7. JavaScript: Control Statements I

Let's all move one place on.

—Lewis Carroll

The wheel is come full circle.

—William Shakespeare

How many apples fell on Newton's head before he took the hint!

—Robert Frost

Objectives

In this chapter you will:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if`...`else` selection statements to choose among alternative actions.
- Use the `while` repetition statement to execute statements in a script repeatedly.
- Implement counter-controlled repetition and sentinel-controlled repetition.
- Use the increment, decrement and assignment operators.

Outline

7.1. Introduction

Before writing a script to solve a problem, we must have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a script, it's equally essential to understand the types of building blocks that are available and to employ proven program-construction principles. In this chapter and Chapter 8, we discuss these issues as we present the theory and principles of *structured programming*.

7.2. Algorithms

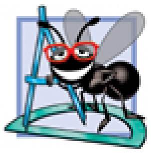
Any computable problem can be solved by executing a series of actions in a specific order. A **procedure** for solving a problem in terms of

1. the **actions** to be executed, and
2. the **order** in which the actions are to be executed

is called an **algorithm**. Correctly specifying the order in which the actions are to execute is important—this is called **program control**. In this chapter and Chapter 8, we investigate the program-control capabilities of JavaScript.

7.3. Pseudocode

Pseudocode is an informal language that helps you develop algorithms. The pseudocode we present here is useful for developing algorithms that will be converted to structured portions of JavaScript programs. Pseudocode is similar to everyday English; it's convenient and user friendly, although it's not an actual computer programming language.



SOFTWARE ENGINEERING OBSERVATION 7.1

Pseudocode is often used to "think out" a script during the script-design process. Carefully prepared pseudocode can easily be converted to JavaScript.

7.4. Control Statements

Normally, statements in a script execute one after the other in the order in which they're written. This process is called **sequential execution**. Various JavaScript statements we'll soon discuss enable you to specify that the next statement to execute may not necessarily be the next one in sequence. This is known as **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The finger of blame was pointed at the **goto statement**, which allowed the programmer to specify a transfer of control to one of a wide range of possible destinations in a program. Research demonstrated that programs could be written without `goto` statements. The notion of so-called **structured programming** became almost synonymous with "**goto elimination**." JavaScript does not have a `goto` statement. Structured programs are clearer, easier to debug and modify and more likely to be bug free in the first place.

Research determined that all programs could be written in terms of only three **control structures**, namely the **sequence structure**, the **selection structure** and the **repetition structure**. The sequence structure is built into JavaScript—unless directed otherwise, the computer executes JavaScript statements one after the other in the order in which they're written (i.e., in sequence). The flowchart segment of Fig. 7.1 illustrates a typical sequence structure in which two calculations are performed in order.

Fig. 7.1. Flowcharting JavaScript's sequence structure.

A **flowchart** is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles; these symbols are connected by arrows called **flowlines**, which indicate the order in which the actions of the algorithm execute.

Like pseudocode, flowcharts often are useful for developing and representing algorithms, although many programmers prefer pseudocode.

Flowcharts show clearly how control structures operate; that's all we use them for in this text.

Consider the flowchart segment for the sequence structure in Fig. 7.1. For simplicity, we use the **rectangle symbol** (or **action symbol**) to indicate *any* type of action, including a *calculation* or an *input/output* operation. The flowlines in the figure indicate the *order* in which the actions are performed—the first action adds `grade` to `total`, then the second action adds `1` to `counter`. JavaScript allows us to have as many actions as we want in a sequence structure. Anywhere a single action may be placed, as we'll soon see, we may place several actions in sequence.

In a flowchart that represents a *complete* algorithm, **oval symbols** containing the words "Begin" and "End" represent the start and end of the algorithm, respectively. In a flowchart that shows only a portion of an algorithm, as in Fig. 7.1, the oval symbols are omitted in favor of using **small circle symbols**, also called **connector symbols**.

Perhaps the most important flowcharting symbol is the **diamond symbol**, also called the **decision symbol**, which indicates that a decision is to be made. We discuss the diamond symbol in the next section.

JavaScript provides three types of selection structures; we discuss each in this chapter and in Chapter 8. The `if` selection statement performs (selects) an action if a condition is *true* or skips the action if the condition is *false*. The `if`...`else` selection statement performs an action if a condition is *true* and performs a *different* action if the condition is *false*. The `switch` selection statement (Chapter 8) performs one of many different actions, depending on the value of an expression.

The **`if`** statement is called a **single-selection statement** because it *selects* or *ignores* a single action (or, as we'll soon see, a single group of actions). The `if`...`else` statement is a **double-selection statement** because it *selects* between two *different* actions (or *groups* of actions). The `switch` statement is a **multiple-selection statement** because it selects among many different actions (or *groups* of actions).

JavaScript provides four repetition statements— `while` , `do` ... `while` , `for` and `for` ... `in` . (`do` ... `while` and `for` are covered in Chapter 8; `for` ... `in` is covered in Chapter 10.) Each of the words `if` , `else` , `switch` , `while` , `do` , `for` and `in` is a JavaScript **keyword**. These words are reserved by the language to implement various features, such as JavaScript's control structures. In addition to keywords, JavaScript has other words that are reserved for use by the language, such as the values `null` , `true` and `false` , and words that are reserved for possible future use. A complete list of JavaScript reserved words is shown in Fig. 7.2.

COMMON PROGRAMMING ERROR 7.1

Using a keyword as an identifier (e.g., for variable names) is a syntax error.

Fig. 7.2. JavaScript reserved keywords.

As we've shown, JavaScript has only eight control statements: sequence, three types of selection and four types of repetition. A script is formed by combining control statements as necessary to implement the script's algorithm. Each control statement is flowcharted with two small circle symbols, one at the *entry point* to the control statement and one at the *exit point*.

Single-entry/single-exit control statements make it easy to build scripts; the control statements are attached to one another by connecting the exit point of one to the entry point of the next. This process is similar to the way in which a child stacks building blocks, so we call it **control-statement stacking**. We'll learn that there's only one other way in which control statements may be connected—**control-statement nesting**. Thus, algorithms in JavaScript are constructed from only eight different types of control statements combined in only two ways.

7.5. `if` Selection Statement

A selection statement is used to choose among alternative courses of action in a script. For example, suppose that the passing grade on an examination is 60 (out of 100). Then the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is *true*, then "Passed" is printed, and the next pseudocode statement in order is "performed" (remember that pseudocode is *not* a *real* programming language). If the condition is *false*, the print statement is *ignored*, and the next pseudocode statement in order is performed.

Note that the second line of this selection statement is *indented*. Such indentation is optional but is highly recommended, because it emphasizes the inherent structure of structured programs. The JavaScript interpreter ignores *white-space characters*—blanks, tabs and newlines used for indentation and vertical spacing.

GOOD PROGRAMMING PRACTICE 7.1

Consistently applying reasonable indentation conventions improves script readability. We use three spaces per indent.

The preceding pseudocode *If* statement can be written in JavaScript as

```
if ( studentGrade >= 60 )
   document.writeln( "<p>Passed</p>" );
```

The JavaScript code corresponds closely to the pseudocode. This similarity is the reason that pseudocode is a useful script-development tool. The statement in the body of the `if` statement outputs the character string `"Passed"` in the HTML5 document.

The flowchart in Fig. 7.3 illustrates the single-selection `if` statement. This flowchart contains what is perhaps the most important flowcharting symbol—the *diamond symbol* (or *decision symbol*), which indicates that a *decision* is to be made. The decision symbol contains an expression, such as a condition, that can be either **true** or **false**. The decision symbol has two flowlines emerging from it. One indicates the path to follow in the script when the expression in the symbol is *true*; the other indicates the path to follow in the script when the expression is *false*. A decision can be made on any expression that evaluates to a value of JavaScript's boolean type (i.e., any expression that evaluates to `true` or `false`—also known as a **boolean expression**).

Fig. 7.3. Flowcharting the single-selection if statement.

SOFTWARE ENGINEERING OBSERVATION 7.2

In JavaScript, any nonzero numeric value in a condition evaluates to `true`*, and 0 evaluates to* `false`*. For strings, any* *string containing one or more characters evaluates to `true`, and the empty string (the string containing no characters, represented as `""`) evaluates to `false`. Also, a variable that's been declared with `var` but has not been assigned a value evaluates to `false`.*

Note that the `if` statement is a single-entry/single-exit control statement. We'll soon learn that the flowcharts for the remaining control statements also contain (besides small circle symbols and flowlines) only rectangle symbols, to indicate the *actions* to be performed, and diamond symbols, to indicate *decisions* to be made. This type of flowchart emphasizes the **action/decision model of programming**. We'll discuss the variety of ways in which actions and decisions may be written.

7.6. `if...else` Selection Statement

The `if` selection statement performs an indicated action only when the condition evaluates to `true`; otherwise, the action is skipped. The **`if...else` selection** statement allows you to specify that *a different* action is to be performed when the condition is `true` than when the condition is `false`. For example, the pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
Else
    Print "Failed"
```

prints `Passed` if the student's grade is greater than or equal to 60 and prints `Failed` if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence (i.e., the next statement after the whole `if...else` statement) is performed. Note that the body of the *Else* part of the statement is also indented.

GOOD PROGRAMMING PRACTICE 7.2

Indent both body statements of an `if … else` statement.

The preceding pseudocode *If...Else* statement may be written in JavaScript as

```
if ( studentGrade >= 60 )
  document.writeln( "<p>Passed</p>" );
else
  document.writeln( "<p>Failed</p>" );
```

The flowchart in Fig. 7.4 illustrates the `if … else` selection statement's flow of control. Once again, note that the only symbols in the flowchart besides small circles and arrows are rectangles for actions and a diamond for a decision.

Fig. 7.4. Flowcharting the double-selection `if … else` statement.

Conditional Operator (`?:`)

JavaScript provides an operator, called the **conditional operator** (**`?:`**), that's closely related to the `if … else` statement. The operator `?:` is JavaScript's only **ternary operator**—it takes *three* operands. The operands together with the `?:` form a **conditional expression**. The first operand is a boolean expression, the second is the value for the conditional expression if the expression evaluates to `true` and the third is the value for the conditional expression if the expression evaluates to `false`. For example, the following statement

```
document.writeln( studentGrade >= 60 ? "Passed" : "Failed" );
```

contains a conditional expression that evaluates to the string `"Passed"` if the condition `studentGrade >= 60` is true and evaluates to the string `"Failed"` if the condition is false. Thus, this statement with the conditional operator performs essentially the same operation as the preceding `if` … `else` statement.

Nested `if...else` Statements

Nested `if` … `else` statements test for multiple cases by placing `if` … `else` statements *inside* `if` … `else` statements. For example, the following pseudocode statement indicates that the script should print `A` for exam grades greater than or equal to 90, `B` for grades in the range 80 to 89, `C` for grades in the range 70 to 79, `D` for grades in the range 60 to 69 and `F` for all other grades:

```
If student's grade is greater than or equal to 90
    Print "A"
Else
    If student's grade is greater than or equal to 80
        Print "B"
    Else
        If student's grade is greater than or equal to 70
            Print "C"
        Else
            If student's grade is greater than or equal to 60
                Print "D"
            Else
                Print "F"
```

This pseudocode may be written in JavaScript as

```
if ( studentGrade >= 90 )
  document.writeln( "A" );
else
  if ( studentGrade >= 80 )
    document.writeln( "B" );
  else
  if ( studentGrade >= 70 )
    document.writeln( "C" );
  else
    if ( studentGrade >= 60 )
      document.writeln( "D" );
    else
      document.writeln( "F" );
```

If `studentGrade` is greater than or equal to 90, all four conditions will be true, but only the `document.writeln` statement after the *first* test will execute. After that particular `document.writeln` executes, the `else` part of the outer `if`...`else` statement is skipped.

GOOD PROGRAMMING PRACTICE 7.3

If there are several levels of indentation, each level should be indented the same additional amount of space.

Most programmers prefer to write the preceding `if` statement in the equivalent form:

```
if ( grade >= 90 )
  document.writeln( "A" );
else if ( grade >= 80 )
  document.writeln( "B" );
else if ( grade >= 70 )
  document.writeln( "C" );
else if ( grade >= 60 )
  document.writeln( "D" );
else
  document.writeln( "F" );
```

The latter form is popular because it avoids the deep indentation of the code to the right. Such deep indentation can force lines to be split and decrease script readability.

Dangling-`else` Problem

It's important to note that the JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`). The following code illustrates the **dangling-`else` problem**. For example,

```
if ( x > 5 )
  if ( y > 5 )
    document.writeln( "<p>x and y are > 5</p>" );
else
  document.writeln( "<p>x is <= 5</p>" );
```

appears to indicate with its indentation that if `x` is greater than `5`, the `if` structure in its body determines whether `y` is also greater than `5`. If so, the body of the nested `if` structure outputs the string `"x and y are > 5"`. Otherwise, it *appears* that if `x` is *not* greater than `5`, the `else` part of the `if`...`else` structure outputs the string `"x is <= 5"`.

Beware! The preceding nested `if` statement does *not* execute as it appears. The interpreter actually interprets the preceding statement as

```
if ( x > 5 )
  if ( y > 5 )
    document.writeln( "<p>x and y are > 5</p>" );
  else
    document.writeln( "<p>x is <= 5</p>" );
```

in which the body of the first `if` statement is a nested `if`...`else` statement. This statement tests whether `x` is greater than `5`. If so, execution continues by testing whether `y` is also greater than `5`. If the second condition is true, the proper string—`"x and y are > 5"`—is displayed. However, if the second condition is false, the string `"x is <= 5"` is displayed, even though we know that `x` is greater than `5`.

To force the *first* nested `if` statement to execute as it was intended originally, we must write it as follows:

```
if ( x > 5 )
{
  if ( y > 5 )
    document.writeln( "<p>x and y are > 5</p>" );
}
else
  document.writeln( "<p>x is <= 5</p>" );
```

The braces (`{}`) indicate to the JavaScript interpreter that the second `if` statement is in the *body* of the first `if` statement and that the `else` is matched with the *first* `if` statement.

Blocks

The `if` selection statement expects only *one* statement in its body. To include *several* statements in an `if` statement's body, enclose the statements in braces (`{` and `}`). This also can be done in the `else` section of an `if` ... `else` statement. A set of statements contained within a pair of braces is called a **block**.

SOFTWARE ENGINEERING OBSERVATION 7.3

A block can be placed anywhere in a script that a single statement can be placed.

SOFTWARE ENGINEERING OBSERVATION 7.4

Unlike individual statements, a block does not *end with a semicolon. However, each statement within the braces of a block should end with a semicolon.*

The following example includes a block in the `else` part of an `if` ... `else` statement:

```
if ( grade >= 60 )
   document.writeln( "<p>Passed</p>" );
else
{
   document.writeln( "<p>Failed</p>" );
   document.writeln( "<p>You must take this course again.</p>" );
}
```

In this case, if `grade` is less than 60, the script executes *both* statements in the body of the `else` and prints

```
Failed
You must take this course again.
```

Note the braces surrounding the two statements in the `else` clause. These braces are important. Without them, the statement

```
document.writeln( "<p>You must take this course again.</p>" );
```

would be *outside* the body of the `else` part of the `if` and would execute *regardless* of whether the grade is less than 60.

Syntax errors (e.g., when one brace in a block is left out of the script) are caught by the interpreter when it attempts to interpret the code containing the syntax error. They prevent the browser from executing the code. While many browsers notify users of errors, that information is of little use to them. That's why it's important to validate your JavaScripts and thoroughly test them. A **logic error** (e.g., the one caused when both braces around a block are left out of the script) also has its effect at execution time. A **fatal logic error** causes a script to fail and terminate prematurely. A **nonfatal logic error** allows a script to continue executing, but it produces incorrect results.

SOFTWARE ENGINEERING OBSERVATION 7.5

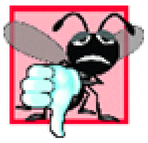
Just as a block can be placed anywhere a single statement can be placed, it's also possible to have no statement at all (the empty statement) in such places. We represent the empty statement by placing a semicolon (;) where a statement would normally be.

7.7. `while` Repetition Statement

A *repetition structure* (also known as a **loop**) allows you to specify that a script is to repeat an action while some condition remains *true*. The pseudocode statement

```
While there are more items on my shopping list
   Purchase next item and cross it off my list
```

describes the repetition that occurs during a shopping trip. The condition "there are more items on my shopping list" may be true or false. If it's true, then the action "Purchase next item and cross it off my list" is performed. This action is performed *repeatedly* while the condition remains true. The statement(s) contained in the *While* repetition structure constitute its body. The body of a loop such as the *While* structure may be a single statement or a block. Eventually, the condition becomes false—when the last item on the shopping list has been purchased and crossed off the list. At this point, the repetition terminates, and the first pseudocode statement after the repetition structure "executes."



COMMON PROGRAMMING ERROR 7.2

If the body of a `while` statement never causes the while statement's condition to become true, a logic error occurs. Normally, such a repetition structure will never terminate—an error called an **infinite loop**. *Many browsers show a dialog allowing the user to terminate a script that contains an infinite loop.*

As an example of a `while` statement, consider a script segment designed to find the first power of 2 larger than 1000. Variable `product` begins with the value 2. The statement is as follows:

```
var product = 2;

while ( product <= 1000 )
  product = 2 * product;
```

When the `while` statement finishes executing, `product` contains the result 1024. The flowchart in Fig. 7.5 illustrates the flow of control of the preceding `while` repetition statement. Once again, note that (besides small circles and arrows) the flowchart contains *only* a rectangle symbol and a diamond symbol.

Fig. 7.5. Flowcharting the `while` repetition statement.

When the script enters the `while` statement, `product` is 2. The script repeatedly multiplies variable `product` by 2, so `product` takes on the values 4, 8, 16, 32, 64, 128, 256, 512 and 1024 successively. When `product` becomes 1024, the condition `product <= 1000` in the `while` statement becomes `false`. This terminates the repetition, with 1024 as `product`'s final value. Execution continues with the next statement after the `while` statement. [*Note:* If a `while` statement's condition is initially `false`, the body statement(s) will *never* execute.]

The flowchart clearly shows the repetition. The flowline emerging from the rectangle wraps back to the decision, which the script tests each time through the loop until the decision eventually becomes false. At this point, the `while` statement exits, and control passes to the next statement in the script.

7.8. Formulating Algorithms: Counter-Controlled Repetition

To illustrate how to develop algorithms, we solve several variations of a class-average problem. Consider the following problem statement:

> *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

The class average is equal to the sum of the grades divided by the number of students (10 in this case). The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation and display the result.

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** to control the number of times a set of statements executes. In this example, repetition terminates when the counter exceeds 10. In this section, we present a pseudocode algorithm (Fig. 7.6) and the corresponding script (Fig. 7.7). In the next section, we show how to develop pseudocode algorithms. Counter-controlled repetition often is called **definite repetition**, because the number of repetitions is known before the loop begins executing.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
   Input the next grade
   Add the grade into the total
   Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 7.6. Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```
<!DOCTYPE html>

<!-- Fig. 7.7: average.html -->
<!-- Counter-controlled repetition to calculate a class average. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Class Average Program</title>
    <script>

       var total; // sum of grades
       var gradeCounter; // number of grades entered
       var grade; // grade typed by user (as a string)
       var gradeValue; // grade value (converted to integer)
       var average; // average of all grades

       // initialization phase
       total = 0; // clear total
       gradeCounter = 1; // prepare to loop

       // processing phase
       while ( gradeCounter <= 10 ) // loop 10 times
       {

         // prompt for input and read grade from user
         grade = window.prompt("Enter integer grade:", "0" );

         // convert grade from a string to an integer
         gradeValue = parseInt( grade );

         // add gradeValue to total
         total = total + gradeValue;

         // add 1 to gradeCounter
         gradeCounter = gradeCounter + 1;
      } // end while

      // termination phase
      average = total / 10;   // calculate the average

      // display average of exam grades
      document.writeln(
         "<h1>Class average is " + average + "</h1>" );

    </script>
  </head><body></body>
</html>
```

Fig. 7.7. Counter-controlled repetition to calculate a class average.

Variables Used in the Algorithm

Note the references in the algorithm to a total and a counter. A **total** is a variable in which a script accumulates the sum of a series of values. A counter is a variable a script uses to count—in this case, to count the number of grades entered. Variables that store totals should normally be initialized to zero before they're used in a script.

Lines 11–15 declare variables `total`, `gradeCounter`, `grade`, `gradeValue`, `average`. The variable `grade` will store the *string* the user types into the `prompt` dialog. The variable `gradeValue` will store the integer value of the `grade` the user enters into the `prompt` dialog.

Initializing Variables

Lines 18–19 are assignments that initialize `total` to `0` and `gradeCounter` to `1`. Note that variables `total` and `gradeCounter` are initialized before they're used in a calculation.

COMMON PROGRAMMING ERROR 7.3

Not initializing a variable that will be used in a calculation results in a logic error that produces the value `NaN` *("Not a Number").*

The `while` Repetition Statement

Line 22 indicates that the `while` statement continues iterating while the value of `gradeCounter` is less than or equal to 10. Line 26 corresponds to the pseudocode statement "*Input the next grade.*" The statement displays a `prompt` dialog with the prompt `"Enter integer grade:"` on the screen.

After the user enters the `grade`, line 29 converts it from a string to an integer. We *must* convert the string to an integer in this example; otherwise, the addition operation in line 32 will be a *string-concatenation*.

Next, the script updates the `total` with the new `gradeValue` entered by the user. Line 32 adds `gradeValue` to the previous value of `total` and assigns the result to `total`. This statement seems a bit strange, because it does not follow the rules of algebra. Keep in mind that JavaScript operator precedence evaluates the addition (`+`) operation before the assignment (=) operation. The value of the expression on the *right* side of the assignment operator always *replaces* the value of the variable on the *left* side.

The script now is ready to increment the variable `gradeCounter` to indicate that a grade has been processed and to read the next grade from the user. Line 35 adds `1` to `gradeCounter`, so the condition in the `while` statement will eventually become `false` and terminate the loop. After this statement executes, the script continues by testing the condition in the `while` statement in line 22. If the condition is still `true`, the statements in lines 26–35 repeat. Otherwise the script continues execution with the first statement in sequence after the body of the loop (i.e., line 39).

Calculating and Displaying the Results

Line 39 assigns the results of the average calculation to variable `average`. Lines 42–43 write a line of HTML5 text in the document that displays the string `"Class average is "` followed by the value of variable `average` as an `<h1>` element.

Testing the Program

Open the HTML5 document in a web browser to execute the script. This script parses any user input as an integer. In the sample execution in Fig. 7.7, the sum of the values entered (100, 88, 93, 55, 68, 77, 83, 95, 73 and 62) is 794. Although the script treats all input as integers, the averaging calculation in the script does not produce an integer. Rather, the calculation produces a **floating-point number** (i.e., a number containing a decimal point). The average of the 10 integers input by the user in this example is 79.4. If your script requires the user to enter floating-point numbers, you can convert the user input from strings to numbers using the JavaScript function `parseFloat`, which we introduce in Section 9.2.

SOFTWARE ENGINEERING OBSERVATION 7.6

If the string passed to `parseInt` contains a floating-point numeric value, `parseInt` simply `truncates` the floating-point part. For example, the string `"27.95"` results in the integer `27`, and the string `"-123.45"` results in the integer `-123`. If the string passed to `parseInt` does begin with a numeric value, `parseInt` returns `NaN` (not a number). If you need to know whether `parseInt` returned `NaN`, JavaScript provides the function `isNaN`, which determines whether its argument has the value `NaN` and, if so, returns `true`; otherwise, it returns `false`.

Floating-Point Numbers

JavaScript actually represents all numbers as floating-point numbers in memory. Floating-point numbers often develop through division, as shown in this example. When we divide 10 by 3, the result is 3.3333333..., with the sequence of 3s repeating *infinitely*. The computer allocates only a *fixed* amount of space to hold such a value, so the stored floating-point value can be only an approximation. Although floating-point numbers are not always 100 percent precise, they have numerous applications. For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits. When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643. The point here is that few applications require such high-precision floating-point values, so calling this number simply 98.6 is fine for many applications.

A Note About Input Via `prompt` Dialogs

In this example, we used `prompt` dialogs to obtain user input. Typically, such input would be accomplished via form elements in an HTML5 document, but this requires additional scripting techniques that are introduced starting in Chapter 9. For now, we'll continue to use `prompt` dialogs.

7.9. Formulating Algorithms: Sentinel-Controlled Repetition

Let's generalize the class-average problem. Consider the following problem:

> *Develop a class-averaging script that will process an arbitrary number of grades each time the script is run.*

In the first class-average example, the number of grades (10) was known in advance. In this example, no indication is given of how many grades the user will enter. The script must process an *arbitrary* number of grades. How can the script determine when to stop the input of grades? How will it know when to calculate and display the class average?

One way to solve this problem is to use a special value called a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate the end of data entry. The user types in grades until all legitimate grades have been entered. Then the user types the sentinel value to indicate that the last grade has been entered. Sentinel-controlled repetition is often called **indefinite repetition**, because the number of repetitions is not known before the loop begins executing.

Clearly, you must choose a sentinel value that *cannot* be confused with an acceptable input value. –1 is an acceptable sentinel value for this problem, because grades on a quiz are normally nonnegative integers from 0 to 100. Thus, an execution of the class-average script might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The script would compute and print the class average for the grades 95, 96, 75, 74 and 89 (–1 is the sentinel value, so it should *not* enter into the average calculation).

Developing the Pseudocode Algorithm with Top-Down, Stepwise Refinement: The Top and First Refinement

We approach the class-average script with a technique called **top-down, stepwise refinement**, a technique that's essential to the development of well-structured algorithms. We begin with a pseudocode representation of the **top**:

```
Determine the class average for the quiz
```

The top is a single statement that conveys the script's overall purpose. As such, the top is, in effect, a *complete* representation of a script. Unfortunately, the top rarely conveys sufficient detail from which to write the JavaScript algorithm. Therefore we must begin a refinement process. First, we divide the top into a series of smaller tasks and list them in the order in which they need to be performed, creating the following **first refinement**:

```
Initialize variables
Input, sum up and count the quiz grades
Calculate and print the class average
```

Here, only the sequence structure is used; the steps listed are to be executed in order, one after the other.

SOFTWARE ENGINEERING OBSERVATION 7.7

Each refinement, as well as the top itself, is a complete *specification of the algorithm; only the level of detail varies.*

Proceeding to the Second Refinement

To proceed to the next level of refinement (the **second refinement**), we commit to specific variables. We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the string representation of each grade as it's input, a variable to store the value of the grade after it's converted to an integer and a variable to hold the calculated average. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize total to zero
Initialize gradeCounter to zero
```

Only the variables *total* and *gradeCounter* are initialized before they're used; the variables *average*, *grade* and *gradeValue* (for the calculated average, the user input and the integer representation of the *grade*, respectively) need not be initialized, because their values are determined as they're calculated or input.

The pseudocode statement

```
Input, sum up and count the quiz grades
```

requires a repetition statement that successively inputs each grade. We do not know in advance how many grades are to be processed, so we'll use *sentinel-controlled repetition*. The user will enter legitimate grades, one at a time. After entering the last legitimate grade, the user will enter the sentinel value. The script will test for the sentinel value after the user enters each grade and will terminate the loop when the sentinel value is encountered. The second refinement of the preceding pseudocode statement is then

```
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
   Add this grade into the running total
   Add one to the grade counter
   Input the next grade (possibly the sentinel)
```

In pseudocode, we do *not* use braces around the pseudocode that forms the body of the *While* structure. We simply indent the pseudocode under the *While* to show that it belongs to the body of the *While*. Remember, pseudocode is only an *informal* development aid.

The pseudocode statement

```
Calculate and print the class average
```

may be refined as follows:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
```

```
   Print the average
Else
   Print "No grades were entered"
```

We test for the possibility of **division by zero**—a logic error that, if undetected, would cause the script to produce invalid output. The complete second refinement of the pseudocode algorithm for the class-average problem is shown in Fig. 7.8.

ERROR-PREVENTION TIP 7.1

When performing division by an expression whose value could be zero, explicitly test for this case, and handle it appropriately in your script (e.g., by displaying an error message) rather than allowing the division by zero to occur.

SOFTWARE ENGINEERING OBSERVATION 7.8

Many algorithms can be divided logically into three phases: an initialization phase *that initializes the script variables, a* processing phase *that inputs data values and adjusts variables accordingly, and a* termination phase *that calculates and prints the results.*

The Complete Second Refinement

The pseudocode algorithm in Fig. 7.8 solves the more general class-average problem. This algorithm was developed after only two refinements. Sometimes more refinements are necessary.

```
Initialize total to zero
Initialize gradeCounter to zero
```

```
Input the first grade (possibly the sentinel)

While the user has not as yet entered the sentinel
   Add this grade into the running total
   Add one to the grade counter
   Input the next grade (possibly the sentinel)

If the counter is not equal to zero
   Set the average to the total divided by the counter
   Print the average
Else
   Print "No grades were entered"
```

Fig. 7.8. Sentinel-controlled repetition to solve the class-average problem.

SOFTWARE ENGINEERING OBSERVATION 7.9

You terminate the top-down, stepwise refinement process after specifying the pseudocode algorithm in sufficient detail for you to convert the pseudocode to JavaScript. Then, implementing the JavaScript is normally straightforward.

SOFTWARE ENGINEERING OBSERVATION 7.10

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution.

SOFTWARE ENGINEERING OBSERVATION 7.11

Many experienced programmers write scripts without ever using script-development tools like pseudocode. As they see it, their ultimate goal is to solve the problem on a computer, and writing pseudocode merely delays the production of final outputs. Although this approach may work for simple and familiar problems, it can lead to serious errors in large, complex projects.

Implementing Sentinel-Controlled Repetition to Calculate a Class Average

Figure 7.9 shows the JavaScript and a sample execution. Although each grade is an integer, the averaging calculation is likely to produce a number with a decimal point (a real number).

In this example, we see that control structures may be *stacked* on top of one another (in sequence) just as a child stacks building blocks. The `while` statement (lines 29–43) is followed immediately by an `if`…`else` statement (lines 46–55) in sequence. Much of the code in this script is identical to the code in Fig. 7.7, so we concentrate in this example on the new features.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.9: average2.html -->
4  <!-- Sentinel-controlled repetition to calculate a class average. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Class Average Program: Sentinel-controlled Repetition</title>
9      <script>
10
```

```
var total; // sum of grades
var gradeCounter; // number of grades entered
var grade; // grade typed by user (as a string)
var gradeValue; // grade value (converted to integer)
var average; // average of all grades

// initialization phase
total = 0; // clear total
gradeCounter = 0; // prepare to loop

// processing phase
// prompt for input and read grade from user
grade = window.prompt(
    "Enter Integer Grade, -1 to Quit:", "0" );

// convert grade from a string to an integer
gradeValue = parseInt( grade );

while ( gradeValue != -1 )
{
  // add gradeValue to total
  total = total + gradeValue;

  // add 1 to gradeCounter
  gradeCounter = gradeCounter + 1;

  // prompt for input and read grade from user
  grade = window.prompt(
      "Enter Integer Grade, -1 to Quit:", "0" );

  // convert grade from a string to an integer
  gradeValue = parseInt( grade );
} // end while

// termination phase
if ( gradeCounter != 0 )
```

```
         {
            average = total / gradeCounter;

            // display average of exam grades
            document.writeln(
               "<h1>Class average is " + average + "</h1>" );
         } // end if
         else
            document.writeln( "<p>No grades were entered</p>" );

      </script>
   </head><body></body>
</html>
```

Fig. 7.9. Sentinel-controlled repetition to calculate a class average.

Line 19 initializes `gradeCounter` to `0`, because no grades have been entered yet. Remember that the script uses *sentinel-controlled repetition*. To keep an accurate record of the number of grades entered, the script increments `gradeCounter` only after processing a valid grade value.

Script Logic for Sentinel-Controlled Repetition vs. Counter-Controlled Repetition

Note the difference in logic for sentinel-controlled repetition as compared with the counter-controlled repetition in Fig. 7.7. In counter-controlled repetition, we read a value from the user during each iteration of the `while` statement's body for the specified number of iterations. In sentinel-controlled repetition, we read one value (lines 23–24) and convert it to an integer (line 27) before the script reaches the `while` statement. The script uses this value to determine whether the script's flow of control should enter the body of the `while` statement. If the `while` statement's condition is `false` (i.e., the user typed the sentinel as the first grade), the script ignores the body of the `while` statement (i.e., no grades were entered). If the condition is `true`, the body begins execution and processes the value entered by the user (i.e., adds the value to the `total` in line 32). After processing the value, the script increments `gradeCounter` by 1 (line 35), inputs the next `grade` from the user (lines 38–39) and converts the `grade` to an integer (line 42), before the end of the `while` statement's body. When the script reaches the closing right brace (}) of the body in line 43, execution continues with the next test of the condition of the `while` statement (line 29), using the new value just entered by the user to determine whether the `while` statement's body should execute again. Note that the next value always is input from the user immediately before the script evaluates the condition of the `while` statement. This order allows us to determine whether the value just entered by the user is the sentinel value *before* processing it (i.e., adding it to the `total`). If the value entered *is* the sentinel value, the `while` statement terminates and the script does not add the value to the `total`.

Note the block in the `while` loop in Fig. 7.9 (lines 30–43). Without the braces, the last three statements in the body of the loop would fall *outside* the loop, causing the code to be interpreted incorrectly, as follows:

```
while ( gradeValue != -1 )
  // add gradeValue to total
  total = total + gradeValue;

// add 1 to gradeCounter
gradeCounter = gradeCounter + 1;
```

```
// prompt for input and read grade from user
grade = window.prompt(
   "Enter Integer Grade, -1 to Quit:", "0" );

// convert grade from a string to an integer
gradeValue = parseInt( grade );
```

This interpretation would cause an *infinite loop* in the script if the user did not input the sentinel `-1` as the first input value in lines 23–24 (i.e., before the `while` statement).

7.10. Formulating Algorithms: Nested Control Statements

Let's work through another complete problem. We once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and write a corresponding script.

Consider the following problem statement:

> *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing exam. Naturally, the college wants to know how well its students performed. You've been asked to write a script to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam and a 2 if the student failed.*
>
> *Your script should analyze the results of the exam as follows:*

1. *Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the script requests another test result.*

2. *Count the number of test results of each type.*

3. *Display a summary of the test results indicating the number of students who passed and the number of students who failed.*

4. *If more than eight students passed the exam, print the message "Bonus to instructor!"*

After reading the problem statement carefully, we make the following observations:

1. The script must process test results for 10 students. A counter-controlled loop will be used.

2. Each test result is a number—either a 1 or a 2. Each time the script reads a test result, the script must determine whether the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2.

3. Two counters are used to keep track of the exam results—one to count the number of students who passed the exam and one to count the number of students who failed the exam.

After the script processes all the results, it must decide whether more than eight students passed the exam. Let's proceed with top-down, step-wise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide whether a bonus should be paid
```

Once again, it's important to emphasize that the top is a complete representation of the script, but that several refinements are necessary before the pseudocode can be evolved naturally into JavaScript. Our first refinement is as follows:

```
Initialize variables
Input the 10 exam grades and count passes and failures
Print a summary of the exam results and decide whether a bonus should be paid
```

Here, too, even though we have a complete representation of the entire script, further refinement is necessary. We now commit to specific variables. Counters are needed to record the passes and failures; a counter will be used to control the looping process, and a variable is needed to store the user input. The pseudocode statement

```
Initialize variables
```

may be refined as follows:

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one
```

Note that only the counters for the number of passes, the number of failures and the number of students are initialized. The pseudocode statement

```
Input the 10 exam grades and count passes and failures
```

requires a loop that successively inputs the result of each exam. Here, it's known in advance that there are precisely 10 exam results, so counter-controlled repetition is appropriate. Inside the loop (i.e., *nested* within the loop), a double-selection structure will determine whether each exam result is a pass or a failure and will increment the appropriate counter accordingly. The refinement of the preceding pseudocode statement is then

```
While student counter is less than or equal to ten
  Input the next exam result
  If the student passed
      Add one to passes
  Else
      Add one to failures
  Add one to student counter
```

Blank lines can be used to set off the *If…Else* control structure to improve script readability. The pseudocode statement

```
Print a summary of the exam results and decide whether a bonus should be
paid
```

may be refined as follows:

```
Print the number of passes
Print the number of failures
If more than eight students passed
    Print "Bonus to instructor!"
```

Complete Second Refinement of Pseudocode and Conversion to JavaScript

The complete second refinement appears in Fig. 7.10. Note that blank lines are also used to set off the *While* statement for script readability.

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

While student counter is less than or equal to ten
   Input the next exam result

   If the student passed
       Add one to passes
   Else
       Add one to failures
   Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
   Print "Bonus to Instructor!"
```

Fig. 7.10. Examination-results problem pseudocode.

This pseudocode is now refined sufficiently for conversion to JavaScript. The JavaScript and two sample executions are shown in Fig. 7.11.

```
<!DOCTYPE html>

<!-- Fig. 7.11: analysis.html -->
<!-- Examination-results calculation. -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Analysis of Examination Results</title>
    <script>

      // initializing variables in declarations
      var passes = 0; //  number of passes
      var failures = 0; //  number of failures
      var student = 1; // student counter
      var result; // an exam result

      // process 10 students; counter-controlled loop
      while ( student <= 10 )
      {
        result = window.prompt( "Enter result (1=pass,2=fail)", "0" );

        if ( result == "1" )
          passes = passes + 1;
        else
          failures = failures + 1;

        student = student + 1;
      } // end while

      // termination phase
      document.writeln( "<h1>Examination Results</h1>" );
      document.writeln( "<p>Passed: " + passes +
        "; Failed: " + failures + "</p>" );

      if ( passes > 8 )
```

```
            document.writeln( "<p>Bonus to instructor!</p>" );

      </script>
   </head><body></body>
</html>
```

Fig. 7.11. Examination-results calculation.

Lines 12–15 declare the variables used to process the examination results. Note that JavaScript allows *variable initialization* to be incorporated into declarations (`passes` is assigned `0` , `failures` is assigned `0` and `student` is assigned `1`). Some scripts may require reinitialization at the beginning of each repetition; such reinitialization would normally occur in assignment statements.

The processing of the exam results occurs in the `while` statement in lines 18–28. Note that the `if` ... `else` statement in lines 22–25 in the loop tests only whether the exam result was 1; it assumes that all other exam results are 2. Normally, you should validate the values input by the user (i.e., determine whether the values are correct).

GOOD PROGRAMMING PRACTICE 7.4

When inputting values from the user, validate the input to ensure that it's correct. If an input value is incorrect, prompt the user to input the value again. The HTML5 self-validating controls can help you check the formatting of your data, but you may need additional tests to check that properly formatted values make sense in the context of your application.

7.11. Assignment Operators

JavaScript provides several additional assignment operators (called **compound assignment operators**) for abbreviating assignment expressions. For example, the statement

```
c = c + 3;
```

can be abbreviated with the **addition assignment operator**, `+=` , as

```
c += 3;
```

The `+=` operator adds the value of the expression on the *right* of the operator to the value of the variable on the *left* of the operator and stores the result in the variable on the *left* of the operator. Any statement of the form

variable = *variable operator expression*;

where *operator* is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we'll discuss later in the text), can be written in the form

variable operator = *expression*;

Thus, the assignment `c += 3` adds `3` to `c`. Figure 7.12 shows the arithmetic assignment operators, sample expressions using these operators and explanations of the meaning of the operators.

Fig. 7.12. Arithmetic assignment operators.

7.12. Increment and Decrement Operators

JavaScript provides the unary **increment operator** (**++**) and **decrement operator** (**--**) (summarized in Fig. 7.13). If a variable `c` is incremented by 1, the increment operator, `++`, can be used rather than the expression `c = c + 1` or `c += 1`. If an increment or decrement operator is placed *before* a variable, it's referred to as the **preincrement** or **predecrement operator**, respectively. If an increment or decrement operator is placed *after* a variable, it's referred to as the **postincrement** or **postdecrement operator**, respectively.

Fig. 7.13. Increment and decrement operators.

Preincrementing (or predecrementing) a variable causes the script to increment (decrement) the variable by 1, then use the new value of the variable in the expression in which it appears. Postincrementing (postdecrementing) the variable causes the script to use the current value of the variable in the expression in which it appears, then increment (decrement) the variable by 1.

The script in Fig. 7.14 demonstrates the difference between the preincrementing and postincrementing versions of the `++` increment operator. Postincrementing the variable `c` causes it to be incremented *after* it's used in the `document.writeln` method call (line 17). Preincrementing the variable `c` causes it to be incremented *before* it's used in the `document.writeln` method call (line 24). The script displays the value of `c` before and after the `++` operator is used. The decrement operator (`--`) works similarly.

```
1  <!DOCTYPE html>
2
3  <!-- Fig. 7.14: increment.html -->
4  <!-- Preincrementing and Postincrementing. -->
5  <html>
6    <head>
7      <meta charset = "utf-8">
8      <title>Preincrementing and Postincrementing</title>
9      <script>
10
```

```
         var c;

         c = 5;
         document.writeln( "<h3>Postincrementing</h3>" );
         document.writeln( "<p>" + c ); // prints 5
         // prints 5 then increments
         document.writeln( " " + c++ );
         document.writeln( " " + c + "</p>" ); // prints 6

         c = 5;
         document.writeln( "<h3>Preincrementing</h3>" );
         document.writeln( "<p>" + c ); // prints 5
         // increments then prints 6
         document.writeln( " " + ++c );
         document.writeln( " " + c + "</p>" ); // prints 6

      </script>
   </head><body></body>
</html>
```

Fig. 7.14. Preincrementing and postincrementing.

GOOD PROGRAMMING PRACTICE 7.5

For readability, unary operators should be placed next to their operands, with no intervening spaces.

The three assignment statements in Fig. 7.11 (lines 23, 25 and 27, respectively),

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written more concisely with assignment operators as

```
passes += 1;
failures += 1;
student += 1;
```

with preincrement operators as

```
++passes;
++failures;
++student;
```

or with postincrement operators as

```
passes++;
failures++;
student++;
```

When incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the *same* effect, and the predecrement and postdecrement forms have the same effect. It's only when a variable appears in the context of a larger expression that preincrementing the variable and post-incrementing the variable have different effects. Predecrementing and postdecrementing behave similarly.

COMMON PROGRAMMING ERROR 7.4

Attempting to use the increment or decrement operator on an expression other than a **left-hand-side expression***—commonly called an* **lvalue***—is a syntax error. A left-hand-side expression is a variable or expression that can appear on the left side of an assignment operation. For example, writing* `++(x + 1)` *is a syntax error, because* `(x + 1)` *is not a left-hand-side expression.*

Figure 7.15 lists the precedence and associativity of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column describes the associativity of the operators at each level of precedence. The conditional operator (`?:`), the unary operators increment (`++`) and decrement (`--`) and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` associate from *right to left*. All other operators shown here associate from *left to right*. The third column names the groups of operators.

Fig. 7.15. Precedence and associativity of the operators discussed so far.

7.13. Web Resources

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. Be sure to visit the related Resource Centers on HTML5 (www.deitel.com/html5/) and CSS3 (www.deitel.com/css3/).

Summary

Section 7.2 Algorithms

- Any computable problem can be solved by executing a series of actions in a specific order.
- A procedure (p. 215) for solving a problem in terms of the actions (p. 215) to execute and the order in which the actions are to execute (p. 215) is called an algorithm (p. 215).
- Specifying the order in which the actions are to be executed in a computer program is called program control (p. 215).

Section 7.3 Pseudocode

- Pseudocode (p. 215) is an informal language that helps you develop algorithms.
- Carefully prepared pseudocode may be converted easily to a corresponding script.

Section 7.4 Control Statements

- Normally, statements in a script execute one after the other, in the order in which they're written. This process is called sequential execution (p. 215).
- Various JavaScript statements enable you to specify that the next statement to be executed may not necessarily be the next one in sequence. This is known as transfer of control (p. 216).

- All scripts could be written in terms of only three control structures—namely, the sequence structure, (p. 216) the selection structure (p. 216) and the repetition structure (p. 216).

- A flowchart (p. 216) is a graphical representation of an algorithm or of a portion of an algorithm. Flowcharts are drawn using certain special-purpose symbols, such as rectangles (p. 216), diamonds (p. 217), ovals (p. 216) and small circles (p. 216); these symbols are connected by arrows called flowlines (p. 216), which indicate the order in which the actions of the algorithm execute.

- JavaScript provides three selection structures. The `if` selection statement (p. 217) performs an action only if a condition is true. The `if...else` selection statement performs an action if a condition is true and a different action if the condition is false. The `switch` selection statement performs one of many different actions, depending on the value of an expression.

- JavaScript provides four repetition statements— `while` (p. 217), `do` … `while` , `for` and `for` … `in` .

- Keywords (p. 217) cannot be used as identifiers (e.g., for variable names).

- Single-entry/single-exit control structures (p. 218) make it easy to build scripts. Control statements are attached to one another by connecting the exit point of one control statement to the entry point of the next. This procedure is called control-statement stacking (p. 218). There's only one other way control statements may be connected: control-statement nesting (p. 218).

Section 7.5 `if` Selection Statement

- The JavaScript interpreter ignores white-space characters: blanks, tabs and newlines used for indentation and vertical spacing. Programmers insert white-space characters to enhance script clarity.

- A decision can be made on any expression that evaluates to `true` or `false` (p. 218).

- The indentation convention you choose should be carefully applied throughout your scripts. It's difficult to read scripts that do not uniform spacing conventions.

Section 7.6 `if`...`else` Selection Statement

- The conditional operator (`?:` ; p. 220) is closely related to the `if`...`else` statement. Operator `?:` is JavaScript's only ternary operator—it takes three operands. The operands together with the `?:` operator form a conditional expression (p. 220). The first operand is a boolean expression, the second is the value for the conditional expression if the boolean expression evaluates to true and the third is the value for the conditional expression if the boolean expression evaluates to false.

- Nested `if`...`else` statements (p. 220) test for multiple cases by placing `if`...`else` statements inside other `if`...`else` structures.

- The JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`).

- The `if` selection statement expects only one statement in its body. To include several statements in the body, enclose the statements in a block (p. 222) delimited by braces (`{` and `}`).

- A logic error (p. 223) has its effect at execution time. A fatal logic error (p. 223) causes a script to fail and terminate prematurely. A nonfatal logic error (p. 223) allows a script to continue executing, but the script produces incorrect results.

Section 7.7 `while` Repetition Statement

- The `while` repetition statement allows the you to specify that an action is to be repeated while some condition remains true.

Section 7.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition (p. 225) is often called definite repetition, because the number of repetitions is known before the loop begins executing.

- Uninitialized variables used in mathematical calculations result in logic errors and produce the value `NaN` (not a number).

- JavaScript represents all numbers as floating-point numbers in memory. Floating-point numbers (p. 228) often develop through division. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can only be an approximation.

Section 7.9 Formulating Algorithms: Sentinel-Controlled Repetition

- In sentinel-controlled repetition, a special value called a sentinel value (also called a signal value, a dummy value or a flag value, p. 228) indicates the end of data entry. Sentinel-controlled repetition is often called indefinite repetition (p. 229), because the number of repetitions is not known in advance.

- It's necessary to choose a sentinel value that cannot be confused with an acceptable input value.

- Top-down, stepwise refinement (p. 229) is a technique essential to the development of well-structured algorithms. The top (p. 229) is a single statement that conveys the overall purpose of the script. As such, the top is, in effect, a complete representation of a script. The stepwise refinement process divides the top into a series of smaller tasks. Terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to JavaScript.

Section 7.10 Formulating Algorithms: Nested Control Statements

- Control statements can be nested to perform more complex tasks.

Section 7.11 Assignment Operators

- JavaScript provides the arithmetic assignment operators `+=`, `-=`, `*=`, `/=` and `%=` (p. 238), which abbreviate certain common types of expressions.

Section 7.12 Increment and Decrement Operators

- The increment operator, `++` (p. 239), and the decrement operator, `--` (p. 239), increment or decrement a variable by 1, respectively. If the operator is prefixed to the variable, the variable is incremented or decremented by 1, then used in its expression. If the operator is postfixed to the variable, the variable is used in its expression, then incremented or decremented by 1.

Self-Review Exercises

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

a. All scripts can be written in terms of three types of control statements: ________, ________ and ________.

b. The ________ double-selection statement is used to execute one action when a condition is true and another action when that condition is false.

c. Repeating a set of instructions a specific number of times is called ________ repetition.

d. When it's not known in advance how many times a set of statements will be repeated, a(n) ________ (or a(n) ________, ________ or ________) value can be used to terminate the repetition.

7.2 Write four JavaScript statements that each add 1 to variable `x`, which contains a number.

7.3 Write JavaScript statements to accomplish each of the following tasks:

a. Assign the sum of `x` and `y` to `z`, and increment the value of `x` by 1 after the calculation. Use only one statement.

b. Test whether the value of the variable `count` is greater than 10. If it is, print `"Count is greater than 10"`.

c. Decrement the variable `x` by 1, then subtract it from the variable `total`. Use only one statement.

d. Calculate the remainder after `q` is divided by `divisor`, and assign the result to `q`. Write this statement in two different ways.

7.4 Write a JavaScript statement to accomplish each of the following tasks:

a. Declare variables `sum` and `x`.

b. Assign `1` to variable `x`.

c. Assign `0` to variable `sum`.

d. Add variable `x` to variable `sum`, and assign the result to variable `sum`.

e. Print `"The sum is: "`, followed by the value of variable `sum`.

7.5 Combine the statements you wrote in Exercise 7.4 into a script that calculates and prints the sum of the integers from 1 to 10. Use the `while` statement to loop through the calculation and increment statements. The loop should terminate when the value of `x` becomes 11.

7.6 Determine the value of each variable after the calculation is performed. Assume that, when each statement begins executing, all variables have the integer value 5.

a. `product *= x++;`

b. `quotient /= ++x;`

7.7 Identify and correct the *errors* in each of the following segments of code:

a.

```
while ( c <= 5 ) {
  product *= c;
  ++c;
```

b.

```
if ( gender == 1 )
  document.writeln( "Woman" );
else;
  document.writeln( "Man" );
```

7.8 What is wrong with the following `while` repetition statement?

```
while ( z >= 0 )
  sum += z;
```

Answers to Self-Review Exercises

7.1

a. Sequence, selection and repetition.

b. `if` ... `else` .

c. Counter-controlled (or definite).

d. Sentinel, signal, flag or dummy.

7.2

```
x = x + 1;
x += 1;
++x;
x++;
```

7.3

a. `z = x++ + y;`

b.

```
if ( count > 10 )
  document.writeln( "Count is greater than 10" );
```

c. `total -= --x;`

d.

```
q %= divisor;
q = q % divisor;
```

7.4

a. `var sum, x;`

b. `x = 1 ;`

c. `sum = 0 ;`

d. `sum += x;` or `sum = sum + x;`

e. `document.writeln( "The sum is: " + sum );`

7.5 The solution is as follows:

```
<!DOCTYPE html>

<!-- Exercise 7.5: ex08_05.html -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Sum the Integers from 1 to 10</title>
    <script>
      var sum; // stores the total
       var x; //counter control variable

       x = 1;
       sum = 0;

       while ( x <= 10 )
       {
          sum += x;
          ++x;
```

```
         } // end while
         document.writeln( "The sum is: " + sum );
      </script>
   </head><body></body>
</html>
```

7.6

a. `product = 25, x = 6;`

b. `quotient = 0.833333 … , x = 6;`

7.7

a. Error: Missing the closing right brace of the `while` body.

Correction: Add closing right brace after the statement `++c;` .

b. Error: The `;` after `else` causes a logic error. The second output statement always executes.

Correction: Remove the semicolon after `else` .

7.8 The value of the variable `z` is never changed in the body of the `while` statement. Therefore, if the loop-continuation condition `(z >= 0)` is true, an *infinite loop* is created. To prevent the creation of the infinite loop, `z` must be decremented so that it eventually becomes less than 0.

Exercises

7.9 Identify and correct the *errors* in each of the following segments of code. [*Note:* There may be more than one error in each piece of code; unless declarations are present, assume all variables are properly declared and initialized.]

a.

```
if ( age >= 65 );
  document.writeln( "Age greater than or equal to 65" );
else
  document.writeln( "Age is less than 65" );
```

b.

```
var x = 1, total;
while ( x <= 10 )
{
  total += x;
  ++x;
}
```

c.

```
var x = 1;
var total = 0;
While ( x <= 100 )
  total += x;
  ++x;
```

d.

```
var y = 5;
while ( y > 0 )
{
  document.writeln( y );
  ++y;
```

7.10 Without running it, determine what the following script prints:

```
<!DOCTYPE html>

<!-- Exercise 7.10: ex08_10.html -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Mystery Script</title>
    <script>

       var y;
       var x = 1;
       var total = 0;

       while ( x <= 10 )
       {
          y = x * x;
          document.writeln( "<p>" + y + "</p>" );
          total += y;
          ++x;
       } // end while

       document.writeln( "<p>Total is " + total + "</p>" );

     </script>
   </head><body></body>
</html>
```

For Exercises 7.11–7.14, perform each of the following steps:

a. Read the problem statement.

b. Formulate the algorithm using pseudocode and top-down, stepwise refinement.

c. Define the algorithm in JavaScript.

d. Test, debug and execute the JavaScript.

e. Process three complete sets of data.

7.11 Drivers are concerned with the mileage obtained by their automobiles. One driver has kept track of several tankfuls of gasoline by recording the number of miles driven and the number of gallons used for each tankful. Develop a script that will take as input the miles driven and gallons used (both as integers) for each tankful. The script should calculate and output HTML5 text that displays the number of miles per gallon obtained for each tankful and prints the combined number of miles per gallon obtained for all tankfuls up to this point. Use `prompt` dialogs to obtain the data from the user.

7.12 Develop a script that will determine whether a department-store customer has exceeded the credit limit on a charge account. For each customer, the following facts are available:

a. Account number

b. Balance at the beginning of the month

c. Total of all items charged by this customer this month

d. Total of all credits applied to this customer's account this month

e. Allowed credit limit

The script should input each of these facts from a `prompt` dialog as an integer, calculate the new balance (= *beginning balance* + *charges* – *credits*), display the new balance and determine whether the new balance exceeds the customer's credit limit. For customers whose credit limit is exceeded, the script should output HTML5 text that displays the message "Credit limit exceeded."

7.13 A large company pays its salespeople on a commission basis. The salespeople receive $200 per week, plus 9 percent of their gross sales for that week. For example, a salesperson who sells $5000 worth of merchandise in a week receives $200 plus 9 percent of $5000, or a total of $650. You have been supplied with a list of the items sold by each salesperson. The values of these items are as follows:

Item	Value
1	239.99
2	129.75
3	99.95
4	350.89

Develop a script that inputs one salesperson's items sold for last week, calculates the salesperson's earnings and outputs HTML5 text that displays the salesperson's earnings.

7.14 Develop a script that will determine the gross pay for each of three employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time and a half" for all hours worked in excess of 40 hours. You're given a list of the employees of the company, the number of hours each employee worked last week and the hourly rate of each employee. Your script should input this information for each employee, determine the employee's gross pay and output HTML5 text that displays the employee's gross pay. Use `prompt` dialogs to input the data.

7.15 The process of finding the *largest* value (i.e., the maximum of a group of values) is used frequently in computer applications. For example, a script that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode algorithm and then a script that inputs a series of 10 single-digit numbers as characters, determines the largest of the numbers and outputs a message that displays the largest number. Your script should use three variables as follows:

a. `counter` : A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed);

b. `number` : The current digit input to the script;

   c. `largest` : The largest number found so far.

7.16 Write a script that uses looping to print the following table of values. Output the results in an HTML5 table. Use CSS to center the data in each column.

7.17 Using an approach similar to that in Exercise 7.15, find the *two* largest values among the 10 digits entered. [*Note*: You may input each number only once.]

7.18 Without running it, determine what the following script prints:

```
<!DOCTYPE html>

<!-- Exercise 7.18: ex08_18.html -->
<html>
  <head>
    <meta charset = "utf-8">
    <title>Mystery Script</title>
    <script>

       var row = 10;
       var column;

       while ( row >= 1 )
```

```
        {
           column = 1;
           document.writeln( "<p>" );

           while ( column <= 10 )
           {
              document.write( row % 2 == 1 ? "<" : ">" );
              ++column;
           } // end while

           --row;
           document.writeln( "</p>" );
        } // end while

      </script>
   </head><body></body>
</html>
```

7.19 ***(Dangling-Else Problem)*** Determine the output for each of the given segments of code when `x` is `9` and `y` is `11`, and when `x` is `11` and `y` is `9`. Note that the interpreter ignores the indentation in a script. Also, the JavaScript interpreter always associates an `else` with the previous `if`, unless told to do otherwise by the placement of braces (`{}`). You may not be sure at first glance which `if` an `else` matches. This situation is referred to as the “dangling-else” problem. We’ve eliminated the indentation from the given code to make the problem more challenging. [*Hint:* Apply the indentation conventions you have learned.]

a.

```
if ( x < 10 )
if ( y > 10 )
document.writeln( "<p>*****</p>" );
else
document.writeln( "<p>#####</p>" );
document.writeln( "<p>$$$$$</p>" );
```

b.

```
if ( x < 10 )
{
if ( y > 10 )
document.writeln( "<p>*****</p>" );
}
else
{
document.writeln( "<p>#####</p>" );
document.writeln( "<p>$$$$$</p>" );
}
```

7.20 A palindrome is a number or a text phrase that reads the same backward and forward. For example, each of the following five-digit integers is a palindrome: 12321, 55555, 45554 and 11611. Write a script that reads in a five-digit integer and determines whether it's a palindrome. If the number is not five digits long, display an `alert` dialog indicating the problem to the user. Allow the user to enter a new value after dismissing the `alert` dialog. [*Hint:* It's possible to do this exercise with the techniques learned in this chapter. You'll need to use both division and remainder operations to "pick off" each digit.]

7.21 Write a script that outputs HTML5 text that keeps displaying in the browser window the multiples of the integer 2—namely, 2, 4, 8, 16, 32, 64, etc. Your loop should *not terminate* (i.e., you should create an *infinite loop*). What happens when you run this script?

7.22 A company wants to transmit data over the telephone, but it's concerned that its phones may be tapped. All of its data is transmitted as four-digit integers. It has asked you to write a script that will *encrypt* its data so that the data may be transmitted more securely. Your script should read a four-digit integer entered by the user in a `prompt` dialog and encrypt it as follows: Replace each digit by *(the sum of that digit plus 7) modulus 10*. Then swap the first digit with the third, and swap the second digit with the fourth. Then output HTML5 text that displays the encrypted integer.

7.23 Write a script that inputs an encrypted four-digit integer (from Exercise 7.22) and *decrypts* it to form the original number.