CHAPTER 2

The Grammar of JavaScript

Learning a programming language is a lot like learning any new language: There are words to learn, punctuation to understand, and a new set of rules to master. And just as you need to learn the grammar of French to speak French, you must become familiar with the grammar of JavaScript to program JavaScript. This chapter covers the concepts that all JavaScript programs rely on.

If you've had any experience with JavaScript programming, many of these concepts may be old hat, so you might just skim this chapter. But if you're new to JavaScript, or you're still not sure about the fundamentals, this chapter introduces you to basic (but crucial) topics.

Statements

A JavaScript *statement* is a basic programming unit, usually representing a single step in a JavaScript program. Think of a statement as a sentence: Just as you string sentences together to create a paragraph (or a chapter, or a book), you combine statements to create a JavaScript program. In the last chapter you saw several examples of statements. For example:

```
alert('Hello World!');
```

This single statement opens an alert window with the message "Hello World!" in it. In many cases, a statement is a single line of code. Each statement ends with a semicolon—it's like a period at the end of a sentence. The semicolon makes it clear that the step is over and that the JavaScript interpreter should move onto the next action.

Note: Officially, putting a semicolon at the end of a statement is optional, and some programmers leave them out to make their code shorter. Don't be one of them. Leaving off the semicolon makes reading your code more difficult and, in some cases, causes JavaScript errors. If you want to make your JavaScript code more compact so that it downloads more quickly, see page 502.

The general process of writing a JavaScript program is to type a statement, enter a semicolon, press Return to create a new, blank line, type another statement, followed by a semicolon, and so on and so on until the program is complete.

Commands

JavaScript and Web browsers let you use various commands to make things happen in your programs and on your Web pages. For example, the *alert( )* command you encountered earlier makes the Web browser open a dialog box and display a message. These commands are usually called *functions* or *methods,* and are like verbs in a sentence. They get things done.

Some commands, like *alert( )* or *document.write( )*, which you encountered on page 29, are specific to Web browsers. In other words, they only work with Web pages, so you won't find them when programming in other environments that use JavaScript (like, for example, when scripting Adobe applications like Acrobat or Dreamweaver or in Flash's JavaScript-based ActionScript).

Other commands are universal to JavaScript and work anywhere JavaScript works. For example, *isNaN( )* is a command that checks to see if a particular value is a number or not—this command comes in handy when you want to check if a visitor has correctly supplied a number for a question that requires a numerical answer (for example, "How many widgets would you like?"). You'll learn about *isNaN( )* and how to use it in Chapter 4 on page 137.

JavaScript has many different commands, which you'll learn about throughout this book. One quick way to identify a command in a program is by the use of parentheses. For example, you can tell *isNaN( )* is a command, because of the parentheses following *isNaN*.

In addition, JavaScript lets you create your own functions, so you can make your scripts do things beyond what the standard JavaScript commands offer. You'll learn about functions in Chapter 3, starting on page 97.

Types of Data

You deal with different types of information every day. Your name, the price of food, the address of your doctor's office, and the date of your next birthday are all information that is important to you. You make decisions about what to do and how to live your life based on the information you have. Computer programs are

no different. They also rely on information to get things done. For example, to calculate the total for a shopping cart, you need to know the price and quantity of each item ordered. To customize a Web page with a visitor's name ("Welcome Back, *Kotter*"), you need to know his or her name.

Programming languages usually categorize information into different types, and treat each type in a different way. In JavaScript, the three most common types of data are *number*, *string*, and *Boolean*.

Numbers

Numbers are used for counting and calculating; you can keep track of the number of days until summer vacation, or calculate the cost of buying two tickets to a movie. Numbers are very important in JavaScript programming: you can use numbers to keep track of how many times a visitor has visited a Web page, to specify the exact pixel position of an item on a Web page, or to determine how many products a visitor wants to order.

In JavaScript, a number is represented by a numeric character; *5*, for example, is the number five. You can also use fractional numbers with decimals, like 5.25 or 10.3333333. JavaScript even lets you use negative numbers, like –130.

Since numbers are frequently used for calculations, your programs will often include mathematical operations. You'll learn about *operators* on page 48, but just to provide an example of using JavaScript with numbers, say you wanted to print the total value of 5 plus 15 on a Web page; you could do that with this line of code:

```
document.write(5 + 15);
```

This snippet of JavaScript adds the two numbers together and prints the total (20) onto a Web page. There are many different ways to work with numbers, and you'll learn more about them starting on page 134.

Strings

To display a name, a sentence, or any series of letters, you use strings. A *string* is just a series of letters and other symbols enclosed inside of quote marks. For example, *'Welcome Hal'*, and "You are here" are both examples of strings. You used a string in the last chapter with the alert command—*alert('Hello World!');*.

A string's opening quote mark signals to the JavaScript interpreter that what follows is a string—just a series of symbols. The interpreter accepts the symbols literally, rather than trying to interpret the string as anything special to JavaScript like a command. When the interpreter encounters the final quote mark, it understands that it has reached the end of the string and continues onto the next part of the program.

You can use either double quote marks (*"hello world"*) or single quote marks (*'hello world'*) to enclose the string, but you must make sure to use the *same type* of quote mark at the beginning and end of the string (for example, *"this is not right'*

isn't a valid string because it begins with a double-quote mark but ends with a single-quote.)

So, to pop-up an alert box with the message *Warning, warning!* you could write:

```
alert('Warning, warning!');
```

or

```
alert("Warning, warning!");
```

You'll use strings frequently in your programming—when adding alert messages, when dealing with user input on Web forms, and when manipulating the contents of a Web page. They're so important that you'll learn a lot more about using strings starting on page 116.

FREQUENTLY ASKED QUESTION

Putting Quotes into Strings

When I try to create a string with a quote mark in it, my program doesn't work. Why is that?

In JavaScript, quote marks indicate the beginning and end of a string, even when you don't want them to. When the JavaScript interpreter encounters the first quote mark, it says to itself, "Ahh, here comes a string." When it reaches a matching quote mark, it figures it has come to the end of the string. That's why you can't create a string like this: "He said, "Hello."". In this case, the first quote mark (before the word "He") marks the start of the string, but as soon as the JavaScript interpreter encounters the second quote mark (before the word "Hello"), it figures that the string is over, so you end up with the string *"He said, "* and the *Hello.* part, which creates a JavaScript error.

There are a couple of ways to get around this conundrum. The easiest method is to use single quotes to enclose a string that has one or more double quotes inside it. For example, *'He said,* "Hello."' is a valid string–the single quotes create the string, and the double quotes inside are a *part* of the string. Likewise, you can use double quotes to enclose a string that has a single quote inside it: "This isn't fair" for example.

Another method is to tell the JavaScript interpreter to just treat the quote mark inside the string literally–that is, treat the quote mark as part of the string, not the end of the string. You do this using something called an *escape character*. If you precede the quote mark with a backward slash (\), the quote is treated as part of the string. You could rewrite the above example like this: *"He said, \"Hello.\ ""*. In some cases, an escape character is the only choice. For example: *'He said,* "This isn\'t fair."' Because the string is enclosed by single quotes, the lone single quote in the word "isn't" has to have a backward slash before it: *isn\'t*.

You can even escape quote marks when you don't necessarily have to–as a way to make it clear that the quote mark should be taken literally. For example. *'He said, "Hello."'*. Even though you don't need to escape the double quotes (since single quotes surround the entire string) some programmers do it anyway so that it's clear to them that the quote mark is just a quote mark.

Booleans

Whereas numbers and strings offer infinite possibilities, the Boolean data type is simple. It is either one of two values: *true* or *false*. You'll encounter Boolean data types when you create JavaScript programs that respond intelligently to user input

and actions. For example, if you want to make sure a visitor supplied an email address before submitting a form, you can add logic to your page by asking the simple question: "Did the user type in a valid email address?" The answer to this question is a Boolean value: either the email address is valid (true) or it's not (false). Depending on the answer to the question, the page could respond in different ways. For example, if the email address is valid (true), then submit the form; if it is not valid (false), then display an error message and prevent the form from being submitted.

You'll learn how Boolean values come into play when adding logic to your programs in the box on page 80.

Variables

You can type a number, string, or Boolean value directly into your JavaScript program, but these data types work only when you already have the information you need. For example, you can make the string "Hi Bob" appear in an alert box like this:

```
alert('Hi Bob');
```

But that statement only makes sense if everyone who visits the page is named Bob. If you want to present a personalized message for different visitors, the name needs to be different depending on who is viewing the page: 'Hi Mary,' 'Hi Joseph,' 'Hi Ezra,' and so on. Fortunately, all programming languages provide something known as a *variable* to deal with just this kind of situation.

A variable is a way to store information so that you can later use and manipulate it. For example, imagine a JavaScript-based pinball game where the goal is to get the highest score. When a player first starts the game, her score will be zero, but as she knocks the pinball into targets, the score will get bigger. In this case, the *score* is a variable since it starts at 0 but changes as the game progresses—in other words, a variable holds information that can *vary*. See Figure 2-1 for an example of another game that uses variables.

Think of a variable as a kind of basket: you can put an item into a basket, look inside the basket, dump out the contents of a basket, or even replace what's inside the basket with something else. However, even though you might change what's inside the basket, it still remains the same basket.

Creating a Variable

Creating a variable is a two-step process that involves *declaring* the variable and naming it. In JavaScript to create a variable named *score* you would type:

```
var score;
```

The first part, *var*, is a JavaScript keyword that creates, or, in programming-speak, *declares* the variable. The second part of the statement, *score*, is the variable's name.

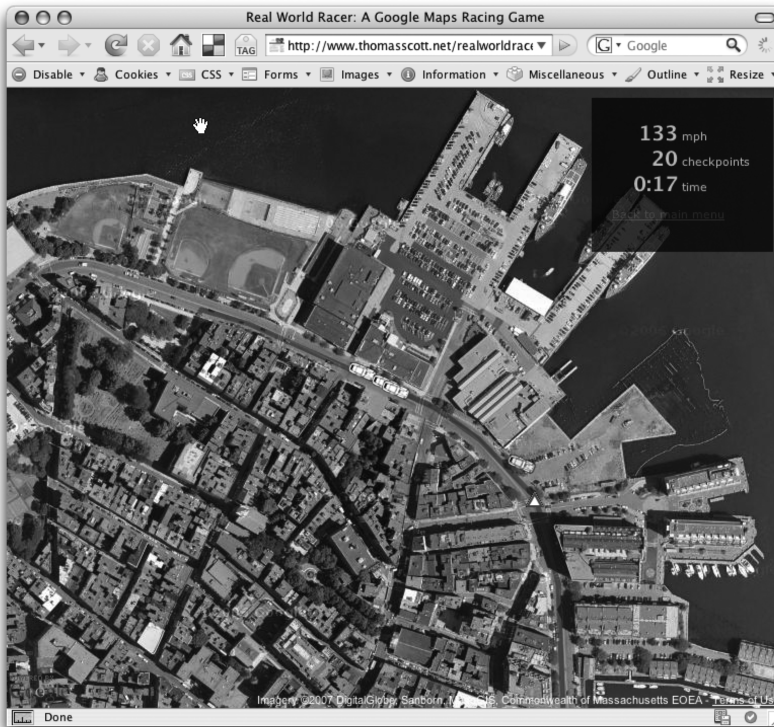

Figure 2-1: *The game Real World Racer (*www.thomasscott.net/realworldracer*) merges JavaScript with Google Maps technology to let you race your way along any road in the world. The game tracks your speed, time, and the number of checkpoints you've crossed (in the top-right box). These are all examples of variables since they change value as the game goes on.*

What you name your variables is up to you, but there are a few rules you must follow when naming variables:

- **Variable names must begin with a letter, $, or _.** In other words, you can't begin a variable name with a number or punctuation: so *1thing,* and *&thing* won't work, but *score, $score,* and *_score* are fine.
- **Variable names can only contain letters, numbers, $, and _.** You can't use spaces or any other special characters anywhere in the variable name: *fish&chips* and *fish and chips* aren't legal, but *fish_n_chips* and *plan9* are.
- **Variable names are case-sensitive.** The JavaScript interpreter sees uppercase and lowercase letters as distinct, so a variable named *SCORE* is different from a variable named *score*, which is also different from variables named *sCoRE* and *Score*.
- **Avoid keywords.** Some words in JavaScript are specific to the language itself: *var* for example is used to create a variable, so you can't name a variable *var*. In addition, some words, like *alert, document,* and *window*, are considered special properties of the Web browser. You'll end up with a JavaScript error if you try to use those words as variable names. You can find a list of some reserved words in Table 2-1. Not all of these reserved words will cause problems in all browsers, but it's best to steer clear of these names when naming variables.

Table 2-1. *Some words are reserved for use by JavaScript and the Web browser. Avoid using them as variable names.*

JavaScript keywords	Reserved for future use	Reserved for browser
break	abstract	alert
case	boolean	blur
catch	byte	closed
continue	char	document
default	class	focus
delete	const	frames
do	debugger	history
else	double	innerHeight
finally	enum	innerWidth
for	export	length
function	extends	location
if	final	navigator
in	float	open
instanceof	goto	outerHeight
new	implements	outerWidth
return	import	parent
switch	int	screen
this	interface	screenX
throw	long	screenY
try	native	statusbar
typeof	package	window
var	private	
void	protected	
while	public	
with	short	
	static	
	super	
	synchronized	
	throws	
	transient	
	volatile	

In addition to these rules, aim to make your variable names clear and meaningful. Naming variables according to what type of data you'll be storing in them makes it much easier to look at your programming code and immediately understand what's going on. For example, *score* is a great name for a variable used to track a player's game score. The variable name *s* would also work, but the single letter "s" doesn't give you any idea about what's stored in the variable.

Likewise, make your variable names easy to read. When you use more than one word in a variable name, either use an underscore between words or capitalize the first letter of each word after the first. For example, *imagepath* isn't as easy to read and understand as *image_path* or *imagePath*.

Tip: If you want to declare a bunch of variables at one time, you can do it in a single line of code like this:

```
var score, players, game_time;
```

This line of code creates three variables at once.

Using Variables

Once a variable is created, you can store any type of data that you'd like in it. To do so, you use the = sign. For example, to store the number 0 in a variable named *score*, you could type this code:

```
var score;
score = 0;
```

The first line of code above creates the variable; the second line stores the number 0 in the variable. The equal sign is called an *assignment operator*, because it's used to assign a value to a variable. You can also create a variable and store a value in it with just a single JavaScript statement like this:

```
var score = 0;
```

You can store strings, numbers and Boolean values in a variable:

```
var firstName = 'Peter';
var lastName = 'Parker';
var age = 22;
var isSuperHero = true;
```

Tip: To save typing, you can declare multiple variables with a single *var* keyword, like this:

```
var x, y, z;
```

You can even declare and store values into multiple variables in one JavaScript statement:

```
var isSuperHero=true, isAfraidOfHeights=false;
```

Once you've stored a value in a variable, you can access that value simply by using the variable's name. For example, to open an alert dialog box and display the value stored in the variable *score*, you'd type this:

```
alert(score);
```

Notice that you don't use quotes with a variable—that's just for strings, so the code *alert('score')* will display the word "score" and not the value stored in the variable *score*. Now you can see why strings have to be enclosed in quote marks: the JavaScript interpreter treats words without quotes as either special JavaScript objects (like the *alert( )* command) or a variable name.

FREQUENTLY ASKED QUESTION

Spaces, Tabs, and Carriage Returns in JavaScript

JavaScript seems so sensitive about typos. How do I know when I'm supposed to use space characters, and when I'm not allowed to?

You must put a space between keywords: *varscore=0*, for example, doesn't create a new variable named *score*. The JavaScript interpreter needs the space between *var* and *score* to identify the *var* keyword: *var score=0*. However, space isn't necessary between keywords and symbols like the assignment operator (=) or the semicolon that ends a statement.

JavaScript interpreters ignore extra space, so you're free to insert extra spaces, tabs and carriage returns to format your code. For example, you don't need a space on either side of an assignment operator, but you can add them if you find it easier to read. Both of the lines of code below work:

```
var formName='signup';
var formRegistration = 'newsletter' ;
```

In fact, you can insert as many spaces as you'd like, and even insert carriage returns within a statement. So both of the following statements also work:

```
var formName     =      'signup';
var formRegistration
              =
      'newsletter';
```

Of course, just because you can insert extra space, doesn't mean you should. The last two examples are actually harder to read and understand because of the extra space. So the general rule of thumb is add extra space if it makes your code easier to understand. You'll see examples of how space can make code easier to read with arrays (page 58) and with JavaScript Object Literals (page 188).

One important exception to the above rules: you can't insert a carriage return inside a string; in other words you can't split a string over two lines in your code like this:

```
var name = 'Bob
         Smith';
```

Inserting a carriage return (pressing the Enter or Return key) like this produces a JavaScript error and your program won't run.

Note: You only need to use the *var* keyword once–when you first create the variable. After that, you're free to assign new values to the variable without using *var*.

Working with Data Types and Variables

Storing a particular piece of information like a number or string in a variable is usually just a first step in a program. Most programs also manipulate data to get new results. For example, add a number to a score to increase it, multiply the number of items ordered by the cost of the item to get a grand total, or personalize a generic message by adding a name to the end: "Good to see you again, Igor." JavaScript provides various *operators* to modify data. An operator is simply a symbol or word that can change one or more values into something else. For example, you use the + symbol—the addition operator—to add numbers together. There are different types of operators for the different data types.

Basic Math

JavaScript supports basic mathematical operations such as addition, division, subtraction, and so on. Table 2-2 shows the most basic math operators and how to use them.

***Table 2-2.** Basic math with JavaScript*

Operator	What it does	How to use it
+	Adds two numbers	5 + 25
–	Subtracts one number from another	25 – 5
*	Multiplies two numbers	5 * 10
/	Divides one number by another	15/5

You may be used to using an x for multiplication (4×5, for example), but in JavaScript, you use the * symbol to multiply two numbers.

You can also use variables in mathematical operations. Since a variable is only a container for some other value like a number or string, using a variable is the same as using the contents of that variable.

```
var price = 10;
var itemsOrdered = 15;
var totalCost = price * itemsOrdered;
```

The first two lines of code create two variables (*price* and *itemsOrdered*) and store a number in each. The third line of code creates another variable (*totalCost*) and stores the results of multiplying the value stored in the *price* variable (10) and the value stored in the *itemsOrdered* variable. In this case, the total (150) is stored in the variable *totalCost*.

This sample code also demonstrates the usefulness of variables. Suppose you write a program as part of a shopping cart system for an e-commerce Web site. Throughout the program, you need to use the price of a particular product to make various calculations. You could code the actual price throughout the program (for example, say the product cost 10 dollars, so you type 10 in each place in the program that price is used). However, if the price ever changes, you'd have to locate and change each line of code that uses the price. By using a variable, on the other hand, you can set the price of the product somewhere near the beginning of the program. Then, if the price ever changes, you only need to modify the one line of code that defines the product's price to update the price throughout the program:

```
var price = 20;
var itemsOrdered = 15;
var totalCost = price * itemsOrdered;
```

There are lots of other ways to work with numbers (you'll learn a bunch starting on page 134), but you'll find that you most frequently use the basic math operators listed in Table 2-2.

The Order of Operations

If you perform several mathematical operations at once—for example, you total up several numbers then multiply them all by 10—you need to keep in mind the order in which the JavaScript interpreter performs its calculations. Some operators take precedence over other operators, so they're calculated first. This fact can cause some unwanted results if you're not careful. Take this example:

```
4 + 5 * 10
```

You might think this simply is calculated from left to right: 4 + 5 is 9 and 9 * 10 is 90. It's not. The multiplication actually goes first, so this equation works out to 5 * 10 is 50, plus 4 is 54. Multiplication (the * symbol) and division (the / symbol) take precedence over addition (+) and subtraction (-).

To make sure that the math works out the way you want it, use parentheses to group operations. For example, you could rewrite the equation above like this:

```
(4 + 5) * 10
```

Any math that's performed inside parentheses happens first, so in this case the 4 is added to 5 first and the result, 9, is then multiplied by 10. If you do want the multiplication to occur first, it would be clearer to write that code like this:

```
4 + (5*10);
```

Combining Strings

Combining two or more strings to make a single string is a common programming task. For example, if a Web page has a form that collects a person's first name in one form field and his last name in a different field, you need to combine the two fields to get his complete name. What's more, if you want to display a message letting the user know his form information was submitted, you need to combine the generic message with the person's name: "John Smith, thanks for your order."

Combining strings is called *concatenation*, and you accomplish it with the + operator. Yes, that's the same + operator you use to add number values, but with strings it behaves a little differently. Here's an example:

```
var firstName = 'John';
var lastName = 'Smith';
var fullName = firstName + lastName;
```

In the last line of code above, the contents of the variable *firstName* are combined (or concatenated) with the contents of the variable *lastName*—the two are literally joined together and the result is placed in the variable *fullName*. In this example, the resulting string is "JohnSmith"—there isn't a space between the two names, since concatenating just fuses the strings together. In many cases (like this one), you need to add an empty space between strings that you intend to combine:

```
var firstName = 'John';
var lastName = 'Smith';
var fullName = firstName + ' ' + lastName;
```

The ' ' in the last line of this code is a single quote, followed by a space, followed by a final single quote. This code is simply a string that contains an empty space. When placed between the two variables in this example, it creates the string "John Smith". This last example also demonstrates that you can combine more than two strings at a time; in this case, three strings.

Combining Numbers and Strings

Most of the mathematical operators only make sense for numbers. For example, it doesn't make any sense to multiply 2 and the string 'eggs'. If you try this example, you'll end up with a special JavaScript value *NaN*, which stands for "not a number." However, there are times when you may want to combine a string with a number. For example, say you want to present a message on a Web page that specifies how many times a visitor has been to your Web site. The number of times she's visited is a *number*, but the message is a *string*. In this case, you use the + operator to do two things: convert the number to a string and concatenate it with the other string. Here's an example:

```
var numOfVisits = 101;
var message = 'You have visited this site ' + numOfVisits + ' times.';
```

In this case, *message* contains the string "You have visited this site 101 times." The JavaScript interpreter recognizes that there is a string involved, so it realizes it won't be doing any math (no addition). Instead, it treats the + as the concatenation operator, and at the same time realizes that the number should be converted to a string as well.

This example may seem like a good way to print words and numbers in the same message. In this case, it's obvious that the number is part of a string of letters that makes up a complete sentence, and whenever you use the + operator with a string value and a number, the JavaScript interpreter converts the number to a string.

That feature, known as *automatic type conversion,* can cause problems, however. For example, if a visitor answers a question on a form ("How many pairs of shoes would you like?") by typing a number (*2*, for example), that input is treated like a string—*'2'*. So you can run into a situation like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = numOfShoes + numOfSocks;
```

You'd expect the value stored in *totalItems* to be 6 (2 shoes + 4 pairs of socks). Instead, because the value in *numOfShoes* is a string, the JavaScript interpreter converts the value in the variable *numOfSocks* to a string as well, and you end up with the string '24' in the *totalItems* variable. There are a couple of ways to prevent this error.

First, you add + to the beginning of the *string* that contains a number like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = +numOfShoes + numOfSocks;
```

Adding a + sign before a variable (make sure there's no space between the two) tells the JavaScript interpreter to try to convert the string to a number value—if the string only contains numbers like '2', you'll end up with the string converted to a number. In this example, you end up with 6 (2 + 4). Another technique is to use the *Number( )* command like this:

```
var numOfShoes = '2';
var numOfSocks = 4;
var totalItems = Number(numOfShoes) + numOfSocks;
```

Number( ) converts a string to a number if possible. (If the string is just letters and not numbers, you get the *NaN* value to indicate that you can't turn letters into a number.)

In general, you'll most often encounter numbers as strings when getting input from a visitor to the page; for example, when retrieving a value a visitor entered into a form field. So, if you need to do any addition using input collected from a form or other source of visitor input, make sure you run it through the *Number( )* command first.

Changing the Values in Variables

Variables are useful because they can hold values that change as the program runs—a score that changes as a game is played, for example. So how do you change a variable's value? If you just want to replace what's contained inside a variable, assign a new value to the variable. For example:

```
var score = 0;
score = 100;
```

However, you'll frequently want to keep the value that's in the variable and just add something to it or change it in some way. For example, with a game score you never just give a new score, you always add or subtract from the current score. To add to the value of a variable, you use the variable's name as part of the operation like this:

```
var score = 0;
score = score + 100;
```

That last line of code may appear confusing at first, but it uses a very common technique. Here's how it works: All of the action happens to the right of the = sign first; that is, the *score + 100* part. Translated, it means "take what's currently stored in *score* (0) and then add 100 to it." The result of that operation is *then* stored back into the variable *score*. The final outcome of these two lines of code is that the variable score now has the value of 100.

The same logic applies to other mathematical operations like subtraction, division, or multiplication:

```
score = score - 10;
score = score * 10;
score = score / 10;
```

In fact, performing math on the value in a variable and then storing the result back into the variable is so common that there are shortcuts for doing so with the four main mathematical operations, as pictured in Table 2-3.

Table 2-3. *Shortcuts for performing math on a variable*

Operator	What it does	How to use it	The same as
`+=`	Adds value on the right side of equal sign to the variable on the left.	`score += 10;`	`score = score + 10;`
`-=`	Subtracts value on the right side of the equal sign from the variable on the left.	`score -= 10;`	`score = score - 10;`
`*=`	Multiplies the variable on the left side of the equal sign and the value on the right side of the equal sign.	`score *= 10;`	`score = score * 10`
`/=`	Divides the value in the variable by the value on the right side of the equal sign.	`score /= 10`	`score = score / 10`
`++`	Placed directly after a variable name, ++ adds 1 to the variable.	`score++`	`score = score + 1`
`--`	Placed directly after a variable name, -- subtracts 1 from the variable.	`score--`	`score = score - 1`

The same rules apply when concatenating a string to a variable. For example, say you have a variable with a string in it and want to add another couple of strings onto that variable:

```
var name = 'Franklin';
var message = 'Hello';
message = message + ' ' + name;
```

As with numbers, there's a shortcut operator for concatenating a string to a variable. The += operator adds the string value to the right of the = sign to the end of the variable's string. So the last line of the above code could be rewritten like this:

```
message += ' ' + name;
```

You'll see the += operator frequently when working with strings, and throughout this book.