Appendix A. Leftovers: The top ten topics (we didn't cover)

We've covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter. Actually, we *did* originally include everything you need to know about JavaScript Programming (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it. So we threw most of it away, and kept the best bits for this Top Ten appendix. This really *is* the end of the book. Except for the index, of course (a must-read!).

#1 jQuery

jQuery is a JavaScript library that is aimed at reducing and simplifying much of the JavaScript code and syntax that is needed to work with the DOM and add visual effects to your pages. jQuery is an enormously popular library that is widely used and expandable through its plug-in model.

Now, there's nothing you can do in jQuery that you can't do with JavaScript (as we said, jQuery is just a JavaScript library); however, it does have the power to reduce the amount of code you need to write.

jQuery's popularity speaks for itself, although it can take some getting used to if you are new to it. Let's check out a few things you can do in jQuery and we encourage you to take a closer look if you think it might be for you.

NOTE

A working knowledge of jQuery is a good skill these days on the job front and for understanding others' code.

For starters, remember all the `window.onload` functions we wrote in this book? Like:

```
window.onload = function() {
    alert("the page is loaded!");
}
```

Here's the same thing using jQuery:

```
$(document).ready(function() {
    alert("the page is loaded!");
});
```

← Just like our version, when the document is ready, invoke my function.

Or you can shorten this even more, to:

```
$(function() {
    alert("the page is loaded!");
});
```

This is cool, but as you can see it takes a little getting used to at first. No worries, it becomes second-nature fast.

So what about getting elements from the DOM? That's where jQuery shines. Let's say you have an `<a>` element in your page with an id of "buynow" and you want to assign a click handler to the click event on that element (like we've done a few times in this book). Here's how you do that:

So what's going on here? First we're setting up a function that is called when the page is loaded.

```
$(function() {
   $("#buynow").click(function() {
       alert("I want to buy now!");
   });
});
```

Next we're grabbing the element with a "buynow" id (notice jQuery uses CSS syntax for selecting elements).

And then we're calling a jQuery method, click, on the result to set the onclick handler.

That's really just the beginning; we can just as easily set the click handler on every `<a>` element in the page:

```
$(function() {
   $("a").click(function() {
       alert("I want to buy now!");
   });
});
```

To do that, all we need to do is use the tag name.

Compare this to the code you'd write to do this if we were using JavaScript without jQuery.

Or, we can do things that are much more complex:

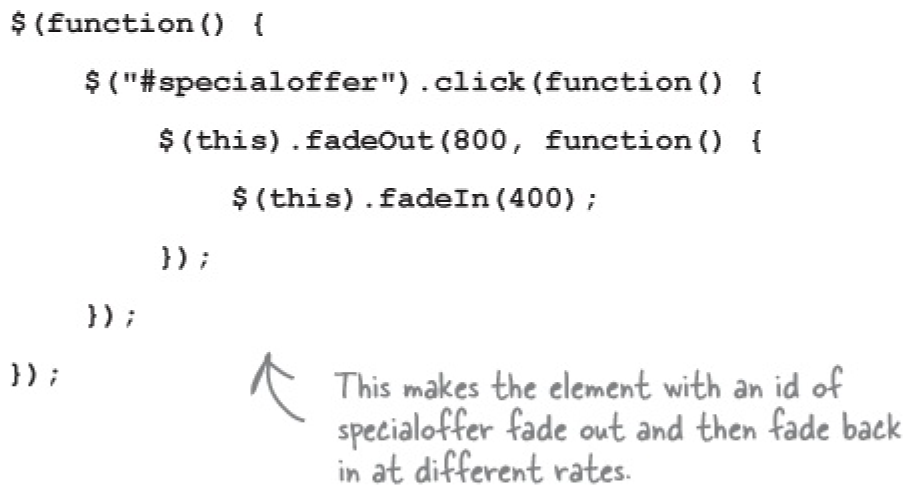
Like find all the <li> elements that are children of the element with an id of playlist.

```
$(function() {
   $("#playlist > li").addClass("favorite");
});
```

And then add the class "favorite" to all the elements.

Actually this is jQuery just getting warmed up; jQuery can do things much more sophisticated than this.

There's a whole 'nother side of jQuery that allows you to do interesting interface transformations on your elements, like this:



```
$(function() {
    $("#specialoffer").click(function() {
        $(this).fadeOut(800, function() {
            $(this).fadeIn(400);
        });
    });
});
```

This makes the element with an id of specialoffer fade out and then fade back in at different rates.

As you can see, there's a lot you can do with jQuery, and we haven't even talked about how we can use jQuery to talk to web services, or all the plug-ins that work with jQuery. If you're interested, the best thing you can do is point your browser to **http://jquery.com/** and check out the tutorials and documentation there.

NOTE

And, check out Head First jQuery too!

#2 Doing more with the DOM

We've touched on some of the things you can do with the DOM in this book, but there's a lot more to learn. The objects that represent the document in your page—that is, the `document` object, and the various element objects—are chock full of properties and methods you can use to interact with and manipulate your page.

You already know how to use `document.getElementById` and `document.getElementsByTagName` to get elements from the page. The `document` object has these other methods you can use to get elements, too:

```
document.getElementsByClassName
```

Pass this method the name of a class, and you'll get back all elements that have that class, as a NodeList.

```
document.getElementsByName
```

This method retrieves elements that have a name attribute with a value that matches the name you pass it.

```
document.querySelector
```

This method takes a selector (just like a CSS selector) and returns the first element that matches.

```
document.querySelectorAll
```

This method also takes a selector, but returns all the elements that match, as a NodeList.

Here's how you'd use `document.querySelector` to match a list item element with the class "song" that's nested in a `<ul>` element with the id "playlist":

```
var li = document.querySelector("#playlist .song");
```

This says match the element with the id playlist, and then match the first element with the class song.

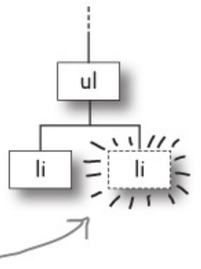
Notice how this selector is just like one you'd write in CSS?

What if you want to add new elements to your page from your code? You can use a combination of `document` object methods and element methods to do that, like this:

```
var newItem = document.createElement("li");
newItem.innerHTML = "Your Random Heart";
var ul = document.getElementById("playlist");
ul.appendChild(newItem);
```

First, we create a new <li> element, and set its content to a string.

Then we get the <ul> element we want to add the new <li> to (as a child element), and append the <li> to the <ul>.



There's a lot more you can do with the DOM using JavaScript. For a good introduction, check out *Head First HTML5 Programming*.

#3 The Window Object

You've heard of the DOM, but you should know there's also a BOM, or Browser Object Model. It's not really an official standard, but all browsers support it through the `window` object. You've seen the `window` object in passing when we've used the `window.onload` property, and you'll remember we can assign an event handler to this property that is triggered when the browser has fully loaded a page.

You've also seen the `window` object when we've used the `alert` and `prompt` methods, even though it might not have been obvious. The reason it wasn't obvious is that `window` is the object that acts as the global namespace. When you declare any kind of global variable or define a global function, it is stored as a property in the `window` object. So for every call we made to `alert`, we could have instead called `window.alert`, because it's the same thing.

Another place you've used the window object without knowing it is when you've used the `document` object to do things like get elements from the DOM with `document.getElementById`. The `document` object is a property of `window`, so we could write `window.document.getElementById`. But, just like with `alert`, we don't have to, because `window` is the global object, and it is the default object for all the properties and methods we use from it.

In addition to being the global object, and supplying the `onload` property and the `alert` and `prompt` methods, the `window` object supplies other interesting browser-based properties and methods. For instance, it's common to make use of the width and height of the browser window to tailor a web page experience to the size of the browser. You can access these values like this:

```
window.innerWidth
window.innerHeight
```

Use these properties to get the browser window's width and height in pixels. Note that older browsers don't always expose these properties.

Check out the W3C documentation[2] for more on the `window` object. Here are a few common methods and properties:

```
window.close()
```

This method closes the browser window.

```
window.setTimeout()
window.setInterval()
```

You already know these methods; they're supplied by the window object.

```
window.print()
```

Initiates printing the page to your printer.

```
window.confirm()
```

This method is similar to prompt, only it gives the user the choice of an Okay or Cancel button.

```
window.history
```

This property is an object containing the browsing history.

```
window.location
```

This property is the URL of the current page. You can also set this property to direct the browser to load a new page.

#4 Arguments

An object named `arguments` is available in every function when that function is called. You won't ever see this object in the parameter list, but it is available nevertheless every time a function is called, in the variable `arguments`.

The `arguments` object contains every argument passed to your function, and it can be accessed in an array-like manner. You can use `arguments` to create a function that accepts a variable number of arguments, or create a function that does different things depending on the number of arguments passed to it. Let's see how `arguments` works with this code:

We're not going to define any formal parameters for now. We'll just use the arguments object.

Like an array, arguments has a length property.

```
function printArgs() {
    for (var i = 0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
```

And we can access each argument using array notation.

```
printArgs("one", 2, 1+2, "four");
```

Here we call printArgs with four arguments.

JAVASCRIPT CONSOLE

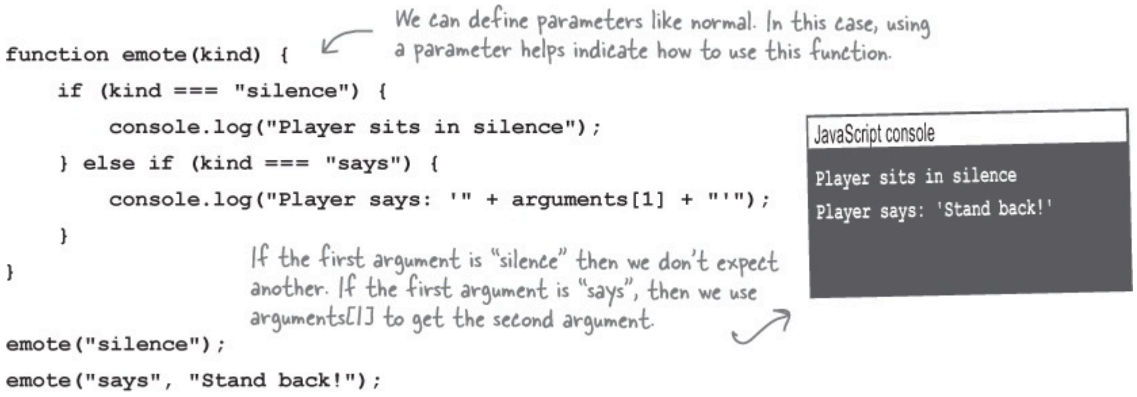
```
one

2

3

four
```

While `arguments` looks just like an array, it is not actually an array; it's an object. It has a `length` property, and you can iterate over it and access items in it using bracket notation, but that's where the similarity with an array ends. Also, note that you can use both parameters and the `arguments` object in the same function. Let's write one more piece of code to see how a function with a variable number of arguments might be written:



```
function emote(kind) {
    if (kind === "silence") {
        console.log("Player sits in silence");
    } else if (kind === "says") {
        console.log("Player says: '" + arguments[1] + "'");
    }
}

emote("silence");
emote("says", "Stand back!");
```

We can define parameters like normal. In this case, using a parameter helps indicate how to use this function.

If the first argument is "silence" then we don't expect another. If the first argument is "says", then we use arguments[1] to get the second argument.

JavaScript console

```
Player sits in silence
Player says: 'Stand back!'
```

#5 Handling exceptions

JavaScript is a fairly forgiving language, but now and then things go wrong—wrong enough that the browser can't continue executing your code. When that happens, your page stops working, and if you look in the console you're likely to see an error. Let's take a look at an example of some code that causes an error. Start by creating a simple HTML page with a single element in the body:

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

Now, add the following JavaScript:

```
window.onload = function() {
    var message = document.getElementById("messge");
    message.innerHTML = "Here's the message!";
};
```

We're making an error in this code. Can you see what we did wrong?

Load the page in your browser, make sure the console is open, and you'll get an error. Can you see what went wrong? We mistyped the id of the `<div>` element, so when your code tries to retrieve that `<div>` element it fails, and the variable `message` is null. And you can't access the `inner-HTML` property of null.

JAVASCRIPT CONSOLE

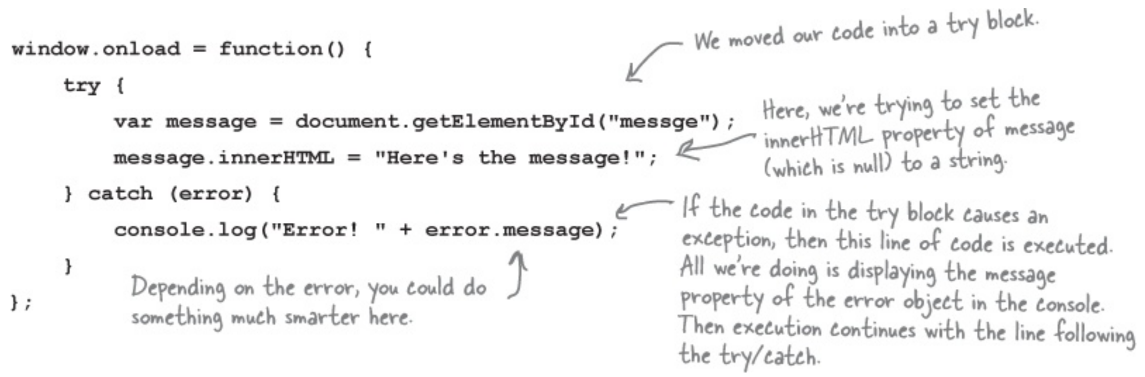
```
Uncaught TypeError: Cannot set

property 'innerHTML' of null
```

When you get an error that causes your code to stop executing like this one does, it's called an exception. JavaScript has a mechanism, called try/catch, that you can use to watch for exceptions and catch them when they happen. The idea is that if you can catch one of these exceptions, rather than your code just stopping, you can take an alternative action (try something else, offer the user a different experience, etc.).

Try/catch

The way you use try/catch is like this: you put the code you want to try in the try block, and then you write a catch block that contains code that will be executed in case anything goes wrong with the code in the try block. The catch keyword is followed by parentheses that contain a variable name (that acts a lot like a function parameter). If something goes wrong and an exception is caught, the variable will be assigned to a value related to the exception, often an Error object. Here's how you use a try/catch statement:



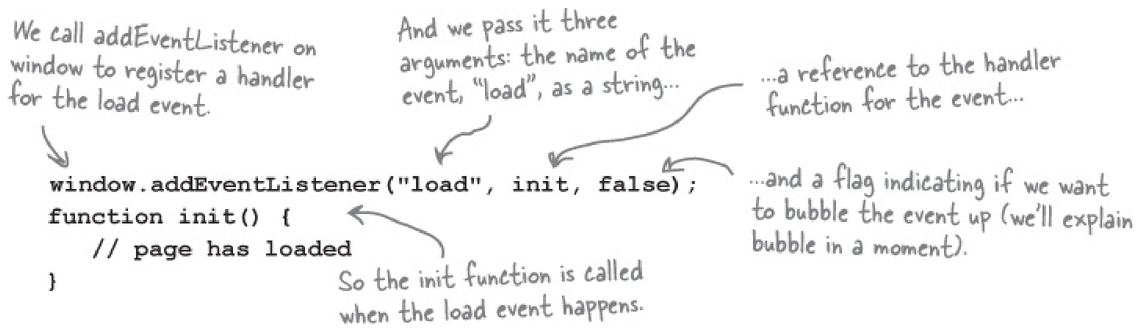
```
window.onload = function() {
    try {
        var message = document.getElementById("messge");
        message.innerHTML = "Here's the message!";
    } catch (error) {
        console.log("Error! " + error.message);
    }
};
```

We moved our code into a try block.
Here, we're trying to set the innerHTML property of message (which is null) to a string.
If the code in the try block causes an exception, then this line of code is executed. All we're doing is displaying the message property of the error object in the console. Then execution continues with the line following the try/catch.
Depending on the error, you could do something much smarter here.

#6 Adding event handlers with addEventListener

In this book, we used object properties to assign event handlers to events. For instance, when we wanted to handle the load event, we assigned an event handler to the `window.onload` property. And when we wanted to handle a button click, we assigned an event handler to that button's `onclick` property.

This is a convenient way of assigning event handlers. But sometimes, you might need a more general way of assigning event handlers. For instance, if you want to assign multiple handlers for one event type, you can't do that if you use a property like `onload`. But you can with a method named `addEventListener`:



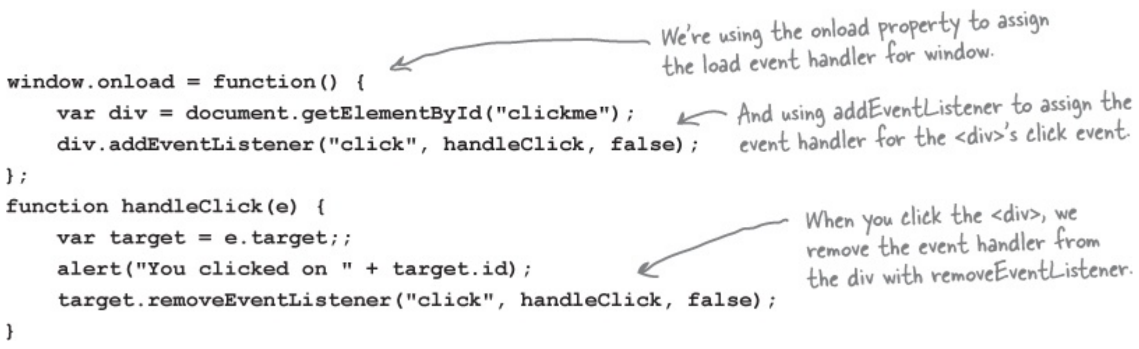
```
window.addEventListener("load", init, false);
function init() {
    // page has loaded
}
```

We call addEventListener on window to register a handler for the load event.
And we pass it three arguments: the name of the event, "load", as a string...
...a reference to the handler function for the event...
...and a flag indicating if we want to bubble the event up (we'll explain bubble in a moment).
So the init function is called when the load event happens.

You can assign a second load event handler to window simply by calling `addEventListener` again, passing a different event handler function reference as the second argument. This is handy if you want to split your initialization code into two separate functions, but remember—you won't know which handler will be called first, so keep that in mind as you're designing your code.

The third argument to `addEventListener` determines if the event is "bubbled up" to parent elements. This doesn't make a difference for the load event (because the window object is at the top level), but if you have, say, a `<span>` element nested inside a `<div>` element, and you click on the `<span>` but want the `<div>` to receive the event, then you can set bubble to true instead of false.

It's totally fine to mix and match using the event properties, like `onload`, with `addEventListener`. Also, if you add an event handler with `addEventListener`, you can remove it later with `removeEventListener`, like this:



```
window.onload = function() {
    var div = document.getElementById("clickme");
    div.addEventListener("click", handleClick, false);
};
function handleClick(e) {
    var target = e.target;;
    alert("You clicked on " + target.id);
    target.removeEventListener("click", handleClick, false);
}
```

We're using the onload property to assign the load event handler for window.
And using addEventListener to assign the event handler for the <div>'s click event.
When you click the <div>, we remove the event handler from the div with removeEventListener.

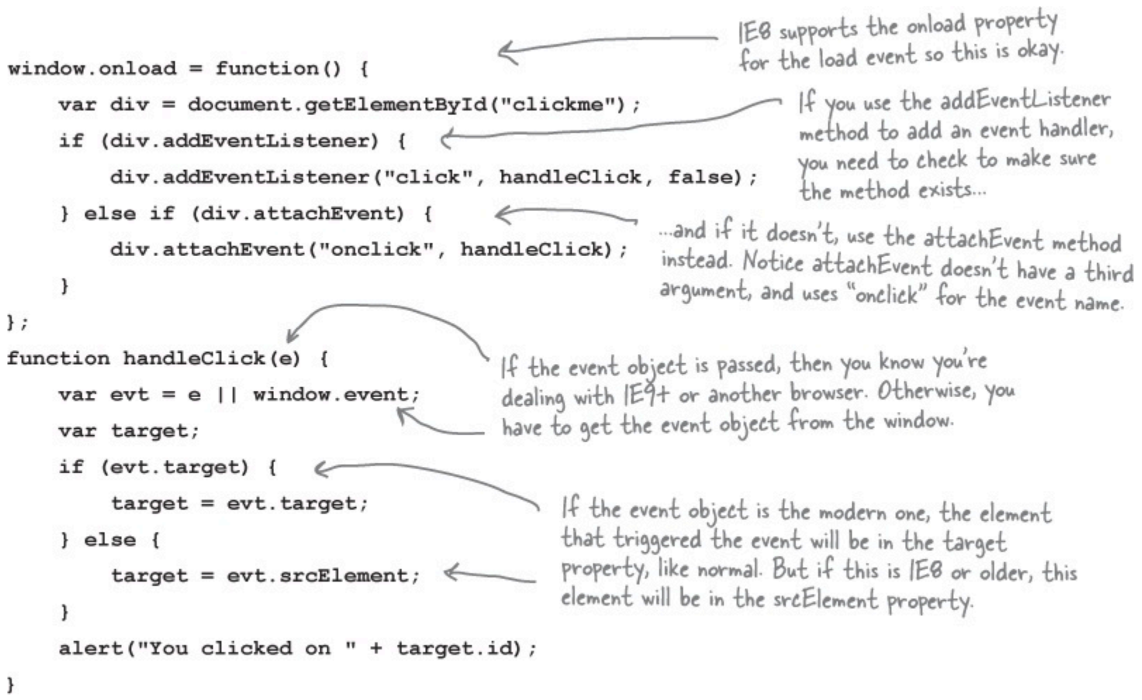
Event handling in IE8 and older

We've handled a few different kinds of events in this book—mouse clicks, load events, key presses, and more—and hopefully you've been using a modern browser and the code has worked for you. However, if you are writing a web page that handles events (and what web page doesn't?) and you're concerned that some of the people in your audience may be using versions of Internet Explorer (IE) that are version 8 or older, you need to be aware of an issue with event handling.

Unfortunately, IE handled events differently from other browsers until IE9. You could use properties like `onclick` and `onload` to set event handlers across all browsers, however, the way that older IE browsers handle the event object is different. In addition, if you happen to be using the standardized `addEventListener` method, IE didn't support this method until IE9 and later. Here are the main issues to be aware of:

- ❏ IE8 and older browsers do support most of the “on” properties you can use to assign event handlers.
- ❏ IE8 and older browsers use a method named `attachEvent` instead of the `addEventListener` method.
- ❏ When an event is triggered and your event handler is called, instead of passing an event object to the handler, IE8 and older store the event object in the window object.

So, if you want to be sure that your code works across all browsers, including IE8 and older browsers, then you can manage these differences like this:



```
window.onload = function() {
    var div = document.getElementById("clickme");
    if (div.addEventListener) {
        div.addEventListener("click", handleClick, false);
    } else if (div.attachEvent) {
        div.attachEvent("onclick", handleClick);
    }
};
function handleClick(e) {
    var evt = e || window.event;
    var target;
    if (evt.target) {
        target = evt.target;
    } else {
        target = evt.srcElement;
    }
    alert("You clicked on " + target.id);
}
```

IE8 supports the onload property for the load event so this is okay.

If you use the addEventListener method to add an event handler, you need to check to make sure the method exists...

...and if it doesn't, use the attachEvent method instead. Notice attachEvent doesn't have a third argument, and uses “onclick” for the event name.

If the event object is passed, then you know you're dealing with IE9+ or another browser. Otherwise, you have to get the event object from the window.

If the event object is the modern one, the element that triggered the event will be in the target property, like normal. But if this is IE8 or older, this element will be in the srcElement property.

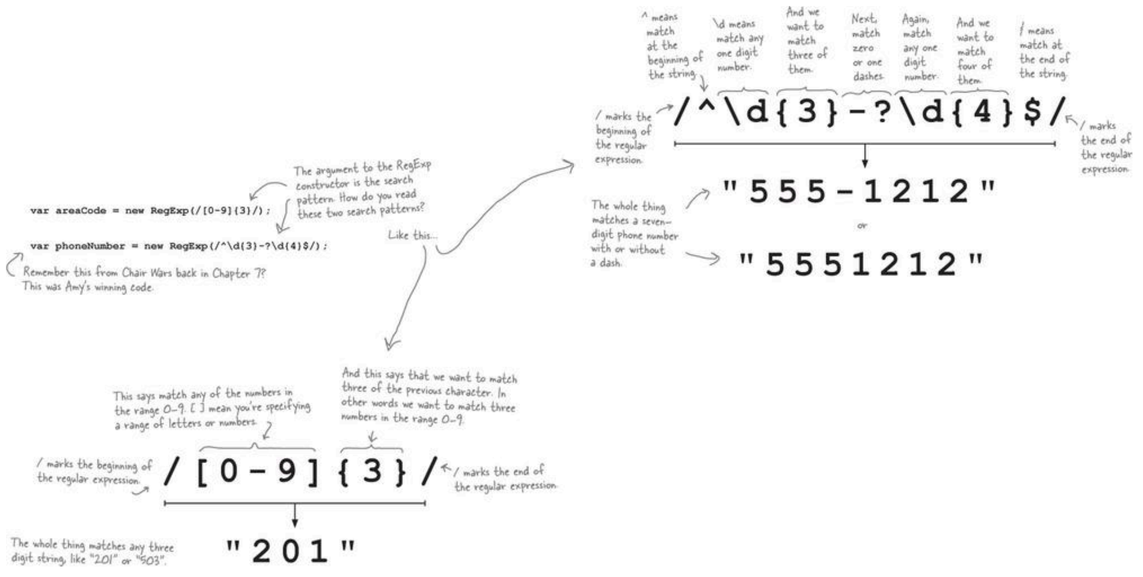
#7 Regular Expressions

You’ve seen the RegExp object in passing in this book—“RegExp” stands for *regular expression*, which is, formally, a grammar for describing patterns in text. For instance, using a regular expression, you could write an expression that matches all text that starts with “t” and ends with “e”, with at least one “a” and no more than two “u”s in between.

Regular expressions can get complex fast. In fact, regular expressions can almost seem like an alien language when you first try to read them. But you can get started with simple regular expressions fairly quickly, and if you like them, check out a good reference on the topic.

The RegExp constructor

Let’s take a look at a couple of regular expressions. To create a regular expression, pass a search pattern to the RegExp constructor, between two slashes, like this: The key to understanding regular expressions is learning how to read the search patterns. These search patterns are the most complex part of regular expressions, so we’ll work through the two examples here, and you’ll have to explore the rest on your own.



```
var areaCode = new RegExp(/[0-9]{3}/);

var phoneNumber = new RegExp(/^\d{3}-?\d{4}$/);
```

The argument to the RegExp constructor is the search pattern. How do you read these two search patterns?
Remember this from Chair Wars back in Chapter 7? This was Amy's winning code.
Like this...
^ means match at the beginning of the string.
\d means match any one digit number.
And we want to match three of them.
Next, match zero or one dashes.
Again, match any one digit number.
And we want to match four of them.
$ means match at the end of the string.
/ marks the beginning of the regular expression.
/^\d{3}-?\d{4}$/
/ marks the end of the regular expression.
The whole thing matches a seven-digit phone number with or without a dash.
"555-1212"
or
"5551212"
This says match any of the numbers in the range 0-9. [ ] mean you're specifying a range of letters or numbers.
And this says that we want to match three of the previous character. In other words we want to match three numbers in the range 0-9.
/ marks the beginning of the regular expression.
/[0-9]{3}/
/ marks the end of the regular expression.
The whole thing matches any three digit string, like "201" or "503".
"201"

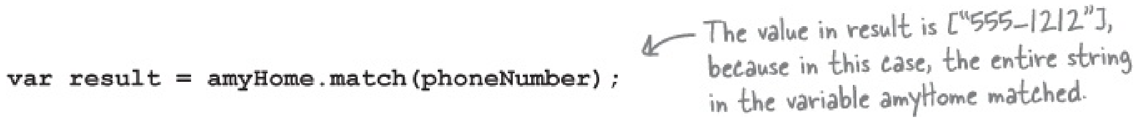
The key to understanding regular expressions is learning how to read the search patterns. These search patterns are the most complex part of regular expressions, so we’ll work through the two examples here, and you’ll have to explore the rest on your own.

Using a RegExp object

To use a regular expression, you first need a string to search:

```
var amyHome = "555-1212";
```

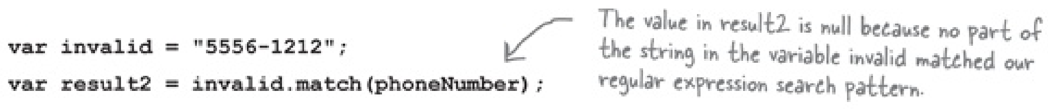
Then, you match the regular expression to the string by calling the `match` method on the string, and passing the regular expression object as an argument:



```
var result = amyHome.match(phoneNumber);
```

The value in result is ["555-1212"], because in this case, the entire string in the variable amyHome matched.

The result is an array containing any parts of the string that matched. If the result is null, then nothing in the string matched the regular expression:



```
var invalid = "5556-1212";
var result2 = invalid.match(phoneNumber);
```

The value in result2 is null because no part of the string in the variable invalid matched our regular expression search pattern.

Once you've got a regular expression, like `phoneNumber`, you can just keep using it to match as many strings as you like.

#8 Recursion

When you give a function a name it allows you to do something quite interesting: call that function from within the function. We call this *recursion*, or a recursive function call.

Now why would you need such a thing? Well, some problems are inherently recursive. Here's one from mathematics: an algorithm to compute the Fibonacci number series. The Fibonacci number series is:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144... and so on.

To compute a Fibonacci number we start by assuming:

Fibonacci of 0 is 1

Fibonacci of 1 is 1

and then to compute any other number in the series we simply add together the two previous numbers in the series. So:

Fibonacci of 2 is Fibonacci of 1 + Fibonacci of 0 = 2

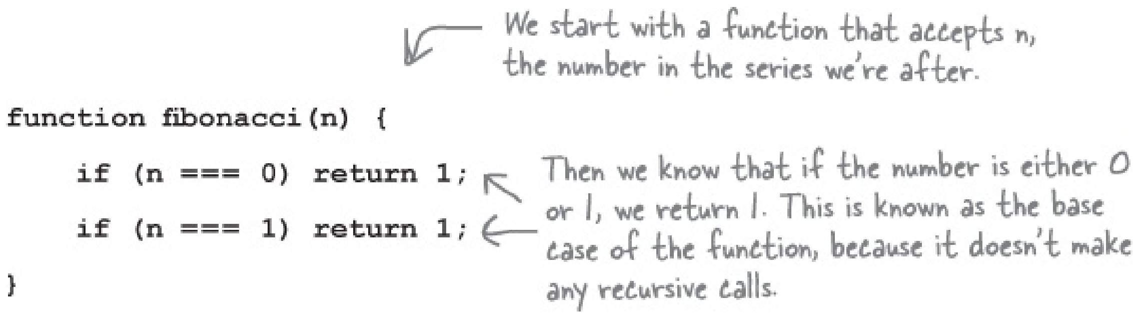
Fibonacci of 3 is Fibonacci of 2 + Fibonacci of 1 = 3

Fibonacci of 4 is Fibonacci of 3 + Fibonacci of 2 = 5

and so on... The algorithm to compute Fibonacci numbers is inherently recursive because you compute the next number using the results of the previous two Fibonacci numbers.

We can make a recursive function to compute Fibonacci numbers like this: to compute the Fibonacci of the number n, we call the Fibonacci function with the argument n-1 and call the Fibonacci function with the argument n-2, and then add the results together.

Let's do that in code. We'll start by handling the cases of 0 and 1:

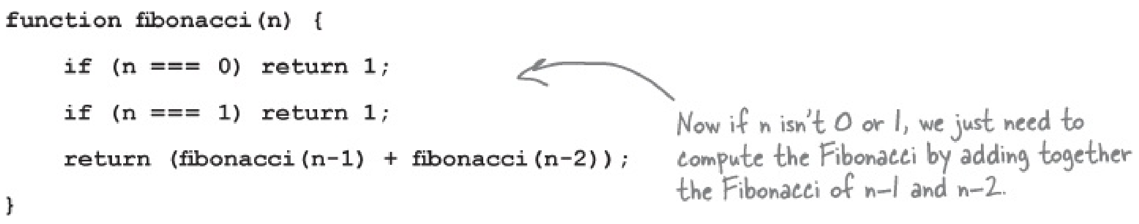


```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
}
```

We start with a function that accepts n, the number in the series we're after.
Then we know that if the number is either 0 or 1, we return 1. This is known as the base case of the function, because it doesn't make any recursive calls.

These are the *base cases*—that is, the cases that don't rely on previous Fibonacci numbers to compute—and it is usually good to write them first. From there you can think like this: "To compute a Fibonacci number n, I return the result of adding the Fibonacci of n-1 and the Fibonacci of n-2.

Let's do that...



```
function fibonacci(n) {
    if (n === 0) return 1;
    if (n === 1) return 1;
    return (fibonacci(n-1) + fibonacci(n-2));
}
```

Now if n isn't 0 or 1, we just need to compute the Fibonacci by adding together the Fibonacci of n-1 and n-2.

This looks a little like magic if you've never seen recursion before, but this does compute Fibonacci numbers. Let's clean the code up a little, and test it:

Same code, just written a little better.

```
function fibonacci(n) {
    if (n === 0 || n === 1) {
        return 1;
    } else {
        return (fibonacci(n-1) + fibonacci(n-2));
    }
}
```

And some test code.

```
for (var i = 0; i < 10; i++) {
    console.log("The fibonacci of " + i + " is " + fibonacci(i));
}
```

JAVASCRIPT CONSOLE

```
The fibonacci of 0 is 1

The fibonacci of 1 is 1

The fibonacci of 2 is 2

The fibonacci of 3 is 3

The fibonacci of 4 is 5

The fibonacci of 5 is 8

The fibonacci of 6 is 13

The fibonacci of 7 is 21

The fibonacci of 8 is 34

The fibonacci of 9 is 55
```

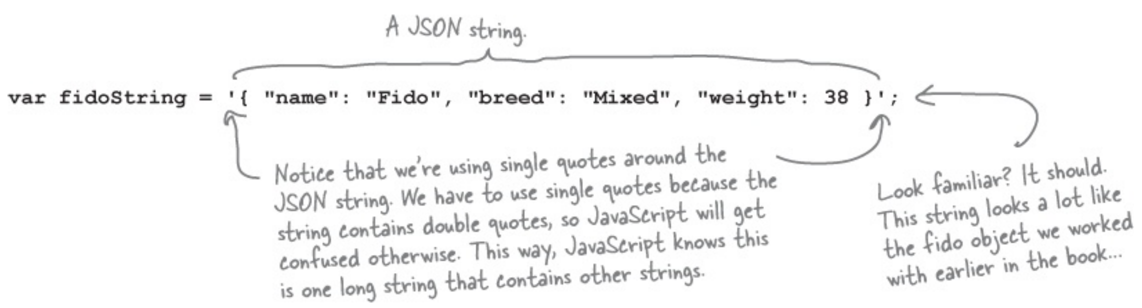
WATCH IT!

Make sure you have a base case.

If recursive code never reaches a base case where the computation ends, it will run forever, like an infinite loop. In other words, the function will continue calling itself over and over, consuming resources until your browser can't take it anymore. So if you write recursive code and your page isn't responding, figure out how to make sure you're getting to the base case.

#9 JSON

Not only is JavaScript the programming language of the Web, it's becoming a common interchange format for storing and transmitting objects. JSON is an acronym for "JavaScript Object Notation" and is a format that allows you to represent a JavaScript object as a string—a string that can be stored and transmitted:



```
var fidoString = '{ "name": "Fido", "breed": "Mixed", "weight": 38 }';
```

A JSON string.
Notice that we're using single quotes around the JSON string. We have to use single quotes because the string contains double quotes, so JavaScript will get confused otherwise. This way, JavaScript knows this is one long string that contains other strings.
Look familiar? It should. This string looks a lot like the fido object we worked with earlier in the book...

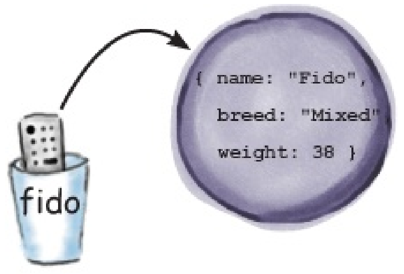
Now, the cool thing about JSON is we can take strings like this and turn them into objects. The way we do it is with a couple of methods supplied by JavaScript `JSON` object: `JSON.parse` and `JSON.stringify`. We'll use the `parse` method to parse the `fidoString` above and turn it into a real dog (well, a JavaScript object anyway):

NOTE

Notice that we're using the JSON object here. JSON is both the name of a string format and an object in JavaScript.

```
var fido = JSON.parse(fidoString);
```

We call the parse method of the JSON object, passing the string above, and we get back...



...a real JavaScript object. We store the reference to the object in the variable fido.

And you can go the other way, too. If you have an object, `fido`, and you want to turn it into a string, you just call the `JSON.stringify` method, like this:

```
var fido = {
    name: "Fido",
    breed: "Mixed",
    weight: 38
};
var fidoString = JSON.stringify(fido);
```

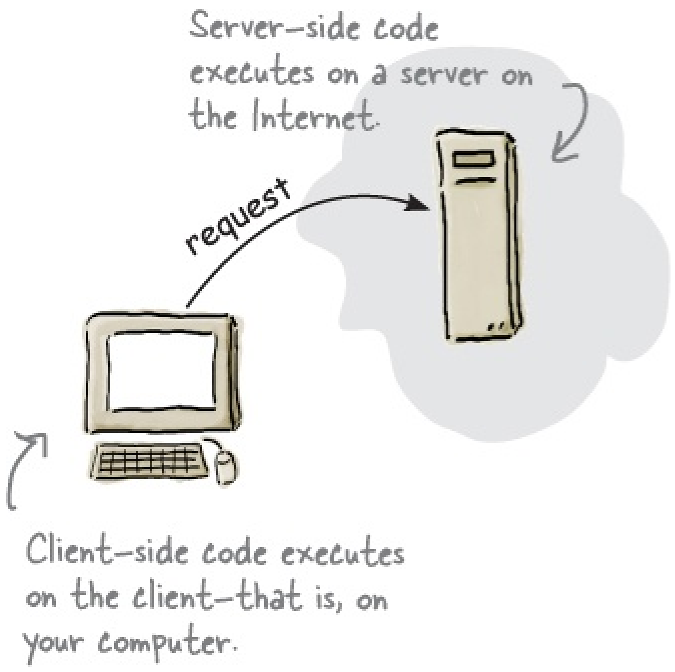
Here, we're taking a JavaScript object...

...and turning it into a string.

Note that the JSON format doesn't work with methods (so you can't include, say, a `bark` method in your JSON string), but it does work with all the primitive types, as well as objects and arrays.

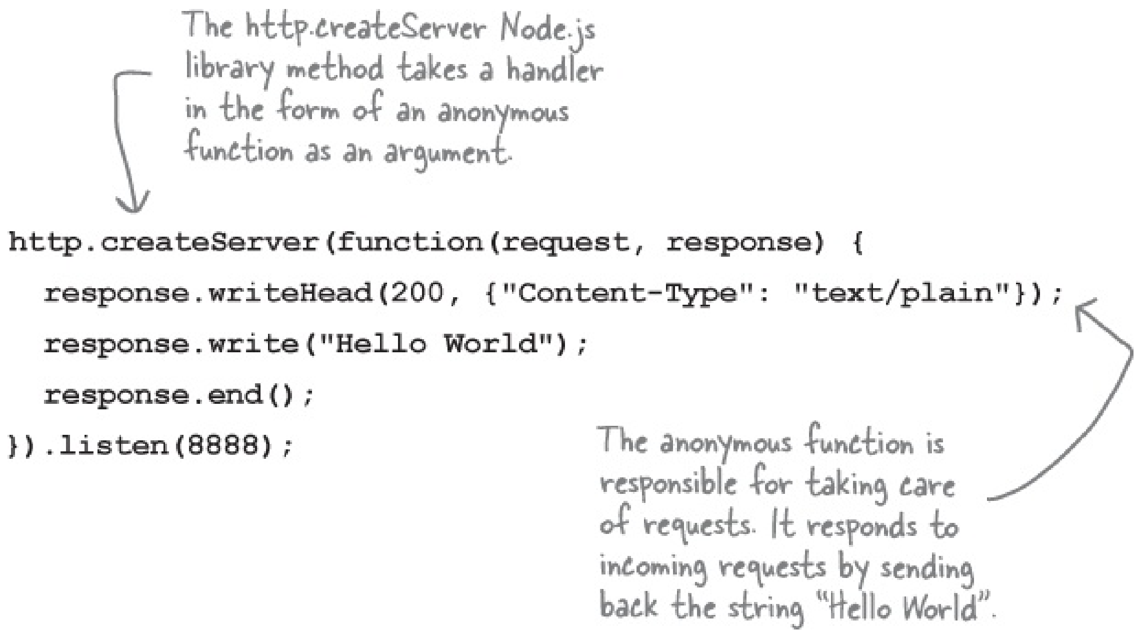
#10 Server-side JavaScript

In this book we've focused on the browser and client-side programming, but there's a whole world of server-side programming where you can now use your JavaScript skills. Server-side programming is typically required for the kinds of web and cloud services you use on the Internet. If you want to create Webville Taco's new online order system, or you think the next big idea is the anti-social network, you'll need to write code that lives and runs in the cloud (on a server on the Internet).


Server-side code executes on a server on the Internet.
request
Client-side code executes on the client—that is, on your computer.

Node.js is the JavaScript server-side technology of choice these days, and it includes its own runtime environment and set of libraries (in the same way client-side JavaScript uses the browser's libraries). And like the browser, Node.js runs JavaScript in a single-threaded model where only one thread of execution can happen at a time. This leads to a programing model similar to the browser that is based on asynchronous events and an event loop.

As an example, the method below starts up a web server listening for incoming web requests. It takes a handler that is responsible for handling those requests when they occur. Notice that the convention for setting up the event handler for incoming requests is to pass an anonymous function to the `createServer` method.



```
http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

The http.createServer Node.js library method takes a handler in the form of an anonymous function as an argument.
The anonymous function is responsible for taking care of requests. It responds to incoming requests by sending back the string "Hello World".

Of course, there is much more to explain and to work through to understand how Node.js works. But, given your knowledge of objects and functions you are well positioned to take this on. Also, explaining Node.js requires at least an entire book of its own, but you'll also find many online tutorials, articles and demonstrations at **http://nodejs.org**.

---

[2] **http://www.w3.org/html/wg/drafts/html/CR/browsers.html#the-window-object**