chapter four

Expressions and Statements

Sure as

one plus one

is two . . .

—Mac Davis

In Chapter 3, we talked about expression evaluation and its close relationship to data values and variables. Most JavaScript statements contain an expression of some kind that requires evaluation. At the heart of these expressions are operators and function calls.

In this chapter, we'll

- Define the terms "expression" and "statement."
- Look at the various types of operators you can use in expressions, including
  - string operators.
  - arithmetic operators.
  - assignment operators.
  - comparison operators.
  - logical operators.
  - the conditional operator.
- Introduce functions.
- Learn how to declare a function with and without parameters.
- Learn how to return a value from a function.

- Study and apply the established order of operations.

Let's get started by taking a closer look at expressions versus statements.

What Is an Expression?

Although we should, by now, have a good feeling for what an expression is, we have yet to define it. An **expression** is any valid set of literals, variables, operators, function calls, and expressions that evaluates to a single value. The resulting single value can be a number, a string, a Boolean, or a special value (null, undefined, Infinity, or NaN); that is, the result of any expression is always one of JavaScript's defined data types or special values. Here are some examples:

```
3 + 7
3 + 7 + 10 + ""
"Dr." + " " + "Pepper"
```

The first expression adds the numbers `3` and `7` and *evaluates to* `10`.

The second expression adds the numbers `3`, `7`, and `10` and an empty string and evaluates to `"20"`, which is a string.

The last expression adds three strings together and evaluates to `"Dr. Pepper"`.

What Is a Statement?

While an expression is any valid set of literals, variables, operators, function calls, and expressions that *evaluates* to a single value, a **statement**, on the other hand, is any set of declarations, method calls, function calls, and expressions that performs some *action*. The possible results of a JavaScript statement are infinite. Here are some examples:

```
var num = 1
document.write("hello")
```

The first statement above declares a variable named `num` and initializes it to the value of 1.

The second statement performs the action of writing "hello" to the document.

Expressions vs. Statements

Statements often contain expressions that have to be evaluated before the specified action can be performed. For instance:

```
document.write("Sum: ", 3 + 7, "<br>")
```

The first *statement* above has to evaluate the *expression* "3 + 7" before it can convert it to a string and write the result between the strings "Sum: " and "
". The statement performs several actions:

- It evaluates the expression "3 + 7."
- It wrote the string "Sum : ".

- It converted the number 10 to a string and wrote that (10 was the result of evaluating the expression "3 + 7").
- Finally, it wrote the string "
".

Here's another example of a statement that has to evaluate an expression before it can perform its intended action:

```
total = 1 + 2
```

This statement first had to evaluate the expression "1 + 2" before it could assign it to `total`.

JavaScript has the following *types* of expressions. Notice that they correspond to the three primitive data types supported by JavaScript that we discussed in Chapter 3.

- Number—evaluates to a number, for example, `15`, `7.57`, or `-3.145`
- String—evaluates to a character string, for example, `"Jane"`, `"Hello"`, or `"9455"`
- Boolean—evaluates to `true` or `false`

This makes sense when we consider that every expression evaluates to a single value and that all JavaScript values can be classified as one of the three primitive data types or one of the special values null, undefined, Infinity, or NaN.

What Is an Operator?

Operators are the workers in expressions. They come primarily in two flavors: unary and binary. A unary operator performs work, or operates, on one operand, whereas a binary operator operates on two operands. Table 4.1 provides some examples.

Operator Flavor	Syntax	Examples
unary	*operand operator* or *operator operand*	-88 count++ !flag
binary	*operand operator operand*	7 + 8 num1 < num2

Table 4.1 Operator Flavors

Types of Operators

The JavaScript language supports many operators. They are easily organized into five categories:

- **String operators**—those operators that work on strings. There are only two.
- **Arithmetic operators, also known as mathematical operators**—those operators that perform mathematical computations.

- **Assignment operators**—those operators that assign a value to a variable, object, or property.
- **Comparison operators**—those operators that compare two values or expressions and return a Boolean value indicating the truth of the comparison.
- **Logical operators, also known as Boolean operators**—those operators that take Boolean values as operands and return a Boolean value indicating the truth of the relationship.

In addition to these five categories of operators, JavaScript supports one special operator, the conditional operator. The conditional operator is the only operator that has three operands. We'll discuss it in more detail later. For now, let's start with string operators and work through them all from there.

Concatenation, the String Operator

There are really only two string operators: the concatenation operator (+) and the concatenation by value operator (+=). The first concatenates two strings together. Thus, the operation

```
"Greetings, " + "Earthlings"
```

evaluates to the string

```
"Greetings, Earthlings"
```

An equivalent operation using variables is

```
var salutation = "Greetings, "
var recipient = "Earthlings"
salutation + recipient
```

The last statement evaluates to

```
"Greetings, Earthlings"
```

The second string operator, concatenation by value (+=), concatenates the string on the right side (or the value of a string variable) to the string value stored in the variable on the left side, then assigns the result back to the left operand variable. Here's an example:

```
var greeting = "Greetings, "
greeting += "Earthlings"
```

The first statement declares a variable named greeting and initializes it to the string "Greetings, ".

The second statement concatenates "Earthlings" onto the value contained in greeting. So the variable greeting now evaluates to

```
"Greetings, Earthlings"
```

Cool, huh?

peek ahead

While there are only two string operators, there are many string methods. We'll look at string methods in detail in the next chapter: Chapter 5.

A common use of this operator is to pile a bunch of HTML statements into a single string for easy writing to a pop-up window. While we're not ready to tackle pop-up windows just yet, we can see how easy it is to cram a bunch of HTML into one tiny little variable.

```
<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>Script 4.1: Using Concatenation By Value</title>
<script language="JavaScript" type="text/javascript"><!--
    var docContents = ""
//-->
</script>
</head>
<body>
<script language="JavaScript" type="text/javascript"><!--
    // now you generate the custom content, maybe from values of form
    // fields or other operations performed on the page
    docContents += "<h1>Dynamically generated page content.</h1>"
    docContents += "<p>More dynamically generated page content. "
    docContents += "Still more dynamically generated page content.</p>"
    docContents += "<p>Yet more dynamically generated page content.</p>"

    // here you would create a new window,
    // you'll learn how in Chapter 11

    document.write(docContents)
//-->
</script>
</body>
</html>
```

Script 4.1 Using Concatenation by Value

Run the code and check out the results:

Figure 4.1 Results of Script 4.1

JavaScript supports only the two string operators just discussed. Attempts to perform mathematical operations (other than + and +=, which are technically string operators when they're used with strings) on strings that cannot be converted to numbers result in NaN. Let's examine arithmetic operators now.

peek ahead You'll learn how to create pop-up windows and write to them dynamically in Chapter 11.

Arithmetic (Mathematical) Operators

Arithmetic operators, that is, operators that perform mathematical operations, should be quite familiar to you. After all, you've been using them since grade school. Because they're so self-explanatory, rather than describe each one in detail, here's a handy chart that describes each operator complete with an example and its name.

Note that all arithmetic operators work on numbers and result in a number. Division by zero results in the numeric value `Infinity`. In some older browsers, division by zero may result in `undefined` or `NaN`; it depends on the browser.

Operator	Name	What It Does	Flavor	Example	Result
+	plus	Adds the two operands	binary	7 + 5	12
-	minus	Subtracts the right operand from the left operand	binary	7 - 5	2
*	multiply	Multiplies the two operands	binary	7 * 5	35
/	divide	Divides the left operand by the right operand and returns the quotient	binary	7/5	1.4
%	modulus (remainder)	Divides the left operand by the right operand and returns the remainder	binary	7%5	2
-	negation	Negates the operand	unary	-7	-7
++	increment	Adds 1 to the operand	unary	*assume x=7* ++x x++	8 (before assignment) 7 (before assignment) 8 (after assignment)

				assume x=7	
--	decrement	Subtracts 1 from the operand	unary	--x	6 (before assignment)
				x--	7 (before assignment) 6 (after assignment)

Table 4.2 Arithmetic Operators

OK, so maybe a few of them do need a little explanation. For instance, what is this modulus operator? I don't remember that one from grade school.

The modulus operator is a remainder operator, that is, it performs division, throws the quotient away, and keeps the remainder. Thus,

```
5 % 3
```

evaluates to

```
2
```

because 3 goes into 5 one time with a remainder of 2.

Let's try another:

```
9 % 3
```

evaluates to

```
0
```

because 9 divided by 3 equals 3 with a remainder of 0.

One more:

```
4.5 % 2
```

evaluates to

```
.5
```

because 2 goes into 4.5 two times with a remainder of .5.

See how that works? It takes a little braintwisting, but not too much.

The increment and decrement operators also require special mention. While incrementing and decrementing are simple in themselves—incrementing adds 1, decrementing subtracts 1—these operators can work pre-evaluation or post-evaluation when used in conjunction with an assignment operator.

Either way, pre- or post-evaluation, the operation is performed, that is, the operand is incremented or decremented. However, if an assignment is involved, the value actually assigned will vary according to which side of the operand the increment or decrement operator is placed. To explain this, let's start with a few variables initialized to zero:

```
var countA = 0
var countB = 0
var num1 = 0
var num2 = 0
```

Now let's use a pre-increment operator. You can tell it's a pre-evaluation operator because the operator precedes the operand.

```
num1 = ++countA
```

Because this statement used a pre-evaluation increment operator, countA is incremented *before* the assignment. So `num1` evaluates to 1:

```
document.write("num1: ", num1, "<br>")
```

Now let's try a post-evaluation increment operator. Post-evaluation operators are placed after the operand. They don't execute until after the rest of the statement has executed, just before the next line of code.

```
num2 = countB++
```

Because this statement used a post-evaluation increment operator, `countB` doesn't get incremented until after the assignment is made to `num2`. So `num2` evaluates to 0, but `countB` holds the value of 1 after the statement is executed:

```
document.write("num2: ", num2, "<br>")
document.write("countB: ", countB, "<br>")
```

If this seems a bit confusing, don't worry. Most of the time, it isn't an issue. Only when an assignment is involved do you need to look closely at it and determine what it is that you want to accomplish. In fact, you may want to avoid using increment and decrement operators in this fashion altogether just because it *is* confusing. It's safer to use them alone, on a line by themselves, to either increment or decrement a variable; *then* perform your assignment on the *next line*. This makes your code easier to read and eliminates the guesswork.

After you've learned more about assignments and assignment operators, reread this section; it may all seem clearer then.

Assignment Operators

As with arithmetic operators, assignment operators should seem pretty straightforward. In essence, every assignment operation either initializes or changes the contents of the variable listed on the left side of the operator. Remember `myCup` from Chapter 3? Let's update its contents a few times to get a better picture of how assignment operators work. For instance, the statement

```
myCup = "lemonade"
```

changes the contents of `myCup` so that `myCup` now holds lemonade. If `myCup` had not yet been assigned a value, the above statement would initialize `myCup` with "lemonade"; otherwise, it replaces `myCup`'s contents with lemonade. Because `myCup` is a variable, we can also say that `myCup` *evaluates* to lemonade. Remember, a variable always *evaluates* to its contents.

The statement

```
myCup += " tea"
```

works a little differently. This statement adds tea to `myCup` of lemonade so that `myCup` now holds both lemonade and tea—an excellent combination, especially on a hot summer day.

Of course we can, at any time, completely replace the contents of `myCup` with something else by making a new assignment:

```
myCup = "ice water"
```

This statement replaces the "lemonade tea" in `myCup` with "ice water".

While we've been using strings to illustrate how assignment operators work, most of the assignment operators listed here, all but equals (=) and add by value (+=), work only on numbers; += also works on strings and = works on any data type. Here's a list of JavaScript's assignment operators:

Operator	Name	Examples	Is Equivalent To	Means	Applies To
=	equals	x = y x = 7		x gets the value of y x gets the value of 7	any data type
+=	add by value	x += y x += 5	x = x + y x = x + 5	x gets the value of x + y x gets the value of x + 5	numbers and strings
-=	subtract by value	x -= y x -= 7	x = x – y x = x -7	x gets the value of x – y x gets the value of x - 7	numbers only
*=	multiply by value	x *= y x *= 5	x = x * y x = x * 5	x gets the value of x * y x gets the value of x * 5	numbers only
/=	divide by value	x /= y x /= 7	x = x/y x = x/7	x gets the value of x/y x gets the value of x/7	numbers only
%=	modulus by value	x %= y x %= 5	x = x % y x = x % 5	x gets the value of x % y x gets the value of x % 5	numbers only

Table 4.3 Assignment Operators

Comparison Operators

Comparison operators should also look familiar to you: like arithmetic operators, you've been using them since grade school. Comparison operators allow you to compare two values or two expressions of any data type. Usually, the two items being compared are of the same data type, so we're comparing apples to apples. It doesn't make much sense to compare apples to oranges. The result of a comparison is always a Boolean truth value: true or false.

Here's a list of JavaScript comparison operators:

Operator	Name	Description	Example (assume: x=7, y=5)	Example Result
==	is equal to	Returns true if the operands are equal	x == y	false
!=	is not equal to	Returns true if the operands are not equal	x != y	true
>	is greater than	Returns true if the left operand is greater than the right operand	x > y	true
>=	is greater than or equal to	Returns true if the left operand is greater than or equal to the right operand	x >= y	true
<	is less than	Returns true if the left operand is less than the right operand	x < y	false
<=	is less than or equal to	Returns true if the left operand is less than or equal to the right operand	x <= y	false
===	is equivalent to	Returns true if the operands are equal and of the same type	x === y	false
!==	is not equivalent to	Returns true if the operands are not equal and/or not of the same type	x !== y	true

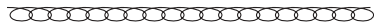
Table 4.4 Comparison Operators

JavaScript often performs conversions for you when you do comparisons of strings and numbers. For instance, should you compare the following:

```
5 == "5"
```

the result is true. Why? After all, the left operand is a number and the right a string. In this and all comparisons except "is equivalent to" (===) and "is not equivalent to" (!==), JavaScript assumes you are trying to compare similar data types and performs the conversions for you. In this example, JavaScript converted the string to a number in order to perform a meaningful comparison.

Comparison operators are often used in conjunction with control structures to direct the flow of a program.



peek ahead

Control structures will be covered in detail in Chapter 6.

Logical (Boolean) Operators

Logical operations, also known as Boolean operations, always result in a truth value: true or false. Perhaps the best way to illustrate logical operators is to use some English statements and let you evaluate whether each statement is true or false.

The && (AND) Operator

Let's first look at the && operator. In order for an AND (&&) statement to be true, both sides of the statement (both operands) must be true. With that in mind, consider this statement. Is it true or false?

```
You are currently reading Chapter 4 && this book is about
Visual Basic.
```

It's false. Why? Because while you are indeed reading Chapter 4, this is *not* a book about Visual Basic. The second part of the statement, the right operand, is false, making the AND (&&) statement false.

Let's try another:

```
The subject of this book is JavaScript && the author is Tina
Spain McDuffie.
```

The result:

```
true
```

Both sides of the AND (&&) statement are true, thus the statement is true.

Now let's take a look at the OR (||) operator.

The || (OR) Operator

In an OR situation, only one side needs to be true in order for the statement to evaluate to true. If both sides are true, the statement is still true. Only if both sides are false will an OR operation evaluate to false. Here's an example:

```
This book is about JavaScript || this book is about Visual
Basic
```

The result:

```
true
```

The first part was true; so even though the second part was false, the overall statement is true. Had we used the AND (&&) operator, the result would have been false.

The ! (NOT) Operator

Last but not least, we have the NOT (!) operator. For example:

```
This book is !Visual Basic.
```

results in

```
true
```

This book is not about Visual Basic, it's about JavaScript, so the statement is true.

The NOT operator is often used to see if the value of a variable is false or to determine if an object doesn't exist. We'll see several examples of these uses throughout the book, after we've covered control structures.

Following is a table listing each logical operator, its associated truth table, and an example using that operator:

Operator	Name	Flavor	Truth Table		Example (isJS=true, isC3=false)	Result
&&	AND	binary	**Expression**	**Result**	isJS && isC3	false
			`true && true`	`true`		
			`true && false`	`false`		
			`false && true`	`false`		
			`false && false`	`false`		
||	OR	binary	**Expression**	**Result**	isJS || isC3	true
			`true || true`	`true`		
			`true || false`	`true`		
			`false || true`	`true`		
			`false || false`	`false`		
!	NOT	unary	**Expression**	**Result**	!isC3	true
			`!true`	`false`		
			`!false`	`true`		

Table 4.5 Logical Operators

The Conditional Operator

The conditional operator is the only JavaScript operator that takes *three* operands. The result of the operation is one of two values, based upon the evaluation of the *condition*. If the condition is true, the first value is the result; if the condition is false, the second value is the result.

The syntax is

```
(condition) ? ValueIfTrue : ValueIfFalse
```

You can use the conditional operator anywhere you would use a standard operator. For example, try the following:

```
var age = 38
(age >= 18) ? "adult" : "minor"
```

It evaluates to

```
"adult"
```

Because the variable `age` holds the value 38, the condition, age >= 18, is true, so the conditional statement evaluates to the first value listed, which is `"adult"`. If the condition had evaluated to false, then the statement would evaluate to the second value, `"minor"`. For example:

```
var age = 7
(age >= 18) ? "adult" : "minor"
```

results in

```
"minor"
```

Cool, huh?

Special Operators

JavaScript supports several special operators that you should be aware of: `delete`, `new`, `this`, `typeof`, and `void`. Let's look at each of them in turn.

`delete`

The `delete` operator allows you to delete an array entry or an object from memory. You need to know more about arrays and objects before you can see a good example of its use. So we'll postpone an example until Chapter 7 when we talk about arrays.

`new`

Most JavaScript built-in objects like `Image` and `Array` have a corresponding constructor function already built into the language. **Constructor** is an object-oriented programming (OOP) term that refers to a function that specifies how to initialize a particular type of object when it is created. Constructor functions generally cannot be called directly. Instead, you use the special operator, **new**, in conjunction with the constructor function to create a new object and initialize it with its constructor.

peek ahead

You'll use the `new` operator to create images for an automatic slideshow in Chapter 8.

For instance, the following JavaScript statement creates and initializes an `Image` object in memory:

```
var myPic = new Image()
```

Note that, in this case, no parameters were provided to the `Image` constructor function. While many constructor functions require parameters to help define an object, some do not. In the case of the `Image` constructor above, its built-in function definition provides for the case of receiving no parameters. You can also, optionally, pass the image's dimensions, width and height, as parameters, in that order.

this

The `this` operator is totally cool and can save you loads of typing, which is always a popular feature with programmers. The special keyword `this` is a shortcut way of referring to the current object. You need to learn more about event handlers, forms, and other objects before looking at a good working example, so we'll save a demonstration of it until that time. For now, just file it away in your brain as a special operator.

peek ahead

You'll get to use `this` to help perform form calculations and validations in Chapters 12 and 13.

typeof

As we saw in Chapter 3, the `typeof` operator lets you easily determine the current data type of any variable. The result of a `typeof` operation is always a *string* indicating the variable's data type. Thus, possible values are `"number"`, `"string"`, `"boolean"`, `"object"`, and `"undefined"`. See Chapter 3 for a working example of this useful operator.

void

The `void` operator tells the interpreter to evaluate an expression and return no value. That's kind of weird, huh? Why would you want to evaluate an expression and then return no value? Well, actually it does come in handy when you want to make sure your program takes no action whatsoever in response to an event. In fact, we'll see `void` in action in the very next chapter.

peek ahead

You'll learn how to use the `void` operator in conjunction with the `javascript:` pseudo-protocol and an `onClick` event handler in Chapter 5.

For now, let's get back to our original discussion about expressions and statements. Recall that an expression is any valid set of literals, variables, operators, function calls, and expressions that evaluates to a single value. A statement is any set of declarations, method calls, function calls, and expressions that performs some *action*. Notice that twice we referred to function calls. So what's a function?

What Is a Function?

A **function** is a block of predefined programming statements whose execution is deferred until the function is "called." A function is a predefined routine that doesn't execute until you "call" it.

You **call a function** by invoking its name with any required or optional parameters. A **function parameter**, also known as an argument, is a data value or data reference that you can pass to the function to work on or use. Parameters make functions more useful and flexible. For instance, you could define a `greetVisitor` function that would greet visitors to a Web site. Here's an example:

```
function greetVisitor() {
   alert("Hello!")
}
```

To call it, you would invoke the function's name:

```
greetVisitor()
```

That's pretty easy, huh?

Passing Parameters to Functions

Wouldn't the `greetVisitor` function be more useful if you could actually greet the visitor by name? That's where function parameters come in. How about you give the visitor's name, as a parameter, to `greetVisitor` so it can use that data when performing its function, which is to greet the visitor by name. Here's what your revised function might look like:

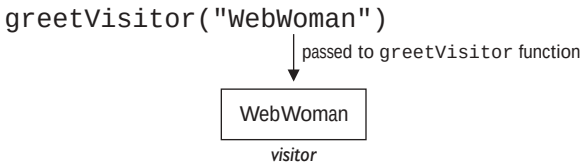
```
function greetVisitor(visitor) {
   alert("Hello, " + visitor + "!")

}
```

To call it, send the visitor's name. You could've acquired the visitor's name in a variety of ways: from a prompt, a form field, or even a cookie stored during a previous visit. In this example, we'll use a simple string.

```
greetVisitor("WebWoman")
```

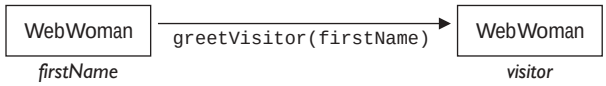
When you call `greetVisitor`, the string "WebWoman" is passed to the variable `visitor`. Here's a picture of what happens:



Were the information stored in the `firstName` variable, the call might look like this:

```
greetVisitor(firstName)
```

Here's a picture:



We'll look more closely at how this works in Chapter 8.

Returning a Value from a Function

Another feature of functions is that they may optionally **return** a value. That is, they can do some processing and spit out (return) a result. This feature makes functions even more useful and flexible. For instance, let's say we need a function to find the area of a rectangle. We'll provide the width and height; we want the function to return the area:

```
function calcRectangleArea(width, height) {
   var area = width * height
   return area
}
```

Here is one way to call it:

```
alert("The area of an 8x5 rectangle is: " + calcRectangleArea(8, 5))
```

The result:

You'll often use functions that return a value in expressions. As you'll see in the next section, the function call usually occurs first, returning some value that is then used in the expression evaluation.

We'll look at functions in much greater detail in Chapter 8. For now, let's move on to the order of operations.

The Order of Operations

When it comes to expression evaluation, it is essential to know which operations have priority, that is, you need to know who gets to go first. For instance, what does the following statement evaluate to?

```
4 + 10/2 * 3 - (1 + 2) * 4
```

If you evaluated the expression left to right, giving each operator equal precedence,

4 + 10 is 14
divided by 2 is 7
times 3 is 21
– 1 is 20
+ 2 is 22
times 4 is 88

Right? Sure, *if* that's how math really worked. But math doesn't work that way. So 88 is the wrong answer. If you have any math background whatsoever, you're already screaming, "No! No! No! That's not how it's done!" And you're right.

Mathematics has an established order of operations:

1. First you work the expressions within parentheses from the inside out.
2. Then you perform any squares, cubes, or other to-the-*n*th-ofs from left to right.
3. Then you multiply and divide from left to right.
4. Finally, you add and subtract from left to right.

Programming languages also have an order of operations. They must. There has to be some established guideline that determines the order of precedence of each programming construct. We saw how important the order of operations was when we worked with the increment and decrement operators. The order of operations directly affected the results of our assignments.

JavaScript's order of operations table is a little longer and more complex than the basic mathematical outline shown above because it also has to factor function calls and such into the equation. Still, the general concept is the same.

The following table describes JavaScript's order of operations. The bitwise operators have been left out to make it simpler, since we won't be covering them in this book.

Order	Description	Operator(s)
1	Parentheses (work from the inside out)	`( )`
2	Member of an object or an array	`. [ ]`
3	Create instance	`new`
4	Function call	`function()`
5	Boolean NOT, negation, positive, increment, decrement, typeof, void, and delete	`! - + ++ -- typeof void delete`
6	Multiplication, division, and modulus	`* / %`
7	Addition, concatenation, and subtraction	`+ -`
8	Relational comparisons	`< <= > >=`

9	Equality, inequality, equivalency, and non-equivalency	`== != === !===`
10	Boolean AND	`&&`
11	Boolean OR	`||`
12	Conditional expression	`?:`
13	Assignment	`= += -= *= /= %=`

Table 4.6 Order of Operations

Some of the descriptions in Table 4.6 refer to stuff we haven't covered yet, like function calls and arrays. Don't worry, all will become clear later. As you complete each new topic, come back and take a look at Table 4.6 again and see how the new topic fits into JavaScript's order of operations.

In the meantime, let's apply the stuff we've learned so far. Let's go back to the expression we looked at earlier and see if we can arrive at the correct answer, which is 7, by applying the correct order of operations step by step.

Original expression:

```
4 + 10/2 * 3 – (1 + 2) * 4
```

Any parentheses? Yes.

```
4 + 10/2 * 3 –    3    * 4   // evaluate parens
```

There are no arrays, objects, instance creations, or function calls (whatever they are), so we can skip 2 thru 4. Any NOTs (!)? No. Any negative or positive signs? No. Any increments or decrements? No.

Next is multiplication, division, and modulus. Any of those? Yes. Let's do them one at a time:

```
4 +   5  * 3 -    3    * 4   // * and / left to right
4 +     15   -       12
```

Next is addition, concatenation, and subtraction. Any of those? There surely are. Let's do those one at a time too:

```
     19       -       12     // + and – left to right
              7
```

Looks like we're done. We didn't even get to comparisons, equality and inequality, Boolean AND, Boolean OR, Boolean NOT, conditional expressions, or assignments.

The order of operations can have a dramatic effect on expressions and statements involving both strings and numbers. Let's explore a few expressions to see how:

```
7 + 5 + "dollars"
```

According to Table 4.6, the addition and concatenation operators have equal precedence. So, we evaluate the expression left to right. The result is

```
12  + "dollars"
  12 dollars          // which is a string
```

Now let's try it the other way:

```
"dollars " + 7 + 5
```

Concatenation and addition have equal precedence, so it's left to right again:

```
"dollars 7" + 5      // add 7 to the string dollars
dollars 75           // we're still adding to a string
```

Because the first operand, `"dollars "`, is a string, the + operator is the concatenation operator. So instead of adding 7, 7 is converted to a string and concatenated to the string `"dollars "`. The same is true for the 5.

Summary

An expression is any valid set of literals, variables, operators, and expressions that evaluates to a single value. A statement is any set of declarations, method calls, function calls, and expressions that performs some action.

The JavaScript language is rich with operators that we can use in expressions, including

- String operators that work on strings.
- Arithmetic operators, used for mathematical operations.
- Assignment operators, used to assign a value to a variable or object property.
- Comparison operators, used to compare two values or expressions.
- Logical operators, used to determine the truth value of a Boolean expression.
- The conditional operator, used to choose one of two values based upon the result of a condition. The condition can be a comparison, a truth value, or the result of a logical operation.

The JavaScript language also supports several special operators:

- `delete`—used to delete an array element or object.
- `new`—used to create new objects with a constructor function call.
- `this`—used to refer to the current object.
- `typeof`—used to determine the data type of a variable, object property, object, or special value.
- `void`—used to prevent an expression from returning a value.

Expressions and statements sometimes include function calls. A function is a deferred set of programming statements whose execution is deferred until the function is called. Functions may optionally accept parameters and/or return a value. Those that return a value are often used in expressions.

Like any other programming language or mathematical system, JavaScript has an established order of operations to determine operator precedence.

Review Questions

1. What is an expression?
2. What type(s) of value can an expression evaluate to? Be specific.
3. What's a statement?
4. What's the difference between an expression and a statement?
5. What's an operator?
6. What string operators does JavaScript support?
7. What type of operators performs mathematical computations?
8. What does the % operator do?
9. What's the difference between the pre-increment operator and the post-increment operator? How can using the wrong one affect your program?
10. Does it always make a difference to the result of your program whether you use a pre- or post-increment or decrement operator?
11. Describe assignment operators. Name three assignment operators.
12. What type of value does a comparison operation result in?
13. What's the difference between = and ==?
14. What the difference between == and ===?
15. List and describe three logical operators.
16. What data type is returned by a logical operation?
17. What operator is the only one that takes three operands?
18. What's a function?
19. How do you call a function?
20. What's a parameter?
21. What does it mean to say that a function can *return* a value?
22. What is the purpose of having an order of operations?
23. Which comes first, multiplication or modulus?
24. Which comes first, a less than or equal to comparison or an inequality comparison?

Exercises

1. Specify what each of the following expressions evaluates to:
   a. 4 + 9
   b. 4 + 10 - 5 + 2
   c. 4 * 2 + 7 - 1
   d. 7 + 4 * 2 - 1
   e. 3 - 2 * 6
   f. 4%2 * 98
   g. 1 + 4%2 * 75
   h. 6 + 25/5
   i. 4 + 5%3 + 7
   j. 2 * 4 * 8 - 6 * 2
   k. 8/4 %2
   l. 5 * “4”
   m. 2 * 4 +”5"
   n. “4” - 2

2. For each of the statements, *in the following sequence*, specify what `myVar` evaluates to after the statement executes in JavaScript.
   a. `var myVar = 5`
   b. `myVar *= 5`
   c. `myVar = myVar % 6`
   d. `myVar += "45"`
   e. `myVar = "Sam"`
   f. `myVar = true`
   g. `myVar = (!myVar && true)`
   h. `myVar = ((5 == 5) || (2 >= 5)) || (!(6 >= 10))`
   i. `myVar = 6 * 2 + 10/5 - 5%2`
   j. `myVar = 6 * ((2 + 10/5) - 5%2)`
3. Specify what each of the following expressions evaluates to. Assume that num1=8, num2=7, num3=-5, string1="8", and string2="7".
   a. num3 ==num2
   b. num1 < 8
   c. num1 <= 8
   d. num2 > 7
   e. num3 < num1
   f. num2 != num3
   g. string1 == num1
   h. string2 === num2
   i. string2 < string1
   j. num1 < string1
4. Specify what each of the following expressions evaluates to. Assume that flag = false, isEmpty = true, validString=false, and validNum = true.
   a. !isEmpty
   b. flag
   c. isEmpty && flag
   d. validNum || validString
   e. validString && !isEmpty
   f. !isEmpty && validNum
5. Specify what values count and newCount hold after each of the following statements, *in order*.
   a. var count = 0
   b. count++
   c. newCount = count++
   d. —count
   e. count %= newCount
6. Write a conditional statement that evaluates to "happy" if `mouthCurve` is less than 180 and "sad" if `mouthCurve` is greater than or equal to 180.

7. Given the following function:

```
function greet(message, visitor) {
      document.write("<h1>", message, ", ", visitor,
                     "</h1>")
}
```

   a. Write the appropriate JavaScript to call the greet function so that it will write "Hullo, Sam."
   b. Write the appropriate JavaScript to call the greet function so that it will write "Greetings and Salutations, Wilbur."
   c. Write the appropriate JavaScript to call the greet function so that it will write "Greetings, Earthlings."

Scripting Exercises

Create a folder named assignment04 to hold the documents you create during this assignment. Save a copy of the personal home page you last modified as home.html in the assignment04 folder.

1. Examine the following script:

```
<script language="JavaScript" type="text/javascript"><!--
      var num1 = prompt("Please enter a number: ", 0)
      var num2 = prompt("Please enter a number: ", 0)
      var sum = num1 + num2

      document.write("You entered: ", num1, " and ",
                     num2, "<br>")
      document.write("Sum: ", sum)
//-->
</script>
```

   Try running it and enter two numbers when prompted. The script does not do what was intended. Fix it so it does.

2. Write a script that prompts the visitor for two numbers, then displays the two numbers and their sum, difference, product, quotient, and modulus in the document. For instance, if the visitor entered 5 and 2, the results would display as follows:

```
You entered: 5 and 2
Sum:  7
Difference: 3
Product:  10
Quotient: 2.5
Modulus: 1
```

3. Write a script that prompts the visitor for two values, then displays the two entries, their data type, and the truth of some comparisons, including is equal to, is not equal to, is equivalent to, is not equivalent to, is less than, is less than or equal to, is greater than, and is greater than or equal to. For instance, if the visitor entered 5 and "2", the results would display as follows:

```
You entered: 5 (number) and 2 (string)
5 == 2? false
5 != 2? true
5 === 2? false
5 !== 2? true
5 < 2? false
5 <= 2? false
5 > 2? true
5 >= 2? true
```

4. Write a script that prompts the visitor for three numbers, then calculates and displays their average.
5. Write a script that prompts the visitor for the number of hours he or she worked that week and his hourly rate. Calculate and display the visitor's expected gross pay for that week. Don't forget that hours over 40 are paid at time and a half. Here's what the display should look like if the visitor entered 35 hours at $10/hour:

```
Total Hours Worked:  35
Regular Pay:  35 hours @ $10/hour = $350
Overtime Pay:  0 hours @ $15/hour = $0
Total Pay: $350
```

6. Create a new document named currency.html.
   a. Look up the current exchange rates for converting U.S. dollars to three different foreign currencies. You can look exchange rates up at http://www.x-rates.com/.
   b. Write a script that will acquire an amount in U.S. dollars from the visitor and display its equivalent in each of the foreign currencies whose exchange rate you looked up.
   c. Your output should look similar to this:

```
$55.78 in U.S. Currency is equivalent to:
35.9870 GBP (British Pounds)
435.084 HKD (Hong Kong Dollars)
542.516 MXN (Mexican Pesos)
```