**JavaScript for Kids: A Playful Introduction to Programming**

**Nick Morgan**

Published by No Starch Press

To Philly (and Pancake)

**About the Author**

Nick Morgan is a frontend engineer at Twitter. He loves all programming languages but has a particular soft spot for JavaScript. Nick lives in San Francisco (the foggy part) with his fiancée and their fluffy dog, Pancake. He blogs at skilldrick.co.uk.

**About the Illustrator**

Miran Lipovaca is the author of *Learn You a Haskell for Great Good!*. He enjoys boxing, playing bass guitar, and, of course, drawing. He has a fascination with dancing skeletons and the number 71, and when he walks through automatic doors he pretends that he’s actually opening them with his mind.

**About the Technical Reviewer**

Angus Croll is the author of *If Hemingway Wrote JavaScript*, and he is obsessed with JavaScript and literature in equal measure. He works on Twitter’s UI framework team, where he co-authored the Flight framework. He writes the influential *JavaScript, JavaScript* blog and speaks at conferences worldwide. He tweets at @angustweets.

**Acknowledgments**

So many thanks to my wonderful fiancée, Philly, for her encouragement and support during the past 18 months. I truly couldn’t have done it without her. And thanks to Pancake, our dog, for graciously allowing me to use him in my code examples.

Thanks to Angus, without whom I wouldn’t be here, in San Francisco, writing this book. Angus referred me to Twitter back in 2011, and then in 2013 suggested to Bill Pollock that I might be interested in writing this book you’re holding. And to top it all, he agreed to be the technical reviewer, catching a great number of JavaScript faux pas.

Thanks to Bill Pollock, Seph Kramer, Riley Hoffman, Tyler Ortman, and everyone else at No Starch Press, who patiently guided me through the process of writing this book. Special thanks to Bill and Seph for massaging my writing into its current form.

Thanks to the young reviewers River Bradley, Damien Champ, and Alex Chu, who had some great feedback on the early PDFs.

Finally, thanks to Miran Lipovaca. I’ve been a fan of Miran for years — his book *Learn You a Haskell for Great Good* is one of my favorite programming books, and his illustrations for it are amazing. Finding out he’d be illustrating my book was like a dream come true. His pictures for this book are better than I could have imagined, and I’m humbled to have had the chance to work with him.

**Introduction**

Welcome to *JavaScript for Kids*! In this book, you’ll learn to program with JavaScript, the language of the Web. But more than that, you’ll become a programmer — someone who not only *uses* computers but also *controls* them. Once you learn to program, you can bend computers to your will and make them do whatever you want!

JavaScript is a great programming language to learn because it’s used everywhere. Web browsers like Chrome, Firefox, and Internet Explorer all use JavaScript. With the power of JavaScript, web programmers can transform web pages from simple documents into full-blown interactive applications and games.

But you’re not limited to building web pages. JavaScript can run on web servers to create whole websites and can even be used to control robots and other hardware!

**Who Should Read This Book?**

This book is for anyone who wants to learn JavaScript or to start programming for the first time. The book is designed to be kid friendly, but it can serve as a first programming book for beginners of all ages.

With this book, you’ll build up your knowledge of JavaScript gradually, starting with JavaScript’s simple data types, before moving onto complex types, control structures, and functions. After that you’ll learn how to write code that reacts when the user moves the mouse or presses a key on the keyboard. Finally, you’ll learn about the canvas element, which lets you use JavaScript to draw and animate anything you can imagine!

Along the way, you’ll create a few games to stretch your programming skills and put what you’ve learned to good use.

**How to Read This Book**

First off, read it in order! That might sound like a silly thing to say, but lots of people want to jump straight into the fun stuff, like making games. But each chapter is meant to build on what was covered in earlier chapters, so if you begin at the beginning, you’ll have an easier time when you get to the games.

Programming languages are like spoken languages: you have to learn the grammar and the vocabulary, and this takes time. The only way to improve is by writing (and reading) a lot of code. As you write more and more JavaScript, you’ll find certain parts of the language become second nature, and eventually you’ll become a fluent writer of JavaScript.

As you read, I encourage you to type out and test the code examples throughout the book. If you don’t fully understand what’s going on, try making small changes to see what effect they have. If the changes don’t have the effect you expected, see if you can find out why.

Above all, work through the “Try It Out” and Programming Challenges sections. Typing out the code that appears in the book is a good first step, but you’ll understand programming at a deeper level when you start writing your own code. If you find a challenge interesting, then keep at it! Come up with your own challenges to build even more onto the programs you’ve written.

You’ll find sample solutions to the programming challenges (as well as the code files for the games and other examples) at *http://nostarch.com/javascriptforkids/*. Try looking at the solutions after you’ve solved a challenge, so you can compare your approach to mine. Or, if you’re stuck, you can check the solution for hints. But remember that these are just *sample* solutions. There are many, many different ways to accomplish the same goal in JavaScript, so don’t worry if you end up with a completely different solution from mine!

If you come across a word and you don’t know what it means, check the glossary at the back of the book. The glossary contains definitions for many of the programming terms you’ll encounter in this book.

**What’s in this Book?**

**Chapter 1** gives you a quick introduction to JavaScript and gets you started writing JavaScript in Google Chrome.

**Chapter 2** introduces variables and the basic data types used by JavaScript: numbers, strings, and Booleans.

**Chapter 3** is all about arrays, which are used to hold lists of other pieces of data. **Chapter 4** is about objects, which contain pairs of keys and values.

**Chapter 5** is an introduction to HTML, the language used to create web pages.

**Chapter 6** shows you how to gain more control over your code using if statements, for loops, and other control structures.

**Chapter 7** puts together everything you’ve learned so far to create a simple Hangman word-guessing game.

**Chapter 8** shows you how to write your own functions so you can group together and reuse blocks of code.

**Chapter 9** introduces jQuery, a tool that makes it easy to control web pages using JavaScript.

**Chapter 10** shows you how to use timeouts, intervals, and event handlers to make your code more interactive.

**Chapter 11** uses functions, jQuery, and event handlers to create a game called Chapter 11 **Chapter 12** teaches a style of programming called *object-oriented programming*.

**Chapter 13** introduces the canvas element, which allows you to draw graphics on a web page with JavaScript.

**Chapter 14** builds on the animation techniques you learned in Chapter 10 so you can create animations with canvas, and **Chapter 15** shows you how to control those canvas animations with the keyboard.

In **Chapter 16 and Chapter 17**, you’ll program a complete Snake game, using everything you learned in the previous 15 chapters!

The **Afterword** gives you some ideas for how to learn even more about programming. The **Glossary** contains definitions for many of the new words you’ll encounter.

**Have Fun!**

One last thing to remember: Have fun! Programming can be a playful and creative activity, just like drawing or playing a game (in fact, you’ll be drawing and playing games with JavaScript a lot in this book). Once you get the hang of how to write code, the only limit is your imagination. Welcome to the amazing world of computer programming — I hope you have a blast!

**Part I. Fundamentals**

**Chapter 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.

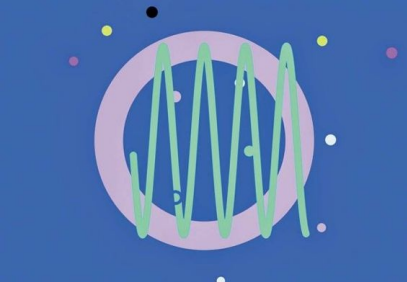
JavaScript lets you play music and create amazing visual effects. For example, you can fly through an interactive music video created by HelloEnjoy for Ellie Goulding’s song “Lights” (*http://lights.helloenjoy.com/*), as shown in Figure 1-1.

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.)

**Writing Some JavaScript**

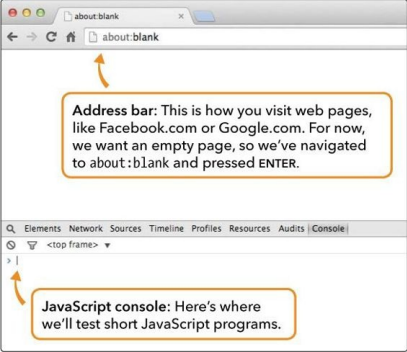
Let’s write a bit of simple JavaScript in Google Chrome (*http://www.google.com/chrome/*). Install Chrome on your computer (if it’s not already installed), and then open it and type **about:blank** in the address bar. Now press ENTER and you’ll see a blank page, like the one in Figure 1-4.

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.



**Chapter 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;

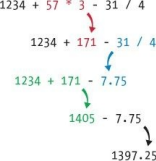
49.38

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. The order of operations: multiplication, division, addition, subtraction*

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.



**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:

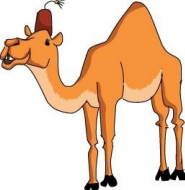
numberOfcandies / numberOfSiblings;

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 ➊, 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;

➊ 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 ➊, we get 86400, which is the number of seconds in a day.

var hoursInADay = 24;

var secondsInADay = secondsInAnHour \* hoursInADay;

➊ 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:

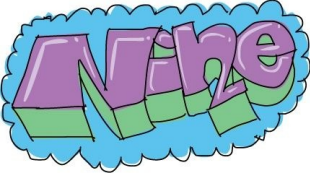
numberNine + numberNine;

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:

➊ var greeting = "Hello ";

var myName = "Nick";

greeting + myName;

"Hello Nick"

The extra space inside the quotes at ➊ 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.



*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.



*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:

➊ var sillyString = "hELlo THERE, hOW ARE yOu doINg?";

➋ var lowerString = sillyString.toLowerCase();

➌ var firstCharacter = lowerString[0];

➍ var firstCharacterUpper = firstCharacter.toUpperCase();

➎ var restOfString = lowerString.slice(1);

➏ firstCharacterUpper + restOfString;

"Hello there, how are you doing?"

Let’s go through this line by line. At ➊, we create a new variable called sillyString and save the string we want to modify to that variable. At ➋, 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 ➌, 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 ➍, we create an uppercase version of firstCharacter ("H") and call that firstCharacterUpper.

At ➎, 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 ➏, 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 ➋ through ➏ 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,” “andand,” 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!

**|| (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.

**Chapter 3. Arrays**

So far we’ve learned about numbers and strings, which are types of data that you can store and use in your programs. But numbers and strings are kind of boring. There’s not a lot that you can do with a string on its own. JavaScript lets you create and group together data in more interesting ways with *arrays*. An array is just a list of other JavaScript data values.

For example, if your friend asked you what your three favorite dinosaurs were, you could create an array with the names of those dinosaurs, in order:

var myTopThreeDinosaurs = ["T-Rex", "Velociraptor", "Stegosaurus"];

So instead of giving your friend three separate strings, you can just use the single array myTopThreeDinosaurs.

**Why Should You Care About Arrays?**

Let’s look at dinosaurs again. Say you want to use a program to keep track of the many kinds of dinosaurs you know about. You could create a variable for each dinosaur, like this:

var dinosaur1 = "T-Rex";

var dinosaur2 = "Velociraptor";

var dinosaur3 = "Stegosaurus";

var dinosaur4 = "Triceratops";

var dinosaur5 = "Brachiosaurus";

var dinosaur6 = "Pteranodon";

var dinosaur7 = "Apatosaurus";

var dinosaur8 = "Diplodocus";

var dinosaur9 = "Compsognathus";

This list is pretty awkward to use, though, because you have nine different variables when you could have just one. Imagine if you were keeping track of 1000 dinosaurs! You’d need to create 1000 separate variables, which would be almost impossible to work with.



It’s like if you had a shopping list, but every item was on a different piece of paper. You’d have one piece of paper that said “eggs,” another piece that said “bread,” and another piece that said “oranges.” Most people would write the full list of things they want to buy on a single piece of paper. Wouldn’t it be much easier if you could group all nine dinosaurs together in just one place?

You can, and that’s where arrays come in.

**Creating an Array**

To create an array, you just use square brackets, []. In fact, an empty array is simply a pair of square brackets, like this:

[];

[]

But who cares about an empty array? Let’s fill it with our dinosaurs!

To create an array with values in it, enter the values, separated by commas, between the square brackets. We can call the individual values in an array *items* or *elements*. In this example, our elements will be strings (the names of our favorite dinosaurs), so we’ll write them with quote marks. We’ll store the array in a variable called dinosaurs:

var dinosaurs = ["T-Rex", "Velociraptor", "Stegosaurus", 

"Triceratops", "Brachiosaurus", "Pteranodon", "Apatosaurus", 

"Diplodocus", "Compsognathus"];

**NOTE**

*Because this is a book and the page is only so wide, we can’t actually fit the whole array on one line. The is to show where we’ve put the code onto an extra line because the page is too narrow. When you type this into your computer, you can type it all on one line.*

Long lists can be hard to read on one line, but luckily that’s not the only way to format (or lay out) an array. You can also format an array with an opening square bracket on one line, the list of items in the array each on a new line, and a closing square bracket, like this:

var dinosaurs = [

"T-Rex",

"Velociraptor",

"Stegosaurus",

"Triceratops",

"Brachiosaurus",

"Pteranodon",

"Apatosaurus",

"Diplodocus",

"Compsognathus"

];

If you want to type this into your browser console, you’ll need to hold down the SHIFT key when you press the ENTER key for each new line. Otherwise the JavaScript interpreter will think you’re trying to execute the current, incomplete, line. While we’re working in the interpreter, it’s easier to write arrays on one line.

Whether you choose to format the items in an array on one line or on separate lines, it’s all the same to JavaScript. However many line breaks you use, JavaScript just sees an array — in this example, an array containing nine strings.

**Accessing an Array’s Elements**

When it’s time to access elements in an array, you use square brackets with the *index* of the element you want, as you can see in the following example:

dinosaurs[0];

"T-Rex"

dinosaurs[3];

"Triceratops"

An *index* is the number that corresponds to (or matches) the spot in the array where a value is stored. Just as with strings, the first element in an array is at index 0, the second is at index 1, the third at index 2, and so on. That’s why asking for index 0 from the dinosaurs array returns "T-Rex" (which is first in the list), and index 3 returns "Triceratops" (which is fourth in the list).

It’s useful to be able to access individual elements from an array. For example, if you just wanted to show someone your absolute favorite dinosaur, you wouldn’t need the whole dinosaurs array. Instead you would just want the first element:

dinosaurs[0];

"T-Rex"



**Setting or Changing Elements in an Array**

You can use indexes in square brackets to set, change, or even add elements to an array. For example, to replace the first element in the dinosaurs array ("T-Rex") with "Tyrannosaurus Rex", you could do this:

dinosaurs[0] = "Tyrannosaurus Rex";

After you’ve done that, the dinosaurs array would look like this:

["Tyrannosaurus Rex", "Velociraptor", "Stegosaurus", "Triceratops",

"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",

"Compsognathus"]

You can also use square brackets with indexes to add new elements to an array. For example, here’s how you could create the dinosaurs array by setting each element individually with square brackets:

var dinosaurs = [];

dinosaurs[0] = "T-Rex";

dinosaurs[1] = "Velociraptor";

dinosaurs[2] = "Stegosaurus";

dinosaurs[3] = "Triceratops";

dinosaurs[4] = "Brachiosaurus";

dinosaurs[5] = "Pteranodon";

dinosaurs[6] = "Apatosaurus";

dinosaurs[7] = "Diplodocus";

dinosaurs[8] = "Compsognathus";

dinosaurs;

["T-Rex", "Velociraptor", "Stegosaurus", "Triceratops",

"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",

"Compsognathus"]

First we create an empty array with var dinosaurs = []. Then, with each following line we add a value to the list with a series of dinosaurs[] entries, from index 0 to index 8. Once we finish the list, we can view the array (by typing dinosaurs;). We see that JavaScript has stored all the names ordered according to the indexes.

You can actually add an element at any index you want. For example, to add a new (made-up) dinosaur at index 33, you could write the following:

dinosaurs[33] = "Philosoraptor";

dinosaurs;

["T-Rex", "Velociraptor", "Stegosaurus", "Triceratops",

"Brachiosaurus", "Pteranodon", "Apatosaurus", "Diplodocus",

"Compsognathus", undefined × 24 "Philosoraptor"]

The elements between indexes 8 and 33 will be undefined. When you output the array, Chrome helpfully tells you how many elements were undefined, rather than listing them all individually.

**Mixing Data Types in an Array**

Array elements don’t all have to be the same type. For example, the next array contains a number (3), a string ("dinosaurs"), an array (["triceratops", "stegosaurus", 3627.5]), and another number (10):

var dinosaursAndNumbers = [3, "dinosaurs", ["triceratops", 

"stegosaurus", 3627.5], 10];

To access an individual element in this array’s inner array, you would just use a second set of square brackets. For example, while dinosaursAndNumbers[2]; returns the entire inner array, dinosaursAndNumbers[2][0]; returns only the first element of that inner array, which is "triceratops".

dinosaursAndNumbers[2];

["triceratops", "stegosaurus", 3627.5]

dinosaursAndNumbers[2][0];

"triceratops"

When we type dinosaursAndNumbers[2][0];, we tell JavaScript to look at index 2 of the array dinosaursAndNumbers, which contains the array ["triceratops", "stegosaurus", 3627.5], and to return the value at index 0 of that second array. Index 0 is the first value of the second array, which is "triceratops". Figure 3-1 shows the index positions for this array.



*Figure 3-1. The index positions of the main array are labeled in red, and the indexes of the inner array are labeled in blue.*

**Working with Arrays**

*Properties* and *methods* help you work with arrays. Properties generally tell you something about the array, and methods usually do something to change the array or return a new array. Let’s have a look.

**Finding the Length of an Array**

Sometimes it’s useful to know how many elements there are in an array. For example, if you kept adding dinosaurs to your dinosaurs array, you might forget how many dinosaurs you have.

The length property of an array tells you how many elements there are in the array. To find the length of an array, just add .length to the end of its name. Let’s try it out. First we’ll make a new array with three elements:

var maniacs = ["Yakko", "Wakko", "Dot"];

maniacs[0];

"Yakko"

maniacs[1];

"Wakko"

maniacs[2];

"Dot"

To find the length of the array, add .length to maniacs:

maniacs.length;

3

JavaScript tells us that there are 3 elements in the array, and we already know they have the index positions 0, 1, and 2. This gives us a useful piece of information: the last index in an array is always the same number as the length of the array minus 1. This means that there is an easy way to access the last element in an array, however long that array is:

maniacs[maniacs.length - 1];

"Dot"

Here, we’re asking JavaScript for an element from our array. But instead of entering an index number in the square brackets, we use a little bit of math: the length of the array minus 1. JavaScript finds maniacs.length, gets 3, and then subtracts 1 to get 2. Then it returns the element from index 2 — the last maniac in the array, "Dot".

**Adding Elements to an Array**

To add an element to the end of an array, you can use the push method. Add .push to the array name, followed by the element you want to add inside parentheses, like this:

var animals = [];

animals.push("Cat");

1

animals.push("Dog");

2

animals.push("Llama");

3

animals;

["Cat", "Dog", "Llama"]

animals.length;

3

Here we create an empty array with var animals = [];, and then use the push method to add "Cat" to the array. Then, we use push again to add on "Dog" and then "Llama". When we display animals;, we see that "Cat", "Dog", and "Llama" were added to the array, in the same order we entered them.



The act of running a method in computer-speak is known as *calling* the method. When you call the push method, two things happen. First, the element in parentheses is added to the array. Second, the new length of the array is returned. That’s why you see those numbers printed out every time you call push.

To add an element to the beginning of an array, you can use .unshift(*element*), like this:

animals;

["Cat", "Dog", "Llama"]

➊ animals[0];

"Cat"

animals.unshift("Monkey");

4

animals;

["Monkey", "Cat", "Dog", "Llama"]

animals.unshift("Polar Bear");

5

animals;

["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]

animals[0];

"Polar Bear"

➋ animals[2];

"Cat"

Here we started with the array that we’ve been using, ["Cat", "Dog", "Llama"]. Then, as we add the elements "Monkey" and "Polar Bear" to the beginning of the array with unshift, the old values get pushed along by one index each time. So "Cat", which was originally at index 0 ➊, is now at index 2 ➋.



Again, unshift returns the new length of the array each time it is called, just like push.

**Removing Elements from an Array**

To remove the last element from an array, you can pop it off by adding .pop() to the end of the array name. The pop method can be particularly handy because it does two things: it removes the last element, *and* it returns that last element as a value. For example, let’s start with our animals array, ["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]. Then we’ll create a new variable called lastAnimal and save the last animal into it by calling animals.pop().

animals;

["Polar Bear", "Monkey", "Cat", "Dog", "Llama"]

➊ var lastAnimal = animals.pop(); lastAnimal;

"Llama"

animals;

["Polar Bear", "Monkey", "Cat", "Dog"]

➋ animals.pop();

"Dog"

animals;

["Polar Bear", "Monkey", "Cat"]

➌ animals.unshift(lastAnimal);

4

animals;

["Llama", "Polar Bear", "Monkey", "Cat"]

When we call animals.pop() at ➊, the last item in the animals array, "Llama", is returned and saved in the variable lastAnimal. "Llama" is also removed from the array, which leaves us with four animals. When we call animals.pop() again at ➋, "Dog" is removed from the array and returned, leaving only three animals in the array.

When we used animal.pop() on "Dog", we didn’t save it into a variable, so that value isn’t saved anywhere anymore. The "Llama", on the other hand, was saved to the variable lastAnimal, so we can use it again whenever we need it. At ➌, we use unshift(lastAnimal) to add "Llama" back onto the front of the array. This gives us a final array of ["Llama", "Polar Bear", "Monkey", "Cat"].

Pushing and popping are a useful pair because sometimes you care about only the end of an array. You can push a new item onto the array and then pop it off when you’re ready to use it. We’ll look at some ways to use pushing and popping later in this chapter.



To remove and return the first element of an array, use .shift():

animals;

["Llama", "Polar Bear", "Monkey", "Cat"]

var firstAnimal = animals.shift();

firstAnimal;

"Llama"

animals;

["Polar Bear", "Monkey", "Cat"]

animals.shift() does the same thing as animals.pop(), but the element comes off the beginning instead. At the start of this example, animals is ["Llama", "Polar Bear", "Monkey", "Cat"].

When we call .shift() on the array, the first element, "Llama", is returned and saved in firstAnimal. Because .shift() removes the first element as well as returning it, at the end animals is just ["Polar Bear", "Monkey", "Cat"].

You can use unshift and shift to add and remove items from the beginning of an array just as you’d use push and pop to add and remove items from the end of an array.



**Adding Arrays**

To add two arrays together to make a new, single array, you can use

*firstArray*.concat(*otherArray*). The term concat is short for *concatenate*, a fancy computer science word for joining two values together. The concat method will combine both arrays into a new array, with the values from *firstArray* added in front of those from *otherArray*.

For example, say we have a list of some furry animals and another list of some scaly animals, and we want to combine them. If we put all of our furry animals in an array called furryAnimals and all of our scaly animals in an array called scalyAnimals, entering furryAnimals.concat(scalyAnimals) will create a new array that has the values from the first array at the beginning and the values from the second array at the end.



var furryAnimals = ["Alpaca", "Ring-tailed Lemur", "Yeti"];

var scalyAnimals = ["Boa Constrictor", "Godzilla"];

var furryAndScalyAnimals = furryAnimals.concat(scalyAnimals);

furryAndScalyAnimals;

["Alpaca", "Ring-tailed Lemur", "Yeti", "Boa Constrictor", "Godzilla"]

furryAnimals;

["Alpaca", "Ring-tailed Lemur", "Yeti"]

scalyAnimals;

["Boa Constrictor", "Godzilla"]

Even though *firstArray*.concat(*otherArray*) returns an array containing all the elements from *firstArray* and *secondArray*, neither of the original arrays is changed. When we look at furryAnimals and scalyAnimals, they’re the same as when we created them.

**Joining Multiple Arrays**

You can use concat to join more than two arrays together. Just put the extra arrays inside the parentheses, separated by commas:

var furryAnimals = ["Alpaca", "Ring-tailed Lemur", "Yeti"];

var scalyAnimals = ["Boa Constrictor", "Godzilla"];

var featheredAnimals = ["Macaw", "Dodo"];

var allAnimals = furryAnimals.concat(scalyAnimals, featheredAnimals);

allAnimals;

["Alpaca", "Ring-tailed Lemur", "Yeti", "Boa Constrictor", "Godzilla",

"Macaw", "Dodo"]

Here the values from featheredAnimals get added to the very end of the new array, since they are listed last in the parentheses after the concat method.

concat is useful when you have multiple arrays that you want to combine into one. For example, say you have a list of your favorite books, and your friend also has a list of favorite books, and you want to go see if the books are available to buy all at once at the bookstore. It would be easier if you had only

onelistofbooks. Allyou’dhavetodoisconcatyourlistwithyourfriend’s,andvoilà! Onelistof books.

**Finding the Index of an Element in an Array**

To find the index of an element in an array, use .indexOf(*element*). Here we define the array colors and then ask for the index positions of "blue" and "green" with colors.indexOf("blue") and colors.indexOf("green"). Because the index of "blue" in the array is 2, colors.indexOf("blue") returns 2. The index of "green" in the array is 1, so colors.indexOf("green") returns 1.

var colors = ["red", "green", "blue"];

colors.indexOf("blue");

2

colors.indexOf("green");

1

indexOf is like the reverse of using square brackets to get a value at a particular index; colors[2] is "blue", so colors.indexOf("blue") is 2:

colors[2];

"blue"

colors.indexOf("blue");

2

Even though "blue" appears third in the array, its index position is 2 because we always start counting from 0. And the same goes for "green", of course, at index 1.

If the element whose position you ask for is not in the array, JavaScript returns -1.

colors.indexOf("purple");

-1

This is JavaScript’s way of saying “That doesn’t exist here,” while still returning a number.

If the element appears more than once in the array, the indexOf method will return the first index of that element in the array.

var insects = ["Bee", "Ant", "Bee", "Bee", "Ant"];

insects.indexOf("Bee");

0

**Turning an Array into a String**

You can use .join() to join all the elements in an array together into one big string.

var boringAnimals = ["Monkey", "Cat", "Fish", "Lizard"];

boringAnimals.join();

"Monkey,Cat,Fish,Lizard"

When you call the join method on an array, it returns a string containing all the elements, separated by commas. But what if you don’t want to use commas as the separator?



You can use .join(*separator*) to do the same thing, but with your own chosen separator between each value. The separator is whatever string you put inside the parentheses. For example, we can use three different separators: a hyphen with spaces on either side, an asterisk, and the word *sees* with spaces on either side. Notice that you need quote marks around the separator, because the separator is a string.

var boringAnimals = ["Monkey", "Cat", "Fish", "Lizard"];

boringAnimals.join(" - ");

"Monkey - Cat - Fish - Lizard"

boringAnimals.join("\*")

"Monkey\*Cat\*Fish\*Lizard"

boringAnimals.join(" sees ")

"Monkey sees Cat sees Fish sees Lizard"

This is useful if you have an array that you want to turn into a string. Say you have lots of middle names and you’ve got them stored in an array, along with your first and last name. You might be asked to give your full name as a string. Using join, with a single space as the separator, will join all your names together into a single string:

var myNames = ["Nicholas", "Andrew", "Maxwell", "Morgan"];

myNames.join(" ");

"Nicholas Andrew Maxwell Morgan"

If you didn’t have join, you’d have to do something like this, which would be really annoying to type out:

myNames[0] + " " + myNames[1] + " " + myNames[2] + " " + myNames[3];

"Nicholas Andrew Maxwell Morgan"

Also, this code would work only if you had exactly two middle names. If you had one or three middle names, you’d have to change the code. With join, you don’t have to change anything — it prints out a string with all of the elements of the array, no matter how long the array is.

If the values in the array aren’t strings, JavaScript will convert them to strings before joining them together:

var ages = [11, 14, 79];

ages.join(" ");

"111479"

**Useful Things to Do with Arrays**

Now you know lots of different ways to create arrays and play around with them. But what can you actually do with all these properties and methods? In this section, we’ll write a few short programs that show off some useful things to do with arrays.

**Finding Your Way Home**

Picture this: your friend has come over to your house. Now she wants to show you her house. The only problem is that you’ve never been to her house before, and later you’ll have to find your way back home on your own.

Luckily, you have a clever idea to help you with your problem: on the way to your friend’s house, you’ll keep a list of all the landmarks you see. On the way back, you’ll go through the list in reverse and check items off the end of the list every time you pass a landmark so you know where to go next.

**Building the Array with Push**

Let’s write some code that would do exactly that. We start off by creating an empty array. The array starts off empty because you don’t know what landmarks you’ll see until you actually start walking to your friend’s house. Then, for each landmark on the way to your friend’s house, we’ll push a description of that landmark onto the end of the array. Then, when it’s time to go home, we’ll pop each landmark off the array.



var landmarks = [];

landmarks.push("My house");

landmarks.push("Front path");

landmarks.push("Flickering streetlamp");

landmarks.push("Leaky fire hydrant");

landmarks.push("Fire station");

landmarks.push("Cat rescue center");

landmarks.push("My old school");

landmarks.push("My friend's house");

Here we create an empty array named landmarks and then use push to store all the landmarks you pass on the way to your friend’s house.

**Going in Reverse with pop**

Once you arrive at your friend’s house, you can inspect your array of landmarks. Sure enough, the first item is "My house", followed by "Front path", and so on through the end of the array, with the final item "My friend's house". When it’s time to go home, all you need to do is pop off the items one by one, and you’ll know where to go next.

landmarks.pop();

"My friend's house"

landmarks.pop();

"My old school"

landmarks.pop();

"Cat rescue center"

landmarks.pop();

"Fire station"

landmarks.pop();

"Leaky fire hydrant"

landmarks.pop();

"Flickering streetlamp"

landmarks.pop();

"Front path"

landmarks.pop();

"My house"

Phew, you made it home!

Did you notice how the first landmark you put in the array was also the last one you got out of it? And the last landmark you put in the array was the first one that came out? You might have thought that you’d always want the first item you put in to be the first item you get out, but you can see that it’s sometimes helpful to go back through an array in reverse.



It’s actually very common to use a process like this in larger programs, which is why JavaScript makes pushing and popping so easy.

**NOTE**

*This technique is known as a* stack *in computer-speak. Think of it like a stack of pancakes. Every time you cook a new pancake, it goes on top (like push), and every time you eat one, it comes off the top (like pop). Popping a stack is like going back in time: the last item you pop is the first one you pushed. It’s the same with pancakes: the last pancake you eat is the first one that was cooked. In programming jargon, this is also called* Last In, First Out (LIFO). *The alternative to LIFO is* First In, First Out (FIFO). *This is also known as a* queue, *because it acts like a queue (or line) of people. The first person to join the queue is the first person to be served.*

**Decision Maker**

We can use arrays in JavaScript to build a program to make decisions for us (like a Magic 8-Ball). First, though, we need to find out how to get random numbers.

**Using Math.random()**

We can produce random numbers using a special method called Math.random(), which returns a random number between 0 and 1 each time it’s called. Here’s an example:

Math.random();

0.8945409457664937

Math.random();

0.3697543195448816

Math.random();

0.48314980138093233

It’s important to note that Math.random() always returns a number *less than* 1 and will never return 1 itself.

If you want a bigger number, just multiply the result of calling Math.random(). For example, if you wanted numbers between 0 and 10, you would multiply Math.random() by 10:

Math.random() \* 10;

7.648027329705656

Math.random() \* 10;

9.7565904534421861

Math.random() \* 10;

0.21483442978933454

**Rounding Down with Math.floor()**

We can’t use these numbers as array indexes, though, because indexes have to be whole numbers with nothing after the decimal point. To fix that, we need another method called Math.floor(). This takes a number and rounds it down to the whole number below it (basically getting rid of everything after the decimal point).

Math.floor(3.7463463);

3

Math.floor(9.9999);

9

Math.floor(0.793423451963426);

0

We can combine these two techniques to create a random index. All we need to do is multiply Math.random() by the length of the array and then call Math.floor() on that value. For example, if the length of the array were 4, we would do this:

Math.floor(Math.random() \* 4);

2 // could be 0, 1, 2, or 3

Every time you call the code above, it returns a random number from 0 to 3 (including 0 and 3). Because Math.random() always returns a value less than 1, Math.random() \* 4 will never return 4 or anything higher than 4.

Now, if we use that random number as an index, we can select a random element from an array:

var randomWords = ["Explosion", "Cave", "Princess", "Pen"];

var randomIndex = Math.floor(Math.random() \* 4);

randomWords[randomIndex];

"Cave"

Here we use Math.floor(Math.random() \* 4); to pick a random number from 0 to 3. Once that

random number is saved to the variable randomIndex, we use it as an index to ask for a string from the array randomWords.

In fact, we could shorten this by doing away with the randomIndex variable altogether and just say:

randomWords[Math.floor(Math.random() \* 4)];

"Princess"

**The Complete Decision Maker**

Now let’s create our array of phrases, and we can use this code to pick a random one. This is our decision maker! I’m using comments here to show some questions you might want to ask your computer.

var phrases = [

"That sounds good",

"Yes, you should definitely do that",

"I'm not sure that's a great idea",

"Maybe not today?",

"Computer says no."

];

// Should I have another milkshake?

phrases[Math.floor(Math.random() \* 5)];

"I'm not sure that's a great idea"

// Should I do my homework?

phrases[Math.floor(Math.random() \* 5)];

"Maybe not today?"

Here we created an array called phrases that stores different pieces of advice. Now, every time we have a question, we can ask for a random value from the phrases array, and it will help us make a decision!

Notice that because our array of decisions has five items, we multiply Math.random() by 5. This will always return one of five index positions: 0, 1, 2, 3, or 4.

**Creating a Random Insult Generator**

We can extend the decision maker example to create a program that generates a random insult every time you run it!

var randomBodyParts = ["Face", "Nose", "Hair"];

var randomAdjectives = ["Smelly", "Boring", "Stupid"];

var randomWords = ["Fly", "Marmot", "Stick", "Monkey", "Rat"];

// Pick a random body part from the randomBodyParts array:

➊ var randomBodyPart = randomBodyParts[Math.floor(Math.random() \* 3)];

// Pick a random adjective from the randomAdjectives array:

➋ var randomAdjective = randomAdjectives[Math.floor(Math.random() \* 3)];

// Pick a random word from the randomWords array:

➌ var randomWord = randomWords[Math.floor(Math.random() \* 5)];

// Join all the random strings into a sentence:

var randomInsult = "Your " + randomBodyPart + " is like a " + 

randomAdjective + " " + randomWord + "!!!";

randomInsult;

"Your Nose is like a Stupid Marmot!!!"



Here we have three arrays, and in lines ➊, ➋, and ➌, we use three indexes to pull a random word from each array. Then, we combine them all in the variable randomInsult to create a complete insult. At ➊ and ➋ we’re multiplying by 3 because randomAdjectives and randomBodyParts both contain three elements. Likewise, we’re multiplying by 5 at ➌ because randomWords is five elements long. Notice that we add a string with a single space between randomAdjective and randomWord. Try running this code a few times — you should get a different random insult each time!

**TRY IT OUT!**

If you wanted to be really clever, you could replace line ➌ with this:

var randomWord = randomWords[Math.floor(Math.random() \* 

randomWords.length)];

We know that we always need to multiply Math.random() by the length of the array, so using randomWords.length means we don’t have to change our code if the length of the array changes.

Here’s another way to build up our random insult:

var randomInsult = ["Your", randomBodyPart, "is", "like", "a", 

randomAdjective, randomWord + "!!!"].join(" ");

"Your Hair is like a Smelly Fly!!!"

In this example, each word of the sentence is a separate string in an array, which we join with the space character. There’s only one place where we *don’t* want a space, which is in between randomWord and "!!!". In this case, we use the + operator to join those two strings without the space.

**What You Learned**

As you’ve seen, JavaScript arrays are a way to store a list of values. Now you know how to create and work with arrays, and you have many ways of accessing their elements.

Arrays are one of the ways JavaScript gives you to bring multiple values together into one place. In the next chapter, we’ll look at objects, which are another way of storing multiple values as a single unit. Objects use *string keys* to access the elements, rather than number indexes.

**Programming Challenges**

Try out these challenges to practice the skills you learned in this chapter.

**#1: NEW INSULTS**

Make your own random insult generator with your own set of words.

**#2: MORE SOPHISTICATED INSULTS**

Extend the random insult generator so it generates insults like “Your [body part] is more [adjective] than a [animal]’s [animal body part].” (Hint: You’ll need to create another array.)

**#3: USE + OR JOIN?**

Make two versions of your random insult generator: one that uses the + operator to create the string, and one that creates an array and joins it with " ". Which do you prefer, and why?

**#4: JOINING NUMBERS**

How could you turn the array [3, 2, 1] into the string "3 is bigger than 2 is bigger than 1" using the join method?

**Chapter 4. Objects**

Objects in JavaScript are very similar to arrays, but objects use strings instead of numbers to access the different elements. The strings are called *keys* or *properties*, and the elements they point to are called *values*. Together these pieces of information are called *key-value pairs*. While arrays are mostly used to represent lists of multiple things, objects are often used to represent single things with multiple characteristics, or *attributes*. For example, in Chapter 3 we made several arrays that listed different animal names. But what if we wanted to store different pieces of information about one animal?

**Creating Objects**

We could store lots of information about a single animal by creating a JavaScript object. Here’s an object that stores information about a three-legged cat named Harmony.

var cat = {

"legs": 3,

"name": "Harmony",

"color": "Tortoiseshell"

};

Here we create a variable called cat and assign an object to it with three key-value pairs. To create an object, we use curly brackets, {}, instead of the straight brackets we used to make arrays. In between the curly brackets, we enter key-value pairs. The curly brackets and everything in between them are called an *object literal*. An object literal is a way of creating an object by writing out the entire object at once.



**NOTE**

*We’ve also seen array literals (for example, ["a", "b", "c"]), number literals (for example, 37), string literals (for example, "moose"), and Boolean literals (true and false).* Literal *just means that the whole value is written out at once, not built up in multiple steps.*

*For example, if you wanted to make an array with the numbers 1 through 3 in it, you could use the array literal [1, 2, 3]. Or you could create an empty array and then use the push method to add 1, 2, and 3 to the array. You don’t always know at first what’s going to be in your array or object, which is why you can’t always use literals to build arrays and objects.*

Figure 4-1 shows the basic syntax for creating an object.



*Figure 4-1. The general syntax for creating an object*

When you create an object, the key goes before the colon (:), and the value goes after. The colon acts a bit like an equal sign — the values on the right get assigned to the names on the left, just like when you create variables. In between each key-value pair, you have to put a comma. In our example, the commas

are at the ends of the lines — but notice that you don’t need a comma after the last key-value pair (color: "Tortoiseshell"). Because it’s the last key-value pair, the closing curly bracket comes next, instead of a comma.

**Keys Without Quotes**

In our first object, we put each key in quotation marks, but you don’t necessarily need quotes around the keys — this is a valid cat object literal as well:

var cat = {

legs: 3,

name: "Harmony",

color: "Tortoiseshell"

};

JavaScript knows that the keys will always be strings, which is why you can leave out the quotes. If you don’t put quotes around the keys, the unquoted keys have to follow the same rules as variable names: spaces aren’t allowed in an unquoted key, for example. If you put the key in quotes, then spaces are allowed:

var cat = {

legs: 3,

"full name": "Harmony Philomena Snuggly-Pants Morgan",

color: "Tortoiseshell"

};

Note that, while a key is always a string (with or without quotes), the value for that key can be any kind of value, or even a variable containing a value.



You can also put the whole object on one line, but it can be harder to read like that: var cat = { legs: 3, name: "Harmony", color: "Tortoiseshell" };

**Accessing Values in Objects**

You can access values in objects using square brackets, just like with arrays. The only difference is that instead of the index (a number), you use the key (a string).

cat["name"];

"Harmony"

Just as the quotes around keys are optional when you create an object literal, the quotes are also optional when you are accessing keys in objects. If you’re not going to use quotes, however, the code looks a bit different:

cat.name;

"Harmony"

This style is called *dot notation*. Instead of typing the key name in quotes inside square brackets after the object name, we just use a period, followed by the key, without any quotes. As with unquoted keys in object literals, this will work only if the key doesn’t contain any special characters, such as spaces.

Instead of looking up a value by typing its key, say you wanted to get a list of all the keys in an object. JavaScript gives you an easy way to do that, using Object.keys():

var dog = { name: "Pancake", age: 6, color: "white", bark: "Yip yap 

yip!" };

var cat = { name: "Harmony", age: 8, color: "tortoiseshell" };

Object.keys(dog);

["name", "age", "color", "bark"]

Object.keys(cat);

["name", "age", "color"]

Object.keys(*anyObject*) returns an array containing all the keys of *anyObject*.