# Lecture 8 JavaScript: Control Statements, Part 1

Instructor: Sudad H. Abed
Desgined By: Dabin Ding
UCM



## 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.





# 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
  - the **actions** to be executed, and
  - the **order** in which the actions are to be executed

  is called an **algorithm**

# 7.3 Pseudocode

- Pseudocode
  - An informal language that helps you develop algorithms
  - Pseudocode is similar to everyday English; it's convenient and user friendly, although it's not an actual computer programming language

```
2. 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
Set the class average to the total divided by ten
Print the class average.
```



**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

- Sequential execution
  - Execute statements in the order they appear in the code
- Transfer of control
  - Changing the order in which statements execute
- All scripts can be written in terms of only three control statements
  - sequence
  - selection
  - repetition



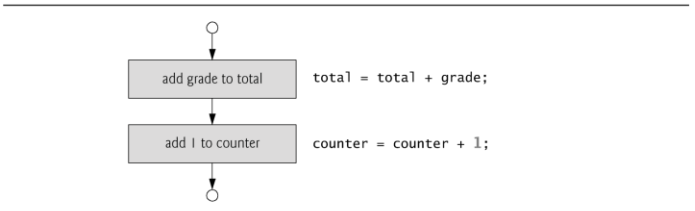



**Fig. 7.1** | Flowcharting JavaScript's sequence structure.

# 7.4 Control Statements (Cont.)

- Flowchart
  - A graphical representation of an algorithm or of a portion of an algorithm
  - Drawn using certain special-purpose symbols, such as rectangles, diamonds, ovals and small circles
  - Symbols are connected by arrows called **flowlines**, which indicate the order in which the actions of the algorithm execute



# 7.4 Control Statements (Cont.)

- In a flowchart that represents a *complete* algorithm, **oval symbols** containing the words "Begin" and "End" represent the start and the end of the algorithm.
- In a flowchart that shows only a portion of an algorithm, 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.

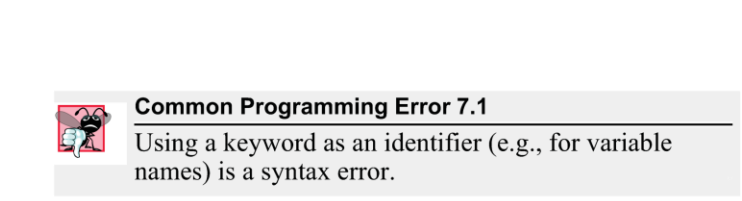
# 7.4 Control Statements (Cont.)

- JavaScript provides three selection structures.
  - The `if` statement either performs (selects) an action if a condition is true or skips the action if the condition is false.
    - Called a single-selection statement because it selects or ignores a single action or group of actions.
  - The `if…else` statement performs an action if a condition is true and performs a different action if the condition is false.
    - Double-selection statement because it selects between two different actions or group of actions.
  - The `switch` statement performs one of many different actions, depending on the value of an expression.
    - Multiple-selection statement because it selects among many different actions or groups of actions.



# 7.4 Control Statements (Cont.)

- JavaScript provides four repetition statements, namely, `while`, `do…while`, `for` and `for…in`.
- 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.



**Common Programming Error 7.1**

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



**JavaScript reserved keywords**

| | | | | |
|---|---|---|---|---|
| break | case | catch | continue | default |
| delete | do | else | false | finally |
| for | function | if | in | instanceof |
| new | null | return | switch | this |
| throw | true | try | typeof | var |
| void | while | with | | |

*Keywords that are reserved but not used by JavaScript*

| | | | | |
|---|---|---|---|---|
| class | const | enum | export | extends |
| implements | import | interface | let | package |
| private | protected | public | static | super |
| yield | | | | |

**Fig. 7.2** | JavaScript reserved keywords.

# 7.4 Control Statements (Cont.)

- **Single-entry/single-exit control statements** 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.
  - **Control-statement stacking.**
- There is only one other way control statements may be connected
  - **Control-statement nesting**



# 7.5 `if` Selection Statement

- The JavaScript interpreter ignores *white-space characters*
  - blanks, tabs and newlines used for indentation and vertical spacing
- 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`).
- The indentation convention you choose should be carefully applied throughout your scripts
  - It is difficult to read scripts that do not use uniform spacing conventions

**Good Programming Practice 7.1**

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



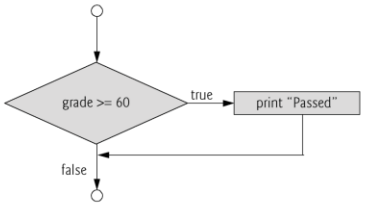



**Fig. 7.3** | Flowcharting the single-selection `if` statement.

**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`.



# 7.6 `if...else` Selection Statement

- Allows you to specify that different actions should be performed when the condition is true and when the condition is false.

**Good Programming Practice 7.2**

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



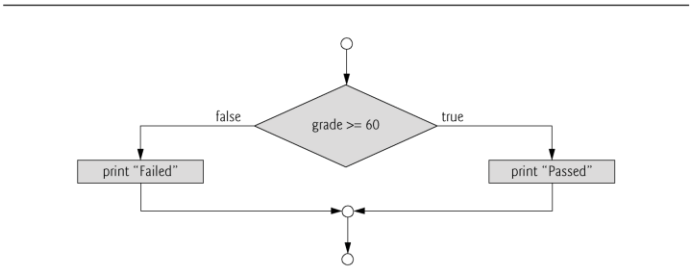



**Fig. 7.4** | Flowcharting the double-selection `if...else` statement.

## 7.6 `if...else` Selection Statement (Cont.)

- Conditional operator (`?:`)
  - Closely related to the `if…else` statement
  - JavaScript's only ternary operator—it takes three operands
  - The operands together with the `?:` operator form a conditional expression
  - The first operand is a boolean expression
  - The second is the value for the conditional expression if the boolean expression evaluates to `true`
  - Third is the value for the conditional expression if the boolean expression evaluates to `false`



## 7.6 `if...else` Selection Statement (Cont.)

- Nested `if…else` statements
  - Test for multiple cases by placing `if…else` statements inside other `if…else` statements
- 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, enclose the statements in braces (`{` and `}`)
  - A set of statements contained within a pair of braces is called a block

**Good Programming Practice 7.3**

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





**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.



## 7.6 `if...else` Selection Statement (Cont.)

- A **logic error** 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 the script 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

- `while`
  - Allows you to specify that an action is to be repeated while some condition remains true
  - The body of a loop may be a single statement or a block
  - Eventually, the condition becomes false and repetition terminates

**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 . Many browsers show a dialog allowing the user to terminate a script that contains an infinite loop.



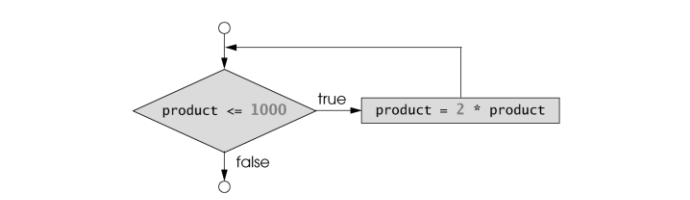



**Fig. 7.5** | Flowcharting the `while` repetition statement.

# 7.8 Formulating Algorithms: Counter-Controlled Repetition

- Counter-controlled repetition
  - Often called definite repetition, because the number of repetitions is known before the loop begins executing
- A *total* is a variable in which a script accumulates the sum of a series of values
  - Variables that store totals should normally be initialized to zero before they are used in a script
- A *counter* is a variable a script uses to count—typically in a repetition statement



```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Input the next grade
6      Add the grade into the total
7      Add one to the grade counter
8
9  Set the class average to the total divided by ten
10 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

```

**Fig. 7.7** | Counter-controlled repetition to calculate a class average. (Part 1 of 4.)



```
         // 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

```

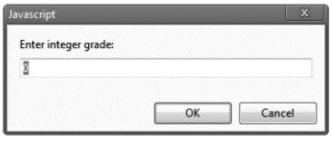
**Fig. 7.7** | Counter-controlled repetition to calculate a class average. (Part 2 of 4.)

```
38          // termination phase
39          average = total / 10;   // calculate the average
40
41          // display average of exam grades
42          document.writeln(
43             "<h1>Class average is " + average + "</h1>" );
44
45       </script>
46    </head><body></body>
47 </html>
```

**Fig. 7.7** | Counter-controlled repetition to calculate a class average. (Part 3 of 4.)



a) This dialog is displayed 10 times. User input is 100, 88, 93, 55, 68, 77, 83, 95, 73 and 62. User enters each grade and presses **OK**.




b) The class average is displayed in a web page


**Fig. 7.7** | Counter-controlled repetition to calculate a class average. (Part 4 of 4.)

**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").



## 7.8 Formulating Algorithms: Counter-Controlled Repetition

- JavaScript represents all numbers as floating-point numbers in memory
- Floating-point numbers 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

**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`.



## 7.9 Formulating Algorithms: Sentinel-Controlled Repetition

- Sentinel-controlled repetition
  - Special value called a sentinel value (also called a signal value, a dummy value or a flag value) indicates the end of data entry
  - Often is called indefinite repetition, because the number of repetitions is not known in advance
- Choose a sentinel value that cannot be confused with an acceptable input value

# 7.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Top-down, stepwise refinement
  - A technique that is essential to the development of well-structured algorithms
  - Approach begins with pseudocode of the top, the statement that conveys the script's overall purpose
  - Divide the top into a series of smaller tasks and list them in the order in which they need to be performed—the first refinement
  - Second refinement commits to specific variables



**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.

**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.

```
1   Initialize total to zero
2   Initialize gradeCounter to zero
3
4   Input the first grade (possibly the sentinel)
5
6   While the user has not as yet entered the sentinel
7       Add this grade into the running total
8       Add one to the grade counter
9       Input the next grade (possibly the sentinel)
10
11  If the counter is not equal to zero
12      Set the average to the total divided by the counter
13      Print the average
14  Else
15      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.

# 7.9 Formulating Algorithms: Sentinel-Controlled Repetition (Cont.)

- Control statements may be stacked on top of one another in sequence



```
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
11          var total; // sum of grades
12          var gradeCounter; // number of grades entered
13          var grade; // grade typed by user (as a string)
14          var gradeValue; // grade value (converted to integer)
15          var average; // average of all grades
16
17          // initialization phase
18          total = 0; // clear total
19          gradeCounter = 0; // prepare to loop
20
```

**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average. (Part 1 of 4.)

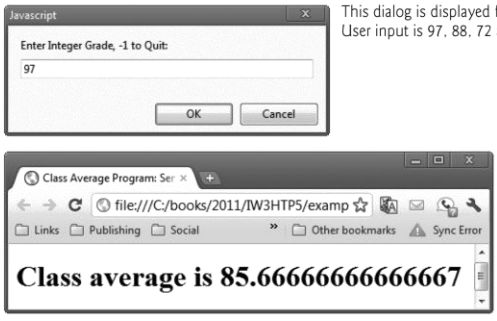
```
21          // processing phase
22          // prompt for input and read grade from user
23          grade = window.prompt(
24               "Enter Integer Grade, -1 to Quit:", "0" );
25
26          // convert grade from a string to an integer
27          gradeValue = parseInt( grade );
28
29          while ( gradeValue != -1 )
30          {
31             // add gradeValue to total
32             total = total + gradeValue;
33
34             // add 1 to gradeCounter
35             gradeCounter = gradeCounter + 1;
36
37             // prompt for input and read grade from user
38             grade = window.prompt(
39                  "Enter Integer Grade, -1 to Quit:", "0" );
40
41             // convert grade from a string to an integer
42             gradeValue = parseInt( grade );
43          } // end while
```

**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average. (Part 2 of 4.)



```
44
45          // termination phase
46          if ( gradeCounter != 0 )
47          {
48             average = total / gradeCounter;
49
50             // display average of exam grades
51             document.writeln(
52                "<h1>Class average is " + average + "</h1>" );
53          } // end if
54          else
55             document.writeln( "<p>No grades were entered</p>" );
56
57       </script>
58    </head><body></body>
59 </html>
```

**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average. (Part 3 of 4.)




This dialog is displayed four times. User input is 97, 88, 72 and –1.

**Fig. 7.9** | Sentinel-controlled repetition to calculate a class average. (Part 4 of 4.)



## 7.10 Formulating Algorithms: Nested Control Statements

- Control structures may be nested inside of one another

```
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.



```
<!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

```

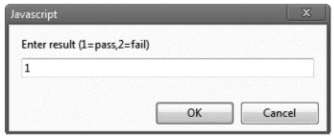
**Fig. 7.11** | Examination-results calculation. (Part 1 of 4.)

```
17          // process 10 students; counter-controlled loop
18          while ( student <= 10 )
19          {
20             result = window.prompt( "Enter result (1=pass,2=fail)", "0" );
21
22             if ( result == "1" )
23                passes = passes + 1;
24             else
25                failures = failures + 1;
26
27             student = student + 1;
28          } // end while
29
30          // termination phase
31          document.writeln( "<h1>Examination Results</h1>" );
32          document.writeln( "<p>Passed: " + passes +
33             "; Failed: " + failures + "</p>" );
34
35          if ( passes > 8 )
36             document.writeln( "<p>Bonus to instructor!</p>" );
37
38       </script>
39    </head><body></body>
40 </html>
```

**Fig. 7.11** | Examination-results calculation. (Part 2 of 4.)



a) This dialog is displayed 10 times. User input is 1, 2, 1, 1, 1, 1, 1, 1, 1 and 1.




b) Nine students passed and one failed, therefore "Bonus to instructor!" is printed.

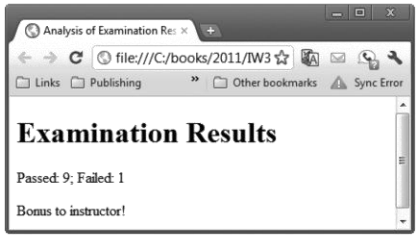
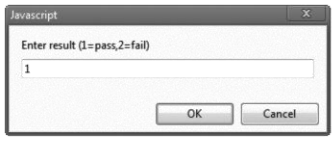



**Fig. 7.11** | Examination-results calculation. (Part 3 of 4.)

c) This dialog is displayed 10 times. User input is 1, 2, 1, 2, 2, 1, 2, 2, 1 and 1.




d) Five students passed and five failed, so no bonus is paid to the instructor.

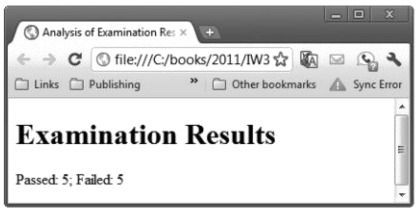



**Fig. 7.11** | Examination-results calculation. (Part 4 of 4.)



**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 the arithmetic assignment operators +=, -=, *=, /= and %=, which abbreviate certain common types of expressions.



| Assignment operator | Initial value of variable | Sample expression | Explanation | Assigns |
|---|---|---|---|---|
| += | c = 3 | c += 7 | c = c + 7 | 10 to c |
| -= | d = 5 | d -= 4 | d = d - 4 | 1 to d |
| *= | e = 4 | e *= 5 | e = e * 5 | 20 to e |
| /= | f = 6 | f /= 3 | f = f / 3 | 2 to f |
| %= | g = 12 | g %= 9 | g = g % 9 | 3 to g |

**Fig. 7.12** | Arithmetic assignment operators.

# 7.12 Increment and Decrement Operators

- The increment operator, ++, and the decrement operator, --, 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.



| Operator | Example | Called | Explanation |
|---|---|---|---|
| ++ | ++a | preincrement | Increment a by 1, then use the new value of a in the expression in which a resides. |
| ++ | a++ | postincrement | Use the current value of a in the expression in which a resides, then increment a by 1. |
| -- | --b | predecrement | Decrement b by 1, then use the new value of b in the expression in which b resides. |
| -- | b-- | postdecrement | Use the current value of b in the expression in which b resides, then decrement b by 1. |

**Fig. 7.13** | Increment and decrement operators.

```
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
11          var c;
12
13          c = 5;
14          document.writeln( "<h3>Postincrementing</h3>" );
15          document.writeln( "<p>" + c ); // prints 5
16          // prints 5 then increments
17          document.writeln( " " + c++ );
18          document.writeln( " " + c + "</p>" ); // prints 6
19
```

**Fig. 7.14** | Preincrementing and postincrementing. (Part 1 of 2.)



```
20          c = 5;
21          document.writeln( "<h3>Preincrementing</h3>" );
22          document.writeln( "<p>" + c ); // prints 5
23          // increments then prints 6
24          document.writeln( " " + ++c );
25          document.writeln( " " + c + "</p>" ); // prints 6
26
27       </script>
28    </head><body></body>
29 </html>
```

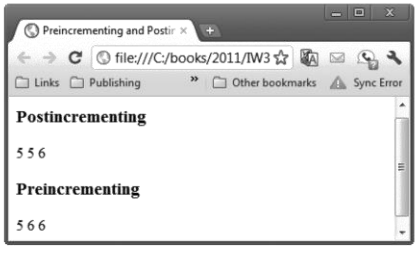



**Fig. 7.14** | Preincrementing and postincrementing. (Part 2 of 2.)

**Good Programming Practice 7.5**

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



## 7.12 Increment and Decrement Operators (Cont.)

- 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
- When a variable appears in the context of a larger expression, preincrementing the variable and postincrementing 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.



| Operator | Associativity | Type |
|---|---|---|
| `++ --` | right to left | unary |
| `* / %` | left to right | multiplicative |
| `+ -` | left to right | additive |
| `< <= > >=` | left to right | relational |
| `== != === !===` | left to right | equality |
| `?:` | right to left | conditional |
| `= += -= *= /= %=` | right to left | assignment |

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

# In Class Exercise

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

| N | 10*N | 100*N | 1000*N |
|---|---|---|---|
| 1 | 10 | 100 | 1000 |
| 2 | 20 | 200 | 2000 |
| 3 | 30 | 300 | 3000 |
| 4 | 40 | 400 | 4000 |
| 5 | 50 | 500 | 5000 |
| 6 | 60 | 600 | 6000 |

- 7.22