



1

WHAT IS JAVASCRIPT?

Computers are incredibly powerful machines, capable of performing amazing feats like playing competitive chess, serving thousands of web pages, or making millions of complex calculations in less than a few seconds. But deep down, computers are actually pretty dumb. Computers can *only* do exactly what we humans tell them to do. We tell computers how to behave using computer programs, which are just sets of instructions for the computers to follow. Without programs, computers can't do anything at all!

MEET JAVASCRIPT

Even worse, computers can't understand English or any other spoken language. Computer programs are written in a *programming language* like JavaScript. You might not have heard of JavaScript before, but you've certainly used it. The JavaScript programming language is used to write programs that run in web pages. JavaScript can control how a web page looks or make the page respond when a viewer clicks a button or moves the mouse.

Sites like Gmail, Facebook, and Twitter use JavaScript to make it easier to send email, post comments, or browse websites. For example, when you're on Twitter reading tweets from @nostarch and you see more tweets at the bottom of the page as you scroll down, that's JavaScript in action.

You only have to visit a couple of websites to see why JavaScript is so exciting.

- JavaScript lets you play music and create amazing visual effects. For example, you can fly through an interactive music video created by HelloEnjoy for Ellie Goulding's song "Lights" (<http://lights.helloenjoy.com/>), as shown in Figure 1-1.
- JavaScript lets you build tools for others to make their own art. Patatap (<http://www.patatap.com/>) is a kind of virtual "drum machine" that creates all kinds of cool noises—and cool animations to go along with them—as shown in Figure 1-2.



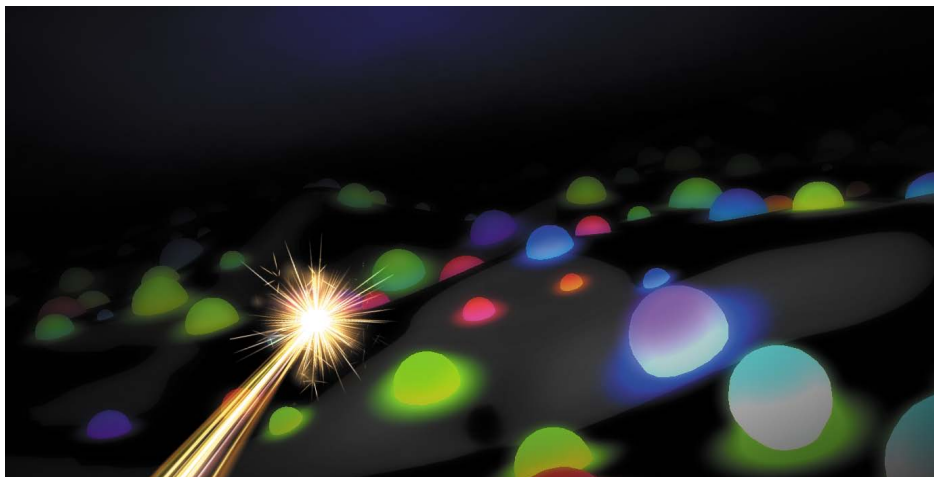


Figure 1-1: You control the flashing cursor in HelloEnjoy's "Lights" music video.

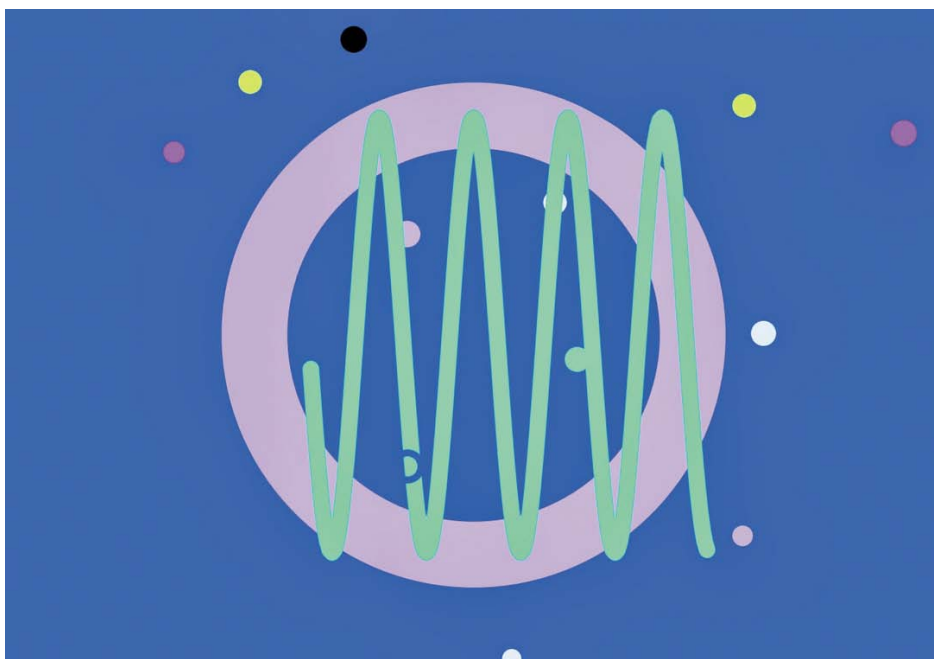


Figure 1-2: When you visit Patatap, try pressing a bunch of keys to make different noises!

- JavaScript lets you play fun games. *CubeSlam* (<https://www.cubeslam.com/>) is a 3D re-creation of the classic game Pong, which looks a little like air hockey. You can play against one of your friends or a computer-generated bear, as shown in Figure 1-3.



Figure 1-3: The CubeSlam game is programmed entirely in JavaScript!

WHY LEARN JAVASCRIPT?

JavaScript isn't the only programming language out there—in fact, there are literally hundreds of programming languages. But there are many reasons to learn JavaScript. For one, it's a lot easier (and more fun) to learn than many other programming languages. But perhaps best of all, in order to write and run JavaScript programs, all you need is a web browser like Internet Explorer, Mozilla Firefox, or Google Chrome. Every web browser comes with a JavaScript *interpreter* that understands how to read JavaScript programs.

Once you've written a JavaScript program, you can send people a link to it, and they can run it in a web browser on their computer, too! (See "Sharing Your Code Using JSFiddle" on page 297.)

WRITING SOME JAVASCRIPT

Let's write a bit of simple JavaScript in Google Chrome (<http://www.google.com/chrome/>). Install Chrome on your computer (if it's not already installed), and then open it and type **about:blank** in the address bar. Now press ENTER and you'll see a blank page, like the one in Figure 1-4.

We'll begin by coding in Chrome's JavaScript console, which is a secret way programmers can test out short JavaScript programs. On Microsoft Windows or Linux, hold down the CTRL and SHIFT keys and press J. On Mac OS, hold down the COMMAND and OPTION keys and press J.

If you've done everything correctly, you should see a blank web page and, beneath that, a blinking cursor (|) next to a right angle bracket (>), as shown in Figure 1-4. That's where you'll write JavaScript!

NOTE

The Chrome console will color your code text; for example, the text you input will be blue, and output will be colored based on its type. In this book, we'll use similar colors for our code text wherever we're using the console.

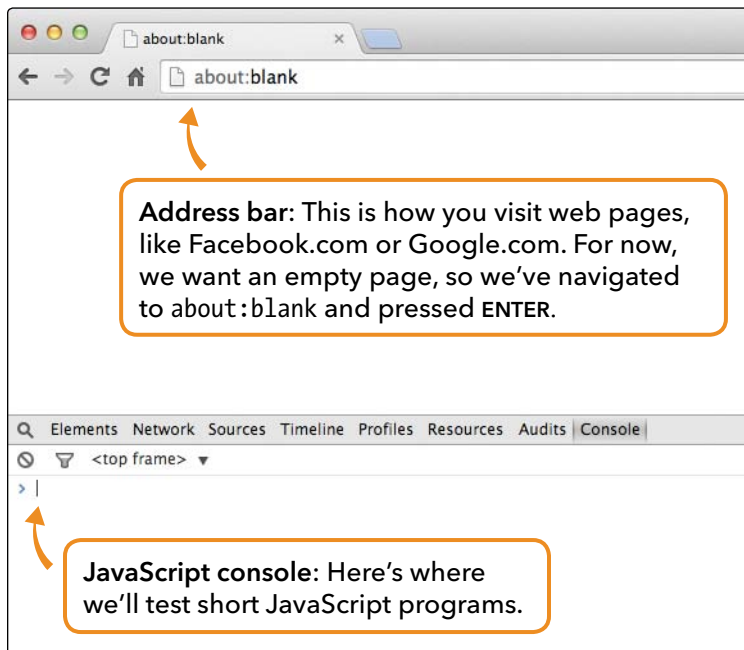


Figure 1-4: Google Chrome's JavaScript console

When you enter code at the cursor and press ENTER, JavaScript should run, or *execute*, your code and display the result (if any) on the next line. For example, type this into the console:

```
3 + 4;
```

Now press ENTER. JavaScript should output the answer (7) to this simple bit of addition on the following line:

```
3 + 4;  
7
```

Well, that's easy enough. But isn't JavaScript more than a glorified calculator? Let's try something else.

THE STRUCTURE OF A JAVASCRIPT PROGRAM

Let's create something a bit sillier—a JavaScript program to print a series of cat faces that look like this:

```
=^.^=
```

Unlike our addition program, this JavaScript program will take up multiple lines. To type the program into the console, you'll have to add new lines by pressing SHIFT-ENTER at the end of each line. (If you just press ENTER, Chrome will try to execute what you've written, and the program won't work as expected. I warned you that computers were dumb!)

Type this into your browser console:

```
// Draw as many cats as you want!  
var drawCats = function (howManyTimes) {  
  for (var i = 0; i < howManyTimes; i++) {  
    console.log(i + " =^.^=");  
  }  
};  
  
drawCats(10); // You can put any number here instead of 10.
```



At the very end, press ENTER instead of SHIFT-ENTER. When you do that, you should see the following output:

```
0 =^.^=  
1 =^.^=  
2 =^.^=  
3 =^.^=  
4 =^.^=  
5 =^.^=  
6 =^.^=  
7 =^.^=  
8 =^.^=  
9 =^.^=
```

If you made any typos, your output might look very different or you might get an error. That's what I mean when I say computers are dumb—even a simple piece of code must be perfect for a computer to understand what you want it to do!



I won't go through exactly how this code *works* for now (we'll return to this program in Chapter 8), but let's look at some of the features of this program and of JavaScript programs in general.

SYNTAX

Our program includes lots of symbols, including parentheses `()`, semicolons `;`, curly brackets `{}`, plus signs `+`, and a few words that might seem mysterious at first (like `var` and `console.log`). These are all part of JavaScript's *syntax*—that is, JavaScript's rules for how to combine symbols and words to create working programs.

When you're learning a new programming language, one of the trickiest parts is getting used to the rules for how to write different kinds of instructions to the computer. When you're first starting out, it's easy to forget when to include parentheses, or to mix up the order in which you need to include certain values. But as you practice, you'll start to get the hang of it.

In this book, we'll go slow and steady, introducing new syntax little by little so that you can build increasingly powerful programs.

COMMENTS

The first line in our `cats` program is this:

```
// Draw as many cats as you want!
```

This is called a *comment*. Programmers use comments to make it easier for other programmers to read and understand their code. The computer ignores comments completely. Comments in JavaScript start with two forward slashes (`//`). Everything following the slashes (on the same line) is ignored by the JavaScript interpreter, so the comments don't have any effect on how a program is executed—they are just there to provide a description.

In the code in this book, you'll see comments that describe what's happening in the code. As you write your own code, add your own comments. Then when you look at your code later, your comments will remind you how the code works and what's happening in each step.

There's another code comment on the last line of our program. Remember, everything after that `//` isn't run by the computer!

```
drawCats(10); // You can put any number here instead of 10.
```

Code comments can be on their own line, or they can come after your code. If you put the `//` at the front, like this:

```
// drawCats(10);
```

... nothing will happen! Chrome sees the whole line as a comment, even if it's JavaScript.

Once you start reading JavaScript code out in the wild world, you'll also see comments that look like this:

```
/*  
Draw as many cats  
as you want!  
*/
```

This is a different style of commenting, which is typically used for comments that are longer than one line. But it does the same thing: everything between the `/*` and the `*/` is a comment that the computer won't run.

WHAT YOU LEARNED

In this chapter, you learned a bit about what JavaScript is and what it can be used for. You also learned how to run JavaScript using the Google Chrome browser and tried out a sample program. All of the code examples in this book, unless I say otherwise, can (and should!) be used in Chrome's JavaScript console. Don't just read the code—try typing things out! It's the only way to learn to program.

In the next chapter, you'll start learning the fundamentals of JavaScript, beginning with the three basic types of information you can work with: numbers, strings, and Booleans.





2

DATA TYPES AND VARIABLES

Programming is all about manipulating data, but what *is* data? *Data* is information that we store in our computer programs. For example, your name is a piece of data, and so is your age. The color of your hair, how many siblings you have, where you live, whether you're male or female—these things are all data.

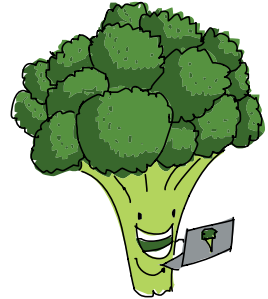
In JavaScript, there are three basic types of data: numbers, strings, and Booleans. Numbers are used for representing, well, numbers! For example, your age can be represented as a number, and so can your height. Numbers in JavaScript look like this:

```
5;
```

Strings are used to represent text. Your name can be represented as a string in JavaScript, as can your email address. Strings look like this:

```
"Hi, I'm a string";
```

Booleans are values that can be true or false. For example, a Boolean value about you would be whether you wear glasses. Another could be whether you like broccoli. A Boolean looks like this:



```
true;
```

There are different ways to work with each data type. For example, you can multiply two numbers, but you can't multiply two strings. With a string, you can ask for the first five characters. With Booleans, you can check to see whether two values are both true. The following code example illustrates each of these possible operations.

```
99 * 123;  
12177  
"This is a long string".slice(0, 4);  
"This"  
true && false;  
false
```

All data in JavaScript is just a combination of these types of data. In this chapter, we'll look at each type in turn and learn different ways to work with each type.

NOTE

You may have noticed that all of these commands end with a semicolon (;). Semicolons mark the end of a particular JavaScript command or instruction (also called a statement), sort of like the period at the end of a sentence.

NUMBERS AND OPERATORS

JavaScript lets you perform basic mathematical operations like addition, subtraction, multiplication, and division. To make these calculations, we use the symbols +, -, *, and /, which are called *operators*.

You can use the JavaScript console just like a calculator. We've already seen one example, adding together 3 and 4. Let's try something harder. What's 12,345 plus 56,789?

```
12345 + 56789;  
69134
```

That's not so easy to work out in your head, but JavaScript calculated it in no time.

You can add multiple numbers with multiple plus signs:

```
22 + 33 + 44;  
99
```

JavaScript can also do subtraction . . .

```
1000 - 17;  
983
```

and multiplication, using an asterisk . . .

```
123 * 456;  
56088
```

and division, using a forward slash . . .

```
12345 / 250;  
49.38
```

You can also combine these simple operations to make something more complex, like this:

```
1234 + 57 * 3 - 31 / 4;  
1397.25
```

Here it gets a bit tricky, because the result of this calculation (the answer) will depend on the order that JavaScript does

each operation. In math, the rule is that multiplication and division always take place before addition and subtraction, and JavaScript follows this rule as well.

Figure 2-1 shows the order JavaScript would follow. First, JavaScript multiplies $57 * 3$ and gets 171 (shown in red). Then it divides $31 / 4$ to get 7.75 (shown in blue). Next it adds $1234 + 171$ to get 1405 (shown in green). Finally it subtracts $1405 - 7.75$ to get 1397.25, which is the final result.

What if you wanted to do the addition and the subtraction first, before doing the multiplication and division? For example, say you have 1 brother and 3 sisters and 8 candies, and you want to split the candies equally among your 4 siblings? (You've already taken your share!) You would have to divide 8 by your number of siblings.

Here's an attempt:

```
8 / 1 + 3;  
11
```

That can't be right! You can't give each sibling 11 candies when you've only got 8! The problem is that JavaScript does division before addition, so it divides 8 by 1 (which equals 8) and then adds 3 to that, giving you 11. To fix this and make JavaScript do the addition first, we can use *parentheses*:

```
8 / (1 + 3);  
2
```

That's more like it! Two candies to each of your siblings. The parentheses force JavaScript to add 1 and 3 *before* dividing 8 by 4.

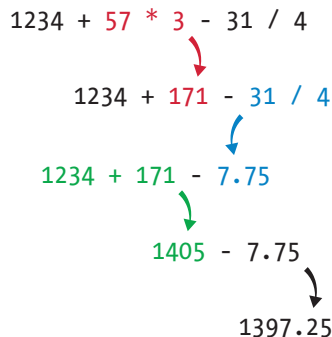
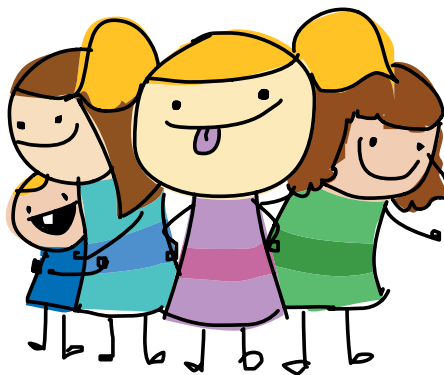


Figure 2-1: The order of operations: multiplication, division, addition, subtraction



TRY IT OUT!

Let's say your friend is trying to use JavaScript to work out how many balloons to buy. She's throwing a party and wants everyone to have 2 balloons to blow up. There were originally 15 people coming, but then she invited 9 more. She tries the following code:

```
15 + 9 * 2;  
33
```

But that doesn't seem right.

The problem is that the multiplication is happening before the addition. How would you add parentheses to make sure that JavaScript does the addition first? How many balloons does your friend really need?

VARIABLES

JavaScript lets you give names to values using *variables*. You can think of a variable as a box that you can fit one thing in. If you put something else in it, the first thing goes away.

To create a new variable, use the keyword `var`, followed by the name of the variable. A *keyword* is a word that has special meaning in JavaScript. In this case, when we type `var`, JavaScript knows that we are about to enter the name of a new variable. For example, here's how you'd make a new variable called `nick`:

```
var nick;  
undefined
```

We've created a new variable called `nick`. The console spits out `undefined` in response. But this isn't an error! That's just what JavaScript does whenever a command doesn't return a value. What's a return value? Well, for example, when you typed `12345 + 56789`;, the console returned the value `69134`. Creating a variable in JavaScript doesn't return a value, so the interpreter prints `undefined`.

To give the variable a value, use the equal sign:

```
var age = 12;  
undefined
```

Setting a value is called *assignment* (we are assigning the value 12 to the variable age). Again, undefined is printed, because we're creating another new variable. (In the rest of my examples, I won't show the output when it's undefined.)

The variable age is now in our interpreter and set to the value 12. That means that if you type age on its own, the interpreter will show you its value:

```
age;  
12
```

Cool! The value of the variable isn't set in stone, though (they're called *variables* because they can *vary*), and if you want to update it, just use = again:

```
age = 13;  
13
```

This time I didn't use the var keyword, because the variable age already exists. You need to use var only when you want to *create* a variable, not when you want to change the value of a variable. Notice also, because we're not creating a new variable, the value 13 is returned from the assignment and printed on the next line.

This slightly more complex example solves the candies problem from earlier, without parentheses:

```
var numberOfSiblings = 1 + 3;  
var numberOfCandies = 8;  
numberOfCandies / numberOfSiblings;  
2
```

First we create a variable called numberOfSiblings and assign it the value of 1 + 3 (which JavaScript works out to be 4). Then we create the variable numberOfCandies and assign 8 to it. Finally, we write numberOfCandies / numberOfSiblings. Because numberOfCandies is 8 and numberOfSiblings is 4, JavaScript works out 8 / 4 and gives us 2.

NAMING VARIABLES

Be careful with your variable names, because it's easy to misspell them. Even if you just get the capitalization wrong, the JavaScript interpreter won't know what you mean! For example, if you accidentally used a lowercase *c* in `numberOfCandies`, you'd get an error:

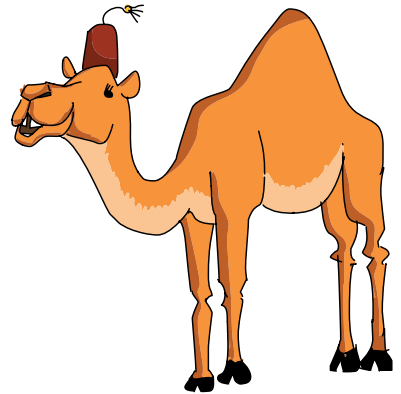
```
numberOfcandies / numberOfSiblings;  
ReferenceError: numberOfcandies is not defined
```

Unfortunately, JavaScript will only do exactly what you ask it to do. If you misspell a variable name, JavaScript has no idea what you mean, and it will display an error message.

Another tricky thing about variable names in JavaScript is that they can't contain spaces, which means they can be difficult to read. I could have named my variable `numberofcandies` with no capital letters, which makes it even harder to read because it's not clear where the words end. Is this variable “numb erof can dies” or “numberofcan dies”? Without the capital letters, it's hard to tell.

One common way to get around this is to start each word with a capital letter as in `NumberOfCandies`. (This convention is called *camel case* because it supposedly looks like the humps on a camel.)

The standard practice is to have variables start with a lowercase letter, so it's common to capitalize each word except for the first one, like this: `numberOfCandies`. (I'll follow this version of the camel case convention throughout this book, but you're free to do whatever you want!)



CREATING NEW VARIABLES USING MATH

You can create new variables by doing some math on older ones. For example, you can use variables to find out how many seconds there are in a year—and how many seconds old you are! Let's start by finding the number of seconds in an hour.

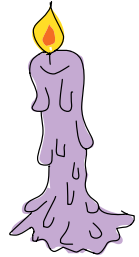
SECONDS IN AN HOUR

First we create two new variables called `secondsInAMinute` and `minutesInAnHour` and make them both 60 (because, as we know, there are 60 seconds in a minute and 60 minutes in an hour). Then we create a variable called `secondsInAnHour` and set its value to the result of multiplying `secondsInAMinute` and `minutesInAnHour`. At ❶, we enter `secondsInAnHour`, which is like saying, “Tell me the value of `secondsInAnHour` right now!” JavaScript then gives you the answer: it’s 3600.

```
var secondsInAMinute = 60;
var minutesInAnHour = 60;
var secondsInAnHour = secondsInAMinute * minutesInAnHour;
❶ secondsInAnHour;
3600
```

SECONDS IN A DAY

Now we create a variable called `hoursInADay` and set it to 24. Next we create the variable `secondsInADay` and set it equal to `secondsInAnHour` multiplied by `hoursInADay`. When we ask for the value `secondsInADay` at ❶, we get 86400, which is the number of seconds in a day.



```
var hoursInADay = 24;
var secondsInADay = secondsInAnHour * hoursInADay;
❶ secondsInADay;
86400
```

SECONDS IN A YEAR

Finally, we create the variables `daysInAYear` and `secondsInAYear`. The `daysInAYear` variable is assigned the value 365, and the variable `secondsInAYear` is assigned the value of `secondsInADay` multiplied by `daysInAYear`. Finally, we ask for the value of `secondsInAYear`, which is 31536000 (more than 31 million)!

```
var daysInAYear = 365;
var secondsInAYear = secondsInADay * daysInAYear;
secondsInAYear;
31536000
```

AGE IN SECONDS

Now that you know the number of seconds in a year, you can easily figure out how old you are in seconds (to the nearest year). For example, as I'm writing this, I'm 29:

```
var age = 29;
age * secondsInAYear;
914544000
```

To figure out your age in seconds, enter the same code, but change the value in age to *your* age. Or just leave out the age variable altogether and use a number for your age, like this:

```
29 * secondsInAYear;
914544000
```

I'm more than 900 million seconds old! How many seconds old are you?

INCREMENTING AND DECREMENTING

As a programmer, you'll often need to increase or decrease the value of a variable containing a number by 1. For example, you might have a variable that counts the number of high-fives you received today. Each time someone high-fives you, you'd want to increase that variable by 1.

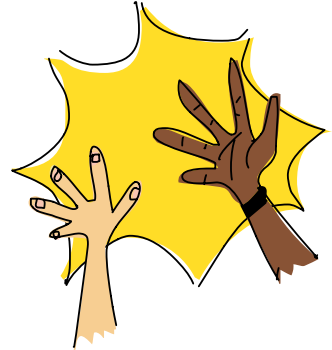
Increasing by 1 is called *incrementing*, and decreasing by 1 is called *decrementing*. You increment and decrement using the operators ++ and --.

```
var highFives = 0;
++highFives;
1
++highFives;
2
--highFives;
1
```

When we use the ++ operator, the value of highFives goes up by 1, and when we use the -- operator, it goes down by 1. You can also put these operators *after* the variable. This does the same thing, but the value that gets returned is the value *before* the increment or decrement.

```
highFives = 0;  
highFives++;  
0  
highFives++;  
1  
highFives;  
2
```

In this example, we set `highFives` to 0 again. When we call `highFives++`, the variable is incremented, but the value that gets printed is the value *before* the increment happened. You can see at the end (after two increments) that if we ask for the value of `highFives`, we get 2.



+= (PLUS-EQUALS) AND -= (MINUS-EQUALS)

To increase the value of a variable by a certain amount, you could use this code:

```
var x = 10;  
x = x + 5;  
x;  
15
```

Here, we start out with a variable called `x`, set to 10. Then, we assign `x + 5` to `x`. Because `x` was 10, `x + 5` will be 15. What we're doing here is using the old value of `x` to work out a new value for `x`. Therefore, `x = x + 5` really means “add 5 to `x`.”

JavaScript gives you an easier way of increasing or decreasing a variable by a certain amount, with the `+=` and `-=` operators. For example, if we have a variable `x`, then `x += 5` is the same as saying `x = x + 5`. The `-=` operator works in the same way, so `x -= 9` would be the same as `x = x - 9` (“subtract 9 from `x`”). Here's an example using both of these operators to keep track of a score in a video game:

```
var score = 10;  
score += 7;  
17  
score -= 3;  
14
```

In this example, we start with a score of 10 by assigning the value 10 to the variable `score`. Then we beat a monster, which increases `score` by 7 using the `+=` operator. (`score += 7` is the same as `score = score + 7`.) Before we beat the monster, `score` was 10, and `10 + 7` is 17, so this operation sets `score` to 17.

After our victory over the monster, we crash into a meteor and `score` is reduced by 3. Again, `score -= 3` is the same as `score = score - 3`. Because `score` is 17 at this point, `score - 3` is 14, and that value gets reassigned to `score`.

TRY IT OUT!

There are some other operators that are similar to `+=` and `-=`. For example, there are `*=` and `/=`. What do you think these do? Give them a try:

```
var balloons = 100;  
balloons *= 2;  
???
```

What does `balloons *= 2` do? Now try this:

```
var balloons = 100;  
balloons /= 4;  
???
```

What does `balloons /= 4` do?

STRINGS

So far, we've just been working with numbers. Now let's look at another type of data: *strings*. Strings in JavaScript (as in most programming languages) are just sequences of characters, which can include letters, numbers, punctuation, and spaces. We put strings between quotes so JavaScript knows where they start and end. For example, here's a classic:

```
"Hello world!";  
"Hello world!"
```

To enter a string, just type a double quotation mark (") followed by the text you want in the string, and then close the string with another double quote. You can also use single quotes ('), but to keep things simple, we'll just be using double quotes in this book.

You can save strings into variables, just like numbers:

```
var myAwesomeString = "Something REALLY awesome!!!";
```

There's also nothing stopping you from assigning a string to a variable that previously contained a number:

```
var myThing = 5;  
myThing = "this is a string";  
"this is a string"
```

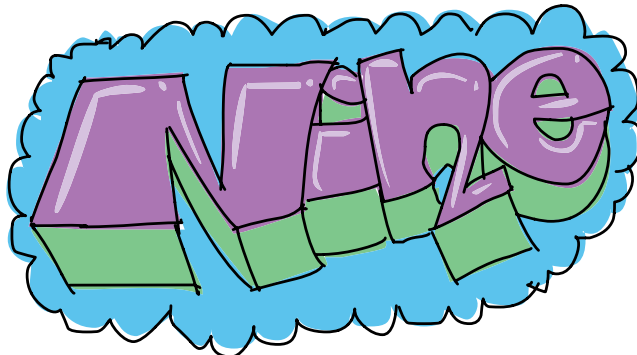
What if you put a number between quotes? Is that a string or a number? In JavaScript, a string is a string (even if it happens to have some characters that are numbers). For example:

```
var numberNine = 9;  
var stringNine = "9";
```

numberNine is a number, and stringNine is a string. To see how these are different, let's try adding them together:

```
numberNine + numberNine;  
18  
stringNine + stringNine;  
"99"
```

When we add the number values 9 and 9, we get 18. But when we use the + operator on "9" and "9", the strings are simply joined together to form "99".



JOINING STRINGS

As you just saw, you can use the + operator with strings, but the result is very different from using the + operator with numbers. When you use + to join two strings, you make a new string with the second string attached to the end of the first string, like this:

```
var greeting = "Hello";  
var myName = "Nick";  
greeting + myName;  
"HelloNick"
```

Here, we create two variables (greeting and myName) and assign each a string value ("Hello" and "Nick", respectively). When we add these two variables together, the strings are combined to make a new string, "HelloNick".

That doesn't look right, though—there should be a space between Hello and Nick. But JavaScript won't put a space there unless we specifically tell it to by adding a space in one of the original strings:

```
❶ var greeting = "Hello ";  
var myName = "Nick";  
greeting + myName;  
"Hello Nick"
```

The extra space inside the quotes at ❶ puts a space in the final string as well.

You can do a lot more with strings other than just adding them together. Here are some examples.

FINDING THE LENGTH OF A STRING

To get the length of a string, just add .length to the end of it.

```
"Supercalifragilisticexpialidocious".length;  
34
```

You can add .length to the end of the actual string or to a variable that contains a string:

```
var java = "Java";  
java.length;  
4
```

```
var script = "Script";
script.length;
6
var javascript = java + script;
javascript.length;
10
```

Here we assign the string "Java" to the variable `java` and the string "Script" to the variable `script`. Then we add `.length` to the end of each variable to determine the length of each string, as well as the length of the combined strings.

Notice that I said you can add `.length` to “the actual string *or to a variable* that contains a string.” This illustrates something very important about variables: anywhere you can use a number or a string, you can also use a variable containing a number or a string.

GETTING A SINGLE CHARACTER FROM A STRING

Sometimes you want to get a single character from a string. For example, you might have a secret code where the message is made up of the second character of each word in a list of words. You’d need to be able to get just the second characters and join them all together to create a new word.

To get a character from a particular position in a string, use square brackets, `[]`. Just take the string, or the variable containing the string, and put the number of the character you want in a pair of square brackets at the end. For example, to get the first character of `myName`, use `myName[0]`, like this:

```
var myName = "Nick";
myName[0];
"N"
myName[1];
"i"
myName[2];
"c"
```

Notice that to get the first character of the string, we use 0 rather than 1. That’s because JavaScript (like many other programming languages) starts counting at zero. That means when

you want the first character of a string, you use 0; when you want the second one, you use 1; and so on.

Let's try out our secret code, where we hide a message in some words' second characters. Here's how to find the secret message in a sequence of words:


```
var codeWord1 = "are";
var codeWord2 = "tubas";
var codeWord3 = "unsafe";
var codeWord4 = "?!";
codeWord1[1] + codeWord2[1] + codeWord3[1] + codeWord4[1];
"run!"
```

Again, notice that to get the second character of each string, we use 1.

CUTTING UP STRINGS

To “cut off” a piece of a big string, you can use `slice`. For example, you might want to grab the first bit of a long movie review to show as a teaser on your website. To use `slice`, put a period after a string (or a variable containing a string), followed by the word `slice` and opening and closing parentheses. Inside the parentheses, enter the start and end positions of the slice of the string you want, separated by a comma. Figure 2-2 shows how to use `slice`.

These two numbers
set the **start** and **end** of the slice.



```
"a string".slice(1, 5)
```

Figure 2-2: How to use `slice` to get characters from a string

For example:

```
var longString = "My long string is long";
longString.slice(3, 14);
"long string"
```

The first number in parentheses is the number of the character that begins the slice, and the second number is the number of

the character *after* the last character in the slice. Figure 2-3 shows which characters this retrieves, with the start value (3) and stop value (14) highlighted in blue.

M	y		l	o	n	g		s	t	r	i	n	g		i	s		l	o	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Figure 2-3: In the example above, *slice* grabs the characters shown in the gray box.

Here we basically tell JavaScript, “Pull a slice out of this longer string starting at the character at place 3 and keep going until you hit place 14.”

If you include only one number in the parentheses after *slice*, the string that it slices will start from that number and continue all the way to the end of the string, like this:

```
var longString = "My long string is long";
longString.slice(3);
"long string is long"
```

CHANGING STRINGS TO ALL CAPITAL OR ALL LOWERCASE LETTERS

If you have some text that you just want to shout, try using *toUpperCase* to turn it all into capital letters.

```
"Hello there, how are you doing?".toUpperCase();
"HELLO THERE, HOW ARE YOU DOING?"
```

When you use *toUpperCase()* on a string, it makes a new string where all the letters are turned into uppercase.

You can go the other way around, too:

```
"hELLo THERE, hOW ARE yOu doING?".toLowerCase();
"hello there, how are you doing?"
```

As the name suggests, *toLowerCase()* makes all of the characters lowercase. But shouldn't sentences always start with a capital letter? How can we take a string and make the first letter uppercase but turn the rest into lowercase?

NOTE

See if you can figure out how to turn "hELlo THERE, hOW ARE yOu doING?" into "Hello there, how are you doing?" using the tools you just learned. If you get stuck, review the sections on getting a single character and using slice. Once you're done, come back and have a look at how I did it.

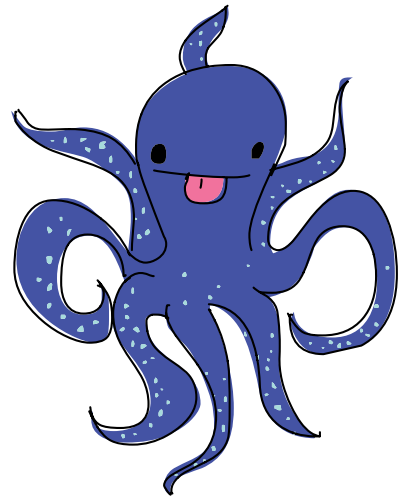
Here's one approach:

```
❶ var sillyString = "hELlo THERE, hOW ARE yOu doING?";
❷ var lowerString = sillyString.toLowerCase();
❸ var firstCharacter = lowerString[0];
❹ var firstCharacterUpper = firstCharacter.toUpperCase();
❺ var restOfString = lowerString.slice(1);
❻ firstCharacterUpper + restOfString;
"Hello there, how are you doing?"
```

Let's go through this line by line. At ❶, we create a new variable called `sillyString` and save the string we want to modify to that variable. At ❷, we get the lowercase version of `sillyString` ("hello there how are you doing?") with `.toLowerCase()` and save that in a new variable called `lowerString`.

At ❸, we use `[0]` to get the first character of `lowerString` ("h") and save it in `firstCharacter` (0 is used to grab the first character). Then, at ❹, we create an uppercase version of `firstCharacter` ("H") and call that `firstCharacterUpper`.

At ❺, we use `slice` to get all the characters in `lowerString`, starting from the second character ("ello there how are you doing?") and save that in `restOfString`. Finally, at ❻, we add `firstCharacterUpper` ("H") to `restOfString` to get "Hello there, how are you doing?".



Because values and variables can be substituted for each other, we could turn lines ❷ through ❸ into just one line, like this:

```
var sillyString = "hEllo THERE, hOW ARE yOu doING?";  
sillyString[0].toUpperCase() + sillyString.slice(1).toLowerCase();  
"Hello there, how are you doing?"
```

It can be confusing to follow along with code written this way, though, so it's a good idea to use variables for each step of a complicated task like this—at least until you get more comfortable reading this kind of complex code.

BOOLEANS

Now for Booleans. A *Boolean* value is simply a value that's either true or false. For example, here's a simple Boolean expression.

```
var javascriptIsCool = true;  
javascriptIsCool;  
true
```

In this example, we created a new variable called `javascriptIsCool` and assigned the Boolean value `true` to it. On the second line, we get the value of `javascriptIsCool`, which, of course, is `true`!

LOGICAL OPERATORS

Just as you can combine numbers with mathematical operators (+, -, *, /, and so on), you can combine Boolean values with Boolean operators. When you combine Boolean values with Boolean operators, the result will always be another Boolean value (either `true` or `false`).

The three main Boolean operators in JavaScript are `&&`, `||`, and `!`. They may look a bit weird, but with a little practice, they're not hard to use. Let's try them out.

&& (AND)

`&&` means “and.” When reading aloud, people call it “and,” “and-and,” or “ampersand-ampersand.” (*Ampersand* is the name of the character `&`.) Use the `&&` operator with two Boolean values to see if they're *both* true.

For example, before you go to school, you want to make sure that you've had a shower *and* you have your backpack. If both are true, you can go to school, but if one or both are false, you can't leave yet.

```
var hadShower = true;
var hasBackpack = false;
hadShower && hasBackpack;
false
```

Here we set the variable `hadShower` to `true` and the variable `hasBackpack` to `false`. When we enter `hadShower && hasBackpack`, we are basically asking JavaScript, “Are both of these values true?” Since they aren't both true (you don't have your backpack), JavaScript returns `false` (you're not ready for school).

Let's try this again, with both values set to `true`:

```
var hadShower = true;
var hasBackpack = true;
hadShower && hasBackpack;
true
```

Now JavaScript tells us that `hadShower && hasBackpack` is `true`. You're ready for school!



II (OR)

The Boolean operator `||` means “or.” It can be pronounced “or,” or even “or-or,” but some people call it “pipes,” because programmers call the `|` character a *pipe*. You can use this operator with two Boolean values to find out whether *either* one is true.

For example, say you're still getting ready to go to school and you need to take a piece of fruit for lunch, but it doesn't matter whether you take an apple or an orange or both. You can use JavaScript to see whether you have at least one, like this:

```
var hasApple = true;
var hasOrange = false;
```

```
hasApple || hasOrange;  
true
```

`hasApple || hasOrange` will be true if either `hasApple` or `hasOrange` is true, or if both are true. But if *both* are false, the result will be false (you don't have any fruit).

! (NOT)

`!` just means “not.” You can call it “not,” but lots of people call it “bang.” (An exclamation point is sometimes called a *bang*.) Use it to turn false into true or true into false. This is useful for working with values that are opposites. For example:

```
var isWeekend = true;  
var needToShowerToday = !isWeekend;  
needToShowerToday;  
false
```

In this example, we set the variable `isWeekend` to true. Then we set the variable `needToShowerToday` to `!isWeekend`. The bang converts the value to its opposite—so if `isWeekend` is true, then `!isWeekend` is *not* true (it's false). So when we ask for the value of `needToShowerToday`, we get false (you don't need to shower today, because it's the weekend).

Because `needToShowerToday` is false, `!needToShowerToday` will be true:

```
needToShowerToday;  
false  
!needToShowerToday;  
true
```

In other words, it's *true* that you do *not* need to shower today.

COMBINING LOGICAL OPERATORS

Operators get interesting when you start combining them. For example, say you should go to school if it's *not* the weekend *and* you've showered *and* you have an apple *or* you have an orange. We could check whether all of this is true with JavaScript, like this:

```
var isWeekend = false;  
var hadShower = true;  
var hasApple = false;
```

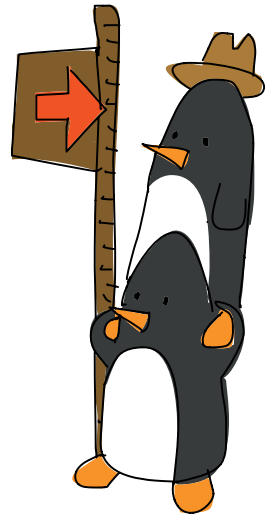
```
var hasOrange = true;  
var shouldGoToSchool = !isWeekend && hadShower && (hasApple || hasOrange);  
shouldGoToSchool;  
true
```

In this case, it's not the weekend, you have showered, and you don't have an apple but you do have an orange—so you should go to school.

`hasApple || hasOrange` is in parentheses because we want to make sure JavaScript works out that bit first. Just as JavaScript calculates `*` before `+` with numbers, it also calculates `&&` before `||` in logical statements.

COMPARING NUMBERS WITH BOOLEANS

Boolean values can be used to answer questions about numbers that have a simple yes or no answer. For example, imagine you're running a theme park and one of the rides has a height restriction: riders must be at least 60 inches tall, or they might fall out! When someone wants to go on the ride and tells you their height, you need to know if it's greater than this height restriction.



GREATER THAN

We can use the greater-than operator (`>`) to see if one number is greater than another. For example, to see if the rider's height (65 inches) is greater than the height restriction (60 inches), we could set the variable `height` equal to 65 and the variable `heightRestriction` equal to 60, and then use `>` to compare the two:

```
var height = 65;  
var heightRestriction = 60;  
height > heightRestriction;  
true
```

With `height > heightRestriction`, we're asking JavaScript to tell us whether the first value is greater than the second. In this case, the rider is tall enough!

What if a rider were exactly 60 inches tall, though?

```
var height = 60;
var heightRestriction = 60;
height > heightRestriction;
false
```

Oh no! The rider isn't tall enough! But if the height restriction is 60, then shouldn't people who are exactly 60 inches be allowed in? We need to fix that. Luckily, JavaScript has another operator, `>=`, which means "greater than or equal to":

```
var height = 60;
var heightRestriction = 60;
height >= heightRestriction;
true
```

Good, that's better—60 *is* greater than or equal to 60.

LESS THAN

The opposite of the greater-than operator (`>`) is the less-than operator (`<`). This operator might come in handy if a ride were designed only for small children. For example, say the rider's height is 60 inches, but riders must be no more than 48 inches tall:

```
var height = 60;
var heightRestriction = 48;
height < heightRestriction;
false
```

We want to know if the rider's height is *less* than the restriction, so we use `<`. Because 60 is not less than 48, we get `false` (someone whose height is 60 inches is too tall for this ride).

And, as you may have guessed, we can also use the operator `<=`, which means "less than or equal to":

```
var height = 48;
var heightRestriction = 48;
height <= heightRestriction;
true
```

Someone who is 48 inches tall is still allowed to go on the ride.

EQUAL TO

To find out if two numbers are exactly the same, use the triple equal sign (`===`), which means “equal to.” But be careful not to confuse `===` with a single equal sign (`=`), because `===` means “are these two numbers equal?” and `=` means “save the value on the right in the variable on the left.” In other words, `===` asks a question, while `=` assigns a value to a variable.



When you use `=`, a variable name has to be on the left and the value you want to save to that variable must be on the right. On the other hand, `===` is just used for comparing two values to see if they’re the same, so it doesn’t matter which value is on which side.

For example, say you’re running a competition with your friends Chico, Harpo, and Groucho to see who can guess your secret number, which is 5. You make it easy on your friends by saying that the number is between 1 and 9, and they start to guess. First you set `mySecretNumber` equal to 5. Your first friend, Chico, guesses that it’s 3 (`chicoGuess`). Let’s see what happens next:

```
var mySecretNumber = 5;
var chicoGuess = 3;
mySecretNumber === chicoGuess;
false
var harpoGuess = 7;
mySecretNumber === harpoGuess;
false
var grouchoGuess = 5;
mySecretNumber === grouchoGuess;
true
```

The variable `mySecretNumber` stores your secret number. The variables `chicoGuess`, `harpoGuess`, and `grouchoGuess` represent your friends’ guesses, and we use `===` to see whether each guess is the same as your secret number. Your third friend, Groucho, wins by guessing 5.

When you compare two numbers with `===`, you get `true` only when both numbers are the same. Because `grouchoGuess` is 5 and `mySecretNumber` is 5, `mySecretNumber === grouchoGuess` returns `true`. The other guesses didn’t match `mySecretNumber`, so they returned `false`.

You can also use `===` to compare two strings or two Booleans. If you use `===` to compare two different types—for example, a string and a number—it will always return `false`.

DOUBLE EQUALS

Now to confuse things a bit: there's another JavaScript operator (double equals, or `==`) that means “equal-ish.” Use this to see whether two values are the same, even if one is a string and the other is a number. All values have some kind of type. So the number 5 is different from the string “5”, even though they basically look like the same thing. If you use `===` to compare the number 5 and the string “5”, JavaScript will tell you they’re not equal. But if you use `==` to compare them, it will tell you they’re the same:

```
var stringNumber = "5";
var actualNumber = 5;
stringNumber === actualNumber;
false
stringNumber == actualNumber;
true
```

At this point, you might be thinking to yourself, “Hmm, it seems much easier to use double equals than triple equals!” You have to be very careful, though, because double equals can be very confusing. For example, do you think 0 is equal to `false`? What about the string “false”? When you use double equals, 0 is equal to `false`, but the string “false” is not:

```
0 == false;
true
"false" == false;
false
```

This is because when JavaScript tries to compare two values with double equals, it first tries to make them the same type. In this case, it converts the Boolean into a number. If you convert Booleans to numbers, `false` becomes 0, and `true` becomes 1. So when you type `0 == false`, you get `true`!

Because of this weirdness, it’s probably safest to just stick with `===` for now.

TRY IT OUT!

You've been asked by the local movie theater managers to implement some JavaScript for a new automated system they're building. They want to be able to work out whether someone is allowed into a PG-13 movie or not.

The rules are, if someone is 13 or over, they're allowed in. If they're not over 13, but they are accompanied by an adult, they're also allowed in. Otherwise, they can't see the movie.

```
var age = 12;  
var accompanied = true;  
???
```

Finish this example using the age and accompanied variables to work out whether this 12-year-old is allowed to see the movie. Try changing the values (for example, set age to 13 and accompanied to false) and see if your code still works out the right answer.



UNDEFINED AND NULL

Finally, we have two values that don't fit any particular mold. They're called undefined and null. They're both used to mean "nothing," but in slightly different ways.

undefined is the value JavaScript uses when it doesn't have a value for something. For example, when you create a new variable, if you don't set its value to anything using the = operator, its value will be set to undefined:

```
var myVariable;  
myVariable;  
undefined
```

The null value is usually used when you want to deliberately say “This is empty.”

```
var myNullVariable = null;  
myNullVariable;  
null
```

At this point, you won’t be using undefined or null very often. You’ll see undefined if you create a variable and don’t set its value, because undefined is what JavaScript will always give you when it doesn’t have a value. It’s not very common to set something to undefined; if you feel the need to set a variable to “nothing,” you should use null instead.

null is used only when you actually want to say something’s not there, which is very occasionally helpful. For example, say you’re using a variable to track what your favorite vegetable is. If you hate all vegetables and don’t have a favorite, you might set the favorite vegetable variable to null.

Setting the variable to null would make it obvious to anyone reading the code that you don’t have a favorite vegetable. If it were undefined, however, someone might just think you hadn’t gotten around to setting a value yet.

WHAT YOU LEARNED

Now you know all the basic data types in JavaScript—numbers, strings, and Booleans—as well as the special values null and undefined. Numbers are used for math-type things, strings are used for text, and Booleans are used for yes or no questions. The values null and undefined are there to give us a way to talk about things that don’t exist.

In the next two chapters, we’ll look at arrays and objects, which are both ways of joining basic types to create more complex collections of values.