ROCKABLE

Includes Screen Casts!

GETTING GOOD WITH JAVASCRIPT

Andrew Burgess

ROCKABLE

Rockablepress.com
Envato.com

© Rockable Press 2011

All rights reserved. No part of this publication may be reproduced or redistributed in any form without the prior written permission of the publishers.

Prepared exclusively for admin@wedoev.com

Contents

Getting Started 6

Aren't There Already Enough JavaScript Books? 6
What Will I Learn? 7
Where Did JavaScript Come From? 7
How Do I Get Started? 8
Conventions Used in This Book 11
Summary 13

The Basics 15

Variables 15
Types 16
Semicolons 23
Comments 24
Operators 25
Conditional Statements 34
Looping Statements 40
Summary 44

More Basics 46

Functions 46
Type Methods 58
Summary 72

More Concepts and Best Practices 74

this 74
Object Oriented JavaScript 83
Object Methods 87
Closure 90
Errors 94

Prepared exclusively for admin@wedoev.com

Testing your JavaScript 97
Organizing your JavaScript 108
Optimizing your JavaScript 112
Summary 113

Working with HTML **115**

Kids, Meet the DOM 115
Nodes 117
Finding Elements 118
Traversing the DOM 120
Adding, Removing, and Modifying Elements 125
Events 131
The DOM, In Sum 137
Wrapping Up 139

Appendix A: Further Study **141**

Appendix B: What We Didn't Cover **143**

About The Author **144**

Your Screencasts **145**

Prepared exclusively for admin@wedoev.com

Prepared exclusively for admin@wedoev.com

Getting Started

Thanks for buying "Getting Good with JavaScript." I'm pretty sure you've made the right decision (if *I'm* allowed to say that). This introductory chapter will get you acquainted the subject under the microscope. Let's roll!

Aren't There Already Enough JavaScript Books?

There's a pretty good chance that you rolled your eyes when you first saw this book. If you've followed the online development community for any time at all, you'll know that JavaScript is a pretty hot topic. Everybody is writing about it. So why *yet another* book on JavaScript?

Well, yet another JavaScript book because I hope this book fills a niche. From the beginning, I wanted this book to be one that would take someone who knows little or nothing about JavaScript and get them up to speed in very little time…while only teaching current methods and best practices. This means that there's much more to JavaScript than you'll find in this book. There are many things you really don't need to know when you get started; I won't be spending time on them. I've seen other beginner books that cover things I've left out, and that's okay, because they're aiming for comprehensiveness. But not in this book: here, you'll only learn what you'll actually use as you dip your toes into JavaScript. That other stuff can wait.

Since I've tried to keep this book short enough to read in a weekend (well, maybe a long weekend), many of the things we'll look at aren't exhaustively covered. This doesn't mean I'll skim over things. It just means that there are more advanced techniques that aren't important for you to learn at this time. Don't worry: you'll

Prepared exclusively for admin@wedoev.com

get a solid grounding in what you need to know to get you off the ground. Then, check out some of the resources in the Appendices for further study.

What Will I Learn?

Before we get started, let's talk about what this book covers. In this chapter, you'll get acquainted with what JavaScript is and where it came from. We'll also talk about how to set up an environment for coding JavaScript.

After that, we'll cover the basics in Chapters 2 and 3. You'll learn all about the core syntax: what to actually *type*. Chapter 4 will cover many best practices and important concepts for writing JavaScript. Then, in Chapter 5, I'll introduce you to the basics of writing JavaScript that interacts with web pages.

Where Did JavaScript Come From?

JavaScript was developed by a gentleman named Brendan Eich, who was working for Netscape (a now non-existent web browser maker) at the time—he built the whole language in less than two weeks. Originally, it was called *Mocha*, then *LiveScript*, and finally *JavaScript*. The first browser to support JavaScript was Netscape Navigator 2.0, way back in late 1995.

For a long time, JavaScript wasn't considered to be much of a language. The "big idea" on the web was

ROCK* TIP

It's important to note that JavaScript IS NOT Java. There are really only two things that JavaScript and Java have in common. First and most obvious is the word "Java" (which was used to make JavaScript attractive to Java developers; I know, really). The only other things is a bit of similar syntax: both languages use C-style curly braces. That's really all.

Prepared exclusively for admin@wedoev.com

supposed to be Java Applets. However, they failed, and eventually, JavaScript's "Good Parts" became well-known and soon even loved by many people.

Now, JavaScript is the most used programming language in the world. It's used not only in web browsers, but also desktop apps, mobile phone apps, and now even on the server. You definitely won't be wasting your time learning JavaScript.

How Do I Get Started?

Now that you're pumped to learn JavaScript, how do we get started? Well, you'll learn how to actually code JavaScript in the following chapters. But where is that code going to go?

Although it's hard to find a place where JavaScript *won't* run, the overwhelming majority of JavaScript—especially the JavaScript you'll be writing for now—will be in web pages, to run in the browser. There are two ways to get JavaScript into your HTML page.

First, you can embed your JavaScript code right inside `script` tags.

```
<script>
    alert("I'm learning JavaScript");
</script>
```

You don't have to understand the code in the tags for now; just know that it works. When your browser encounters a script tag, it will interpret the code that you've written and execute it (if appropriate).

Prepared exclusively for admin@wedoev.com

The second way to get JavaScript onto your page is to link to a .js file:

```
<script src="somefile.js"></script>
```

Yes: unfortunately that closing script tag is required, even though you don't put anything inside it. As you might guess, the `src` attribute is short for "source." This will download the JavaScript file and process it, just like the inline code.

Two more notes about including scripts:

- For the most part, you'll want to include your `script` tags right before your closing body tag. Actually, you can put them anywhere you want, but it's best for performance reasons to put it at the end of the body…or write your code to wait until the page has completed loading before it begins executing. We'll see how to do this later on (and no, you can't jump ahead to "Another Short Rabbit-Trail on Script Loading" on page 136).
- You might see code in which the `script` tags have this attribute: `type="text/javascript"`. This used to be required, but isn't in HTML5 (or course, you should then be using an HTML5 doctype).

Firebug

As you learn about JavaScript, you'll find the Firefox plugin Firebug invaluable. Go to that link and install it, right after you install the latest version of Firefox.

After installing Firebug, you'll have a little bug icon in the lower right corner of the status bar of your Firefox window; if it's coloured, that means it's enabled for the website you're on; otherwise it will be black and white.

Prepared exclusively for admin@wedoev.com

Fig. 1-1. Firebug activated indicator

At least, that's where it's been for a long time. If you're on Firefox 4, however, it's now on the toolbar:

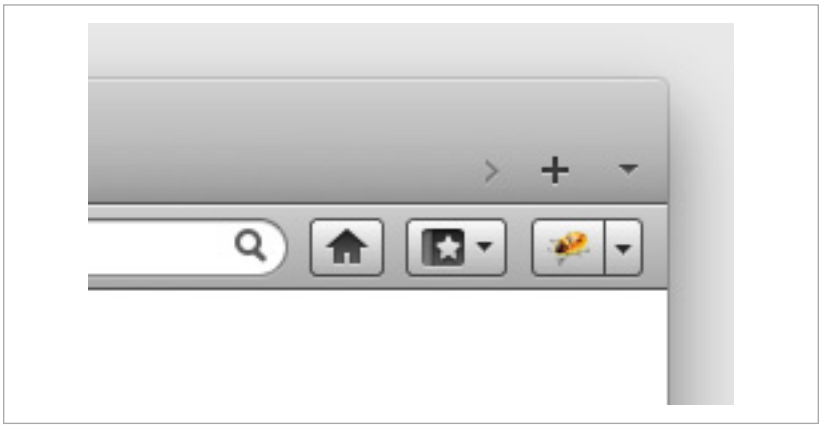

Fig. 1-2. Firebug activated indicator in Firefox 4

To enable Firebug on a website, just click that icon. When you click on this, the Firebug panel will pop up. It'll look something like this:

Prepared exclusively for admin@wedoev.com

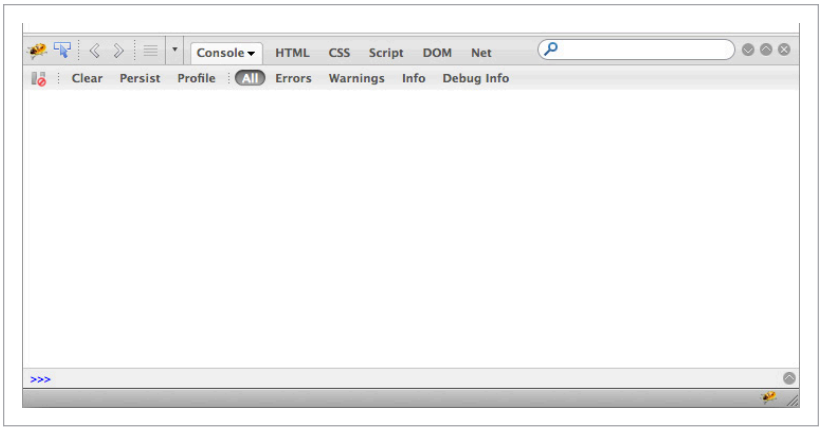

Fig. 1-3. The Firebug panel

We could write a whole tutorial on Firebug, but the panel you'll find useful in this book is the console panel. You can find it by clicking the console tab (Firebug should open to it by default; it's pictured above).

At the bottom of the panel, you'll see a prompt (»>) where you can type some JavaScript. This is like the command line, where you type in some code, hit *return* (or *enter*), and you'll get the result. If you need to type in more than one line of code at a time, press the upward-facing arrow at the end of the line; you'll be able to type in more than one line, and then click the *Run* button at the bottom of the expanded panel to execute it.

Conventions Used in This Book

One more administrative note before we continue: many of the code examples in the book will look like this:

Example 1.1

```
alert("Hello, reader"); // Hello, reader
```

Prepared exclusively for admin@wedoev.com

See how there's a header on this code snippet: "Example 1.1"? All code snippets with headers have an accompanying file—called something like 'example_1_1.html'—that you can find in the 'code' folder which you've already downloaded with the book. I want you to run these, but even better, I want you to open them up and change the code, mess around, and learn why some things work and others don't.

If a snippet doesn't have a title, it's just a quick example that needs to be shown in code, but won't give you any result if you execute it by itself.

You won't fully understand this now, but I need to explain it, and you'll understand it soon: in the snippet above, I've used the **`alert`** function to show the result of the code; this function will pop up an alert box, with the text we give it. Notice the two backslashes at the end of the line: those denote a comment in the code (more on this soon); in our case, I've put the result you should see in the alert box in the comment, so you can compare results.

Sometimes, I'll use **`console.log`** instead of **`alert`**; instead of popping up an alert box, this will write the information to the Firebug console. This the more popular way of getting output when writing JavaScript. We'll use both, so that you're comfortable with both.

Example 1.2

```
console.log("Hi there");  // Hi there
console.log(1 + 4); // 5
```

Open the 'Example 1.2' file in Firefox, and open Firebug (actually you'll have to refresh the page after you open Firebug, because nothing will log if Firebug isn't enabled for that page). You should see two lines in the console, one saying "Hi there", and the other saying "5."

Prepared exclusively for admin@wedoev.com

Sometimes the code will be longer than can fit on the width of this book's page. If so, there will be a continuation marker (▸ or ▹) at the end of the line. This means the text on the following line is intended to be typed together with the previous line as one. For example, the following is all typed as one line, with no break:

```
msg = "the time is " + hour + " o'clock"; // addition ▹
operator used on strings
```

A hollow arrow indicates that a space is permissible between the character as the end of the starting line and the first character of the second; a solid arrow indicates that there should be no space. The marker itself is not typed in.

Summary

I hope you're getting excited, because I know I am! We're about to undertake a journey that will change your life forever (well, at least it could). Let's swing into Chapter 2 to begin talking about the basics of JavaScript.

Prepared exclusively for admin@wedoev.com

2

Prepared exclusively for admin@wedoev.com

The Basics

Here's where we actually get down and dirty; in this chapter, we're going to cover the nuts and bolts of JavaScript: all the little seemingly-unrelated pieces that will build the core of your JavaScript knowledge.

Variables

In a minute, we'll be discussing data types in JavaScript, but we need to cover variables first. Depending how you look at this, we might be putting the cart before the horse, but we'll figure this all out. For now, know that we have values like text and numbers in JavaScript.

Variables are basically labels that you give to your values. The variable points to the place in memory where the value is stored. So, how do you make one of these variables?

Example 2.1

```
var name = "Joe",
    age = 45,
    job = "plumber";

alert( name ); // Joe
```

There are several things to notice here: firstly, when creating variables, you must put the keyword `var` before the variable name. Next comes the variable name. You can name your variables whatever you want, within these boundaries: you can use any combination of letters, numbers, dollar signs, and underscores, as long as you don't start with a number. As you can see, you can declare multiple variables at once by separating the expressions with commas. It's a good idea to do this: at the top of your code,

Prepared exclusively for admin@wedoev.com

and at the top of every function, declare all the variables you'll need within that file or function (functions: coming soon to a page near you).

But, what do you do if you don't know what the value of that variable should be when you create it? Maybe it's a value that the user is going to give you later on, or something you need to calculate. It's still a good idea to declare the variable, but you don't have to give it a value:

```
var name;

name = "Jim";
```

When you declare the variable without giving it a value, the JavaScript engine gives it the default value of **`undefined`**. Then, you can give it a new value later.

I know I already said it, but it's important that you understand it: variables are just labels for values. This means they are interchangeable for the most part.

With that out of the way, let's look at the types of data we have in JavaScript.

Types

The first step in learning JavaScript (after we know what a variable is) is to understand two things:

- What basic data types are available, and what they can do
- How the language is actually written (the syntax)

Prepared exclusively for admin@wedoev.com

We're going to start by looking at types of data you can use in JavaScript: the parts that will keep your programs running. Then, we'll move on to looking at syntax: that's *how* to code things so the JavaScript compiler—the *thing* that reads your code—will understand it.

String

A string is simply the way to handle any kind of text in JavaScript. Whenever you're working with text, you'll be using a string. A string of text must be surrounded by either double or single quotes; there's no difference. Here are some examples:

```
"Here is some text"

'This text is surrounded by single quotes'
```

If you want to include a quote character inside your string, then you'll have to escape it. To escape a character inside a string means to precede the character with a backslash. This basically tells the JavaScript interpreter that the character after the slash shouldn't be interpreted normally; it's something special. In this case, the normal interpretation of a quote is to close the string.

Example 2.2

```
alert( "John says \"What are you doing?\"" ); ▷
// John says "What are you doing?"
alert( '"Let\'s go out for lunch."' ); ▷
// "Let's go out for lunch."
```

Notice how, in that last string, I don't have to escape the double quotes, because the string is delimited by single quotes. You only have to escape the quote-type that holds your string.

Prepared exclusively for admin@wedoev.com

Numbers

While some programming languages offer different types of numbers, JavaScript has only one. In most cases, you'll simply type the number of your choice.

```
10

3.14159
```

JavaScript makes no distinction (at least for you, the coder) between numbers with and without decimals. For extra large or extra small numbers, you can use scientific notation:

Example 2.3

```
alert( 3.45e5 ); // 345000
alert( 123e-3 ); // 0.123
```

The first number above represents 3.45×10^5, or 345,000. The second represents 123×10^{-3}, or 0.123. Be careful though: JavaScript doesn't accept numbers with more that 17 decimal places. It will cut off any digits after that.

ROCK* TIP

This is a good time to mention that numbers in JavaScript aren't always that reliable. For example, type `0.1 + 0.2` *into Firebug. The result is* `0.30000000000000004`*; what? Why does this happen? The reasons are pretty technical, and they're based on the number specification JavaScript uses. Thankfully, as you start out with JavaScript, you won't be doing too much number crunching. However, consider this a general rule of thumb: when adding and subtracting with decimals, multiply them first. After calculating, divide. We haven't discussed operators, yet, but this is the way we'd do it:* `(( 0.1 * 10 ) + ( 0.2 * 10 )) / 10` *This will come up with the right answer:* `0.3`*. I know this looks painful, but hey, that's JavaScript: at times, terribly frustrating (but mostly fun and cool).*

!

Booleans

There are two Boolean values: **`true`** and **`false`**. There are many places you'll use Boolean values in your JavaScript code, and when necessary, you'll use these values. However, more

Prepared exclusively for admin@wedoev.com

often you'll evaluate other expressions for their "truthiness" or "falsiness." We'll see more about how to do this later on in this chapter, when discussing looping and conditionals.

Null and Undefined

Most languages have a value that is the "non-existent" value. By non-existent, I mean that when you try to get a value that doesn't exist (like a variable with no value), you'll get `undefined` back. There's another value in this strain, and that's `null`. There isn't usually a need to distinguish between `null` and `undefined`; but if you need to "get rid of" the value in a variable, you can set it to `null`.

Object

What we've looked at so for are considered primitive values; they're the raw pieces that every language has, in some form or another. Now, we move on to reference values. Reference values are different from primitive values (i.e. strings, numbers, booleans, `null`, and `undefined`) in that they are passed around your code by reference. I know that's abstract, so we'll come back to this concept when we can put it in context (if you can't wait, that's on page 77, when we're talking about calling and applying functions).

Objects are one of JavaScript's most powerful features. Think of an object as a wrapper for a bunch of primitive or reference values. Here's an example:

Prepared exclusively for admin@wedoev.com

```
var obj = {
    name    : "Andrew",
    age     : 20,
    awake   : true,
    company : {
        name : "XYZ Ltd.",
        id   : 12345
    }
};
```

An object is delimited by curly braces; inside the object, we have a list of key-value pairs. A key-value pair is just what it sounds like: a pair of values, one of which is the key that points to another. If I wanted to get my name out of this object, I'd ask the object for it's **`name`** property—that's what we call the value in a key-value pair. Also notice that each pair, except for the last one, ends with a comma; that's important. What's not important is the spacing; I've added some extra spacing here to make our object pretty, but I could have jammed it all on one line, *sans* any spacing, and it would be the same.

As you can see, any type of value can be a property of an object. I've stuffed a string, a number, a Boolean, and even another object in there. You can access properties in two ways: either the dot notation or the square bracket notation, as showcased below:

Example 2.4

```
var obj = { name : "Joe", age : 10 };

alert( obj.name ); // "Joe"

alert( obj["age"] ) ; // 10
```

Creating custom objects like this will be important for your JavaScript applications. It allows you to make your own custom

Prepared exclusively for admin@wedoev.com

values that aren't built into JavaScript. Need a `Person` object? No problem. Or what about a tabbed container? JavaScript objects have you covered. We'll come back to custom objects in the next chapter, when we dive ankle-deep into the dirty waters of object-oriented programming.

A final note about objects: you don't need to put all the properties you want them to have in them right away; you can add them later. We'll see more about this later.

Array

Most programming languages have arrays, which are a great way to hold bunches of data. They're very similar to objects, but without the keys.

```
["Buy milk", "Feed cat", "Pick up dry cleaning", "Deposit ▷
Cheque"]
```

An array is delimited by brackets and each value is separated by a comma. Of course, you aren't limited to strings, as I've used here. Also, you don't have to use only values of the same type in a single array.

```
["string", 20, false, null, { "id" : 8888 }]
```

If you've like to access an item from an array, you can use square bracket notation. You would do this to access the first item in the array:

Example 2.5

```
var arr = ["string", 20, false];
alert( arr[0] ); // "string"
```

Prepared exclusively for admin@wedoev.com

Array item indices are zero-based, which means the first item is 0, the second is 1, and so on.

Date

So far, all the values we've looked at have been literal; this means that we haven't used any "real" code to give the JavaScript interpreter our values; we've just inputted the raw string, number, Boolean, or whatever. The Date value is the first value we'll look at that can't be created that way.

So, to make a JavaScript Date object, you'll do this:

```
new Date()
```

This introduces some new syntax, so let's go over that first. The **`new`** keyword simply means we want to create a new object; in this case, that's a date object. **`Date`**, in this case, is the name of a function. We'll discuss functions in the next chapter, but they're basically wrappers for related lines of code. You call a function—that means run the code inside it—by placing parentheses after the function name.

ROCK* TIP

Actually, we can create all the primitive types in a way similar to this, but I'm not going to teach you that, because there's never a good reason to do so.

!

What you see above will give you a new date object that holds the date and time you created that object…right down to the millisecond. If you want to create a date object for a specific date or time, you'll have to pass in the right parameters. Parameters? Well, if you want a function to use values outside itself, you have to give them to it. To do so, just put them between those parenthesis that called the function. Don't forget to separate multiple parameters with commas.

Prepared exclusively for admin@wedoev.com

There are several parameters you can use to create other dates. You can pass in a string date:

```
new Date("January 17, 2012")
```

Or, you can pass a list of parameters, corresponding to the folowing list: year, month, day, hour, minute, second, millisecond. You don't have to add them all; only the ones that matter. It's important to note that the month number is zero-based. This means that to get January, you use the number 0 and to get December, you use 11.

```
new Date(2009, 3, 20, 17, 30)
```

This will return April 20, 2009 at 5:30:00.0 PM. Your date object will also include time zone information, based on the settings on your computer.

Semicolons

You may have notices the semicolons popping up here are there in the bits of code we've looked at so far. These are important in JavaScript; they delimit statements. So, when should you use semicolons? Well, the short and easy answer is to use them at the end of each line. The more accurate answer is to use one after each expression statement. Vague, eh? Explaining how all this works is a bit beyond this book. When you're beginning, it's easiest to learn where to put your semicolons by looking at other people's code. As you take your cues from the code snippets you see in this book, these rules will hold true **most of the time**:

Prepared exclusively for admin@wedoev.com

- Any single line that you could put in your Firebug console should have a semicolon at the end.

  var someVar = "value"; // semicolon

  someFunction("parameter"); // semicolon

- Any statement that includes a block (lines of code between curly braces) should not have semicolons at the end of the lines that open and close the block (the lines that usually end with an opening or closing curly brace).

  if (true) { // no semicolon doThis(); // semicolon } // no semicolon

I know this code might not make sense right now, but just keep these semicolon rules in the back of your mind.

Comments

Here's something you'll definitely find useful: you can add comments to your JavaScript code. To make a single-line comment, use two backslashes. You can do this for a whole line, or just part of one:

```
// This is a useless comment
var x = 10; // x is now equal to 10
```

If you want multi-line comments, use a backslash and an asterisk; reverse that to end the comment:

```
/* file: navigtion.js
   author: Andrew Burgess
   date: July 2, 2011
   purpose: provides animation for the site navigation
*/
```

Prepared exclusively for admin@wedoev.com

Make sure you don't abuse comments. Don't litter your files with them, and make the comments you do use worthwhile.

Operators

Well, now that we know what values we can use, and how to store them, let's talk about operators. Operators are used to work with variables and values; often, they'll change variables, or return values that you'll want to assign to variables. Let's take a look at some of the operators we've got in our toolbox.

Arithmetic Operators

These are probably the most often used operators. They're your standard math operators, good friends of most people since about grade 2.

Example 2.6

```
var four = 2 + 2, //addition operator
    hour = 24 - 13, // subtraction operator
    seventy = 7 * 10, // multiplication operator
    avg_days_per_week = 365 / 52, // division operator
    remainder = 31 % 2, // modulus operator
    msg = "the time is " + hour + " o'clock"; // addition ▷
operator used on strings

console.log(four); // 4
console.log(hour); // 11
console.log(seventy); // 70
console.log(avg_days_per_week); // 7.019230769230769
console.log(remainder); // 1
console.log(msg); // "the time is 11 o'clock"
```

Prepared exclusively for admin@wedoev.com

Besides the obvious ways these operators work, you can see that we're assigning the value of each operation to a variable. Also, notice that we only need to use one `var` statement, since each variable assignment is separated by commas. Then, notice that + operator is used to concatenate strings. We can even throw a number in there and JavaScript will convert the it to part of the resulting string.

The one operator above that you might not be familiar with is the modulus (%) operator. It returns the remainder or the division of the two numbers. The result for the example above is 1, because `31 / 2` is 15, with remainder 1.

Comparison Operators

Besides acting on values, you'll want to compare them. You'll be familiar with some of these:

Example 2.7

```
alert( 10 < 5 ); // false
alert( 10 > 5 ); // true
alert( 4 <= 3 ); // false
alert( 4 >= 3 ); // true
```

It's not hard at all figure which number is greater, or lesser. Besides "less-than" and "greater-than", we've got "less-than-or-equal-to" and "greater-than-or-equal-to." All of these operators return boolean values.

You can use these operators on strings as well:

Prepared exclusively for admin@wedoev.com

Example 2.8

```
alert( "cat" > "dog" ); // false
alert( "bacon" > "chunks" ); // false
alert( "cats" >= "cats" ); // true
alert( "fishing" <= "sleeping" ); // true
alert( "cat" > "CAT" ); // true
```

How does this work? Well, each letter has a character code, a number that identifies it. It's the numbers that are compared. It's important to note that capital letters don't line up beside their lowercase children. All the capital letters come before the lowercase ones, so `"A" < "z"` is true.

And it's not just letters that have character codes; every character you can type, including ones you can't (like Shift, Escape, and the others in the modifier gang), has a character code. So a string with numbers or punctuation can be compared this way.

How about just checking to see if two values are equal? Don't make the mistake of doing this:

```
var a = "cat",
    b = "dog";

a = b // WRONG: DOES NOT COMPARE
```

Look at this for a second; can you see why you can't use = to compare values? That's the assignment operator; you use a single equal-sign to assign a value to variable. To test for equality, you can use three equal-signs (often called the triple-equals operator).

ROCK* TIP

You might be thinking, "What about two equal-signs? Isn't it dumb to jump from one to three?" Well, there actually is a double-equal operator. Usually, you won't want to use it, because it tries to change the type of your variables. For example, `1 === '1'` *is false, but* `1 == '1'` *is true, because the double-equal operator will convert the string* `"1"` *to the number* 1 *(We'll talk about converting values soon).*

Prepared exclusively for admin@wedoev.com

Example 2.9

```
var a = "cat",
    b = "dog";

console.log( a === b ); // false
console.log( a === "cat" ); // true
```

Unary Operators

All the operators we've looked as so far work with two values, one on each side. There are several operators that only work with one value. First off, we've got the incrementing and decrementing operators:

Example 2.10

```
var i = 0;

i++;

alert(i); // 1

++i;

alert(i); // 2

i--;

alert(i); // 1

--i;

alert(i); // 0
```

Prepared exclusively for admin@wedoev.com

Putting “plus-plus” before or after a number will add one to the number. Predictably, “minus-minus” will subtract one from the number. So, what’s the difference between the prefix version and the postfix version? It’s this: these operators return a value too, just like the other ones. In this case, they return the value of the number they are incrementing; however, the prefix (that the operator on the front) operators increment the number before returning the value, which the postfix operators return the value, then increment the number. Here’s an example:

Example 2.11

```
var num1 = 1, num2, num3;

num2 = ++num1;

console.log("num1: ", num1); // num1: 2
console.log("num2: ", num2); // num2: 2

num3 = num1++;

console.log("num1: ", num1); // num1: 3
console.log("num3: ", num2); // num3: 2
```

Another useful unary operator is `typeof`; this operator will return the type of the given variable:

Example 2.12

```
alert( typeof 10 ); // "number"
alert( typeof "test" ); // "string"
alert( typeof new Date() ); // "object"
```

The only catch here is that arrays aren’t given this type of array; they are called objects, because—technically—they are. But that’s not helpful.

Prepared exclusively for admin@wedoev.com

There are a few other unary operators, but we'll see them in a while. Right now, let's talk about our last two groups of operators: extra assignment operators and Boolean operators.

Assignment Operators

We've looked at assigning with the = operator; but there are a few others that I think are taught best by demonstration:

Example 2.13

```
var num = 10;

num += 5; // same as num = num + 5
alert(num); // 15

num -= 3; // same as num = num - 3
alert(num); // 12

num *= 2; // same as num = num * 2
alert(num); // 24

num /= 6; // same as num = num / 6
alert(num); // 4

num %= 3; // same as num = num % 3
alert(num); // 1
```

As you can see, these assignment operators are used to perform an operation on a value already in a variable and re-assign the new value to the variable. If you run the above code, you should find that **`num`** equals 1 after you run them all.

Prepared exclusively for admin@wedoev.com

Boolean Operators

Boolean operators are used to test the trueness or falseness of a value. Every value in JavaScript can be evaluated as a Boolean. Some values are considered false and others are considered true; you'll see how this is important later on in this chapter. For now, know that these values are considered false:

- `false`
- `null`
- `undefined`
- `""` (an empty string)
- `0`
- `Nan` (what you get when you try to add, subtract, etc. a number and another non-number value)

Every other value is considered true; keep in mind, this includes the strings "false", "null", and so on.

Now then, back to Boolean operators. The first is the logical NOT; the logical NOT operator converts the given value to the opposite Boolean value.

Example 2.14

```
var a = "a string";

alert( !a ); // false

alert( !!a ); // true
```

Since a string with something in it is a true value, using the logical NOT operator will return `false`. It's just like saying "NOT true."

Prepared exclusively for admin@wedoev.com

Notice that using NOT twice gives us **`true`**; this is a great trick to use when you want to convert any value to its Boolean value.

While the logical NOT is a unary operator, the logical AND and logical OR operators use two values. Look at the syntax, then we'll discuss them:

Example 2.15

```
var a = true;
var b = false;

alert( a && b ); // false

alert( a || b ); // true
```

How does this work? And where do these operators do any good? The first one above (**&&**) is the logical AND operator; it only returns **`true`** if both sides of the operator are true. The logical OR (||) returns true as long as just one of the values are true.

So, where is this useful? Well, soon we'll be looking at loops and conditionals, and that's where you'll use this most. In a conditional statement, you'll want to check for certain conditions. If you need two values to be something specific before proceeding, you'll use the logical AND; if only one needs to be true, logical OR will work fine.

I should note that you can use any expression that returns a value; for example:

Prepared exclusively for admin@wedoev.com

Example 2.16

```
var day = 14, year = 2011;

alert( day > 10 || year === 2011 ); // true

alert( day < 20 && year >= 2010 ); // true
```

You should also know that if the first side of the operator determines the answer, the second side won't be evaluated. For example, in the use of logical OR above, `year === 2011` will never be evaluated, because `day > 10` returns true, and OR only requires one to be true. On the other hand, if the first half of a logical AND returns false, the second half won't be evaluated, because both sides need to be true for the whole expression to be true.

This fact—that if the first half of the expression determines the final value, the second isn't tested—is used by some of the JavaScript ninjas to shorten their code a bit. Let's say we want to invoke a function (remember, we'll get to functions soon), but we're not sure if the function exists. We could do this:

```
funcName && funcName();
```

Remember, to execute the function, we have to have the parentheses, so the first part of this code doesn't execute the function, it just returns the function. If the function exists, this will evaluate to true, and the evaluation will continue to the second part, which executes the function. This avoids the error that would be thrown if we tried to execute a function that doesn't exist (and, yes, we will talk about errors).

Prepared exclusively for admin@wedoev.com

Conditional Statements

Several—if not most—of the operators we've just finished looking at are used to determine whether something is true or false. Often, that will be the condition for something further on: if something is true, do this; otherwise, do that. Let's see how this plays out.

If Statements

Pretend for a moment that we're building an app that must output the century a given year is in.

Example 2.17

```
var year = 2011, msg = "In the ";

if (year > 2000) {
    msg += "twenty-first century";
} else {
    msg += "twentieth century";
}

console.log(msg); // In the twenty-first century
```

This is an if-statement. We start with the keyword **`if`**. Then, there's the condition inside parentheses. This is where true or false values and operators that return Booleans come into play. They'll determine which statement runs.

We haven't said much about statements yet. In an if-statement, the statement consists of all the lines of code between the opening and closing curly braces. In our example, that's only one line, which appends the century name to the string **`msg`**. Then, notice another statement after that one: the else-statement. This will only be executed if the condition before it evaluates to false.

Prepared exclusively for admin@wedoev.com

But let's say we're doing something more than a true/false deal. It's not hard to do more:

Example 2.18

```
var year = 11, msg = "In the ";
if (year > 2000) {
    msg += "twenty-first century";
} else if (year > 1900) {
    msg += "twentieth century";
} else {
    msg += "very distant past.";
}

console.log(msg); // In the very distant past
```

As you can see, multiple if-statements are no big deal. In place of the else-statement, just put another if-statement. Bump that `if` right up against the `else`, add a second condition inside parenthesis, and you're off. Of course, this could go on indefinitely; there's no limit to the number of `else if` statements you can use.

Switch Statements

However, let's say we're checking something with several options, like the size of coffee a customer wants. We'd have to do something like this:

Example 2.19

```
var order_size = "medium";

if (order_size === "small") {
    alert("small");
} else if (order_size === "medium") {
    alert("meduim");
```

Prepared exclusively for admin@wedoev.com

```
} else if (order_side === "large") {
    alert("large");
} else if (order_side === "extra large") {
    alert("extra large");
} else {
    alert("something else?");
}
```

Now, I don't know about you, but that last snippet seems like a lot of needless repetition to me. Here's what you can do instead: use a switch statement.

Example 2.20

```
var order_size = "medium";

switch (order_size) {
    case "small":
        alert("small");
    break;
    case "medium":
        alert("medium");
    break;
    case "large":
        alert("large");
    break;
    case "extra large":
        alert("extra large");
    break;
    default:
        alert("something else?");
}
```

Let me introduce you to the switch statement. It's the perfect substitute for long-winded, nested if/else statements. With the switch statement, you only write the expression to evaluate once;

Prepared exclusively for admin@wedoev.com

it goes in parentheses at the top, right after the keyword **`switch`**. Then we open a set of curly braces.

We want to do something based on the evaluation of that expression, so we have any number of cases. The keyword **`case`** is followed by the value that we want the expression to match. In this example, the expression **`order_item`** is compared to each of the coffee sizes. The compared value is then followed by a colon. Next are the lines of code that will be executed if the expression matches the case; if this is only one line, feel free to put it on the same line as the **`case "value":`** part, if you like. It doesn't matter at all, because JavaScript doesn't care about spacing. Don't forget to end your case block with the **`break;`** line. This prevents our case statements from falling through; we'll see what this means in a second.

But first, don't forget about the **`default:`** piece at the end; yup, you guessed it: it's the equivalent of an **`else`**. It executed when we don't match any of the cases.

So, back to falling through: if we didn't include the **`break;`** statement, the code for any case statements under the one we match will also be executed. Here's example 2.20, with the break statement removed:

Example 2.21

```
var order_size = "medium";

switch (order_size) {
    case "small":
        alert("small");
    case "medium":
        alert("medium");
    case "large":
        alert("large");
```

Prepared exclusively for admin@wedoev.com

```
        case "extra large":
            alert("extra large");
        default:
            alert("something else?");
    }
```

Go run this example file: you'll get alerts for "medium," "large," "extra large," and "something else?" At first, this might not make sense; so look at it this way. We can have multiple cases for each code block, meaning that if our condition is one of the cases, the code will execute:

Example 2.22

```
    var order_size = "medium";

    switch (order_size) {
        case "small":
            alert("small");
        break;
        case "medium":
        case "large":
        case "extra large":
            alert("medium, large, or extra large");
        break;
        default:
            alert("something else?");
    }
```

If we now add some code for each case statement (**without** any `break` statements), the same behaviour will take place: the code under each case will run, but it will "fall through" to the next case statement and execute the code there.

You can probably come up with a situation where this would be a neat idea, and save you some coding. However, several of the

Prepared exclusively for admin@wedoev.com

JavaScript bigwigs consider this a bad practice. What it does is pretty subtle, and it could very easily trip you up when debugging. I'd recommend using a `break` for every `case`.

Conditional Operator (Ternary Operator)

There's one more type of conditional statement. Let's say we wanted to do something like this:

```
var temp = 18, // degrees Celsius
    msg;

if (temp > 10) {
    msg = "Today was warm";
} else {
    msg = "Today was cold";
}
```

For setting one variable, that seems like a lot of extra code. Well, using the conditional operator, we can shorten it.

ROCK* TIP

You may see the conditional operator called the "ternary operator"; "ternary" just means it takes three values, just like the unary operators take only one. No one gets confused, though, because there's only one ternary operator in JavaScript.

Example 2.23

```
var temp = 18,
    msg = (temp > 10) ? "Today ▷
was warm" : "Today was cold";

alert(msg);
```

It's pretty simple. in parentheses (and really, the parenthesis aren't required, but it improves readability), put your conditional statement, followed by a question mark. Then, we have the value that's used if the condition is true. The value that's returned if the condition is false comes next, after a colon. The

Prepared exclusively for admin@wedoev.com

conditional operator is great for when your if statements have only two possible outcomes, each of which can be coded in a single line.

Looping Statements

Now that we've got conditional statements down, we're ready to move on to loops. Very often, you'll want to perform the same action several times, each time on a different value. As you might guess, this often goes hand-in-hand with arrays. Let's check out the looping constructs JavaScript gives us.

For Loops

The for loop will often be the first tool you reach for when you need to build a loop. Take a look at the syntax, and then we'll discuss it.

Example 2.24

```
var names = ["Ringo", "John", "Paul", "George"];

for (var i = 0; i < names.length; i++) {
    alert("Say hello to " + names[i]);
}
```

We begin our for loop with the keyword `for`; next we have three statements in a set of parentheses. Notice how the statements are separated by semi-colons, just like all good JavaScript statements. The first statement sets any variables we need for the loop; in this case, we're setting only one variable: `i = 0` ("i" is short for iterator or index). The second statement is an expression that returns a Boolean value; this is evaluated before the loop is executed. If it evaluates `true`, the loop will execute; if it evaluates `false`, the loop will end. In our example, we check to see that `i` is less than the length of the `names` array (more on the length property later). The

Prepared exclusively for admin@wedoev.com

final section performs any action we want to be done after the loop executes; here, we're incrementing `i` by one. Inside the loop, we can work with the values of the array by using the square bracket notation, passing in an index number—this is where the variable `i` comes in.

Now, most of the loops you'll see (and probably write, too) will be structured very similarly to this one, even though **all three statements in the parenthesis are optional**. However, make sure you really understand why this thing works; if you understand what each of these pieces do, your for loops will be more flexible. For example, who says we need to set the variables inside the parentheses?

```
var i = 0;
for ( ; i < names.length; i++) {
    // whatever
}
```

Notice that even though I've taken the variable declaration outside the loop, we still need to put a semicolon after where the statement would go.

On the other hand, why not set two variables in the first part, and "cache" the value of `names.length` in the variable `len`, and use that in the second statement (this really is faster if you're working with a large array).

```
for (var i = 0, len = names.length; i < len; i++) {
    // whatever
}
```

Notice that I haven't separated the two variable declarations with a semicolon; it has to be one statement, so we separate it with comma.

Prepared exclusively for admin@wedoev.com

But then there's the post-loop-actiony part. Why not throw that inside the loop?

```
for (var i = 0; i < names.length; ){
    // whatever
    i++;
}
```

And instead of doing the whole `< names.length` thing, we could just check to see if the value exists in the array.

```
var i = 0;

for ( ; names[i] ; ) {
    //do something
    i++;
}
```

This is an important point that's relevant to more than just the for loop: if you understand what each piece does, you'll be able wield it more effectively. With this in mind, can you guess why you haven't tried removing the Boolean expression from the for loop? Without that, your loop would loop forever (try it, and crash your browser).

While Loop

Here's a structure that might look a bit like the last for loop we wrote:

Prepared exclusively for admin@wedoev.com

Example 2.25

```
var names = ["Ringo", "John", "Paul", "George"],
    i = 0;

while (i < names.length) {
    alert("Say hi to " + names[i]);
    i++;
}
```

A while loop is a simpler form of the for loop. Instead of three statements in the parentheses, there's only one: the Boolean expression. If it's true, the while loop will run again. Of course, it starts with the **`while`** keyword, followed by the parentheses, and then the block statement. If you're using any variables, you have to initialize them before the loop and increment them within the loop. Depending on your task, a while loop can be more elegant than a for loop. What I've shown above is using a while loop for iteration; that's usually better off left to a for loop. A while loop, semantically, should be used *while* something is true:

Example 2.26

```
var not_home = true,
    miles_driven = 0;

while (not_home) {
    miles_driven += 1 // drive another mile
    if (miles_driven === 10) {
        not_home = false;
    }
}

alert(miles_driven);
```

Prepared exclusively for admin@wedoev.com

For Loop, Revisited

There's another form of for loop that you should now about, usually called a for-in loop. It's used for looping over objects. Check it out:

Example 2.27

```
var person = {
      name : "Jules Verne",
        job  : "Author",
        year_of_birth: 1828,
        year_of_death: 1905
    },
    prop;

for (prop in person) {
    alert("His " + prop + " is " + person[prop]);
}
```

As you can see, a for-in loop is much simpler than the regular for loop. Instead of the three statements, we only put one in the parentheses: the name of the iterator variable, the keyword `in`, and the object we're going over. As expected, this will execute the inside of the block for each item or property in the array or object, respectively. There's a caveat that you must be aware of when using the for-in loop with objects, but we'll discuss this more in the next chapter.

Summary

You should be warming into our topic by now. We've covered the different types available in JavaScript, most of the operators, and how to use conditions and loops. In the next chapter, we're going talk about one of the most important features of JavaScript: functions.

Prepared exclusively for admin@wedoev.com

3

Prepared exclusively for admin@wedoev.com

More Basics

Now that we've got the very basics of types and syntax down, let's move on to another incredibly important feature: functions. Pay close attention!

Functions

What is a function? A function is really just an encapsulated bunch of code. We've been looking at a bunch of JavaScript syntax: all of that can go inside functions, and often that's what you'll do. When you put code inside a function, it isn't run right away, like "loose" code. You decide when the code runs by calling the function, and you can do that as often as you want to. This is part of the benefit of functions.

But don't forget the encapsulation part: that means that at the time you invoke, or call, a function, you don't have any control (for the most part) of what goes on inside it. You'll hand the function some values to work with, and it will hand a value back; but what goes on inside it is black-boxed. This turns out to be an important thing that we can take advantage of, as we'll see later.

There's two other important things you should know about functions. First off, functions are first-class objects in JavaScript, which means that *they are objects*, and that *they are no different from any other value*. Let's take these two points a bit further.

Functions are Objects: We saw how to create a object literal earlier. Now, we can't create a function using that object literal notation, but a function can have properties and methods, because it is an object. You can add your own properties and methods to functions, but we'll look at some of the built in ones soon.

Prepared exclusively for admin@wedoev.com

Functions are like other Values: In many programming languages, a function is something very different from primitive values like strings and numbers. However, in JavaScript, that's not the case. You can use functions in most places that you'd use another value, like passing it to another function (which we'll see shortly). Also, you can redefine a function, just like you'd redefine any other variable. This, as you'll see, is very handy.

Now that we understand a bit of function theory, let's look as some syntax.

Function Syntax

As we know, a function is just any number of lines of regular JavaScript. But we need to wrap that in function material. Let's say we have some code that determines an appropriate greeting for our website:

Example 3.1

```
var hour = 10,
message;

if (hour < 12) {
    message = "Good Morning!";
} else if (hour < 17) {
    message = "Good Afternoon!";
} else {
    message = "Good Evening!";
}

console.log(message); // Good Morning!
```

We've created an `hour` variable, with the hypothetical current hour. Then, we use an if statement to figure out in what range our hour is, and assign `message` accordingly.

Prepared exclusively for admin@wedoev.com

So, what do we do to convert this code to a function? Well, first, let's meet the function shell:

```
function getGreeting () {
    // code goes here
}
```

Start with the keyword **`function`**; then, write the name of the function; it can be whatever you want, as long as it follows the rules for variable names (i.e. alphanumeric characters, underscore, and dollar sign, but not starting with a number). Next, parentheses. Then, the lines of code go between curly braces.

Now, you might be tempted to throw all the code we wrote above into a function shell. This isn't wise for two reasons. Firstly, there would be no way to get to the variable **`message`** (more on *function scope* soon). Secondly, with a bit of customization, we can increase the usefulness of the function by tweaking it a bit.

Let's make our function accept a parameter. A parameter is a value that you hand to the function when you ask the function to run. The function can then use the parameters however it needs to. How does this work syntactically?

```
function getGreeting (hour) {
    //code here
}
```

Here, our **`getGreeting`** function accepts one parameter, which we're calling **`hour`**. Parameters are like variable that you can only access from within the function. To invoke this function and give it an hour, we would do this:

```
getGreeting( 10 );
```

Prepared exclusively for admin@wedoev.com

We could store the hour in a variable, but we don't need to, since we won't need it after this.

So, we're passing the current hour to **`getGreeting`**. Now we're ready to write the code inside that function:

Example 3.2

```
function getGreeting(hour) {
    var msg;

    if (hour < 12) {
        msg = "Good Morning!";
    } else if (hour < 17) {
        msg = "Good Afternoon!";
    } else {
        msg = "Good Evening!";
    }
    return msg;
}

var message = getGreeting( 16 ); // 4 PM

alert(message); // Good Afternoon!
```

If you've been following along carefully, there should be only one thing above that you're not sure about: **`return msg`**. What's up with that? To return a value is to pass it back to the code that called the function. That way, we can assign the calling of a function to a variable (as we do above), and it will be whatever value was returned from **`getGreeting`**.

Here are two more things I'd like to do to this function, to make it better. First, there's really no need for the **`msg`** variable, because we only ever do one thing to it. Try this:

Prepared exclusively for admin@wedoev.com

Example 3.3

```
function getGreeting(hour) {
    if (hour < 12) {
        return "Good Morning!";
    } else if (hour < 17) {
        return "Good Afternoon!";
    } else {
        return "Good Evening!";
    }
}

alert( getGreeting( 3 ) ); // Good Morning!
```

This makes it clear that we can have more than one **`return`**. Once we figure out which value we must return, we'll do so right away. I should note that a return statement will be the last line to be executed in a function; once the function returns, it's done; nothing else is executed. Therefore, unless your return statement is inside a conditional statement, you usually won't have any code after the return line in your function.

One more thing: When writing functions, you want them to be user-friendly. Let's do this: if the user doesn't pass in a parameter, we'll provide a reasonable default.

Example 3.4

```
function getGreeting(hour) {

    hour = hour || new Date().getHours();

    if (hour < 12) {
        return "Good Morning!";
    } else if (hour < 17) {
        return "Good Afternoon!";
    } else {
```

Prepared exclusively for admin@wedoev.com

```
            return "Good Evening!";
        }
    }

    alert( getGreeting( new Date().getHours() ) );
```

What's going on here? We're using a logical OR operator; remember what this does: if the first side is true, it will use that; otherwise, it will use the second side. In this case, we're saying that we want to set the hour parameter to either the value of the hour parameter (if the user passed one in), or the current hour. We're using the **`getHours`** function of a current Date object; we'll see more about this in a page or so. If the user didn't pass a parameter, **`hour`** will be **`undefined`**, which is false. Therefore, the function will assign the current hour to **`hour`**. And yes, you can assign new values to parameters like this.

ROCK* TIP

This is the best way to provide default values for parameters. It's a pretty advanced JavaScript tip, but I think you can handle it. Don't feel bad if it's a bit complex, though. Come back to it later and you should have no troubles.

Well, that's out first function. Let's wrap up with discussion on functions with a few important points.

Arguments

We've been calling the values that you pass to a function parameters. There's another common term, and that's *arguments*. In fact, JavaScript itself calls the them arguments. Many other programming languages have error-detection features related to the arguments of a function: they make sure that when calling a function you pass in the right number and type of arguments. JavaScript doesn't offer any of this. These are all perfectly valid ways of calling our **`getGreeting`** function above:

Prepared exclusively for admin@wedoev.com

```
var msg1 = getGreeting(10),

    msg2 = getGreeting(1,2,3,4,5,6, "and on"),

    msg3 = getGreeting(),

    msg4 = getGreeting("five o'clock");
```

And there are many more, of course. Out of all these situations, we've only written our function to work with options 1 and 3. However, it will work with option 2: **`hour`** will be set to 1, but all the following arguments will be lost. Option 4 is a good example of why some error checking is necessary: to guard against string parameters in our function, we should use the **`typeof`** operator to make sure it's a number (after we assert that they did indeed give us a parameter). If it's not a number, we should provide the default: the current hour.

I said that for option 2 above, all the extra arguments are lost. Well, that's not exactly true. JavaScript does provides a way to get to them: it offers an object called **`arguments`** that you can access from inside your functions. You might think that it would make more sense if **`arguments`** was an array…and every other JavaScript programmer would agree with you. However, it's an *array-like object* which means it has some of the characteristics of an array, but not all of them. It has a **`length`** property, which is the number of arguments passed into the function. You're also able to access the parameters with square bracket notations, just like in an array. There are no other similarities with a real array. That is all.

So we could write a **`sum`** function like this: we loop over each argument that was passed in and add it to the variable **`sum`**. Then, we return **`sum`**.

Prepared exclusively for admin@wedoev.com

Example 3.5

```
function sum() {
    var sum = 0, i;

    for ( i = 0 ; arguments[i]; i++) {
        sum += arguments[i];
    }
    return sum;
}

var sum1 = sum(1, 2),
    sum2 = sum(100, 75, 40),
    sum3 = sum(9, 1.4, 8, 2, 79, 3234, 6, 4, 5e3, 5);

alert(sum1); // 2
alert(sum2); // 215
alert(sum3); // 8348.4
```

Scope

We mentioned scope in passing (quite) a few paragraphs ago. Let's discuss that in more detail now. While scope isn't related to functions only, functions are a main part of scope in JavaScript, so this is appropriate timing.

Scope is basically the set of variables that you have access to at a given point in your code. JavaScript has function scope, which means that the scope changes when we enter a function. Let's look at a few examples.

Let's start with the global area. If you write

```
var name = "Bob";
```

Prepared exclusively for admin@wedoev.com

at the top level of a JavaScript file—meaning it's not inside a function—that variable will be accessible from everywhere within the JavaScript environment. Most often, that environment will be a webpage; this means that every other JavaScript file on the page will have access to that **`name`** variable from everywhere within the file. Any variables created in this "global namespace" will have global access-ability.

ROCK* TIP

You might think that having access to variables globally is a great thing. With that, you don't ever have to worry about whether or not you can get to the value you need—because you always can! This is emphatically not the case . . . and we'll talk more about why that is in the next chapter.

As I mentioned, the scope changes when we enter a function, and only when we enter a function (this isn't true of most other languages). We still have access to all the variables outside the function, but now we have a group of variables that can be accessed *from only inside this function*.

For example,

Example 3.6

```
var name = "Bob";

function greet (greeting) {
    var punc = "!!!";
    return greeting + " " + name + punc;
}

alert( greet("Hello") ); // "Hello Bob!!!"
```

This rather simple example demonstrates function scope. The **`name`** variable is global, so we can reach it from inside the function. Arguments of the function, like **`greeting`**, are part of the function's

Prepared exclusively for admin@wedoev.com

scope, so there's no way to access that outside the function. Also part of the function's scope are any variables created inside the function: in this case, that's **punc**.

This will blow your mind: we can nest functions in JavaScript. Nested functions are just like their parent functions, in that they have access to all the scopes "above" them, as well as their own:

Example 3.7

```
var name = "Bob";

function greet(greeting) {

    function get_punc() {
        return (greeting === "Why") ? "?" : "!";
    }

    return greeting + " " + name + get_punc() + " " + ▷
greeting + get_punc();
}

alert( greet("Why") ); // "Why Bob? Why?"
alert( greet("Hi") );  // "Hi Bob! Hi!"
```

Another example that hopefully points out the scope of nested function; the inner function can access the **greeting** argument of its parent function, and if it needed to it could access **name** as well. As you might guess, a function inside a function is useful when you need to execute the same code more than once during the execution of a function: just put that code in a function and call it whenever you need to.

Now that you're understanding that, chew on this for a while:

Prepared exclusively for admin@wedoev.com

Example 3.8

```
var name = "Bob";

function greet(greeting) {

    var name = "Alice";

    return greeting + " " + name + ".";
}

alert( greet("Hello") ); // "hello Alice."
```

What does this return? The name variable inside the function "overwrites" our access to the outside name variable. Now, when we use `name`, it refers to Alice, not Bob. However, once we're back outside the function, name will once again refer to Bob, because there's no way for us to get to Alice outside of her function.

Anonymous Functions

Thus far, the functions we've made all have names. However, we can create functions without names, just be leaving the name out:

```
function () {
    return "this function has no name";
}
```

Well, cool, I guess, you're saying. But what's the point? There's no way to call it. Well, we'll see many places where anonymous functions (for that is what nameless functions are) are useful; here's one that you might use often:

Prepared exclusively for admin@wedoev.com

Example 3.9

```
var hi = function () {
    return "Hi";
};

alert( hi() ); // Hi
```

Yes, that's right; we can assign a function (named or anonymous) to a variable, and execute it with that variable name. Remember, functions are objects and can be assigned to variables just like values.

Here's another use for anonymous functions:

```
var hi = (function () {
    return function () {
        return "Hi";
    }
}());
```

Take a deep breath; this might look like rocket science, but there's not that much going on here. Start with what you know: we're assigning an anonymous function to the variable `hi`. That anonymous function returns another anonymous function. That's all good. Then, we're wrapping the first function in parentheses. This isn't required, but you can wrap pretty much anything you want to in parentheses in JavaScript; it just improves readability. In this case, we do it to remind ourselves that we're doing something else of interest: notice that, after the function, we have a set of parentheses. We know that a set of parentheses after a function name executes that function. Well, they do the same after a function itself: `function () { /**/ }()`. This is called an *anonymous self-invoking function*, because it has no name and it executes itself. Since it runs right away, `hi` is not assigned that

Prepared exclusively for admin@wedoev.com

function, but the function's return value: another function! In this case, it's just like doing this:

```
var hi = function () {
    return "Hi";
};
```

So why not do that? Well, this is just an example. We'll see a real use for this pattern—assigning the returned value of anonymous self-invoking to a variable—later (if you can't wait, read the section on *closure*, on page 90).

I should note that, while this pattern is usually called an anonymous self-invoking function, not everyone agrees with that term. You'll see that name a lot, but Ben Alman—frequent contributor to jQuery and creator of many popular JavaScript projects—prefers the term *immediately-invoked function expressions*. You can read more about why he prefers that term in his blog post on the topic.

ROCK* TIP

What's the difference between a function and a method? Only where they exist. A method is just a function that is a property of an object. For example:

```
var console = {
  log : function () {
    // code here
  }
};
```

You'd call this the same way you access "normal" properties: `console.log()`. Look familiar?

!

Type Methods

In the last chapter, we learned about the value and reference types that JavaScript offers us. Let's conclude by getting to know the methods of these types. Of course, we won't learn them all, but we'll cover all the important ones, the ones you'll use regularly.

Prepared exclusively for admin@wedoev.com

String Methods

length

This one is actually a property, not a method. Predictably, the length property of a string is the number of characters in the string.

Example 3.10

```
alert( "gobsmacked".length ); // 10
```

indexOf

This is the method you'll most often use when you're searching within a string. This method returns a number, the index of your "search term."

Example 3.11

```
var line = "HAL: I'm sorry, Dave. I'm afraid I can't do ▷
that";

alert( line.indexOf("I'm") ); // 5
alert( line.indexOf("I'm", 6) ); // 22
```

You can pass a second parameter as the starting point, if you don't want to start searching at the beginning of your string. Remember, these indices are zero-based—just like the indices of an array—which means that the first letter of the string is indexed 0, the second is 1, and so on. If the substring you searched for doesn't exist, you'll get -1 back.

slice, substr, and substring

That's right; there are three methods for getting part of a string out of a larger string. First, you can use `slice`. It takes two parameters: the first parameter is the starting index, and the second one is

Prepared exclusively for admin@wedoev.com

the ending index, meaning the index of the character after the last character in the desired substring. If you leave the second one off, it will slice until the end of the string.

Example 3.12

```
var greeting = "Hello, Andrew, what's up?",
    name = greeting.slice(7, 13);

alert(name); // Andrew
```

These index parameters can also be negative numbers, which means they "count" from the end of the string. This way, -1 is the last item in the string, -2 is the second last, and so on.

An alternative to `slice` is `substr`. The first parameter is the same as `slice`—the starting index—but the second parameter is the length of the substring:

Example 3.13

```
var greeting = "Hello, Andrew, what's up?",
    name = greeting.substr(7, 6);

alert(name); // Andrew
```

Since you're string can't have a negative length, you can't use a negative number for the second parameter (you can use a negative number for the first parameter, though). Of course, that second parameter is optional, if you want the rest of the string.

Finally, there's `substring`. This works similarly to `slice`, except when it comes to negative values. A negative value for either parameter acts as 0—it refers to the start of the string. The neat thing about `substring` is that, unlike slice, the second parameter can be lower than the first (while still positive). When this is the

Prepared exclusively for admin@wedoev.com

case, **substring** goes back to the character of that index. For example,

Example 3.14

```
var greeting = "Hello, Andrew, what's up?",
    name = greeting.substring(13, 7);

alert(name); // Andrew
```

split

It's easy to pull a string into an array with the **split** method. Just pass the method a parameter determining what character to split the array on.

Example 3.15

```
var arr = "apples oranges peaches bananas".split(" ");
console.log(arr); // ["apples", "oranges", "peaches", ▷
"bananas"]
```

Yes, this is the way lazy programmers create arrays. But it has more use than just as an alternative to an array literal.

toLowerCase and toUpperCase

I'm sure you know exactly what these do: easy converting of a string to upper- or lowercase:

Example 3.16

```
var usa  = "usa".toUpperCase(),
    comment = "THIS MIGHT BE A COMMENT";

comment = comment.substr(0,1) + comment.slice(1).▸
toLowerCase();
```

Prepared exclusively for admin@wedoev.com

```
console.log(usa); // USA
console.log(comment); // This might be a comment
```

Numbers

toExponential

If you'd like to convert a number to its exponential form, this is the method to use. It takes one parameter: the number of decimals places to use. If you leave the parameter out, JavaScript will make sure your number isn't rounded.

Example 3.17

```
alert( 12345..toExponential() ); // "1.2345e+4", meaning ▷
1.2345 x 10^4
alert( 987.65432.toExponential(3) ); // "9.877e+2"
alert( 0.1234.toExponential() ); // "1.234e-1"
```

If your scratching your head at the double-period in the first example above, a short explanation is in order. When JavaScript sees a dot after a number, it expects more digits; it thinks it's a decimal point. So writing `5` is translated as `5.0`. So if we wrote `5.toExponential()`, we'd get an error. We could do `5.0.toEponential()` if we wanted to, or you could save the character and just do the double-dot. Don't worry, in most cases, your numbers will probably be in variables, so it won't make to much of a difference. (You could also wrap the number in parentheses: `(5234).toExponential()`);

toFixed

This method will round your numbers to the number of decimal points you define in the parameter:

Prepared exclusively for admin@wedoev.com

Example 3.18

```
alert( 6..toFixed(4) ); // "6.0000"
alert( 3.14159.toFixed(2) ); // "3.14"
alert( 10.46.toFixed(1) ); // "10.5"
```

You'll find this useful if you're working with money, and need to round to the nearest cent.

toPrecision

This method is for choosing how many digits should represent your number. This includes numbers of both sides of the decimal point:

Example 3.19

```
alert( 15.7896.toPrecision(4) ); // "15.79"
alert( 1940..toPrecision(2) ); // "1.9e+3"
```

Wait, what? How does that last example work? Well, it round the number a precision of 2, which in this case is to the hundreds digit. Then, it converts it to exponential notation.

I'll add here that all three of these number methods don't changed the variable with the number; they return a new value that you can assign to another (or the same) variable:

Example 3.20

```
var a = 300,
    b = a.toFixed(2);

console.log(a, typeof a); // 300
console.log(b, typeof b); // "300.00"
```

You may also have noticed that all three of these methods return strings, not numbers. What are you to do if you want to convert

Prepared exclusively for admin@wedoev.com

them back to numbers? You can use the global function **parseInt** and **parseFloat**. You use **parseInt** when you want to get a whole number (parse integer). You can hand it a string, and if it can make sense of a number within it, it will do so. For example:

Example 3.21

```
alert( parseInt("123", 10) ); // 123
alert( parseInt("45.67", 10) ); // 45
alert( parseInt("$12.00", 10) ); // NaN
```

parseInt will start at the beginning of the string and stop when the character isn't a number. That's why you get "Not a Number" from a string starting with "$." The second parameter is the radix: the base of the number system we want the string to be parsed to. So 10 is decimal, 2 is binary, 8 is octal, and so on.

parseFloat can take decimals and exponential forms:

Example 3.22

```
alert( parseFloat("45.67", 10) ); // 45.67
alert( parseFloat(1940..toPrecision(2), 10) ); // 1900
```

Just like **parseInt**, **parseFloat** stops when the character is not a number, decimal point, or exponential notation.

Date Methods

There are a ton of methods for Date objects, but some are used way more often than others.

Prepared exclusively for admin@wedoev.com

get____

You'll probably use the family of `get_____` methods most often, for getting different pieces of the date. Here are the specifics; you'll notice that most of the values are zero-based:

Method Name	Return Value
`getDate`	day of month, 1 – 31
`getDay`	day of week, 0 – 6
`getFullYear`	year
`getHours`	hour, 0 – 23
`getMilliseconds`	milliseconds, 0 – 999
`getMinutes`	minutes, 0 – 59
`getMonth`	month, 0 – 11
`getSeconds`	seconds, 0 – 59
`getTime`	milliseconds since January 1, 1970
`getTimezoneOffset`	difference between GMT and local time, in minutes

set____

Most of the get methods have a corresponding set method. They each take a single parameter, identical to the return value of their get counterpart. Here are your options:

- `setDate`
- `setFullYear`
- `setHours`
- `setMilliseconds`
- `setMinute`
- `setMonth`
- `setSeconds`
- `setTime`

Prepared exclusively for admin@wedoev.com

parse

Sometimes you'll have a string date that you want to convert to a Date object. `Date.parse` will do that for you. It actually converts it to a number, the number of milliseconds since midnight on January 1, 1970 (known as the Unix epoch). Then, you can plug that number into `new Date()` to get a Date object.

Example 3.23

```
alert( Date.parse("June 18, 1970") ); // 14529600000
alert( Date.parse("2010/11/11") ); // 1289451600000

var d = new Date(Date.parse("1995/04/25")); // Tue Apr 25 ▷
1995 00:00:00 GMT-0400 (EST)
alert(d);
```

Array Methods

Arrays have useful methods as well. Let's check out some of the common ones!

join

This is the reverse of the string's `split` method. It will join all the elements in the array into a string. You can pass a single parameter to `join` that will be put between each element in the array.

Example 3.24

```
alert( ["cats", "dogs", "hamsters", "fish"].join(' ') );
// "cats dogs hamsters fish"
alert( "this should not have spaces".split(" ").join("_") );
// "this_should_not_have_spaces"
```

Prepared exclusively for admin@wedoev.com

Bonus tip here: notice what I did in the last example: I called the join method right off the split method call. How's this work? Well, we know the split method returns an array. So we can call a method on that returned array immediately; we don't have to save it to a variable first. This is called *chaining* methods, and there's no limit to how long you can make the method chain: as long as the return value of one method has the next method in your chain, you're gold. Many JavaScript frameworks (like jQuery) take advantage of this ability.

pop / shift

These two methods remove and return one value from the array. `pop` gets the last item, `shift` gets the first item.

Example 3.25

```
var arr = ["cats", "dogs", "hamsters", "fish"];

console.log( arr.pop() ); // "fish"
console.log( arr.shift() ); // "cats"
console.log( arr ); // ["dogs", "hamsters"]
```

push / unshift

Predictably, these are the opposite of `pop` and `shift`. `push` adds an item to the end of the array, and `unshift` to the beginning.

Example 3.26

```
var arr = ["Beta", "Gamma"];

arr.push("Delta");
console.log(arr); // ["Beta", "Gamma", "Delta"]
```

Prepared exclusively for admin@wedoev.com

```
arr.unshift("Alpha");

console.log(arr); //  ["Alpha", "Beta", "Gamma", "Delta"]
```

Both return the number of items in the array after the new item is pushed in.

reverse

Just guess what it does. Yes, it returns an array with the items reversed.

Example 3.27

```
alert( ["Crockford", "Resig", "Zakas"].reverse() ); ▷
// "Zakas, Resig, Crockford"
```

slice

Sometimes you'll want to slice your array into multiple parts. This is your method. It takes two parameters: the (zero-based) index of the starting element and the index of the element after the last one you want to slice. You can leave the second parameter off to select the rest of the array. You'll get the sub-array back:

Example 3.28

```
alert( ["a", "b", "c", "d", "e"].slice(2, 4) ); ▷
// ["c", "d"]
alert( ["f", "g", "h", "i", "j"].slice(1) ); ▷
// ["g", "h", "i", "j"]
```

sort

The `sort` method will re-order the items in your array however you'd like. If you don't pass in any parameters, JavaScript will sort the elements alphabetically.

Prepared exclusively for admin@wedoev.com

Example 3.29

```
["d", "e", "c", "a", "b"].sort(); // ["a", "b", "c", "d", ▷
"e"]
```

Unfortunately, it sorts numbers "alphabetically," too, which means that the array `[0, 5, 10, 15, 20, 25]` will be sorted to `[0, 10, 15, 20, 25, 5]`. Not good. Instead, you can pass a sorting function into `sort`. That function should take two parameters—say, `a` and `b`—which will be items from the array. Here's how it works: the `sort` method will pass the first and second items of the array to the function as the parameters; then it will pass the second and third items, and so on. Inside the sorting function, you must compare the two items. Then, return one of these values (assuming the first parameter is `a` and the second, `b`):

- If `a` should come before `b`, return a negative number.
- If `a` and `b` are equal, return 0.
- If `b` should come before `a`, return a positive number.

If you're just sorting numbers, this is really easy:

Example 3.30

```
var arr = [5, 2, 3, 4, 1];
arr.sort(function (a, b) {
    return a - b;
});
console.log(arr); // [1, 2, 3, 4, 5];
```

For more on sorting functions, check out this tutorial I wrote on Nettuts+ a while back.

Prepared exclusively for admin@wedoev.com

Math Functions

JavaScript also has a `Math` object; you can't create objects from it, like with `Date`, but it offers some useful mathematic functionality.

min

A helpful function that returns the lowest of the number parameters you pass in:

Example 3.31

```
alert( Math.min(9, 1, 4, 2) ); // 1
```

max

Yes, it's the opposite of `min`:

Example 3.32

```
alert( Math.max(9, 1, 4, 2) ); // 9
```

random

This function returns a random number between 0 and 1. What good is that? Well, you can then multiply it by 10 to get a number between 0 and 10, roughly. Anytime you need a random number, this is your tool:

Example 3.33

```
alert( Math.random() ); // 0.5938208589795977 (you'll ▷
probably get something else)
alert( Math.random() * 10 ); // 6.4271276677027345 (again,▷
your mileage may vary)
```

Prepared exclusively for admin@wedoev.com

round / ceil / floor

`Math.round` is your friend when you want to round a number to the nearest whole number. If you need to round up, use `Math.ceil`. Rounding down? `Math.floor` is here to help.

Example 3.34

```
alert( Math.round(10.4) ); // 10
alert( Math.round(10.5) ); // 11
alert( Math.ceil(10.4) );  // 11
alert( Math.floor(10.5) ); // 10
```

Combining `Math.random` and `Math.floor`, we can write a function that will return a random number between a minimum and maximum number:

Example 3.35

```
function getRandomNumberInRange(min, max) {
    return Math.floor( Math.random() * (max - min + 1) + ▷
min);
}

alert( getRandomNumberInRange(0, 100) ); // 39; you'll ▷
probably get something different.
```

pow

This is the function to use when taking the power of a number. The first parameter is the number you're taking the power of and the second parameter is the power.

Example 3.36

```
alert( Math.pow(2, 3) ); // 8
alert( Math.pow(2, -1) ); // 0.5
```

Prepared exclusively for admin@wedoev.com

Summary

Well, it's been a long haul, but you've pretty much mastered all the basics; you're familiar with all the data types built into JavaScript, and all their methods. You're also proficient using function and control structures (loops and conditionals). With the basics under your belt, we can now move on to some slightly more difficult techniques.

Prepared exclusively for admin@wedoev.com

4

Prepared exclusively for admin@wedoev.com

More Concepts and Best Practices

You now have a good grasp of the basics of JavaScript. This chapter will take that knowledge up a level: we'll discuss some of the more complex JavaScript features, as well as some good practices for coding. So let's not waste another moment!

this

The first this we're going to tackle is **`this`**. Wait, what? That's right: **`this`**. It's rather hard to talk about, because of the name, but it's an important concept to understand. **`This`** is a keyword in JavaScript that you can think of as a dynamic variable that you can't control. The value of **`this`** changes based entirely on where you are in the code. Let's take a look.

For the majority of your code, **`this`** will reference the global object. I've mentioned the global object very briefly, but now is a good time to get to know it better. Remember the **`parseInt`** and **`parseFloat`** functions we talked near the end of the last chapter? These are functions are actually methods of the global object. This means you can access them via this format:

Example 4.1

```
alert( this.parseInt("10.2", 10) ); // 10
alert( this.parseFloat("3.14159") ); // 3.14159
```

Here's a general rule: unless you've done something to change the value of **`this`**, it refers to the global object. When this is the case, anything you'd access without specifying an object is a property of the global object. Yes, this even applies to global-level functions and variables of your own. Try this:

Prepared exclusively for admin@wedoev.com

Example 4.2

```
var my_variable = "value";
function my_function () { return "another_value" }

alert( this.my_variable ); // "value"
alert( this.my_function() ); // "another_value"
```

`this` doesn't really have a proper name, but it does have a property that refers to itself: `window`. If you could see this code, it might look something like the following:

```
var global = {
    window : global
    . . .
};
this = global;
```

This means you can also access your variables and functions as properties of `window`, which is more common than using `this`. It gets a bit more confusing here, because you can access `window` to use things on the global object even when `this` refers to something else. Even then, you don't have to call it off `window`, unless you have a local variable or function overwriting it.

Now that we've (somewhat) got a hold on what `this` is by default, let's look as how we can change it.

The new Keyword

The first way of changing `this` that we'll look at is yet another keyword: `new`. But first, some context.

You know what a JavaScript object is. Keeping most of your related functionality in an appropriate object and accessing it via an appropriate interface is a very (very, very) basic description

Prepared exclusively for admin@wedoev.com

of what's called "Object-Oriented Programming" (OOP). Many programming languages use classes to do OOP. Classes are like blueprints: they aren't the real objects, but they lay out what the objects will have when they are created. JavaScript is very object-oriented, but it doesn't use classes. Instead, it uses prototypes. Prototypes are real objects (the same objects you've already seen), and we can use them as a "living" blueprint for other objects.

With all that said, let's get back to **`new`**. The classes in other languages use constructor functions to construct the new objects. In JavaScript, constructors are regular functions, written to create new object and called with the keyword **`new`** in front of them. Let's take a look at how to write and call this type of function.

Let's say we want to create truck objects. I know: this isn't necessarily practical. You probably won't ever need truck objects in JavaScript, but it's something you'll be familiar with while we discuss **`new`** and it's crew.

So, here's a function that will create a truck:

Example 4.3

```
function Truck(model) {
    this.num_of_tires = 4;
    this.kilometers = 0;
    this.model = model;
}

var my_truck = new Truck("Hercules");

console.log(my_truck);
```

Notice two things here: first, we're defining all these variables as properties of the object **`this`**. Second, we don't return anything from this function. Both of these are because we plan to call this

Prepared exclusively for admin@wedoev.com

function with the **new** keyword. At the end there, you can see how we call this function.

Take note of how we use **new** here. This does two things to the function. First, it changes **this** to a new clean object. Obviously, we then add our custom properties within the function. The second thing **new** does is assure that **this** is returned. Now the value of **my_truck** is just the same as if we had done this:

Example 4.4

```
var my_truck = {
    num_of_tires : 4,
    kilometers : 0,
    model : "Hercules"
};

console.log(my_truck);
```

That's the idea of using **new**: we get a super easy way to create objects. Obviously, you won't use this when you just need one or two simple objects. It's most useful when you have a lot of functionality to pack into an object, or you want to create a lot of similar objects.

We'll come back to **new** a bit later. For now, let's discuss the other ways to change **this**.

call and apply

I mentioned earlier that everything in JavaScript is an object—even functions. Since functions are objects, they can have properties and methods. **call** and **apply** are two methods that functions have by default.

Prepared exclusively for admin@wedoev.com

The only reason for these functions to exist is to change the value of `this` within the function. Let's see how to do it.

Example 4.5

```
function Truck (model, num_of_tires) {
    this.num_of_tires = num_of_tires;
    this.kilometers = 0;
    this.model = model;
}

var basic_vehicle = { year : 2011 };

Truck.call(basic_vehicle, "Speedio", 4);

console.log(basic_vehicle);
```

ROCK* TIP

An aside on primitive vs. reference values: You might think that, although we're modifying an existing object, we will lose those changes unless we assign them to a new variable, or back to basic_vehicle. That would be the case if our variable was a primitive value. However, objects (and arrays) are reference values: values passed by reference. A primitive value (such as a string or number) is stored in a spot in your computer's RAM, and we keep track of it with a variable. When we pass that variable to a function, we copy that value to a new spot in memory, point the parameter name to that spot, and work with that inside the function, so that the original is unharmed. It's different with reference values: when we pass an object or array to a function, the parameter points to the very same spot in memory as the original variable. This means that the this inside Truck.call(basic_vehicle) is the very same basic_object we have outside the function. So there's no need to assign anything, because basic_vehicle now has the properties assigned by Truck. What would happen if we assigned the returned value of Truck.call(basic_vehicle) to a variable? It would be undefined, because Truck without new doesn't return anything.

Prepared exclusively for admin@wedoev.com

Here, we're calling the `Truck` function via it's `call` method. The first parameter of `call` is the object we want to be considered `this` within the function. Since we're not using `new` with this call to `Truck`, it doesn't create a new object or return `this` by default. That's what we want in this case, because we're modifying an existing object.

After the object that will be `this` (you could call this the context of the function), we can pass in any parameters necessary. These will be passed to the function being executed. As you can see in our above example, the second parameter "Speedio" is being passed to `Truck` as its first parameter. The third parameter `6` becomes the second parameter of `Truck`. Of course, this will work for as many parameters as you required

There's an alternate form of `call`, named `apply`. The difference here is that apply takes **only** two parameters. The first is the context object. The second is an array of parameters that will be passed to the function. This is useful if you have the parameters as an array, instead of as loose variables. For example, let's say we ask the user to enter the model of the truck and the number of wheels it has into a textbox. The string `user_input` represents what we may have gotten back from them:

Example 4.6

```
function Truck (model, num_of_tires) {
    this.num_of_tires = num_of_tires;
    this.kilometers = 0;
    this.model = model;
}
```

Prepared exclusively for admin@wedoev.com

```
var basic_vehicle = { year : 2011 },
    user_input = "Speedio 18";

Truck.apply(basic_vehicle, user_input.split(" "));

console.log(basic_vehicle);
```

This is a great place to use **`apply`**. We call **`split`** on that input string and split it on a single space character; this returns an array that would look like this: **`["Speedio", "18"]`**. If we were going to use **`Truck.call`** here, we would cache that array in a variable and pass the its elements individually. like this:

```
var user_array = user_input.split(" ");

Truck.call(basic_vehicle, user_array[0], user_array[1]);
```

Obviously, this is more work; so we can just pass that array to apply, and it will assign the elements of the array to the parameters of the function we're applying it to in order.

If you're like me, and have a hard time remembering whether it's call or apply that takes the array, then notice that **a**rray and **a**pply both start with *a* (and end with "y", and have a double consonant, if that's any help).

Inside an Object

Okay, there's one more way to change the value of **`this`**. We've talked about object literals enough that you're familiar with them. Well, remember that we can use functions as the value of a property—a method, it's usually called. Check this out:

Prepared exclusively for admin@wedoev.com

```
var waitress = {
    name : "Ashley",
    greet: function (customer) {
        customer = customer || " there!";
        return "Hi" + customter + " My name is Ashley; ▷
what can I get you?";
    }
};
```

Behold a waitress object; nothing too complicated is going on here; she's got a **`name`** and a **`greet`** function. However, what would happen if we ran this:

```
waitress.name = "Melissa";
```

That resets her name property; but she'll still be greeting people by calling herself Ashley! That won't do; but how can we reference the name property of the object?

It won't be obvious that we need to change that return message to this:

```
return "Hi" + customter + " My name is " + this.name + "; ▷
what can I get you?";
```

The million dollar question is, why do we do it that way? Let's step through what you're probably thinking. Your first thought might be to use **`name`** as a variable and just put it in there, but since it's a property, that doesn't work. Next, you might try this:

```
return "Hi" + customer + " My name is " + waitress.name + ▷
"; what can I get you?";
```

Yes, that works, but the problem here is that we could do this:

Prepared exclusively for admin@wedoev.com

Example 4.7

```
var waitress = {
    name : "Ashley",
    greet: function (customer) {
        customer = customer || " there!";
        return "Hi" + customer + " My name is " + ▷
waitress.name + "; what can I get you?";   </script>
    }
};

doppelganger = waitress;
waitress = null;

doppelganger.greet(); // failure!
```

Think about this code: remember that objects are a reference value. This means that when we create an object literal, it is stored in memory in only one place. When we set **`doppelganger = waitress`**, both variables point to the same object in memory. At this point, **`doppelganger.greet()`** would work fine. However when we remove **`waitress`**'s "pointer" to the object in memory, things break. What we need is some way to refer to the object itself without using any of the variables that point to it.

Now you see why we use **`this.name`**. Inside an object literal, **`this`** points to that object itself.

Example 4.8

```
var waitress = {
    name : "Ashley",
    greet: function (customer) {
        customer = customer || " there!";
            return "Hi " + customer + " My name is " + ▷
this.name + "; what can I get you?";
    }
```

Prepared exclusively for admin@wedoev.com

```
};
alert( waitress.greet("Joe") ); // Hi Joe My name is ▷
Ashley; what can I get you?
```

Object Oriented JavaScript

In the previous chapter, we talked for a very short while about objects. Now, our discussion of **`this`** has led us back down the trail of objects in JavaScript, and—since everything in JavaScript is an object—I think we should continue.

I should note that we're only going to touch the tip of the tip of the tip of how you can manipulate objects in JavaScript. I have screencast series on Object-Oriented Programming in JavaScript on the Tuts+ Marketplace, and if you'd like to delve into a more advanced view of OOP (in JavaScript, of course), you might want to check that out.

You know that constructor functions coupled with the keyword **`new`** are a decent way of making objects. To recap that, make sure you're grokking this:

```
function Computer (name, ram_amount, memory_amount) {
    this.name = name;
    this.RAM = ram_amount; // in GBs
    this.space = memory_amount; // in MBs
}
```

This is a normal function that will create an object if we call **`new Computer("MacBook", 2, 250000 )`**. Note that this isn't the prototype (or blueprint) of all computer objects; it's the constructor of them.

Prepared exclusively for admin@wedoev.com

What about adding a function to this computer object, such as the ability to store a file? You might be tempted to do something like this:

```
function Computer (name, ram_amount, memory_amount) {
    this.name = name;
    this.RAM = ram_amount;
    this.space = memory_amount;
    this.files = [];
    this.store_file = function (filename, filesize) {
        this.files.push(filename);
        this.space -= filesize;
    };
}
```

That seems innocuous enough; however, there is a problem here. You shouldn't create functions inside your constructor functions because every time you create a new copy of that object, you'll be creating a new copy of that function in memory. There's no need for this, because we can have just one copy of the function that works for all Computer objects. This works because we use **`this`** to refer to the object inside the function.

So what we need is a way to point every copy of the Computer to a **`store_file`** function. It's nice the way we've got it now, because the function is a method, so **`this`** refers to the given object. We can do this—and only need one function—with the prototype of Computer. Here's how that's done:

```
Computer.prototype.store_file = function (filename, filesize) {
    this.files.push(filename);
    this.space -= filesize;
};
```

Remember that functions are objects too, so they can have properties. In this case, we're accessing the **`prototype`**

Prepared exclusively for admin@wedoev.com

property, which is an object that all Computer objects inherit from. What that means is this: when you run `my_computer.store_file("image.jpg", 26)`, the JavaScript engine sees that `my_computer` doesn't have a method called `store_file`, so it looks on `my_computer`'s prototype object, which is `Computer.prototype`. It will find it there, and execute it as a method of `my_computer`. If it didn't find it there, it would continue up the *prototype chain*. In this case, the next step up is `Object.prototype`, which is the top of everything in JavaScript.

I want to make sure you're understanding what is really going on here. There are two main parts to creating an object with a constructor function and the keyword `new`:

1. The constructor function creates an object (denoted by `this` inside the function); within the function, we are assigning the values of properties of the object.
2. The prototype, or parent object, of every object made with a given constructor function is stored at `ConstructorFunction.prototype`. All the methods and properties of this object are accessible from the "children" objects.

This is why it's called "Prototypal Inheritance"; each object inherits from the same prototype. It's important to note that if a given child object has a given property of it's own—say, `my_obj.name`—the prototype won't be checked for the property. Also, note that when you change a property or method from an object, you're changing the property on that object, not the prototype object.

Let's look at an example, just to cement this down:

Prepared exclusively for admin@wedoev.com

Example 4.9

```
function Product(name) {
    if (name) {
        this.name = name;
    }
}
Product.prototype.name = "To be announced";
Product.prototype.rating = 3;

var ipad = new Product("iPad");

alert( ipad.name ); // "iPad";
alert( ipad.rating ); // 3
```

Here, we've created a simple **`Product`**. Right now, you can see that the **`name`** property is right on the object **`ipad`** and the **`rating`** property is being inherited from its prototype.

Example 4.10

```
// ipad from above

ipad.rating = 4;

ipad.rating; // 4

Product.prototype.rating; // 3
```

By changing the rating, we're actually adding a rating parameter to the **`ipad`** object. Since it has its own property called **`rating`** now, when we ask for **`ipad.rating`**, it no longer goes to the prototype… however, the prototype's value for **`rating`** hasn't been changed.

Prepared exclusively for admin@wedoev.com

Object Methods

We discussed methods for every type except Objects at the end of the last chapter. This was for two reasons:

- I wanted to keep all object-related material together, right here.
- Most of the methods you'll use on objects will be ones you create yourself.

However, there are several useful methods that are built into objects. Since every object you create is an `Object` (i.e, inherits from `Object.prototype`), these methods are stored there.

ROCK* TIP

This revelation might get you wondering where the methods for, say, strings, are stored. Well, there is a `String` object, which has a `prototype` property. That's where they are kept. Some string methods are also object methods; since everything is an object—meaning that `String` inherits from `Object.prototype`, too—`String` doesn't need to have its own versions of those methods, except when they are different from the `Object` versions.

!

hasOwnProperty

We talked about using a for-in loop to iterate over arrays in the last chapter; you can use this on objects, too.

Example 4.11

```
function Person(name) {
    this.name = name;
}
Person.prototype.legs = 2;

var person = new Person("Joe"), prop;

for (prop in person) {
```

Prepared exclusively for admin@wedoev.com

```
        console.log(prop + ": " + person[prop]);
    }

    // in console:
    // name : Joe
    // legs: 2
```

However, there's something to be aware of here. If your object is inheriting from a prototype with properties, those properties will be looped over as well. This might make for-in cough up results you didn't want. In this case, objects have a handy method for determining if a given property belongs to the object or its prototype: it's called **`hasOwnProperty`**:

Example 4.12

```
    function Person(name) {
        this.name = name;
    }
    Person.prototype.legs = 2;

    var person = new Person("Joe"), prop;

    for (prop in person) {
        if (person.hasOwnProperty(prop)) {
            console.log(prop + ": " + person[prop]);
        }
    }

    // console:
    // name: Joe
```

It's a bit inefficient to have to write that inner if block every time you want to loop over an object, but get used to it; it's a best practice.

Prepared exclusively for admin@wedoev.com

toString

Every object has a **toString** method. This method is called when the object is to be printed out anywhere; it converts it to a string. The different built-in types do different things: strings obviously don't change; numbers become strings of numbers; dates become nicely formated date strings. But what about general objects?

Example 4.13

```
var o = { name : "Andrew" };
alert( o.toString());  // "[object Object]"
```

This is pretty useless, because it tells you absolutely nothing about the object, other than that it is indeed an object. Use this as a reminder to create a custom **toString** method if you'll need it. You could do this:

Example 4.14

```
var person = {
    name : "Joe",
    age  : 30,
    occupation: "Web Developer",
    toString : function () {
        return this.name + " | " + this.occupation;
    }
};

alert( person.toString() ); // "Joe | Web Developer"
```

Usually, you won't be calling the **toString** method yourself; the JavaScript engine will call it for you when you try to use the object (built-in type or not) where you would use a string.

Prepared exclusively for admin@wedoev.com

valueOf

This method returns a primitive value that would represent your object. For example:

Example 4.15

```
var account = {
    holder : "Andrew",
    balance : 100,
    valueOf : function () {
        return this.balance;
    }
};

alert( account + 10 ); // 110
```

As you can see, when we try to add 10 to **`account`**, we get 110, even though account is an object. Obviously, we can't assign to objects like this, but it works for this kind of thing. This method is like **`toString`**, in that you don't usually call it yourself.

You might be wondering when the JavaScript engine decides to use **`toString`** and when to use **`valueOf`**. That's a bit complicated for this beginner's book; this StackOverflow thread might be of help.

That's it; as you can see there really isn't much in the way of object methods.

Closure

Closure is one of the most important features of JavaScript. If you're familiar with other programming languages (specifically object-oriented ones), you might know about private variables. A

Prepared exclusively for admin@wedoev.com

private variable is a variable that is accessible to the methods of an object, but isn't accessible from outside the object. JavaScript doesn't have this, but we can use closure to "make" private variables. This is just one way of putting closure to work for you. But enough about why it's useful; let's see how to implement it!

Before we begin, remember two important points about JavaScript that contribute to closure:

- Everything is an object, even functions. This means we can return a function from a function.
- Anonymous, self-invoking functions are nameless functions that run immediately and usually only run once.

Now, let's use these two ideas to create a closure. First, let's look at some "normal" JavaScript code:

```
var secretNumber = 1024;

function revealSecret(password) {
    if (password === "please") {
        return secretNumber++;
    }
    return 0;
}
```

This should be old hat to you now. We have a variable called **`secret_number`** and a function that returns that number if the password is correct. The post-increment operator adds one to **`secret_number`** after we have used it (in this case, that means after we have returned it); this way, we get a unique number each time we call the function. If the password is wrong, we get 0.

Prepared exclusively for admin@wedoev.com

However, there's an obvious problem there: there's nothing secret at all about **secret_number**; it's a global variable that everyone has access to. To fix that, why don't we just put it inside the function? Then, we won't be able to view or change it from outside the function.

Hold on, though; we can't do that either. If we do, each time we call the function, it will reset **secret_number** to 1024. What to do? Closure to the rescue! Examine the following:

Example 4.16

```
var revealSecret = (function () {
    var secretNumber = 1024;

    return function (password) {
        if (password === "please") {
            return secretNumber++;
        }
        return 0;
    };
}());

alert( revealSecret("please") ); // 1024

alert( revealSecret("please") ); // 1025
```

Notice that **revealSecret** is being assigned to the return value of the anonymous self-invoking function. Since that function runs right away, **revealSecret** will be whatever is returned. In this case, that's a function. Now, here's the impressive part. Notice that the internal function (the one being returned) references the **secretNumber** variable from its "parent" function's scope. Even after the anonymous function has returned, the inner function will still have access to this variable. That's closure: an inner function having access to the scope of it's parent function, even after the

Prepared exclusively for admin@wedoev.com

parent function has returned. This way, **secretNumber** is set only once, it can safely be incremented each time it's used, and no one can get to it, because it's protected by the scope of the anonymous function. Putting it another way, closure is the concept of exposing limited control of a function's scope outside the function.

Another place where this is useful is in creating modules. Check this out:

```
var a_module = (function () {
    var private_variable = "some value";

    function private_function () {
        // some code;
    }

    return {
        public_property : "something"
        // etc : " ... "
    };
}());
```

This is the module pattern: inside a anonymous self-invoking function, we create variable and methods that we don't have available from the outside. Then, we return an object with public properties and methods that use these private variable and methods. This way, the "guts" that shouldn't be exposed aren't. If you want to read more about the module pattern, check out this post of the YUI Blog.

You probably understand the "how" of closure. Maybe you're still struggling with the "why." After all, it's your code and you know enough not to hurt anything, or change something that shouldn't be changed. Well, building in protection like this is one of the best practices of good coding. It's defensive; coding this way means

Prepared exclusively for admin@wedoev.com

you're aware of the things that can go wrong and are actively preventing them. Also, there's a good chance your code will eventually be used by others; they won't know necessarily how things are "supposed" to work, and coding this way will prevent rampant breakage in such situations.

For example, look back up at the `revealSecret` function about. We might know not to change the `secretNumber` variable, but another developer might not…or we might forget. Basically, any situation where you want to be able to change a value over multiple executions of a function (meaning you can't put the value inside the function), but not have that value changeable from outside the function (meaning you can't put the value outside the function) is a situation where closure comes to your rescue.

It's not just other programmers that you have to prepare for. Nine times out of ten, your programs will interact with data from the user. But who is to say that the user will docilely do as they are told and input the right type of information. Defensive coding sometimes means having a backup plan for situations like these.

Errors

This brings us to errors in your JavaScript. Now, there are really two types of errors: errors you make while coding—things like syntax errors and typos—and errors that are unforeseeable—things are break because of something outside your code, such as the user's input, or a missing feature in the JavaScript engine. While the JavaScript engines usually silently solves small errors (such as a missing semicolon), a larger mistake will make it throw (yes, that's the technical term—throw) an error. Also, we can throw our own errors, if something goes wrong.

Prepared exclusively for admin@wedoev.com

Here's the error-protection syntax:

```
try {
    // code that might cause an error here
} catch (e) {
   // deal with the error here
}
```

We wrap the code that might throw an error in a **`try`** block. If we don't do this, the JavaScript engine will deal with the error itself, which usually means stopping the execution of your code and potentially showing the user the error: not good. If an error is thrown, the error object (which we'll look at soon) is passed to the **`catch`** block (you can call it whatever you want; in this case, I'm calling it **`e`**, which is common). Inside the **`catch`** block, you can deal with the error.

Different web browsers include different properties on the error object. They all include the error name and the error message. Firefox also includes the file name and line number of the error. Chrome includes a stack trace. There are other things, but the main ones are **`name`** and **`message`**, which are the type of error and a description of what happened, respectively. You can throw your own errors using the **`throw`** keyword and creating an error object. You can create this error object as an object literal, or you can use the error constructor functions.

What kinds of things will the JavaScript engine throw an error on? Try calling a function that doesn't exist, or accessing a property of **`null`**; yep, you'll get an error. However, it's useful to be able to throw our own errors, so let's see how it's done:

Prepared exclusively for admin@wedoev.com

Example 4.17

```
function make_coffee(requested_size) {
    var sizes = ["large", "medium", "small"],
        coffee;

    if (sizes.indexOf(requested_size) < 0) {
        throw new Error("We don't offer that size");
    }

    coffee = { size: required_size, added: ["sugar", ▷
"sugar", "cream" ] };

    return coffee;
}

try {
    make_coffee("extra large");
} catch (e) {
    console.log(e.message);
}
```

You might find this to be a common pattern. When there's the possibility of an error, you'll throw it within a function. Then, you'll wrap the calling of that function in a `try` statement and catch the error, if there is one. The way it works is that as soon as the error is thrown, the program jumps to the `catch` statement, so nothing after the error is ever executed. As you can see, we're using the `Error` constructor function, passing in our error message.

Prepared exclusively for admin@wedoev.com

Testing your JavaScript

Let's talk about testing your code. Testing is like pictures of kittens: it's impossible to have too much of it. You should constantly be testing your code. There are two main types of testing: performance testing and unit testing.

- **Unit testing** refers to testing each bit of functionality (each unit) in your code. You may even want to use the practice "Test Driven Development," which means you write your tests first, so you know what the code you are about to write is supposed to do.
- **Performance testing** refers to making sure you have written your code in the fastest way possible. No, I'm not referring to the length of time it took you to code. I mean the length of time is takes your code to run. It's important to have efficient code.

Let's look at each of these types of testing individually.

Unit Testing

So, testing each unit of your code: it sounds like a good idea, but how do you do it? Well, here's the basic process:

- Write your tests
- Run your tests and watch them fail
- Write your code
- Run your tests and watch them pass
- Refactor (clean up) your code
- Rinse and repeat for each feature

Prepared exclusively for admin@wedoev.com

This is the process for test-driven development: by writing the tests first, you'll have a goal to work towards. Don't feel bad about failing tests; at first, you want them to fail. If they don't fail, you've probably misunderstood your goals or written poor tests.

So how do you write these tests? Often, you'll use a testing framework (we'll talk about frameworks and libraries in the next chapter). However, it's not too hard to write your own testing functionality.

Before we write our tests or test functionality, let's ask this: what types of things should we be testing for? Here are the two types of tests common testing frameworks implement:

- **assert:** determines is something is true or not
- **equal:** determines if two things are equal

We could write these functions ourselves…so let's do that now! Note: this will assume that you're testing your JavaScript in a web browser, because this is the case more often than not. This means we're going to deal with the Document Object Model, which you might not know about yet. Don't worry, we cover all the ugly aspects of the DOM in the next chapter. You can jump there if you'd like, or hang on for the ride. It won't be much right now, so you shouldn't have to worry.

Example test.js

```
var TEST = (function () {

    var results = document.createElement("ul");
    results.id = "tests";
    document.body.appendChild(results);

    function log(class_name, msg) {
```

Prepared exclusively for admin@wedoev.com

```
            var li = document.createElement("li");
            li.className = class_name;
            li.appendChild(document.createTextNode(msg));
            results.appendChild(li);
        }

        return {
            assert : function (bool, msg) {
                log((bool) ? "pass" : "fail", msg);
            },
            equal : function (first, second, msg) {
                log((first === second) ? "pass" : "fail", msg)
            }
        };
    }());
```

Notice I'm using the module pattern to create our little testing framework. It creates an unordered list and sticks it on our page. Then, for each test we execute, we stick a new list item in the list. If the test passes, we give the list item a class of "pass"; if it fails, we give it a class of "fail." This way, we can style the list through CSS.

Our framework has two methods The **`assert`** method takes two parameters. The first parameter is the value we're testing. If its Boolean value is true, the test passes. Using the conditional operator makes this extremely easy, because the only difference between a passing test and a failing test is the class on the list item. The second parameter is the message that identifies the test.

The second method, **`equal`**, takes three parameters. If the first two are equal, the test passes. If they aren't, it fails.

Let's step away from testing for a moment to make sure you understand the code above. Since both test functions are pretty similar, I've taken that similar functionality and put it in a private method called **`log`**; it's private because we're using the module

Prepared exclusively for admin@wedoev.com

pattern, and we have access to it via closure. Inside each testing method, we just call log and pass it the class name to give the element and the message to put in the list item. As I mentioned, I'm using the conditional operator to decide whether the test passes or fails right inside the call to the function. Inside `the` log function, we create the list item, give it the appropriate class and test, and append it to the unordered list we created.

ROCK* TIP

I haven't included the CSS that goes along with this little test framework here; if you want to see that, and follow along with these tests, check out the example files for these examples in the downloadable source code.

!

So, let's use the framework now. Obviously, we're not going to build an entire project right here, but let's say we're going to build a Business object. We start by writing the most simple test possible, to test for its existence.

Example 4.18

```
TEST.assert(Business, "Existence of Business function");
```

If you run this now, the test will fail; actually, you won't even get a test, because your browser will throw an error saying there's no such thing as `Business`. That's ok; our test obviously failed. Now, let's write the function:

Example 4.19

```
function Business (name, year_founded) {
}
TEST.assert(Business, "Existence of Business function");
```

Now we have our function; re-run the test and it should pass! Now, let's add some other tests:

Prepared exclusively for admin@wedoev.com

```
var b = new Business("my business", 2000);
TEST.equal(b.name, "my business", "b.name === 'my ▷
business'");
TEST.equal(b.year_of_founding, 2000, "b.year_of_founding ▷
=== 2000");
```

They fail, of course, so let's fix that:

Example 4.20

```
function Business (name, year_founded) {
    this.name = name;
    this.year_of_founding = year_founded;
}

var b = new Business("my business", 2000);
TEST.equal(b.name, "my business", "b.name === 'my ▷
business'");
TEST.equal(b.year_of_founding, 2000, "b.year_of_founding ▷
=== 2000");
```

Now, they should pass. One more:

```
TEST.assert(b.get_age, "b has get_age");
TEST.equal(typeof b.get_age, "function", "b.get_age is a ▷
function");
```

After making sure it fails, you can add the function:

Example 4.21

```
function Business (name, year_founded) {
    this.name = name;
    this.year_of_founding = year_founded;
}
```

Prepared exclusively for admin@wedoev.com

```
Business.prototype.get_age = function () {
    return (new Date().getFullYear()) - this.year_of_ ▸
founding;
}


var b = new Business("my business", 2000);

TEST.assert(b.get_age, "b has get_age");
TEST.equal(typeof b.get_age, "function", "b.get_age is a ▷
function");
```

Now, it should pass.

This is just a very basic example of how you might do testing. There are whole books written about this kind of thing, so I can't really do the topic justice. If you want to learn more, check out "Test-Driven JavaScript Development" by Christian Johansen. You can read an excerpt of the book over on Nettuts+.

Performance Testing

Often, there are several ways to do something with JavaScript. This doesn't mean all ways are equal, however. Very often, one way is faster than another. This is where performance testing comes in. In this type of testing, you test two (or more) different ways of doing something and compare their speeds.

If you want to test something small, there's an informal way of doing this.

- Store the current time
- Run your test

Prepared exclusively for admin@wedoev.com

- Store the new current time
- Subtract the two times to see how long it took

If your code will be running in different browsers, you'll want to run your tests in all those environments. Sometimes—often due to the way the JavaScript engine is implemented—what's fastest in one browser won't be what's fastest in another.

Let's see how to implement one of these tests. But what to test? We'll use an test I did a while ago. Here's the scenario I had: I had two arrays, and I wanted to add all the items from the second array to the first array. There are three ways to do this:

1. Loop over the second array and push each item into the first array one by one.
2. Use array's `concat` method; this returns a new array with the items in both arrays, but we can just reassign the variable that has the first array
3. Use the fact that array's `push` method can take as many parameters as we want to give it and use the function's `apply` method to turn that second array into a list of parameters.

So, let's see how we would test this; I'll go over this piece by piece:

Example 4.22 a

```
var d1, d2, d3, d4, d5, d6,
    arr1 = [1,2], arr2 = [],
    i = 0, len;

```

Prepared exclusively for admin@wedoev.com

```
for ( ; i < 100000; i++) {
    arr2[i] = i;
}

i = 0;
len = arr2.length;
```

This sets up our testing environment. We've got a bunch of variables to keep our times. We have the first array and the second array. We're putting 100,000 items in the second array because modern JavaScript engines are so fast that we need to make a big test, or else everything will happen in zero milliseconds. Now, let's run the first test:

Example 4.22 b

```
d1 = new Date().getTime();

for ( ; i < len; i++) {
    arr1.push(arr2[i]);
}

d2 = new Date().getTime();
```

Here it is; we capture the current time, run the test, and capture the new current time. As you may recall, the **`getTime`** method of Date objects return the number of millisecond since January 1, 1970. If we subtract **`d1`** from **`d2`**, we'll get the number of milliseconds it took to run the test (see a few paragraphs down).

Next test!

Prepared exclusively for admin@wedoev.com

Example 4.22 c

```
arr1 = [1,2];
d3 = new Date().getTime();

arr1 = arr1.concat(arr2);

d4 = new Date().getTime();
```

We have to start by resetting **arr1**, because it was changed in the previous test. We've captured our times, so let's do the last test now:

Example 4.22 d

```
arr1 = [1,2];
d5 = new Date().getTime();

Array.prototype.push.apply(arr1, arr2);

d6 = new Date().getTime();
```

This might seem a bit cryptic. Basically, we're using the fact that the **apply** method of functions takes an array of arguments as it's second parameters. Since we can't just do **arr1.push(arr2)** (that would make **arr2** an item in **arr1**, instead of each of **arr2**'s items), we're doing it this way.

At the end, we can do this:

Example 4.22 e

```
console.log("looping and pushing: ", (d2 - d1) );
console.log("array's concat: ", (d4 - d3) );
console.log("push via apply: ", (d6 - d5) );
```

Prepared exclusively for admin@wedoev.com

The test with the lowest score (the quickest running time) is the fastest. Remember that time might change between browsers, so be sure to check all the browsers your code might be running in. To make this easier, I like to use the web app JSBin; it's made to easily share quick snippets of code. Just paste your tests in the left panel (the JavaScript panel), and hit "Preview." You can also click "Save," and then copy the URL. This is a unique URL that you can use to test that snippet in any browsers. If you want to try this test in JSBin, you can get it at this URL: http://jsbin.com/ejogo5

A note here: When using JSBin, you'll probably want to change the **`console.log`** lines to something like this:

```
document.write("<br />looping and pushing: " +  (d2 - d1) );
document.write("<br />array's concat: " + (d4 - d3) );
document.write("<br />push via apply: " + (d6 - d5) );
```

Fig. 4-1. JSBin

You'll want to run the test a couple times to see which one is really fastest. In my case, running this test in FireFox 3.6.10 shows that "push via apply" is the fastest. However, your mileage will definitely vary.

Prepared exclusively for admin@wedoev.com

This is a pretty rough and dirty way of testing; it's primary appeal is that it's quick and easy. It's really not very scientific. If you want to do some real performance testing, you'll want to check out JSPerf. This site will comprehensively test your code, as well as keep track of the score in all the browsers you test in. Let's see how it works:

Here's what you'll see on the home page:

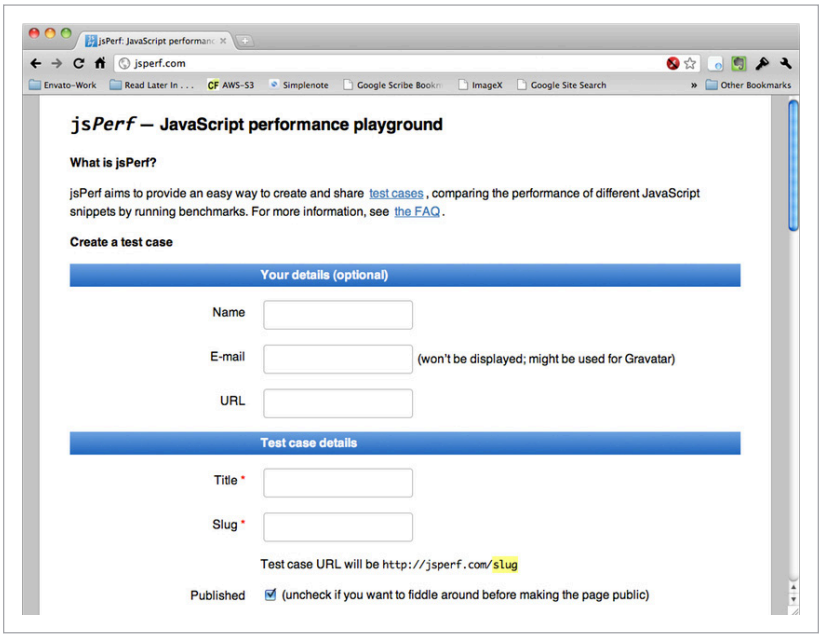

Fig. 4-2. JSPerf

It's pretty simple: just go down the form and put in your code. There's a place to put in any starting HTML or JavaScript code that's needed for the test, but isn't part of the code you want to test. Then, you add a title for each test code snippet and fill in the code. You can have as many test snippets as you want. When you're done, hit "Save Test Case." You'll be taken to the test case page, where you can run the test. Hit run test, and wait. It will run the test, tell you which code snippet was fastest, and by how much. Then, it stores the results from your browser in a table

Prepared exclusively for admin@wedoev.com

below, so you can compare different browsers. Just copy that URL and try the test in a different browser, to see how it performs. Note that JSPerf shows the results as “Operations Per Second”; therefore, a higher number is better.

JSPerf tests are publicly visible, so you can see what tests other people have run.

Organizing your JavaScript

There’s more to programming that rapidly smashing your fingers against the keyboard. There’s the meta part, like the number of files you put your JavaScript code in, or how to organize the code within those files. These are things beginners often don’t think about, so let’s take a look at them.

One File, or Many?

Let’s start with files. As you know from Chapter 1, JavaScript code is stored in a file with the extension “js”; you also know that there are two ways to include JavaScript in your web page:

```
<script src="path/to/script.js"></script>
<script>
    // include that code inline
</script>
```

Because it’s wise to separate your content and functionality, the first method above is better practice. But should all your JavaScript go in one file? Or should you separate it into different files?

When you’re coding, it’s definitely easier to break that code into several files. Since JavaScript global variables (the ones outside functions) are available from a global object accessible everywhere that JavaScript is on the page, you can create variables in one file

Prepared exclusively for admin@wedoev.com

and access them another, as long as both files are linked to in your HTML.

However, splitting your JavaScript into several files comes with a cost: that cost is paid by your website visitors. Here's why: for every file their browser needs to download, it is downloading more than just the code. There's the overhead—like HTTP headers—that comes with the file. Because of this, it's much better to store all your code in one file, to minimize the amount of overhead.

The best medium here would be to develop your code in separate files and then concatenate those files together when you're ready to publish the site. There are programs (such as build scripts) that can help you automate this process. This might be a bit beyond your skill level at this point, but if you're ready for it check out my tutorial on using Ant to compile and compress JavaScript, published on Nettuts+ (See the section below for more on compression).

Global Variables

By now you're familiar with the concept of global variables. You might think this is a great think, since everything is always accessible. However, that's not the case. Here's why: the more you write JavaScript, the more you'll find that the code you use on a given page isn't just yours: you'll use libraries, helper methods, and other code from other developers. If your site has advertising, you'll find that many advertisers insert their ads via JavaScript. If all the code relies on global variables, there's a really good chance that one script will overwrite the variables of another script. Since there's no protection against this built into JavaScript engines, you have to code very defensively. Because of this, it's a best practice to use as few global variables as possible.

How can you do this? Well—and I hate to sound like a broken record, but—here's another place that anonymous, self-invoking

Prepared exclusively for admin@wedoev.com

functions are really useful. If all your code can be contained in one place, you can do this:

```
(function () {

    // all your code goes here

}());
```

This way, you don't have to worry about anyone getting at your variables; they are all safely tucked away inside the scope of the anonymous function. And since functions can access the variables in the scope outside them, you can access any global variables that other libraries set.

But what if you're writing a library, or some similar code that needs to be accessed outside of itself? In cases like this, you'll have to use a global variable. That's not a bad thing, as long as you name and set it properly. You'll probably want to use the module pattern for this if you want to have your private variables and methods. Then, assign a returned object or function (depending on the complexity of your library) to the global variable.

I prefer to give my global variables all uppercase names; I do this for two reasons:

1. It makes it clear to me when I'm reading my code that it's a global variables.
2. It reduces the chance of having that variable overwritten, because most libraries don't do this (yet!).

You might want to adopt this principle.

Prepared exclusively for admin@wedoev.com

The Good Parts and JSLint

Here's a dirty secret I've been hiding all along the way: JavaScript has bad parts. There are several terrible features in JavaScript that you should never ever use. I haven't mentioned any of these anywhere at all in this book, so you currently have a relatively "pure" understanding of JavaScript.

However, you should really know about the bad parts, eventually. I really recommend you get Douglas Crockford's book JavaScript: The Good Parts. It will do more than just teach you what's naughty and what's nice about JavaScript. You'll really be taken to a new level of JavaScript understanding.

ROCK* TIP

Don't misunderstand me here: this doesn't mean that if you hear about something not mentioned in this book, it's a bad part; I just couldn't fit it in! Check out the appendices for more great resources and topics you should look into next.

!

Once you've read the book (or even before!), you can run your JavaScript through JSLint, Douglas Crockford's JavaScript code quality tool. At the top, paste in your code. Then, at the bottom, choose the best practices you want JSLint to check for. If it finds places where you aren't using those best practices, it will let you know.

Remember, your code will probably still run, even if it doesn't pass the JSLint test. This is just a self-check. Also, realize that there are several very reputable JavaScript developers who don't see eye to eye on everything Crockford calls a Bad Part. Ultimately, you'll have to weigh both sides and decide what's right for yourself.

Prepared exclusively for admin@wedoev.com

Fig. 4-3. JSLint

Optimizing your JavaScript

We talked a bit about how one JavaScript file is better for your site visitors than multiple ones. There's something else you can do to improve the time it takes your site to load for a visitor: JavaScript minification. The idea here is that every byte counts when a site visitor is downloading the JavaScript file. To improve that time, we can take out all the whitespace (spaces, tabs, line breaks, etc.) and shorten all the variables names, since these extra bytes are really just there to make our jobs easier, and aren't necessary for the code to run properly.

So, when you're website is finished, you can go through your code and take out all the whitespace.

Actually, that's not necessary. There are command-line tools that do this for you. Since this is only a significant benefit if you have **a**

Prepared exclusively for admin@wedoev.com

lot of JavaScript, I'll leave it to you to explore this topic more. You can check out JSMin, YUI Compressor, or Google Closure. But first, I'd really recommend reading Nicolas Zakas' articles Better JavaScript Minification and JavaScript Minification Part II. Really, this whole idea of making your JavaScript faster is quite a large topic, so if you're really interested, check out Zakas' book High Performance JavaScript.

Summary

We've covered a lot in this chapter. We've looked at Object-Oriented Programming with JavaScript prototypes. We've delved into the concepts of closure and error checking. Then, we discussed the important topics of good coding practices, including testing, organizing, and optimizing your code.

But we've still got a few topics to cover. Next up, we're going to talk about the specifics of working with HTML elements in the browser. It will be slightly intimidating, but you'll by grokking it in no time!

Prepared exclusively for admin@wedoev.com

5

Prepared exclusively for admin@wedoev.com

Working with HTML

Up until now, I've tried to talk about JavaScript in a very open way, meaning that pretty much everything you've learned so far will work in any JavaScript environment. However, there's no hiding the fact that you'll probably be writing most of your JavaScript for websites—that is, for running in a browser. So that's what we'll talk about here.

Kids, Meet the DOM

Sounds simple enough, no? The tough part is that "a browser" could mean

- Internet Explorer
- Firefox
- Google Chrome
- Safari and Mobile Safari
- Opera
- *ad finitum*

Unfortunately, there are *at least* five major browsers your code has to run in, not to mention the different versions of those browsers.

So what? Well, here's how it all plays out: every web browser implements an interface by which you can interact with the HTML elements that are in the webpage that your JavaScript is running on. But let's back up one more step. When the browser loads a page, it converts the HTML text that you have written into a tree of nodes.

Prepared exclusively for admin@wedoev.com

We'll come back to nodes soon; but for now, realize that the browser turns every HTML element into a node. This is half of what the browser brings to the table. The other half is the functionality to access all those things: it's a JavaScript object that allows you to find and manipulate this tree of nodes.

These two parts together are called the **D**ocument **O**bject **M**odel or DOM, for short.

ROCK* TIP

Not sure what a node is? Think of a family tree: each person is a node. In fact, the idea of a family is quite prevalent in the browser: nodes can have siblings, parents and children.

So, why is this all so unfortunate? Well, the main issue is that all the different browsers and versions aren't anywhere near harmonious in their implementation of the DOM standard…especially Internet Explorer. This makes it rather difficult to program, as you're often checking for one feature or another, and performing the same action in two or more ways to get one job done. This isn't fun.

The other problem is that the DOM is a pretty clumsy interface to work with; it's not designed nicely *at all*. Yes, this is definitely an opinion, but it's the opinion of the healthy majority of professional JavaScript developers. However, it's probably not going anywhere anytime soon, so you'll have to get used to it.

Actually, you don't *have* to get used to it…but it's a good idea if you do. Because if it's awkwardness, stacks of JavaScript libraries have been created to ease the pain. You could jump immediately to something like jQuery or Mootools, but it's better to learn the "real" way first.

So, let's do it!

Prepared exclusively for admin@wedoev.com

Nodes

We talked a bit about nodes previously. Every element in your HTML becomes a node in the DOM. Also, every chunk of text in your HTML becomes a node as well (a text node, obviously). Also, the attributes of elements (like the id or class) are nodes. So are the comments in a file.

So, if we had this HTML:

```
<p>We're learning about the <em>Document Object Model ▸
</em>, or <strong>DOM</strong>.</p>
```

We would have a DOM tree like this:

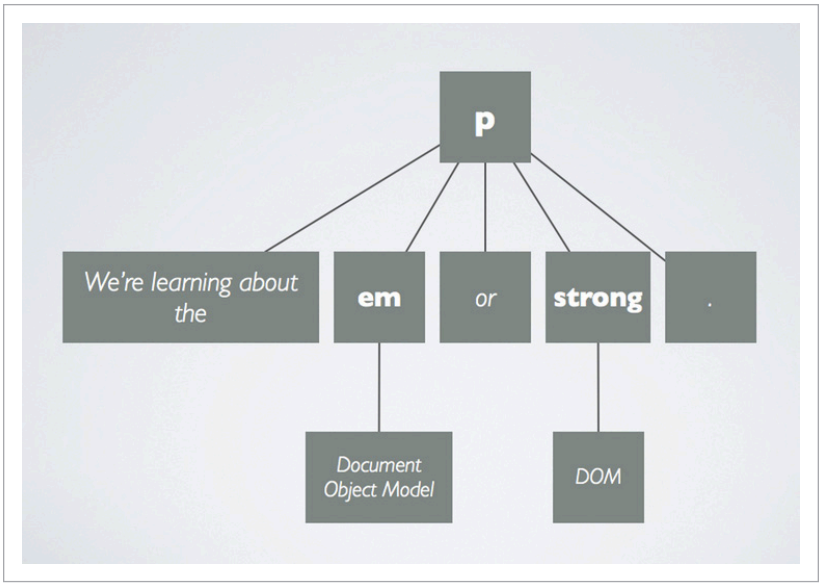

Fig. 5-1. DOM Tree of Nodes

The element nodes are bold and the text nodes are italic. As you can see, each element and bunch of sequential text is a node.

Prepared exclusively for admin@wedoev.com

Finding Elements

We've had enough theory for now; let's look at the interface to the HTML that the DOM gives us.

Often, you'll want to find one or more elements and so something with them. There are a couple of ways to do this.

By Id

Most of the time, you'll want to get a single element with a given id. Here's the way we do it:

Example 5.1

```
console.log( document.►
getElementById("id_name") );
```

Pretty much every interaction with the DOM starts with the `document` object. Here, we're calling the `getElementById` method, and passing it a single string parameter: the id of the object we want to find. If the method finds the element in the DOM, it will return it. If not, it will return null.

ROCK* TIP

But first, a word about compatibility . . . because remember, not all browsers support everything. We're going to be looking at a limited set of DOM functionality in this book, but there will still be some stuff that's not supported everywhere. So, rule of thumb: if I don't mentioned browser support, assume the functionality is supported in the following browsers

- *Internet Explorer 6+*
- *Firefox 3+*
- *Opera 10+*
- *Safari 4+*
- *Chrome 5+*

I think it's fairly safe to assume that site visitors on Firefox, Opera, Chrome, and Safari will be using a recent version of the browser. If there's an issue with one of the above browsers, I'll point that out and try to give a solution.

!

Prepared exclusively for admin@wedoev.com

By Class

If you want to get a group of elements, there are probably two things these elements have in common: the class name, or the HTML tag name (not always, but often). Hence, we can find elements based on those criteria, too. Try this:

Example 5.2

```
console.log( document.getElementsByClassName("warning") );
```

Pretty obvious name, right? However, now the discrepancies start. Internet Explorer 6, 7, and 8 don't have the **`document.getElementByClassName`** function. Although this is unfortunate, this is actually where JavaScript shines: it's relatively easy (if you're familiar with the cross-browser DOM idiosyncrasies) to create your own **`getElementsByClassName`** function. You can find the different ways to do it and how their speeds compare in an article by John Resig. With your current knowledge, you might have a bit of trouble understanding the code, but give it a try. At the very least, you can copy it and use it in your projects.

By Tag Name

If you want to get all the elements with the same tag, you can probably guess what to use:

Example 5.3

```
console.log( document.getElementsByTagName("p") );
```

Predictable, well-supported; nothing to dislike here.

Prepared exclusively for admin@wedoev.com

Traversing the DOM

Most often, you'll get your elements via one of the above functions, do what you want to do to then and move on. However, occasionally you'll want to get a node and work with other nodes close to it. For example, You might have something like this:

```
<div class="product">
    <h2> Product Name </h2>
    <img src="pic.jpg" />
    <p> Description </p>
</div>
```

You'll have, say, a dozen or so of these on a given page. Let's say you want to sort the products alphabetically by name. To do this, you'll need to get the `h2` elements. However you'll also have other `h2`s on the page. This means you need to get all the `div`s with a class "product," and then find the `h2` within each. This requires some DOM traversal skills.

Before we continue with out scenario, let's go back to that idea of a node tree. Trees are actually a very important part of computer science. You know that for a tree to be a tree, there have to be links between parent nodes and child nodes; these are one-way links. It's inefficient to have a link from the parent node to every child node; so instead, trees have a link from every parent node to its first child node. Then, every child node has a link to it's next sibling.

The DOM works exactly this way: every node has a link to its first child and its next sibling; it also has a link to its parent node. That means our HTML snippet above looks like this:

Prepared exclusively for admin@wedoev.com

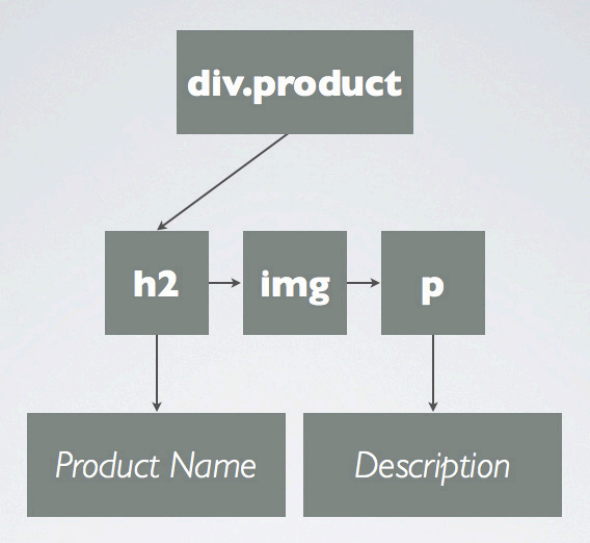

Fig. 5-2. DOM Tree snippet

Now let's move back to our scenario. With the theory we've just learned, we can see that what we need to do is find all the elements with the class "product" and then find the `h2` within it. You might think the `h2` would be the first child, but it actually isn't (yes, I intentionally mis-drew the chart for simplicities sake). This is because any whitespace between the opening `div` tag and the opening `h2` tag is captured in a text node. Then, if there are any HTML comments between those two opening tags, that gets put in a comment node. So there could possibly be many nodes to jump over before we get to the `h2`. What's a developer to do?

Thankfully, each of these different types of nodes has a property called `nodeType`; they are as follows:

- HTML elements = 1
- HTML element attributes = 2
- Text nodes = 3

Prepared exclusively for admin@wedoev.com

- Comment nodes = 8
- The document = 9

So, to find that **h2**, we just have to get the first child of the **div** node; then, we check the **nodeType** property of that node. If it's not 1, we get the next sibling node and check that. Here's the code for that:

Example 5.4

```
var products = document.getElementsByClassName("product"),
    i = 0,
    child;

for ( ; i< products.length; i++) {
    child = products[i].firstChild;
    while (child.nodeType !== 1) {
        child = child.nextSibling;
    }
    console.log(child);
}
```

We start by getting all the elements with that class "product"; remember, since we've getting more than one element, this returns an array (Well, it's not really an array; it's a *NodeList*, which can be iterated over like an array, as we see here). For each node in our list, we get the first child via the **firstChild** property and assign it to the variable **child**. Then, we're using a while loop to find the first node with a **nodeType** of 1 (meaning an HTML element). Our condition says "while the node type of child node is not 1"; this means that if the **firstChild** *is* an element, the loop won't run. If it's not, we'll run the loop, which simply reassigns the **child** variable to **child.nextSibling** (the next sibling of **child**). When we find the first element in the list of children, we can do what we

Prepared exclusively for admin@wedoev.com

want to do. We're not going to do any more than print it out to the console right now, so you can see that is, in fact, the `h2`.

Those are the basics of traversing the DOM. There are a few more links between nodes that we'll wrap up this discussion with.

childNodes

Every node has a property called `childNodes`; it's a NodeList (remember, an array-like object) containing all the child nodes of the node. Don't forget, it's zero-based.

Example 5.5

```
// <ul id="container">
//    <li>one</li>
//    <li>two</li>
// </ul>
var my_node = document.getElementById("container");

my_node.childNodes[1]; // gets the first list item (the ▷
second child node)
```

children

While `childNodes` holds all node types, `children` just includes the element nodes. These are usually what you want, so this is probably more useful that `childNodes`. However, IE 6 - 8 includes comment nodes for some reason, so beware of that.

lastChild

Works just like `firstChild`, except it gets—wait for it—the last child node.

Prepared exclusively for admin@wedoev.com

previousSibling

Just as you'd (hopefully) expect, the `previousSibling` property returns the previous sibling of the node.

parentNode

This is the one-way link every node has to its parent. Think back to our bunch of product `div`s. We wanted to sort them by product name. This meant we needed to find the `h2` element, to sort by. But if we actually got to the sorting step, we would have used `parentNode` to get the `div` parent for each `h2`.

Example 5.6

```
// <ul>
//    <li id="first">one</li>
//    <li>two</li>
// </ul>

var my_node = document.getElementById("first");
console.log( my_node.parentNode); // ul element
```

Others

There are a bunch of properties for traversing the DOM that are incredibly useful, but not incredibly well-supported. They are as follows:

- `firstElementChild`
- `lastElementChild`
- `nextElementSibling`
- `previousElementSibling`

Prepared exclusively for admin@wedoev.com

You can probably guess what they do; because most of the time you just want to work with element nodes, these properties do exactly as their already-seen counterparts do, except they ignore all those other types of nodes that just get in the way.

That's the good news. The bad news is that they aren't supported in IE 6 - 8 or Firefox 3.0 (they are in IE 9 and Firefox 3.5).

Adding, Removing, and Modifying Elements

Obviously, there's more to using the DOM than finding the right elements. Once you're where you need to be, you want to do something. When you boil it down, that something could be one of two things:

- Change the existing DOM tree (by adding, removing, or shuffling nodes)
- Change the information surrounding a given node or set of nodes. That information could be the nodes style, events, attributes, etc.

We'll come back to that second one soon. For now, let's talk about shaking things up in the DOM. Thankfully, everything here has had rock-solid support for years.

Creating Nodes

If we want to add a node to the DOM, we must first create it. To do that, we just use methods of that `document` object.

To create an element, we use `document.createElement`, passing it the tag name of the element we want to create.

Prepared exclusively for admin@wedoev.com

```
var paragraph = document.createElement('p');
```

It's just as easy to create a text node: this time we use the **`document.createTextNode`**:

```
var text_node = document.createTextNode("Put your text ▷
here");
```

Once you've created your nodes, you'll want to put them into the DOM. Let's see how!

Adding Nodes to the DOM

Now that you're somewhat familiar with the notion of trees, you'll realise that to insert a node into the tree, you'll have to do so in reference to another node. Think about it for a moment, and you'll see that you describe any insertion positioning in one of two ways:

- As the last child element of a given node
- As the previous sibling node of a given node

If you want to make your new node the last child of its soon-to-be parent node, use the **`appendChild`** method of nodes.

Assume we have the **`paragraph`** and **`text_node`** elements from above. While they aren't in the DOM yet, their DOM incarnations would look like this, respectively: **`<p></p>`** and **`Put your text here`**.

Example 5.7

```
// <div id="container"></div>

var paragraph = document.createElement("p"),
    textNode = document.createTextNode("This is some text");
```

Prepared exclusively for admin@wedoev.com

```
paragraph.appendChild(textNode);

document.getElementById("container").appendChild(paragraph);
```

Now, the HTML will look like this:

```
<div id="container">
    <p>This is some text.</p>
</div>
```

As you can see, this works for both text nodes and element nodes. To insert the text node we've created into the paragraph element node, we just use **`appendChild`**. We then use the same method to insert the paragraph into the DOM. This will make the paragraph appear on the page.

If you don't want your node appended as the last child of its parent, you'll have to get a reference to a child node you want it inserted before. Then, use the **`insertBefore`** method. Here's how we would insert another text node into our paragraph above, before the text node that's currently alone:

Example 5.8

```
// <div id="container">Text that is already here.</div>

var newTextNode = document.createTextNode("Hello World!"),
    container = document.getElementById("container"),
    textNode = container.firstChild;

container.insertBefore(newTextNode, textNode);
```

Of course, this works for elements nodes as well.

Prepared exclusively for admin@wedoev.com

Removing Nodes

When working with the DOM, it's all give and take. This means you'll occasionally want to remove elements from the DOM.

Removing elements is a bit trickier than adding them; it makes sense that you need to know about the nodes surrounding your node to put it into the DOM. However, shouldn't you be able to just remove an element without worrying about where it is? That's unfortunately not the case. You need to know the element you want to remove *and* its parent.

```
the_parent.removeChild(child_to_remove);
```

Because of this, you'll probably see this a lot:

Example 5.9

```
// <div> Kill this: <span id="kill_me"> Kill me!</span> ▷
</div>

var child = document.getElementById("kill_me");

child.parentNode.removeChild(child);
```

Get the child, and use its **`parentNode`** property to find the parent; then, remove the child you started with. Ah, well; that's the DOM for you.

There's another method of removing DOM nodes, that you'll probably find useful on occasion. That's **`replaceChild`**. Predictably, you hand it the new node and the node you want to replace, and it will swap in the new one while destroying the old one.

Prepared exclusively for admin@wedoev.com

Example 5.10

```
// <div id="container">
//    <p id="remove_me"> Remove Me!</p>
// </div>

var container = document.getElementById("container"),
    old_element = document.getElementById("remove_me"),
    new_element = document.createElement("p");

new_element.appendChild(document.createTextNode("new ▷
element"));

container.replaceChild(new_element, old_element);
```

Modifying a DOM Element

Now that you're familiar with creating, adding, and removing DOM nodes, let's talk about modifying nodes. After all, there are three main actions you can perform on an individual node:

- You can change the attributes.
- You can change the styling.
- You can add and remove event handlers.

Let's start with attributes. If you're not sure what I'm talking about, check this out:

```
<div id="content" class="wide"></div>
<input type="text" name="first_name" value="Bob" />
<span class="warning"></span>
```

Attributes are the `key="value"` part of your HTML elements. Here, I'm using `id`, `class`, `type`, `name`, and `value`, but there are many more.

Prepared exclusively for admin@wedoev.com

There are a couple of ways to work with attributes, but I'm just going to show you what the best, most cross-browser way to do it is. That way is using the methods **`setAttribute`** and **`getAttribute`**. They work as follows:

Example 5.11

```
// <div id="container"></div>

var elem = document.getElementById("container");

elem.setAttribute("class", "modal");
alert( elem.getAttribute("class") ); // "modal"
```

As you can see, we pass two parameters to the first method: the name of the attribute we want to set, and the value for the attribute. When we're getting the attribute, we just have to pass the attribute name to the **`getAttribute`** method. Obviously, there are methods of an element node.

This should work with any attribute you want to set on your element—even the new HTML5 data-* attributes. However, there's an easier way set and get some common attributes.

If you're working with the id or class, you can just do this:

```
elem.id = "content";

elem.id; // "content"

elem.className = "module";

elem.className; // "module";
```

This shorthand notation is the go-to way of setting styling on elements. Watch this:

Prepared exclusively for admin@wedoev.com

Example 5.12

```
// <div id="container">container</div>

var elem = document.getElementById("container");

elem.style.border = "1px solid red";
elem.style.backgroundColor = "green";
```

Yup; it's all the CSS properties that you already know and love, right in your JavaScript. The key thing to note it this: JavaScript property names can't have dashes in them, like they do in CSS. Therefore, use *camelCase* for those, capitalizing every letter after a dash and removing the dashes. This way, **`border-bottom-width`** becomes **`borderBottomWidth`**, and so on.

Another important thing to note is that all the style properties are strings; this makes sense if you think about it, because even numerical values like **`borderBottomWidth`** need to be suffixed with a unit (px). So, it's all strings here.

One more thing: when you're using **`element.style`**, you're working with the inline **`style`** attribute on the element. When you set a styling property, that gets added to the **`style`** attribute. The logical following of this is that you can't get a style attribute that isn't in the inline **`style`** attribute (either coded into the HTML or previously set in your JavaScript).

Events

While events could be considered a part of modifying DOM elements, they deserve a section of their own. But first, what are events?

Prepared exclusively for admin@wedoev.com

Events, put simply, are things that happen to elements in the DOM. Some examples of this might be as follows:

- The page **loaded**.
- An element was **clicked**.
- The mouse **moved over** an element.
- When something like this happens, you may often want to do something. For example, run this script when the page loads. Or, execute this function when an element is clicked.

So, exactly what events are there? Well, you can find a comprehensive list on Wikipedia, but the ones you'll use most often are these:

- `click`
- `mouseover`
- `mouseout`
- `keypress`
- `load`
- `submit`

As you can see, there are two categories of events: things the user of your site does, and things that the browser does. In the above list, the user performs the click, mouseover, mouseout, and keypress events. The browser fires the load event when the page has finished loading, and the submit event when the user clicks a submit button (so it's almost a user-perfromed events).

Prepared exclusively for admin@wedoev.com

Adding Event Handlers

An event is fired on a given DOM element. This means we can wire up an *event handler* (or *event listener*) to a given element. An event handler is a function that will be executed when a given event is fired. Let's look at an example:

Example 5.13

```
// <div id="some_button"> Click Me! </div>
var btn = document.getElementById("some_button");

function welcome_user(evt) {
    alert("Hello there!");
}

btn.addEventListener("click", welcome_user, false);
```

(Note: you'll have to run this example in a browser other than Internet Explorer; we'll resolve this soon.)

What's going on here? Well, the first bit is stuff you're getting used to: getting an element and creating a function. Next, we're calling the element's method **`addEventListener`**. You can probably guess what an event listener does: it "listens" for an event and executes a function when that event occurs. The **`addEventListener`** takes three parameters. The first is the type of event we're listening for; in this case, it's a click event. The events in the sample list above are all valid events as well, and you can find the rest at that Wikipedia page.

The next parameter is the function to run when that event is fired. In this case, we're passing in the **`welcome_user`** function. We could also pass in an anonymous function, if we wanted to. The final parameter decides whether to use capturing or bubbling. This

Prepared exclusively for admin@wedoev.com

requires a short rabbit-trail on the difference between capturing and bubbling.

First off, note that Internet Explorer 8 and below support bubbling only (IE9 and up support both bubbling and capturing), so you'll probably use that 99% of the time. Now, check out this HTML:

```
<div>
    <span> Click Here! </span>
</div>
```

Let's say I click the **span**. Because the **span** is inside the **div**, when that **span** gets clicked, the **div** is clicked as well. So both elements have a click event fire on them. If both have an event handler to harness that event, which one should be executed first? That's the background of capturing and bubbling. The DOM standard says that events should capture first, bubble second. The *capture* phase is when the event starts at the highest element and works down. In this case, the click handler of **div** will fire first. Once the event reaches the bottom—or the innermost element—it will then bubble. The *bubble* phase takes the event from the bottom and brings it back to the top, element by element. To remember which is which, remember that bubbles float up, just like events in the bubbling phase.

So, back to **addEventListener**. That last parameter is whether to handle the event in the capture phase or not. Because IE (before 9) doens't support capturing, it's usually **false** (which means bubbling).

But it's more than bubbling that IE doesn't support. Before version 9, IE had it's own event model (that model is still supported in IE 9, but it supports the standard model as well). Let's check that out.

It's actually pretty simple:

Prepared exclusively for admin@wedoev.com

Example 5.14

```
// <div id="some_button"> Click Me! </div>
var btn = document.getElementById("some_button");

function welcome_user(evt) {
    alert("Hello there!");
}

btn.attachEvent("onclick", welcome_user);
```

Instead of **`addEventListener`**, they user the method **`attachEvent`**. You'll notice that there's no third parameter (because it only supports bubbling). The other difference is that the event name must be prefixed with the word "on." Other than that, there's not much difference.

Removing Event Handlers

If you want to add event listeners, you'll eventually want to remove them. Here's how:

```
// assume the event we wired up above

btn.removeEventListener("click", welcome_user, false);
```

The method is **`removeEventListener`**. The important point is that the three parameters are all the same as the **`addEventListener`** parameters; that's how you target which event you're removing.

To remove events with the IE event model, try this:

```
// assumt the event we wired up above

btn.detachEvent("onclick", welcome_user);
```

Prepared exclusively for admin@wedoev.com

It's pretty simple.

Writing Cross-Browser Event Handlers

The fact that there are two event models means that you need to wire up two event handlers for each event. The easiest way to do that is to write a function that wraps the event behaviour:

```
function addEvent(element, type, fn) {
    if (element.addEventListener) {
        element.addEventListener(type, fn, false);
    } else {
        element.attachEvent("on" + type, fn);
    }
}
```

This is an extremely basic form of this function; there are a ton of ways to improve it. As an exercise, try creating a `removeEvent` function that removes events for both the standard model and the IE model.

Another Short Rabbit-Trail on Script Loading

Remember way back on page 9 (when you were younger, and had less JavaScript in your blood), I recommended you put script tags at the end of the body, unless (and I quote) you "write your code to wait until the page has completed loading before it begins executing." I'm bringing this up for more than nostalgic reasons: events can help you do this.

What you want to do is listen for the `load` event on the `window` object. Here's how that's done (assume we have our `addEvent` function from above):

Prepared exclusively for admin@wedoev.com

Example 5.15

```
addEvent(window, "load", function () {
    alert("do something, for example.");
});
```

There is a lot more about events that we can't cover here. I didn't even mention the event object that gets passed to the function that is called; this object provides information about the event, such as which element the event from fired on. Check out Appendix A for further resources that will teach you about events.

The DOM, In Sum

As you can see, the DOM is occasionally confusing. You're constantly jumping over text nodes; IE has it's own mind for events and more. And we haven't even talked about AJAX or even mentioned the Browser Object Model (which isn't part of the DOM, but closely related). Does this mean you're doomed to awkwardly code JavaScript for the browser for the rest of your life?

Not really. The answer is to write a library: a collection of functions that do all the heavy, cross-browser work for you. But the great part about this is that there are already several incredible JavaScript libraries out there that can make your work simpler.

What are your options? Well, do any of these JavaScript buzzwords sound familiar?

- jQuery
- Mootools
- YUI
- Dojo

Prepared exclusively for admin@wedoev.com

And that's only a few of them. These libraries give you a drop-dead simple way to work with the DOM. To whet your appetite, here's a snippet of raw JavaScript, followed by it's jQuery counterpart:

```
// Raw JavaScript
var leading_p = document.getElementById("content").
firstChild;

while (leading_p.nodeType !== 1 || leading_p.nodeName
!== 'P') {
    leading_p = leading_p.nextSibling;
}

leading_p.style.backgroundColor = "red";

// jQuery

$("#content p:first").css("background-color", "red");
```

'Nuff said, eh? I think you can see the appeal of a library. All these libraries have great sites with tons of good documentation and tutorials; check them out to learn more!

A warning here: while using a JavaScript framework is much simpler than writing the raw JavaScript, you should really understand how to do things with the raw DOM functionality (which is much more than what we've covered here).

Prepared exclusively for admin@wedoev.com

Wrapping Up

Well, That's all we're going to cover in this book. But that's definitely not all there is to learn about JavaScript. While you should now be comfortable writing functions, working with object, and (gently) manipulating the DOM, there's so much more to cover. JavaScript is growing as we speak, and there's a lot more to learn. Check out the Appendices for where to go from here.

Go forth and code JavaScript!

Prepared exclusively for admin@wedoev.com

APPENDICES

Prepared exclusively for admin@wedoev.com

Appendix A: Further Study

There are a ton of great JavaScript resources out there. Here are a few of the most popular one:

- **JavaScript: The Definitive Guide**, by David Flanigan - This book is a great resource for learning JavaScript; it covers all the basics we've covered here, and more, in much more depth. Currently, the fifth edition is available; but the link above goes to the O'Reilly website where you can get the "Rough Cut" of the sixth edition. You might want to follow the author's tweets for further developments.
- **JavaScript: The Good Parts**, by Douglas Crockford - This little book is a gem. It's a great way to become familiar with what's good in JavaScript and how to leverage that good in your programming.
- **Crockford on JavaScript**, by Douglas Crockford - This series of presentations given by Crockford in early 2010 is an excellent resource. You can catch them on the YUI Theater , but the link above goes to Nettuts+ post where I rounded them up for your convenience.
- **Object Oriented JavaScript**, by Stoyan Stefanov - Another great book for starting JavaScripters. You'll learn a lot about the prototypal nature of JavaScript in the chapters Prototype and Inheritance.
- **Nettuts+ JavaScript Posts** - If you're looking for regular, high-quality JavaScript tutorials, check out Nettuts+. I'm one of the staff writers for Nettuts+ and more often than not write about JavaScript. However, there's a ton of other great authors; it's an ever up-to-date resource!

Prepared exclusively for admin@wedoev.com

- **Rey Bango's Must-read list of JavaScript Books** -
  I could go on and on here. However, Rey Bango already has, so I'll direct you to his list. He's listed books for beginner, intermediate, and advanced levels, as well as a bunch of great JavaScript blogs you should be following.

Prepared exclusively for admin@wedoev.com

Appendix B: What We Didn't Cover

You can't learn everything there is to know about JavaScript in a weekend; it's really a lifetime pursuit. Here's a list of a few topics we weren't able to get to in this book:

- Non-base 10 numbers
- reserved words
- labels
- break and continue
- ECMAScript
  - What it is
  - Fifth edition features
- Regular Expressions
- Server-side JavaScript
- AJAX
- Cookies
- JavaScript on the Desktop
- Publish and Subscribe Events (pub-sub)

This is just a short list, but it should give you a few things to look at when continuing your JavaScript education. Good luck!

Prepared exclusively for admin@wedoev.com

About The Author

Andrew Burgess is a Canadian web developer and a staff writer for Nettuts+, where he has published numerous popular tutorials and screencasts. Andrew is also the author of the popular Rockable title called "Getting Good with Git", and a web development reviewer on Envato's Tuts+ Marketplace. As a web developer, he specializes in JavaScript and Ruby. Andrew lives with his family in Oshawa, Canada.

Check out Andrew's personal site at: http://andrewburgess.ca/

Or follow him on Twitter: @Andrew8088

Prepared exclusively for admin@wedoev.com

Your Screencasts

To download your screencasts, please use the links below.

Chapter 1

- http://rockable-extras.s3.amazonaws.com/js-screencasts/js-screencast-1.zip

Chapter 2

- http://rockable-extras.s3.amazonaws.com/js-screencasts/js-screencast-2.zip

Chapter 3

- http://rockable-extras.s3.amazonaws.com/js-screencasts/js-screencast-3.zip

Chapter 4

- http://rockable-extras.s3.amazonaws.com/js-screencasts/js-screencast-4.zip

Chapter 5

- http://rockable-extras.s3.amazonaws.com/js-screencasts/js-screencast-5.zip

Prepared exclusively for admin@wedoev.com

Ever wanted to spice up your websites with a dash of JavaScript, but not known where to start? In **Getting Good with JavaScript**, author **Andrew Burgess** breaks programming in JavaScript down into easy, straight-forward principles and practices.

This book will introduce you to important programming concepts, show you how to write your first scripts, and make you comfortable with JavaScript code. You'll learn :

- The basics of types, variables, and operators
- Best practices for efficient coding
- Testing and optimizing your JavaScript
- Interacting with HTML elements

Andrew Burgess will help you get past the learning curve and get you **Getting Good with Javascript**!

ROCKABLE