side scripting presented in Chapters 25–31.

We now introduce JavaScript programming and present examples that illustrate several important features of JavaScript. Each example is carefully analyzed one line at a time. In Chapters 8–9, we present a detailed treatment of *program development* and *program control* in JavaScript.

7.2 Simple Program: Printing a Line of Text in a Web Page

JavaScript uses notations that may appear strange to nonprogrammers. We begin by considering a simple *script* (or *program*) that displays the text "**`Welcome to JavaScript Programming!`**" in the body of an XHTML document. The Internet Explorer Web browser contains a *JavaScript interpreter*, which processes the commands written in JavaScript. The JavaScript code and its output are shown in Fig. 7.1.

1. Microsoft's version of JavaScript is called *JScript*. JavaScript was originally created by Netscape. Both Netscape and Microsoft have been instrumental in the standardization of JavaScript/JScript by the *ECMA* (*European Computer Manufacturer's Association*) as *ECMAScript*. For information on the current ECMAScript standard, visit **`www.ecma.ch/stand/ecma-262.htm`**. Throughout this book, we refer to JavaScript and JScript generically as JavaScript.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 7.1: welcome.html     -->
6  <!-- Displaying a line of text -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>A First Program in JavaScript</title>
11
12       <script type = "text/javascript">
13          <!--
14          document.writeln(
15             "<h1>Welcome to JavaScript Programming!</h1>" );
16          // -->
17       </script>
18
19    </head><body></body>
20 </html>
```

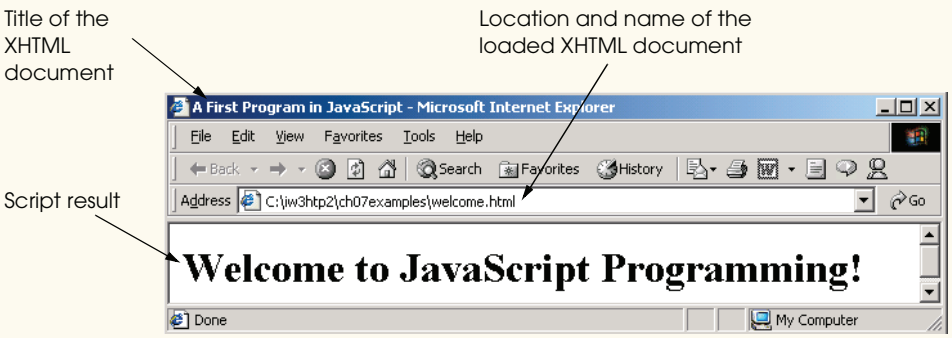


Fig. 7.1 First program in JavaScript.

This program illustrates several important JavaScript features. We consider each line of the XHTML document and script in detail. We have given each XHTML document line numbers for the reader's convenience; those line numbers are not part of the XHTML document or of the JavaScript programs. Lines 14–15 do the "real work" of the script, namely displaying the phrase **`Welcome to JavaScript Programming!`** in the Web page. However, let us consider each line in order.

Line 9 indicates the beginning of the **`<head>`** section of the XHTML 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 will execute before the **`<body>`** of the XHTML document displays. In later chapters on JavaScript and in the chapters on dynamic HTML, we illustrate *inline scripting*, in which JavaScript code is written in the **`<body>`** of an XHTML document.

Line 11 is simply a blank line to separate the **`<script>`** tag at line 12 from the other XHTML elements. This effect helps the script stand out in the XHTML document and makes the document easier to read.

Good Programming Practice 7.1

Place a blank line before **<script>** *and after* **</script>** *to separate the script from the surrounding XHTML elements and to make the script stand out in the document.*

Line 12 uses the **<script>** tag to indicate to the browser that the text which follows is part of a script. The **type** *attribute* specifies the type of file as well as the *scripting language* used in the script—in this case, a **text** file written in **javascript**. Both Microsoft Internet Explorer and Netscape Communicator use JavaScript as the default scripting language. [*Note*: Even though Microsoft calls the language JScript, the **type** attribute specifies **javascript**, to adhere to the ECMAScript standard.]

Lines 14–15 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*. A string is sometimes called a *character string*, a *message* or a *string literal*. We refer to characters between double quotation marks generically as strings. Individual whitespace characters between words in a string are not ignored by the browser. However, if consecutive spaces appear in a string, browsers condense those spaces to a single space. Also, in most cases, browsers ignore leading whitespace characters (i.e., whitespace at the beginning of a string).

Software Engineering Observation 7.1

*Strings in JavaScript can also be enclosed in single quotation marks (**'**).*

Lines 14–15 use the browser's **document** *object*, which represents the XHTML document the browser is currently displaying. The **document** object allows a script programmer to specify text to display in the XHTML document. The browser contains a complete set of objects that allow script programmers to access and manipulate every element of an XHTML document. In the next several chapters, we overview some of these objects. Chapters 13 through 18 provide in-depth coverage of many more objects that a script programmer can manipulate.

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 provide useful services to the *client of the object* (i.e., the script that calls the methods). In lines 14–15, we call the **document** object's **writeln** *method* to write a line of XHTML markup in the XHTML document. The parentheses following the method name **writeln** contain the *arguments* that the method requires to perform its task (or its action). Method **writeln** instructs the browser to display the argument string. If the string contains XHTML 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 XHTML head, because the phrase is enclosed in an **h1** element.

The code elements in lines 14–15, including **document.writeln**, its *argument* in the parentheses (the string) and the *semicolon* (**;**), together are called a *statement*. Every statement should end with a semicolon (also known as the *statement terminator*), although this practice is not required by JavaScript. Line 17 indicates the end of the script.

Good Programming Practice 7.2

Always include the semicolon at the end of a statement to terminate the statement. This notation clarifies where one statement ends and the next statement begins.

Common Programming Error 7.1

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

The **</head>** tag at line 19 indicates the end of the **<head>** section. Also on line 19, the tags **<body>** and **</body>** specify that this XHTML document has an empty body—no XHTML appears in the **body** element. Line 20 indicates the end of this XHTML document.

We are now ready to view our XHTML document in Internet Explorer. Open the XHTML document in Internet Explorer by double-clicking it. If the script contains no syntax errors, it should produce the output shown in Fig. 7.1.

Common Programming Error 7.2

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 the programmer 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 it attempts to execute the statement containing the error. The JavaScript interpreter in Internet Explorer reports all syntax errors by indicating in a separate popup window that a "*runtime error*" occurred (i.e., a problem occurred while the interpreter was running the script).*

Testing and Debugging Tip 7.1

When the interpreter reports a syntax error, the error may not be on 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.

Some older Web browsers do not support scripting. In such browsers, the actual text of a script often will display in the Web page. To prevent this from happening, many script programmers enclose the script code in an XHTML comment, so that browsers which do not support scripts ignore the script. The syntax used is as follows:

```
<script type = "text/javascript">
   <!--
   script code here
   // -->
</script>
```

When a browser that does not support scripts encounters the preceding code, it ignores the **<script>** and **</script>** tags and the script code in the XHTML comment. Browsers that do support scripting will interpret the JavaScript code as expected. [*Note*: Some browsers require the *JavaScript single-line comment* **//** (see Section 7.3 for an explanation) before the ending XHTML comment delimiter (**-->**) to interpret the script properly.]

Portability Tip 7.1

Some browsers do not support the **<script>**...**</script>** *tags. If your document is to be rendered with such browsers, the script code between these tags should be enclosed in an XHTML comment, so that the script text does not get displayed as part of the Web page.*

A script can display **Welcome to JavaScript Programming!** several ways. Figure 7.2 uses two JavaScript statements to produce one line of text in the XHTML document. This example also displays the text in a different color using the CSS **color** property.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 7.2: welcome2.html                    -->
6  <!-- Printing a Line with Multiple Statements -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Printing a Line with Multiple Statements</title>
11
12       <script type = "text/javascript">
13          <!--
14          document.write( "<h1 style = \"color: magenta\">" );
15          document.write( "Welcome to JavaScript " +
16             "Programming!</h1>" );
17          // -->
18       </script>
19
20    </head><body></body>
21 </html>
```

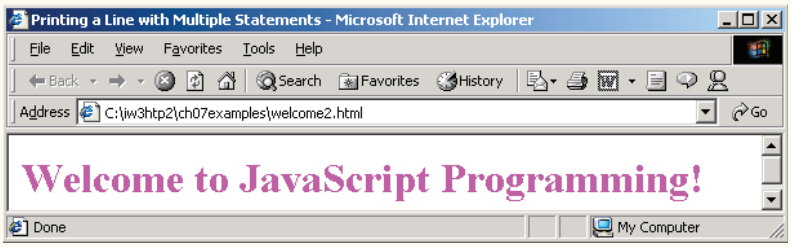


Fig. 7.2 Printing on one line with separate statements.

Most of this XHTML document is identical to Fig. 7.1, so we concentrate only on lines 14–16 of Fig. 7.2, which display one line of text in the XHTML document. The first statement uses **`document`** method ***`write`*** to display a string. Unlike **`writeln`**, **`write`** does not position the output cursor in the XHTML 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 XHTML document.] The next character written in the XHTML document appears immediately after the last character written with **`write`**. Thus, when line 16 executes, the first character written, "**`J`**," appears immediately after the last character displayed with **`write`** (the space character inside the right double quote on line 15). 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 next line. In effect, the two statements in lines 14–16 result in one line of XHTML text. Remember that statements in JavaScript are separated by semicolons (**`;`**). Therefore, lines 15–16 represent one statement. JavaScript allows large statements to be split over many lines. However, you cannot split a statement in the middle of a string.

Common Programming Error 7.3

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

Notice, however, that the two characters "****" and "**"**" are not displayed in the browser. 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 of characters, the next character is combined with the backslash to form an *escape sequence*. The escape sequence **\"** is the *double-quote character*, which causes a double-quote character to be inserted into the string. We use this escape sequence to insert double-quotes around the attribute value for **style**. We discuss escape sequences in greater detail momentarily.

It is important to note that the preceding discussion has nothing to do with the actual rendering of the XHTML 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 unless the browser encounters an XHTML element that explicitly starts a new line—e.g., **
** to start a new line, **<p>** to start a new paragraph, etc.

Common Programming Error 7.4

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

In the next example, we demonstrate that a single statement can cause the browser to display multiple lines through the use of line-break XHTML tags (**
) throughout the string of XHTML text in a **write or **writeln** method call. Figure 7.3 demonstrates the use of line-break XHTML tags. Lines 13–14 produce three separate lines of text when the browser renders the XHTML document.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 7.3: welcome3.html    -->
6  <!-- Printing Multiple Lines    -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head><title>Printing Multiple Lines</title>
10
11       <script type = "text/javascript">
12          <!--
13          document.writeln( "<h1>Welcome to<br />JavaScript" +
14             "<br />Programming!</h1>" );
15          // -->
16       </script>
17
18    </head><body></body>
19 </html>
```

Fig. 7.3 Printing on multiple lines with a single statement (part 1 of 2).

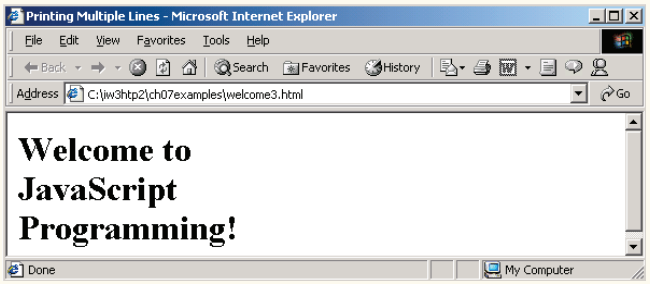


Fig. 7.3 Printing on multiple lines with a single statement (part 2 of 2).

The first several programs in this chapter display text in the XHTML document. Sometimes it is 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 program in Fig. 7.4 displays **`Welcome to JavaScript Programming!`** as three lines in a predefined dialog called an ***`alert`*** *dialog*.

Line 13 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 the first window of Fig. 7.4. The *title bar* of the dialog contains the string **Microsoft Internet Explorer**, 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., *hide*) *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.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 7.4: welcome4.html                    -->
6  <!-- Printing multiple lines in a dialog box -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head><title>Printing Multiple Lines in a Dialog Box</title>
10
11       <script type = "text/javascript">
12          <!--
13             window.alert( "Welcome to\nJavaScript\nProgramming!" );
14          // -->
15       </script>
16
17    </head>
18
19    <body>
20       <p>Click Refresh (or Reload) to run this script again.</p>
21    </body>
22 </html>
```

Fig. 7.4 Displaying multiple lines in a dialog (part 1 of 2).

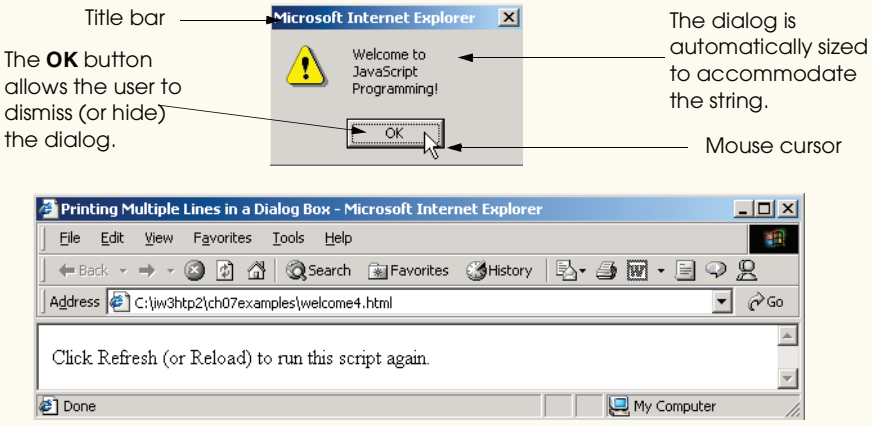


Fig. 7.4 Displaying multiple lines in a dialog (part 2 of 2).

Common Programming Error 7.5

Dialogs display plain text; they do not render XHTML. Therefore, specifying XHTML elements as part of a string to be displayed in a dialog results in the actual characters of the tags being displayed.

Note that the **`alert`** dialog contains three lines of plain text. Normally, a diaolg displays the characters in a string exactly as they appear between the double quotes. Notice, however, that the dialog does not display the two characters "**`\`**" and "**`n`**." The escape sequence **`\n`** is the *newline character*. In a dialog, the newline character causes the *cursor* (the current screen position indicator) to move to the beginning of the next line in the dialog. Some other common escape sequences are listed in Fig. 7.5. The **`\n`**, **`\t`** and **`\r`** escape sequences in the table do not affect XHTML rendering unless they are in a ***`pre`*** *element* (this element displays the text between its tags in a fixed-width font exactly as it is formatted between the tags, including leading whitespace characters and consecutive whitespace characters). The other escape sequences result in characters that will be displayed in plain text dialogs and in XHTML.

Escape sequence	Description
`\n`	Newline. Position the screen cursor at the beginning of the next line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor to the beginning of the current line; do not advance to the next line. Any characters output after the carriage return overwrite the characters previously output on that line.
`\\`	Backslash. Used to represent a backslash character in a string.

Fig. 7.5 Some common escape sequences (part 1 of 2).

Escape sequence	Description
`\"`	Double quote. Used to represent a double quote character in a string contained in double quotes. For example, `window.alert( "\"in quotes\"" );` displays `"in quotes"` in an `alert` dialog.
`\'`	Single quote. Used to represent a single quote character in a string. For example, `window.alert( '\'in quotes\'' );` displays `'in quotes'` in an `alert` dialog.

Fig. 7.5 Some common escape sequences (part 2 of 2).

7.3 Another JavaScript Program: Adding Integers

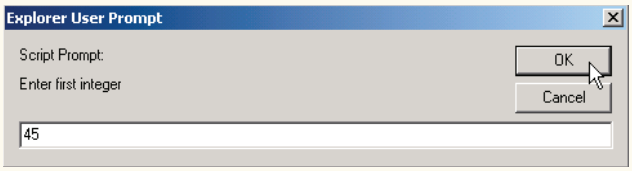
Our next script inputs two *integers* (whole numbers, such as 7, –11, 0 and 31,914) typed by a user at the keyboard, computes the sum of the values and displays the result.

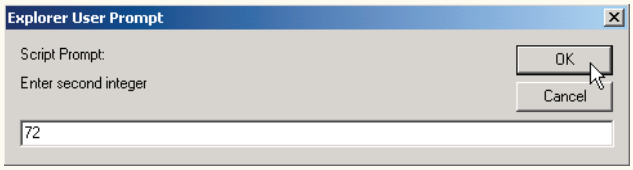
The script uses another predefined dialog box from the **`window`** object, one called a ***`prompt`*** *dialog*, that allows the user to input a value for use in the script. The program displays the results of the addition operation in the XHTML document. Figure 7.6 shows the script and some sample screen captures. [*Note*: In later JavaScript chapters, we will obtain input via GUI components in XHTML forms, as introduced in Chapter 5.]

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Fig. 7.6: Addition.html -->
6  <!-- Addition Program        -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>An Addition Program</title>
11
12       <script type = "text/javascript">
13          <!--
14          var firstNumber,    // first string entered by user
15              secondNumber,   // second string entered by user
16              number1,        // first number to add
17              number2,        // second number to add
18              sum;            // sum of number1 and number2
19
20          // read in first number from user as a string
21          firstNumber =
22             window.prompt( "Enter first integer", "0" );
```

Fig. 7.6 Addition script "in action" (part 1 of 2).

```
23
24          // read in second number from user as a string
25          secondNumber =
26             window.prompt( "Enter second integer", "0" );
27
28          // convert numbers from strings to integers
29          number1 = parseInt( firstNumber );
30          number2 = parseInt( secondNumber );
31
32          // add the numbers
33          sum = number1 + number2;
34
35          // display the results
36          document.writeln( "<h1>The sum is " + sum + "</h1>" );
37          // -->
38       </script>
39
40    </head>
41    <body>
42       <p>Click Refresh (or Reload) to run the script again</p>
43    </body>
44 </html>
```





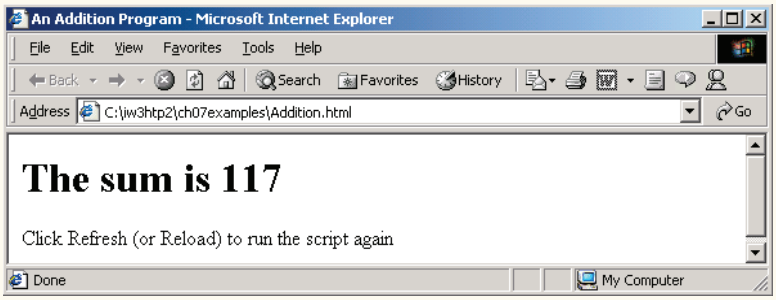


Fig. 7.6 Addition script "in action" (part 2 of 2).

Lines 14–18 are *declarations*. The keyword **`var`** at the beginning of the statement indicates that the words **`firstNumber`**, **`secondNumber`**, **`number1`**, **`number2`** and **`sum`** are the names of *variables*. A variable is a location in the computer's memory where a value

can be stored for use by a program. All variables should be declared with a name in a **var** statement before they are used in a program. Although using **var** to declare variables is not required, we will see in Chapter 10, "JavaScript/JScript: Functions," that **var** sometimes ensures proper behavior of a script.

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 (**$**) that does not begin with a digit and does not contain any spaces. Some valid identifiers are **Welcome**, **$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 a valid identifier, because it contains a space. Remember that JavaScript is *case sensitive*—uppercase and lowercase letters are considered to be different characters, so **firstNumber**, **FiRsTnUmBeR** and **FIRSTNUMBER** are different identifiers.

Good Programming Practice 7.3

Choosing meaningful variable names helps a script to be "self-documenting" (i.e., easy to understand by simply reading the script, rather than having to read manuals or excessive comments).

Good Programming Practice 7.4

By convention, variable-name identifiers begin with a lowercase first letter. Every word in the name after the first word should begin with a capital first letter. For example, identifier **firstNumber** *has a capital* **N** *in its second word,* **Number**.

Common Programming Error 7.6

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

Declarations, like statements, end with a semicolon (**;**) and can be split over several lines (as shown in Fig. 7.6) 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 declaration or in multiple declarations. We could have written five declarations, one for each variable, but the single declaration we used in the program is more concise.

Programmers often indicate the purpose of each variable in the program by placing a JavaScript comment at the end of each line in the declaration. In lines 14–18, *single-line comments* that begin with the characters **//** state the purpose of each variable in the script. This form of comment is called a single-line comment because the comment terminates at the end of the line. A **//** comment can begin at any position in a line of JavaScript code and continues until the end of that 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 7.5

Some programmers prefer to declare each variable on a separate line. This format allows for easy insertion of a descriptive comment next to each declaration.

Another comment notation facilitates the writing of *multiple-line comments*. For example,

```
/* This is a multiple-line
   comment. It can be
   split over many lines. */
```

comments can be spread over several lines. Such comments begin with delimiter `/*` and end with delimiter `*/`. All text between the delimiters of the comment is ignored by the compiler.

Common Programming Error 7.7

Forgetting one of the delimiters of a multiple-line comment is a syntax error.

Common Programming Error 7.8

Nesting multiple-line comments (i.e., placing a multiple-line comment between the delimiters of another multiple-line comment) is a syntax error.

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.

Line 20 is a single-line comment indicating the purpose of the statement in the next two lines. Lines 21–22 allow the user to enter a string representing the first of the two integers that will be added. The `window` object's `prompt` method displays the dialog in Fig. 7.7.

The first argument to `prompt` indicates to 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. The optional second argument is the default string to display in the text field; if the second argument is not supplied, the text field does not display a default value. The user types characters in the text field, then clicks the **OK** button to return the string to the program. [If you type, but nothing appears in the text field, position the mouse pointer in the text field and click the left mouse button to activate the text field.] Unfortunately, JavaScript does not provide a simple form of input that is analogous to writing a line of text with `document.write` and `document.writeln`. For this reason, we normally receive input from a user through a GUI component such as the `prompt` dialog, as in this program, or through an XHTML form GUI component, as we will see in later chapters.

Technically, the user can type anything in the text field of the `prompt` dialog. For this program, if the user either types a noninteger value or clicks the **Cancel** button, a runtime logic error will occur, and the sum of the two values will appear in the XHTML document as *NaN* (*not a number*). In Chapter 12, JavaScript: Objects, we discuss the `Number` object and its methods that can determine whether a value is not a number.

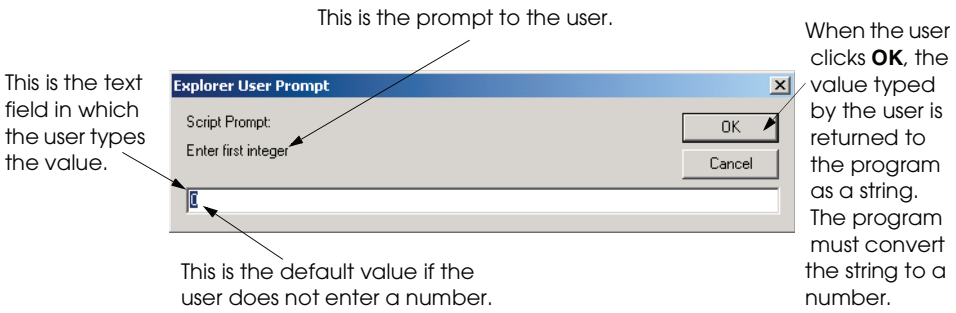


Fig. 7.7 Prompt dialog displayed by the `window` object's `prompt` method.

The statement at lines 21–22 gives the result of the call to the **window** object's **prompt** method (a string containing the characters typed by the user) to variable **firstNumber** by using the *assignment operator*, **=**. The statement is read as, **firstNumber** *gets* the value of **window.prompt("Enter first integer", "0")**. The **=** operator is called a *binary operator*, because it has two *operands*—**firstNumber** and the result of the expression **window.prompt("Enter first integer", "0")**. This entire statement is called an *assignment statement*, because it assigns a value to a variable. The expression to the right of the assignment operator always is evaluated first.

Lines 24 is a single-line comment that indicates the purpose of the statement in lines 25 and 26. The statement displays a **prompt** dialog in which the user types a string representing the second of the two integers to add.

Lines 29–30 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 29 assigns the integer that function **parseInt** returns to the variable **number1**. Any subsequent references to **number1** in the program use this same integer value. Line 30 assigns the integer that function **parseInt** returns to variable **number2**. Any subsequent references to **number2** in the program use this same integer value. [*Note*: 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 operator (**.**). The term *method* implies 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.]

The assignment statement on line 33 calculates the sum of the variables **number1** and **number2** and assigns the result to variable **sum** by using the assignment operator, **=**. The statement is read as "**sum** *gets* the value of **number1 + number2**." Most calculations occur in assignment statements.

Good Programming Practice 7.6

Place spaces on either side of a binary operator. This format makes the operator stand out and makes the program more readable.

After line 33 performs the calculation, line 36 uses **document.writeln** to display the result of the addition. The expression from the preceding statement uses the operator **+** to "add" a string (the literal **"<h1>The sum is "**) and **sum** (the variable containing the integer result of the addition on line 33). 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 concatenated. The result of this operation is a new (and normally longer) string. If we assume that **sum** contains the value **117**, the expression evaluates as follows: JavaScript determines that the two operands of the **+** operator (the string **"<h1>The sum is "** and the integer **sum**) are different types and that one of them is a string. Next, the statement converts the value of variable **sum** to a string and concatenates it with **"<h1>The sum is "**, which results in the string **"<h1>The sum is 117"**. Then, the statement concatenates the string **"</h1>"** to produce the string **"<h1>The sum is 117</h1>"**. The browser renders this string as part of the XHTML document. Note that the automatic conversion of integer **sum** occurs because it is concatenated with the string literal **"<h1>The sum is "**. Also note that the space between **is** and **117** is part of the string **"<h1>The sum is "**.

Common Programming Error 7.9

Confusing the `+` *operator used for string concatenation with the* `+` *operator used for addition can lead to strange results. For example, assuming that integer variable* `y` *has the value* `5`*, the expression* `"y + 2 = " + y + 2` *results in the string* `"y + 2 = 52"`*, not* `"y + 2 = 7"`*, because first the value of* `y` *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 desired result.*

After the browser interprets the `<head>` section of the XHTML document (which contains the JavaScript), it then interprets the `<body>` of the XHTML document (lines 41–43) and renders the XHTML. If you click your browser's **Refresh** (or **Reload**) button, the browser will reload the XHTML document, so that you can execute the script again and add two new integers. [*Note*: In some cases, it may be necessary to hold down the *Shift* key while clicking your browser's **Refresh** (or **Reload**) button, to ensure that the XHTML document reloads properly.]

7.4 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 program in Fig. 7.6, when line 22 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 program converts `firstNumber` to an integer, and the computer places the integer value `45` into location `number1`, as shown in Fig. 7.8.

Whenever a value is placed in a memory location, the value replaces the previous value in that location. The previous value is lost.

When line 26 executes, suppose the user enters the string `72` as the value for `secondNumber`. The program converts `secondNumber` to an integer, the computer places that integer value, `72`, into location `number2` and the memory appears as shown in Fig. 7.9.

Fig. 7.8 Memory location showing the name and value of variable `number1`.

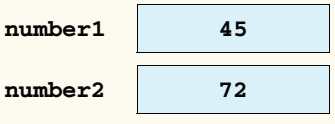


Fig. 7.9 Memory locations after values for variables `number1` and `number2` have been input.

Once the program 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. 7.10. 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, as the computer performed the calculation. When a value is read from a memory location, the process is *nondestructive*.

7.5 Arithmetic

Many scripts perform arithmetic calculations. Figure 7.11 summarizes the *arithmetic operators*. Note the use of various special symbols not used in algebra. The *asterisk* (*****) indicates multiplication; the *percent sign* (**%**) is the *modulus operator*, which is discussed shortly. The arithmetic operators in Fig. 7.11 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**.

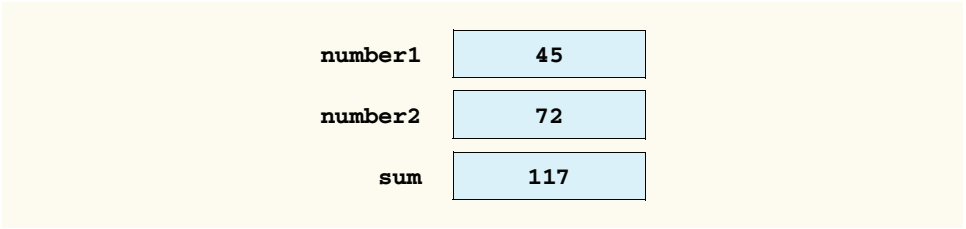


Fig. 7.10 Memory locations after calculating the **sum** of **number1** and **number2**.

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

Fig. 7.11 Arithmetic operators.

JavaScript provides the modulus operator, **%**, which yields the remainder after division. The expression **x % y** yields the remainder after **x** is divided by **y**. Thus, **7.4 % 3.1** yields **1.2**, and **17 % 5** yields **2**. In later chapters, we consider many interesting applications of the modulus operator, such as determining whether one number is a multiple of another. There is no arithmetic operator for exponentiation in JavaScript. (Chapter 9 shows how to perform exponentiation in JavaScript.)

Arithmetic expressions in JavaScript must be written in *straight-line form* to facilitate entering programs 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 in JavaScript expressions in the same manner as in algebraic expressions. For example, to multiply **a** times the quantity **b + c** we write:

```
a * ( b + c )
```

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. Operators in expressions contained between a left parenthesis and its corresponding right parenthesis are evaluated first. Thus, *parentheses may be used to force the order of evaluation to occur in any sequence desired by the programmer.* Parentheses are said to be at the highest level of precedence." In cases of *nested*, or *embedded*, parentheses, the operators in the innermost pair of parentheses are applied first.

2. Multiplication, division and modulus operations are applied next. If an expression contains several multiplication, division and modulus operations, operators are applied from left to right. Multiplication, division and modulus operations are said to have the same level of precedence.

3. Addition and subtraction operations are applied last. 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 are referring to the *associativity* of the operators—the order in which operators of equal priority are evaluated. We will see that some operators associate from right to left. Figure 7.12 summarizes these rules of operator precedence. The table in Fig. 7.12 will be expanded as additional JavaScript operators are introduced. A complete precedence chart is included in Appendix B.

Now, in light of the rules of operator precedence, let us consider several algebraic expressions. Each example lists an algebraic expression and the equivalent JavaScript expression.

Operator(s)	Operation(s)	Order of evaluation (precedence)
`( )`	Parentheses	Evaluated first. If the parentheses are nested, the expression in the innermost pair is evaluated first. If there are several pairs of parentheses on the same level (i.e., not nested), they are evaluated from left to right.
`*`, `/` or `%`	Multiplication Division Modulus	Evaluated second. If there are several such operations, they are evaluated from left to right.
`+` or `-`	Addition Subtraction	Evaluated last. If there are several such operations, they are evaluated from left to right.

Fig. 7.12 Precedence of arithmetic operators.

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

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

JavaScript:
```
m = ( a + b + c + d + e ) / 5;
```

The parentheses are required, because division has higher precedence than that of 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

$$a + b + c + d + \frac{e}{5}$$

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

Algebra: $y = mx + b$

JavaScript:
```
y = m * x + b;
```

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

The following example contains modulus (`%`), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

JavaScript:
```
z = p * r % q + w / x - y;
```

6 1 2 4 3 5

The circled numbers under the statement indicate the order in which JavaScript applies the operators. The multiplication, modulus and division operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than that of addition and subtraction. The addition and subtraction operations are evaluated next. These operations are also applied from left to right.

Not all expressions with several pairs of parentheses contain nested parentheses. For example, the expression

```
a * ( b + c ) + c * ( d + e )
```

does not contain nested parentheses. Rather, these parentheses are on the same level.

To develop a better understanding of the rules of operator precedence, consider the evaluation of a second-degree polynomial ($y = ax^2 + bx + c$):

```
y  =  a  *  x  *  x  +  b  *  x  +  c;
      6     1     2     4     3     5
```

The circled numbers under the preceding statement indicate the order in which JavaScript applies the operators. There is no arithmetic operator for exponentiation in JavaScript; x^2 is represented as **x * x**.

Suppose that **a**, **b**, **c** and **x** are initialized as follows: **a = 2**, **b = 3**, **c = 7** and **x = 5**. Figure 7.13 illustrates the order in which the operators are applied in the preceding second-degree polynomial.

As in algebra, it is acceptable to place unnecessary parentheses in an expression to make the expression clearer. Such unnecessary parentheses are also called *redundant parentheses*. For example, the preceding assignment statement might be parenthesized as follows:

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

Good Programming Practice 7.7

Using parentheses for complex arithmetic expressions, even when the parentheses are not necessary, can make the arithmetic expressions easier to read.

7.6 Decision Making: Equality and Relational Operators

This section introduces a version of JavaScript's **if** structure that allows a program 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** structure is executed. If the condition is not met (i.e., the condition is *false),* the statement in the body of the **if** structure is not executed. We will see an example shortly.

Conditions in **if** structures can be formed by using the *equality operators* and *relational operators* summarized in Fig. 7.14. 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.

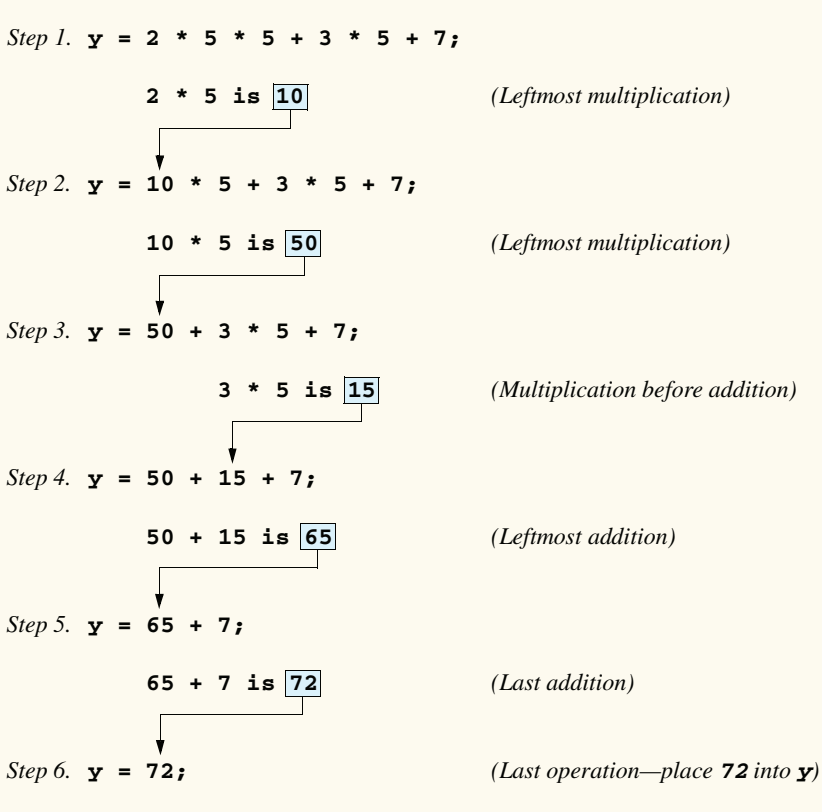


Fig. 7.13 Order in which a second-degree polynomial is evaluated.

Standard algebraic equality operator or relational operator	JavaScript equality or relational operator	Sample JavaScript condition	Meaning of JavaScript condition
Equality operators			
=	`==`	`x == y`	`x` is equal to `y`
≠	`!=`	`x != y`	`x` is not equal to `y`
Relational operators			
>	`>`	`x > y`	`x` is greater than `y`
<	`<`	`x < y`	`x` is less than `y`
≥	`>=`	`x >= y`	`x` is greater than or equal to `y`
≤	`<=`	`x <= y`	`x` is less than or equal to `y`

Fig. 7.14 Equality and relational operators.

Common Programming Error 7.10

It is a syntax error if the operators `==`, `!=`, `>=` *and* `<=` *contain spaces between their symbols, as in* `= =`, `! =`, `> =` *and* `< =`, *respectively.*

Common Programming Error 7.11

Reversing the operators `!=`, `>=` *and* `<=`, *as in* `=!`, `=>` *and* `=<`, *respectively, is a syntax error.*

Common Programming Error 7.12

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."*

The script in Fig. 7.15 uses six **`if`** statements to compare two values input into **`prompt`** dialogs by the user. If the condition in any of the **`if`** statements is satisfied, the assignment statement associated with that **`if`** statement is executed. The user inputs two values through input dialogs. The program stores the values in the variables **`first`** and **`second`**, then converts the values to integers and stores them in variables **`number1`** and **`number2`**. Finally, the program compares the values and displays the results of the comparison in an information dialog. The script and sample outputs are shown in Fig. 7.15.

Lines 15–18 declare the variables used in the script. Remember that variables may be declared in one declaration or in multiple declarations. If more than one name is declared in a declaration (as in this example), the names are separated by commas (**`,`**). This list of names is referred to as a comma-separated list. Once again, notice the comment at the end of each line, indicating the purpose of each variable in the program. Line 21 uses **`window.prompt`** to allow the user to input the first value and to store the value in **`first`**.

Line 24 uses **`window.prompt`** to allow the user to input the second value and to store the value in **`second`**. Lines 27–28 conver the strings to integers and stores them in variables **`number1`** and **`number2`**. Line 30 outputs a line of XHTML text containing the **`<h1>`** head **`Comparison Results`**. Lines 31–32 output a line of XHTML text that indicates the start of a **`<table>`** that has a one-pixel border and is 100% of the browser window's width.

The **`if`** structure (lines 34–36) compares the values of variables **`first`** and **`second`** to test them for equality. If the values are equal, the statement on lines 35–36 outputs a line of XHTML text representing one row of an XHTML table (as indicated by the **`<tr>`** and **`</tr>`** tags). The text in the row contains the result of **`first + " == " + second`**. As in Fig. 7.6, the **`+`** operator is used in this expression to perform string concatenation. If the conditions are true in one or more of the **`if`** structures starting at lines 38, 42, 46, 50 and 54, the corresponding **`document.writeln`** statement(s) output(s) a line of XHTML text representing a row in the XHTML table.

Notice the indentation in the **`if`** statements throughout the program. Such indentation enhances program readability.

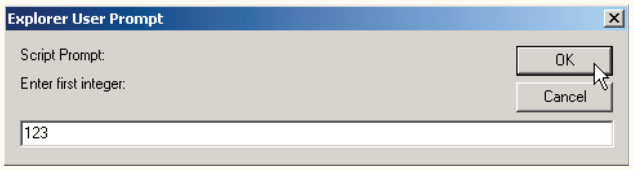
Good Programming Practice 7.8

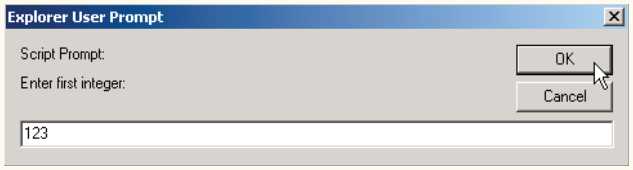
Indent the statement in the body of an **`if`** *structure to make the body of the structure stand out and to enhance program readability.*

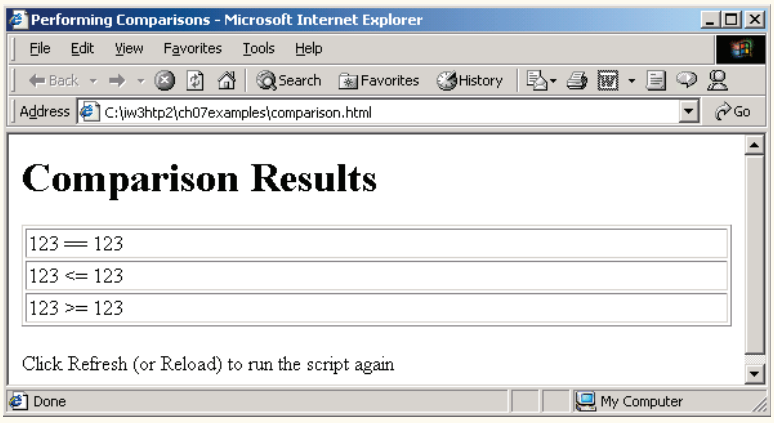
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!-- Fig. 7.14: comparison.html                  -->
<!-- Using if statements, relational operators  -->
<!-- and equality operators                      -->

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Performing Comparisons</title>

      <script type = "text/javascript">
         <!--
         var first,   // first string entered by user
             second,  // second string entered by user
             number1,       // first number to compare
             number2;       // second number to compare

         // read first number from user as a string
         first = window.prompt( "Enter first integer:", "0" );

         // read second number from user as a string
         second = window.prompt( "Enter second integer:", "0" );

         // convert numbers from strings to integers
         number1 = parseInt( first );
         number2 = parseInt( second );

         document.writeln( "<h1>Comparison Results</h1>" );
         document.writeln(
            "<table border = \"1\" width = \"100%\">" );

         if ( number1 == number2 )
            document.writeln( "<tr><td>" + number1 + " == " +
               number2 + "</td></tr>" );

         if ( number1 != number2 )
            document.writeln( "<tr><td>" + number1 + " != " +
               number2 + "</td></TR>" );

         if ( number1 < number2 )
            document.writeln( "<tr><td>" + number1 + " < "  +
               number2 + "</td></tr>" );

         if ( number1 > number2 )
            document.writeln( "<tr><td>" + number1 + " > " +
               number2 + "</td></tr>" );

         if ( number1 <= number2 )
            document.writeln( "<tr><td>" + number1 + " <= " +
               number2 + "</td></tr>" );

```

Fig. 7.15 Using equality and relational operators (part 1 of 3).

```
54            if ( number1 >= number2 )
55               document.writeln( "<tr><td>" + number1 + " >= " +
56                  number2 + "</td></tr>" );
57
58            // Display results
59            document.writeln( "</table>" );
60            // -->
61         </script>
62
63      </head>
64      <body>
65         <p>Click Refresh (or Reload) to run the script again</p>
66      </body>
67   </html>
```







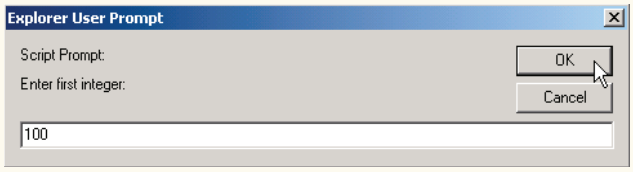


Fig. 7.15 Using equality and relational operators (part 2 of 3).

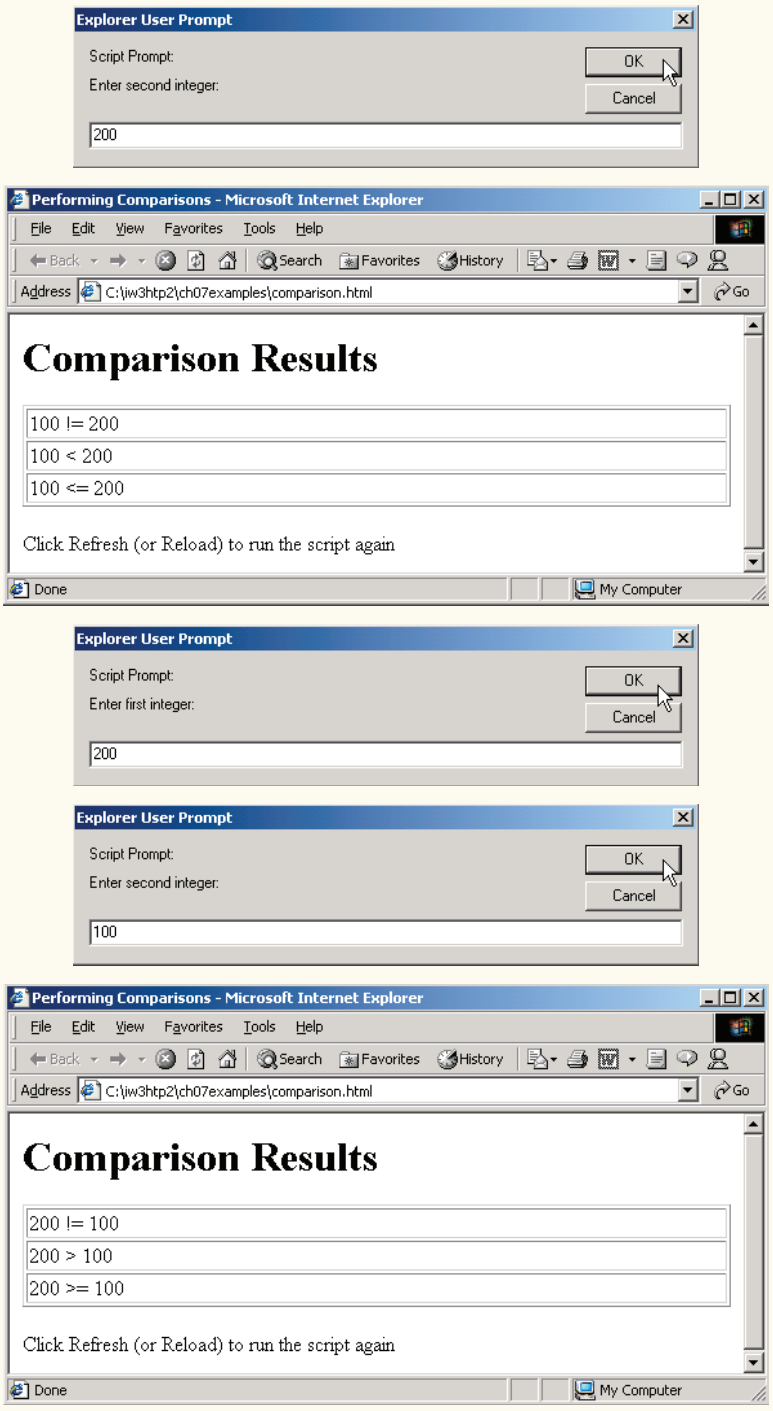


Fig. 7.15 Using equality and relational operators (part 3 of 3).

Good Programming Practice 7.9

Place only one statement per line in a program. This format enhances program readability.

Common Programming Error 7.13

Forgetting the left and right parentheses for the condition in an **`if`** *structure is a syntax error. The parentheses are required.*

Notice that there is no semicolon (**`;`**) at the end of the first line of each **`if`** structure. Such a semicolon would result in a logic error at execution time. For example,

```
if ( number1 == number2 ) ;
   document.writeln( "<tr><td>" + number1 + " == " +
      number2 + "</td></tr>" );
```

would actually be interpreted by JavaScript as

```
if ( number1 == number2 )
   ;

document.writeln( "<tr><td>" + number1 + " == " +
   number2 + "</td></tr>" );
```

where the semicolon on the line by itself—called the *empty statement*—is the statement to execute if the condition in the **`if`** structure is true. When the empty statement executes, no task is performed in the program. The program then continues with the assignment statement, which executes regardless of whether the condition is true or false.

Common Programming Error 7.14

Placing a semicolon immediately after the right parenthesis of the condition in an **`if`** *structure is normally a logic error. The semicolon would cause the body of the* **`if`** *structure to be empty, so the* **`if`** *structure itself would perform no action, regardless of whether its condition is true. Worse yet, the intended body statement of the* **`if`** *structure would now become a statement in sequence after the* **`if`** *structure and would always be executed.*

Notice the use of spacing in Fig. 7.15. Remember that whitespace characters, such as tabs, newlines and spaces, are normally ignored by the compiler. So, statements may be split over several lines and may be spaced according to the programmer's preferences without affecting the meaning of a program. However, it is incorrect to split identifiers and string literals. Ideally, statements should be kept small, but it is not always possible to do so.

Good Programming Practice 7.10

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.

The chart in Fig. 7.16 shows the precedence of the operators introduced in this chapter. The operators are shown from top to bottom in decreasing order of precedence. Notice 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, as we will soon see, first assigns the value `0` to variable `y` and then assigns the result of that assignment, `0`, to `x`.

Good Programming Practice 7.11

Refer to the operator precedence chart when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order in which you expect them to be performed. If you are uncertain about the order of evaluation in a complex expression, 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.

We have introduced many important features of JavaScript, including how to display data, how to input data from the keyboard, how to perform calculations and how to make decisions. In Chapter 8, we build on the techniques of Chapter 7 as we introduce *structured programming*. You will become more familiar with indentation techniques. We will study how to specify and vary the order in which statements are executed; this order is called the *flow of control*.

7.7 JavaScript Internet and World Wide Web Resources

There are a tremendous number of resources for JavaScript programmers on the Internet and World Wide Web. This section lists a variety of JScript, JavaScript and ECMAScript resources available on the Internet and provides a brief description of each. Additional resources for these topics are presented in the subsequent chapters on JavaScript and in other chapters as necessary.

`www.ecma.ch/ecma1/stand/ecma-262.htm`
JScript is Microsoft's version of *JavaScript*—a scripting language that is standardized by the *ECMA* (*European Computer Manufacturer's Association*) as *ECMAScript*. This site is the home of the standard document for ECMAScript.

`msdn.microsoft.com/scripting/default.htm`
The *Microsoft Windows Script Technologies* page includes an overview of JScript, complete with tutorials, FAQs, demos, tools for downloading and newsgroups.

`www.webteacher.com/javascript`
Webteacher.com is an excellent source for tutorials that focus on teaching with detailed explanations and examples. This site is particularly useful for nonprogrammers.

Operators	Associativity	Type
`()`	left to right	parentheses
`*  /  %`	left to right	multiplicative
`+  -`	left to right	additive
`<  <=  >  >=`	left to right	relational
`==  !=`	left to right	equality
`=`	right to left	assignment

Fig. 7.16 Precedence and associativity of the operators discussed so far.

wsabstract.com
Website Abstraction is devoted to JavaScript and provides specialized tutorials and many free scripts. This site is good for beginners, as well as people with prior experience who are looking for help in a specific area of JavaScript.

www.webdeveloper.com/javascript
WebDeveloper.com provides tutorials, tools, and links to many free scripts.

SUMMARY

- The JavaScript language facilitates a disciplined approach to the design of computer programs that enhance Web pages.
- JScript is Microsoft's version of JavaScript—a scripting language that is standardized by the ECMA (European Computer Manufacturer's Association) as ECMAScript.
- The spacing displayed by a browser in a Web page is determined by the XHTML elements used to format the page.
- Often, JavaScripts appear in the **<head>** section of the XHTML 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. Attribute **type** specifies the scripting language used in the script—such as **JavaScript**.
- A string of characters can be contained between double (**"**) or single (**'**) quotation marks.
- A string is sometimes called a character string, a message or a string literal.
- The browser's **document** object represents the XHTML document currently being displayed in the browser. The **document** object allows a script programmer to specify XHTML text to be displayed in the XHTML document.
- The browser contains a complete set of objects that allow script programmers to access and manipulate every element of an XHTML document.
- 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 provide useful services to the client of the object—the script that calls the methods.
- The **document** object's **writeln** method writes a line of XHTML text in the XHTML document.
- The parentheses following the name of a method contain the arguments that the method requires to perform its task (or its action).
- Using **writeln** to write a line of XHTML text into a **document** does not guarantee that a corresponding line of text will appear in the XHTML document. The text displayed is dependent on the contents of the string written, which is subsequently rendered by the browser. The browser will interpret the XHTML elements as it normally does to render the final text in the document.
- Every statement should end with a semicolon (also known as the statement terminator), although none is required by JavaScript.
- JavaScript is case sensitive. Not using the proper uppercase and lowercase letters is a syntax error.
- Sometimes it is useful to display information in windows called dialogs that "pop up" on the screen to grab the user's attention. Dialogs are typically used to display important messages to the user browsing the Web page. The browser's **window** object uses method **alert** to display an alert dialog. Method **alert** requires as its argument the string to be displayed.
- When a backslash is encountered in a string of characters, the next character is combined with the backslash to form an escape sequence. The escape sequence **\n** is the newline character. It causes the cursor in the XHTML document to move to the beginning of the next line in the dialog.

- The keyword **var** is used to declare the names of variables. A variable is a location in the computer's memory where a value can be stored for use by a program. Though you are not required to do so, you should declare all variables with a name in a **var** statement before they are used in a program.
- A variable name can be any valid identifier consisting of letters, digits, underscores (_) and dollar signs (**$**) that does not begin with a digit and does not contain any spaces.
- Declarations end with a semicolon (**;**) and can be split over several lines, with each variable in the declaration separated by a comma (forming a comma-separated list of variable names). Several variables may be declared in one declaration or in multiple declarations.
- Programmers often indicate the purpose of each variable in the program by placing a JavaScript comment at the end of each line in the declaration. A single-line comment 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.
- Multiple-line comments begin with delimiter **/*** and end with delimiter ***/**. All text between the delimiters of the comment is ignored by the compiler.
- 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. The optional second argument is the default string to display in the text field.
- A variable is assigned a value with an assignment statement, using the assignment operator, **=**. The **=** operator is called a binary operator, because it has two operands.
- Function **parseInt** converts its string argument to an integer.
- 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 concatenated.
- Variable names correspond to locations in the computer's memory. Every variable has a name, a type, a size 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.
- The arithmetic operators are binary operators, because they each operate on two operands.
- Operators in arithmetic expressions are applied in a precise sequence determined by the rules of operator precedence.
- Parentheses may be used to force the order of evaluation of operators to occur in any sequence desired by the programmer.
- When we say that operators are applied from left to right, we are referring to the associativity of the operators. Some operators associate from right to left.
- Java's **if** structure allows a program 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** structure is executed. If the condition is not met (i.e., the condition is false), the statement in the body of the **if** structure is not executed.
- Conditions in **if** structures can be formed by using the equality operators and relational operators.

TERMINOLOGY

\" double-quote escape sequence
\n newline escape sequence
<head> section of the XHTML document
<script></script>
addition operator (**+**)
alert dialog
alert method of the **window** object
argument to a method
arithmetic expressions in straight-line form
arithmetic operator

assignment operator (`=`)
assignment statement
attribute
automatic conversion
backslash (`\`) escape character
behavior
binary operator
blank line
case sensitive
character string
client of an object
comma-separated list
comment
condition
data
decision making
declaration
dialog
division operator (`/`)
`document` object
double quotation (`"`) marks
ECMA
ECMAScript
empty statement
equality operators
error message
escape sequence
European Computer Manufacturer's Association (ECMA)
false
identifier
`if` structure
inline scripting
integer
interpreter
JavaScript
JavaScript interpreter
JScript
location in the computer's memory
logic error
meaningful variable names
method
modulus operator (`%`)
multiple-line comment (`/*` and `*/`)
multiplication operator (`*`)
name of a variable
object
operand
operator associativity
operator precedence
parentheses
`parseInt` function
perform an action
program
prompt
`prompt` dialog
`prompt` method of the `window` object
redundant parentheses
relational operators
remainder after division
rules of operator precedence
runtime error
script
scripting language
self-documenting
semicolon (`;`) statement terminator
single quotation (`'`) marks
single-line comment (`//`)
statement
string concatenation
string concatenation operator (`+`)
string literal
string of characters
subtraction operator (`-`)
syntax error
text field
true
`type` attribute of the `<script>` tag
type of a variable
value of a variable
`var` keyword
variable
violation of the language rules
whitespace characters
whole number
`window` object
`write` method of the `document` object
`writeln` method of the `document` object

SELF-REVIEW EXERCISES

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

a) _____ begins a single-line comment.

b) Every statement should end with a _____.

c) The _____ structure is used to make decisions.
d) _____, _____, _____ and _____ are known as whitespace.
e) The _____ object displays alert dialogs and prompt dialogs.
f) _____ are reserved for use by JavaScript.
g) Methods _____ and _____ of the _____ object write XHTML text into an XHTML document.

7.2 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 program is executed.
b) JavaScript considers the variables **number** and **NuMbEr** to be identical.
c) The modulus operator (**%**) can be used only with any numeric operands.
d) The arithmetic operators *****, **/**, **%**, **+** and **-** all have the same level of precedence.
e) Method **parseInt** converts an integer to a string.

7.3 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 text field.
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 XHTML text that will display the message **"This is a JavaScript program"** on one line in the XHTML document.
f) Output a line of XHTML text that will display the message **"This is a JavaScript program"** on two lines in the XHTML document. Use only one statement.

7.4 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" );
```

7.5 Write a statement (or comment) to accomplish each of the following tasks:
a) State that a program will calculate the product of three integers.
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 **xVal** to an integer, and store the result in the variable **x**.
h) Convert **yVal** to an integer, and store the result in the variable **y**.
i) Convert **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 XHTML text containing the string **"The product is "** followed by the value of the variable **result**.

7.6 Using the statements you wrote in Exercise 7.5, write a complete program that calculates and prints the product of three integers.

ANSWERS TO SELF-REVIEW EXERCISES

7.1 a) `//`. b) Semicolon (`;`). c) `if`. d) Blank lines, space characters, newline characters and tab characters. e) `window`. f) Keywords. g) `write`, `writeln`, `document`.

7.2 a) False. Comments do not cause any action to be performed when the program is executed. They are used to document programs 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.

7.3 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 a JavaScript program" );
```
f)
```
document.writeln( "This is a<br />JavaScript program" );
```

7.4 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 `>=`.

7.5 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>" );
```

7.6 The program is as follows:

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
4
5  <!-- Exercise 7.6: product.html -->
6
7  <html xmlns = "http://www.w3.org/1999/xhtml">
8     <head>
9        <title>Product of Three Integers</title>
10
11       <script type = "text/javascript">
12          <!--
13          // Calculate the product of three integers
```

```
14            var x, y, z, result;
15            var xVal, yVal, zVal;
16
17            xVal = window.prompt( "Enter first integer:", "0" );
18            yVal = window.prompt( "Enter second integer:", "0" );
19            zVal = window.prompt( "Enter third integer:", "0" );
20
21            x = parseInt( xVal );
22            y = parseInt( yVal );
23            z = parseInt( zVal );
24
25            result = x * y * z;
26            document.writeln( "<h1>The product is " +
27               result + "<h1>" );
28            // -->
29         </script>
30
31      </head><body></body>
32   </html>
```

EXERCISES

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

a) _____ are used to document a program 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 _____.

d) Calculations are normally performed by _____ statements.

e) A dialog capable of showing a message to the user is displayed with method _____ of object _____.

7.8 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 program performs a sample payroll calculation [*Hint*: Use text that helps to document a program].

7.9 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`**.

7.10 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 set of parentheses is evaluated first in an arithmetic expression? _____.

c) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a _____.

7.11 What displays in the message 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 ) );`

7.12 Which of the following JavaScript statements contain variables whose values are destroyed (i.e., changed or replaced)?

a) `p = i + j + k + 7;`
b) `window.alert( "variables whose values are destroyed" );`
c) `window.alert( "a = 5" );`
d) `stringVal = window.prompt( "Enter string:" );`

7.13 Given $y = ax^3 + 7$, which of the following are correct 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);`

7.14 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 ) ) ) );`

7.15 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 program using the following methods:

a) Using one `document.writeln` statement.
b) Using four `document.write` statements.

7.16 Write a script that asks the user to enter two numbers, obtains the two numbers from the user and outputs XHTML text that displays the sum, product, difference and quotient of the two numbers. Use the techniques shown in Fig. 7.6.

7.17 Write a script that asks the user to enter two integers, obtains the numbers from the user and outputs XHTML text that displays the larger number followed by the words "`is larger`" in an information message dialog. If the numbers are equal, output XHTML text that displays the message "`These numbers are equal`." Use the techniques shown in Fig. 7.15.

7.18 Write a script that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers in an `alert` dialog.

7.19 Write a script that inputs from the user the radius of a circle and outputs XHTML text that displays the circle's diameter, circumference and area. Use the constant value 3.14159 for π. Use the GUI techniques shown in Fig. 7.6. [*Note*: You may also use the predefined constant `Math.PI` for the value of π. 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* = πr^2.

7.20 Write a script that outputs XHTML text that displays in the XHTML document an oval, an arrow and a diamond using asterisks (`*`), as follows [*Note*: Use the `<pre>` and `</pre>` tags to specify that the asterisks should be displayed using a fixed-width font]:

```
*********          ***              *              *
*       *        *     *           ***            * *
*       *       *       *         *****          *   *
*       *       *       *           *           *     *
*       *       *       *           *          *       *
*       *       *       *           *           *     *
*       *       *       *           *            *   *
*       *        *     *            *             * *
*********          ***              *              *
```

7.21 Modify the program you created in Exercise 7.20 to display the shapes without using the **<pre>** and **</pre>** tags. Does the program display the shapes exactly as in Exercise 7.20?

7.22 What does the following code print?

```
document.writeln( "*\n**\n***\n****\n*****" );
```

7.23 What does the following code print?

```
document.writeln( "*" );
document.writeln( "****" );
document.writeln( "******" );
document.writeln( "*****" );
document.writeln( "***" );
```

7.24 What does the following code print?

```
document.write( "*<br />" );
document.write( "****<br />" );
document.write( "*****<br />" );
document.write( "****<br />" );
document.writeln( "***" );
```

7.25 What does the following code print?

```
document.write( "*<br />" );
document.writeln( "****" );
document.writeln( "******" );
document.write( "****<br />" );
document.writeln( "***" );
```

7.26 Write a script that reads five integers and determines and outputs XHTML text that displays the largest integer and the smallest integer in the group. Use only the programming techniques you learned in this chapter.

7.27 Write a script that reads an integer and determines and outputs XHTML text that displays whether it is odd or even. [*Hint*: Use the modulus operator. An even number is a multiple of 2. Any multiple of 2 leaves a remainder of zero when divided by 2.]

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

7.29 Write a script that outputs XHTML text that displays in the XHTML document a checkerboard pattern, as follows:

```
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
* * * * * * * * *
 * * * * * * * * *
```

7.30 Write a script that inputs five numbers and determines and outputs XHTML text that displays the number of negative numbers input, the number of positive numbers input and the number of zeros input.

7.31 Using only the programming techniques you learned in this chapter, write a script that calculates the squares and cubes of the numbers from 0 to 10 and outputs XHTML text that displays the resulting values in an XHTML table format, as follows:

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

[*Note*: This program does not require any input from the user.]