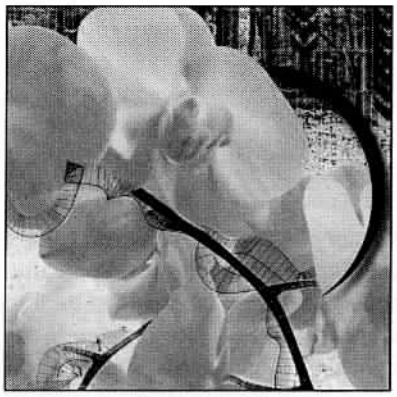

Chapter 2

Foundation JavaScript

In this chapter we delve into the realm of JavaScript. We review its history and the differences with its implementation. We also look into the jQuery code library and how it can save us a lot of time. The rest of the chapter is dedicated to learning the fundamental features of JavaScript, including everything you'll need to know to create amazing games in HTML5 canvas. I'm not going to lie, this is a big chapter; but it's all explained in as much detail as possible to help make your transition into JavaScript an easy one.

An overview of JavaScript

In Chapter 1 we stumbled across JavaScript. Now, we know it's a scripting language, but what *exactly* is it? And, what does it do? I'll answer those questions and others now. In this chapter, I put things in context and take a look at how JavaScript has evolved.

JavaScript was created way back in 1995 by Brendan Eich of Netscape, a browser company (remember it?), with cooperation from Sun Microsystems. When it started out it was actually called Mocha, eventually being renamed to LiveScript, then JavaScript in December of 1995. It was originally created to make the lives of web developers a little easier, as the only real option for adding animations or anything vaguely cool at the time was to use something like Java – a pretty complex programming language. The problem with Java is that the only way you can get it into the browser is by compiling the code (packaging it up) into something called an applet. This is great, but it means everything inside your applet is pretty much cut off from the outside world –it's in a walled garden, or however you want to put it.

JavaScript is much different in that the code is not compiled; it's embedded and interpreted by the browser. This means you don't have to construct huge, complete pieces of code to get it working. Just a few short snippets of code is all it takes, at least get you started. In short, JavaScript was created to be simpler and more forgiving than the stricter programming languages like Java. It was designed to be picked up by web developers who may not have any experience with traditional programming, and who just want to quickly add a bit of shine to their website. It's important to point out that JavaScript is in no way related to Java, apart from sharing a similar name.

> *Note: JavaScript code is interpreted by the browser, which means that each part of the code is analyzed by the browser when it's run. In comparison, compiled code is typically converted into an application that can be executed directly, without any interpretation of the code.*

No story on the Web would be complete without a legal wrangle between two tech companies. The story of JavaScript is no different. Soon after it was released by Netscape, Microsoft was eager to implement the popular scripting language in its Internet Explorer browser. Unfortunately for Microsoft, it failed to get a license from Sun Microsystems, the owners of the JavaScript trademark, so it were forced to call its implementation JScript. This has caused a bit of confusion with some developers, as at first glance it would seem the two scripting languages, JavaScript and JScript, are completely different things. In reality the two are very much the same, although each provides some functionality that the other doesn't. It should probably be mentioned that all the major browsers other than Internet Explorer (Firefox, Chrome, and so on) use JavaScript, not JScript. However, for the sake of our sanity, when I mention JavaScript in this book I'm referring to both Netscape's version and Microsoft's JScript.

> *Note: You may have heard of ECMAScript on your travels around the Web. There is a reason why it has a name so similar to JavaScript; ECMAScript is the standardized scripting language that originated from Netscape's JavaScript. The ECMA specifications are what both JavaScript and JScript aim to support.*

jQuery

If you've never heard of jQuery then you're probably a little perplexed right now. Jay what?! But don't fret, it's something that is going to make your life much easier in the long run. Let me explain.

What is jQuery?

jQuery is a JavaScript library. No, it isn't a giant building full of JavaScript books (who would even go to such a place?) It is effectively a simple wrapper around the most complicated and time-consuming tasks performed in JavaScript – things like traversing the document object model (DOM), event handling, and animation. Don't worry about what those things are just yet; I'll cover them in detail later. The official line from jQuery sums it up quite nicely: "jQuery is designed to change the way that you write JavaScript." It sounds pretty profound doesn't it? To put it another way: jQuery allows you to write less, do more. That's good enough for me!

Why are we using it?

Writing in pure JavaScript without the help from libraries like jQuery isn't quite as simple as it should be. Most of the core features are apparent in all the major browsers, which is fantastic. What isn't fantastic is that a lot of these browsers implement other features in slightly different ways. One example is detecting when a HTML document has finished loading, which is very important (we'll look into it later).

Unfortunately, there is no single way to do this across all the major browsers, which can make it a nightmare to get your JavaScript working the same for all your users. jQuery provides its own functionality for tasks like this, which works consistently across all the major browsers. For example, with one line of jQuery you can make that check to see if a HTML document has finished loading, something that would take tens of lines to cover all browsers if written in pure JavaScript. jQuery isn't doing anything magic here (it still uses the pure JavaScript behind the scenes), it just wraps everything up and does all the laborious browser checks for you. It lets you get on with what you're making, safe in the knowledge that it will work across all the major browsers.

> *Note: There is no specific reason behind choosing the jQuery library over other JavaScript libraries, like Prototype and YUI. On a personal level, I find jQuery very easy to use and, most important, easy to teach. There would be no point me teaching you how to use a JavaScript library that I'm not 100 percent comfortable with. If you already use another library then feel free to ignore the jQuery code and replace it for the equivalent code in your chosen library.*

Isn't this cheating?

Some people, particularly those who have already learned pure JavaScript, believe it's a bad idea to learn JavaScript with the help of libraries like jQuery. Their argument is that if you only learn the jQuery way then you won't have a proper understanding of how JavaScript works, like what to do if things go wrong. While I partly agree with this argument (it *is* wrong not to have an understanding of how JavaScript works), I don't see an issue with learning a library like jQuery at the same time. Let me put it this way: in reality, web developers are pushed for time and are much more interested in getting something working than worrying about if it's going to break in other browsers. Because of this, most people who work with JavaScript will use a library of some sort to make their lives that little bit easier. So why should I teach you a way of using JavaScript that deliberately complicates matters and is different to the way people work in the real world?

Does this mean I won't understand pure JavaScript?

Not at all. My aim in this book is to use jQuery only when it's appropriate. I'll be teaching you the foundations of pure JavaScript with jQuery added in to make things easier. Whenever we use a new feature of jQuery, I will explain what it does, and show you the equivalent way of doing it in pure JavaScript. I'll be doing this not only for the sake of comparison, but to help you understand how JavaScript works, and to enforce the reasons why jQuery makes things simpler. At the end of the day, jQuery is JavaScript, but instead of giving you the individual flour, eggs, butter, and sugar of JavaScript, jQuery just gives you cake. So by using jQuery you're actually using and learning parts of pure JavaScript at the same time.

How do I start using jQuery?

Getting jQuery into your project is really easy – like, one line of code easy. All you have to do is link the jQuery file into your HTML document. Something like the following would do:

```
<script type="text/javascript" src="jquery.js"></script>
```

Now, to do this you would need to go to the jQuery website, download the library, and add it into the same folder as your HTML page. Not an amazingly difficult task. Or, you could let Google do all the legwork and host the jQuery library for you. The second option sounds a little weird, but it makes sense really. If web developers use the library that Google hosts, by using the Google Libraries API, then the users of those websites will have that version of the jQuery file cached on their computer. This means that they won't have to re-download the library if they visit another site that also uses the Google-hosted jQuery. In short, it makes things quicker for the user, and that's a good thing. However, heed this warning: if you do use the Google-hosted file you will need to be connected to the Internet when testing your JavaScript. If you want a method that works offline you'll need to download jQuery and use the method described in the previous example.

To use the Google-hosted version you simply change the `src` attribute of the example above:

```
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1➥
/jquery.min.js"></script>
```

You may notice the jQuery file has `.min` at the end. This just means that the file has been minified – that is, it's been shrunk to be as small as possible. Using the minified version of jQuery is always a good idea once you've finished developing your website, as it's much quicker to download; it's just under seven times smaller than the unminified version.

> *Note: Visit the jQuery website for more information about the library. The documentation on there is particularly useful if you want to learn some of the more advanced features: `http://jquery.com`.*

Adding JavaScript to an HTML page

You should now have an idea of what jQuery is and why you're using it, so let's put everything together and construct a basic HTML page with a bit of JavaScript goodness.

```
<!DOCTYPE html>

<html>
        <head>
                <title>Adding JavaScript to a HTML page</title>
                <meta charset="utf-8">

                <!-- CSS to go here -->

                <script type="text/javascript" src="http://ajax.googleapis.com➥
/ajax/libs/jquery/1/jquery.min.js"></script>

                <script type="text/javascript">
                        $(document).ready(function() {
```

```
                alert("Hello, World!");
            });
        </script>
    </head>

    <body>

    </body>
</html>
```

The first thing you'll notice is that we're using the same HTML5 code from earlier. The only difference is that we've stripped out all the content and added a couple of `script` elements into the head element. This is where all the JavaScript happens.

A `script` element allows us to place JavaScript, and other scripts, within a HTML page. The first `script` element we use pulls in an external file from Google:

```
<script type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></script>
```

To do this we first define the kind of script we're using by setting the `type` attribute to "text/javascript." This lets the browser know that the script we're about to provide is JavaScript rather than something else. We then pull in the jQuery library by setting the `src` attribute to the Google URL. The `src` attribute tells the browser that it will need to grab the script from an external location – external being something that isn't the current HTML page file. You can also use the `src` attribute to grab scripts from the local filesystem.

The second `script` element doesn't have a `src` attribute because we're writing the JavaScript directly inside of the element. It may sound simple, but it's important to distinguish the difference. You don't use an `src` attribute if you're writing the JavaScript directly into the HTML page – sometimes referred to as internal JavaScript.

> *Note: As a rule of thumb, it's always a good idea to put your JavaScript in a separate file and place it in your HTML pages using the `src` attribute. Doing so helps keep things neat and tidy, but also means that you can use the same JavaScript code on multiple pages without having to rewrite it.*

Inside of the second `script` element we find our first piece of real JavaScript:

```
$(document).ready(function() {
    alert("Hello, World!");
});
```

Doesn't look too bad, does it? We'll skip the first and last lines, as I'm going to cover them in more detail in the next section. For now, just know that they are part of a jQuery function that makes sure our JavaScript doesn't run until the HTML document has finished loading.

The line we're interested in right now is the second line, the one with the weird `alert` thing. This piece of JavaScript is one of the simplest ways to make something interesting happen. The `alert` function (we'll

cover functions in more detail later) is used to open a dialog box within the browser that forces the user to read it before they can continue browsing. Sounds a bit annoying really, which it is, but it's a good first example. By passing the `alert` function a string (a bit of text), we're able to change the message that appears inside the dialogue box. When we run our page with "Hello, World!" as the string, we should get something like what you see in Figure 2-1:

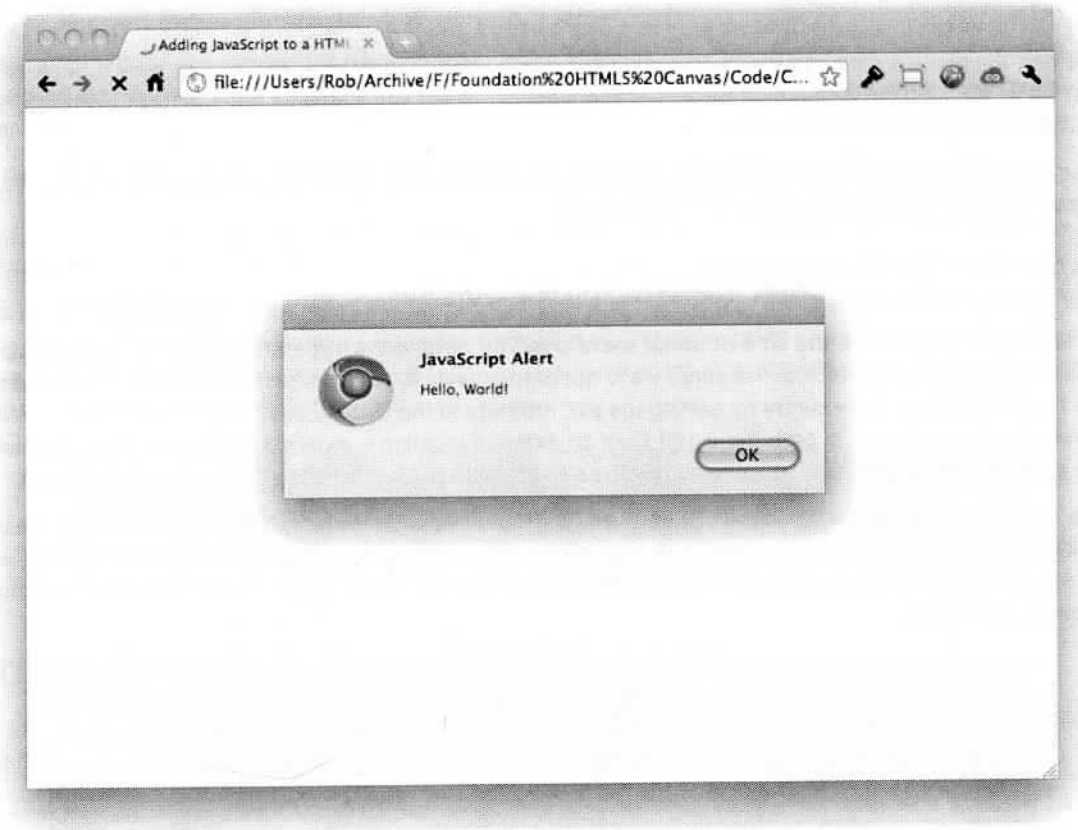

Figure 2-1. Output of the "Hello, World!" example

It works! You probably recognize the dialog box now – they used to be used for all sorts of things, like welcoming users to a website, or warning them when they submitted a form in the wrong way. Luckily for us no one really uses them any more, as they are disruptive and generally annoying. For our purposes the `alert` function will do as a quick and easy way to output a message in JavaScript, for testing purposes of course.

So there you have it, a basic HTML page with a bit of JavaScript thrown in for good measure. That wasn't so hard, was it?

Note: If you ran the example and nothing happened, make sure you are online, as we're using the external Google version of jQuery. If you want to develop offline then download jQuery from the official website and use that file instead, as described in the jQuery section previous to this. If you're still experiencing issues, it's worth checking to see whether JavaScript is enabled in your browser.

Running JavaScript after the page has loaded

I've already mentioned how useful it is to run JavaScript *after* the HTML document has finished loading. But why is it so important? Why can't you just run JavaScript at any time? The answer is twofold. First, if JavaScript ran at the point you placed it in your HTML page, it would start loading and halt the rest of the HTML page from being displayed until it finished. It's generally a good idea to have the entire content load first, then let the JavaScript do its thing afterwards. That way, the user of your website can start looking at the content while the JavaScript loads in the background. Second, if JavaScript ran before the content had finished loading, it wouldn't be able to access any of the content that hadn't loaded yet. This is a massive problem if you're trying to manipulate HTML elements using JavaScript, like we will be later in the book. So we need a method of stalling the JavaScript from running until the HTML document (our page) has finished loading. We have three options available to us: the wrong way, the long way, and the easy way. Let's take a look at all three, as it's important to understand the difference.

The wrong way (the window.onload event)

Originally, developers used something called the `window.onload` event. The way this works is that after a web page has finished loading absolutely everything, it will run `window.onload`. If we converted our alert box example, it would look like this:

```
window.onload = function() {
        alert("Hello, World!");
};
```

It seems pretty clean and simple. So what makes it the wrong way? Well, the problem with `window.onload` is that it's *too* patient. It doesn't run until absolutely every piece of content has finished loading. But hang on a minute, don't we want to wait until all the content is loaded? Yes and no. We want to wait for the content to be accessible, but we don't want to wait for every single piece of content to load and become visible. For example, imagine you have a very large image in your content, one that is many megabytes in size (for your users' sakes, I hope you don't!) When you load your web page that image will take a fairly long time to load, possibly taking so long that you can sit and watch it build up line by line. Now if you were using `window.onload` on this web page, it wouldn't run until the entire image had finished downloading, which could take minutes!

We need a better option, one that runs after the browser is aware of the content, but before the content is actually loaded onto our screen. Fortunately for us there is something that allows us to do just that, the document object model (DOM).

The long way (the DOM)

The DOM is a method of representing and accessing elements of a document like, in the case of a web page, the HTML elements. This allows us to get information about and manipulate all the elements and attributes on a HTML page, directly from JavaScript. Don't worry if the DOM sounds a bit difficult; we'll be talking about it in more detail later in this chapter. The important thing to know right now is that the DOM represents the raw structure of our content, which means it has to be created before the content is displayed on the screen. If we can find out when the DOM has finished loading, we'll know when the content is accessible, regardless of whether it's visible on the screen.

Unfortunately, detecting when the DOM has loaded is a troublesome task. There's just no consistent method across all the major browsers (surprise, surprise). Even if there was, it's not exactly easy to do. The good thing for us is that a few kind souls, like freelance programmer Dean Edwards, have done the hard work for us and uncovered a method that works in each major browser. This is the result of their hard work; you'll soon see why I call this "the long way":

```
// Dean Edwards/Matthias Miller/John Resig
function init() {
        // quit if this function has already been called
        if (arguments.callee.done) return;

        // flag this function so we don't do the same thing twice
        arguments.callee.done = true;

        // kill the timer
        if (_timer) clearInterval(_timer);

        // do stuff
};

/* for Mozilla/Opera9 */
if (document.addEventListener) {
        document.addEventListener("DOMContentLoaded", init, false);
}

/* for Internet Explorer */
/*@cc_on @*/
/*@if (@_win32)
        document.write("<script id=__ie_onload defer
src=javascript:void(0)><\/script>");
        var script = document.getElementById("__ie_onload");
        script.onreadystatechange = function() {
                if (this.readyState == "complete") {
                        init(); // call the onload handler
                }
        };
/*@end @*/

/* for Safari */
```

```
if (/WebKit/i.test(navigator.userAgent)) { // sniff
        var _timer = setInterval(function() {
                if (/loaded|complete/.test(document.readyState)) {
                        init(); // call the onload handler
                }
        }, 10);
}

/* for other browsers */
window.onload = init;
```

Even for a JavaScript veteran, it's not exactly something you want to get your hands dirty with on each and every project. Surely there's an easier way to detect when the DOM has loaded? As luck would have it there is: it's called jQuery.

> *Note: Going through the Dean Edwards script line-by-line is beyond the scope of this book. Visit his website for more information about it and the problems he encountered during its creation: `http://dean.edwards.name/weblog/2006/06/again/`*

The easy way (the jQuery way)

Remember those two lines I told you to forget about in the last section? Well it's time to claw them back from the depths of your memory. Or, I could just show them to you again and save us both a lot of time:

```
$(document).ready(function() {
        // Put the JavaScript you want to run after the page loads in here
});
```

This is jQuery doing what it does best, letting us do something complex in an elegant and easy way. When broken down into its core components we can see how powerful jQuery really is.

The first part, `$(document)`, is a jQuery selector. It allows you to select an element from the DOM to be manipulated, in our case the document object – the root of a web page that contains all of the HTML elements. We'll talk more about jQuery selectors in the DOM section of this chapter.

The second part, `.ready()`, is the juicy bit that does the cool stuff for you. Its sole purpose is to let you know when the DOM has finished loading. You'll notice that this is not only one line, but also really only one word, and it does the same stuff that tens of lines did the long way. However, I should make it clear that the jQuery `.ready()` method is not, as it would seem, magic. It's actually based on the Dean Edwards method, and has just been wrapped up in a ridiculously easy-to-use package. But behind the scenes it's practically exactly the same as what we looked at in the code for the long way.

To finish it all off we need somewhere to put the code we want to run once the DOM is loaded. The `.ready()` method allows us to do just that by using a callback function within the parentheses (rounded brackets) – a piece of JavaScript that is called once some particular event has happened. In our case we

place an empty function in there, within which we can then add the code we want to run once the DOM has loaded. That's really as hard as it gets!

For now, I hope you can start to see the benefits that jQuery has over using pure, raw JavaScript. We aren't cheating; we're simply making things easier for ourselves by letting jQuery handle the repetitive and complex stuff that just gets in the way.

You should now have a good idea of how the jQuery way works. Let's step things up a notch and tackle the fundamental theories behind programming in JavaScript.

Variables and data types

At the most fundamental level we have variables and data types. They are the building blocks of JavaScript that allow us to do some pretty cool stuff.

Variables

If you ever want to remember something when programming, you'll probably use a variable. The sole purpose of a variable is to hold a value (a piece of data) for retrieval at a later date. In this way they are very much like what you may have encountered while learning algebra at school. Remember those crazy formulas you had to work out that had numbers assigned to letters (e.g., x=3)? Those letters are variables, they store a value.

Variables in JavaScript are no different; you create and name a variable (e.g., *x*) then assign a value to it, like a number or some text (e.g., *x=3*). To get the value back again you just refer to the variable you assigned it to; in our case, referring to *x* would return 3. This is particularly important in programming because it allows you to assign a value once, then refer to it again and again in the future by using the variable instead of typing out the value multiple times. You can even change the value of a variable once it has been assigned; hence, it's called a *variable*. This will make more sense soon, so don't worry if I've confused you.

Naming variables

To use a variable you first have to give it a name, something that explains its purpose in as few words as possible. For example, it would be a mistake to name a variable that holds a user's name *myVariable*, or *x*. How would you ever remember its purpose in the future? It's much better to give variables a meaningful name that explains exactly what it's for. For our example, a much better name would be something like *userName*, or even just *name*. With a name like that you'll never forget what value it holds.

> *Note: There are many ways to format the names of variables, but in our case we're going to be using something called camel case. This is when all the words of a variable are joined together, with the first character of each word being a capital letter, apart from the first letter of the entire variable. It's a fairly common method of naming variables, but there is nothing wrong with you using underscores to separate words, or anything else that you're used to.*

When naming variables, you should be aware that you can't use any word you want. The most important thing to remember is that a variable name can only start with a letter (a to z) or an underscore. Numbers can be used, but only after the first character of the name. Also, some words are reserved, which means they can't be used for variables, functions, or objects (more on these later). The Mozilla Developer Center has a great list of all the words you can't use: https://developer.mozilla.org/en/JavaScript/Reference/Reserved_Words

Declaring variables and assigning values

So we now understand what variables are and why we need to use them. We also know how to name them properly so they make sense. The next step is learning how to actually use them in JavaScript or, in developer speak, how to declare them and assign values to them.

Declaring a variable is just another way of saying *creating a variable*. It is a good idea to declare a variable before we use it so we have full control over it, and also so we know that it exists. This is how you would declare a single variable:

```
var userName;
```

There are two things to notice here: the first being that a variable is declared to JavaScript by using the `var` keyword. This basically says that the next word after `var` is the variable name, and that JavaScript should declare a variable with that name. The second thing to notice is that a semi-colon is placed at the end of the line. In some programming languages it is extremely important to end each statement (a block of code) with a semi-colon. JavaScript is different in that it won't break without a semi-colon at the end of a statement, but it's good practice to do so, it looks a lot neater, and it can save you a whole world of pain when debugging code in the future.

After a variable is declared, it's usually a good idea to assign a value to it, or it will be automatically assigned a value of "undefined," which I'll cover later. Assigning a value to a variable is really simple; you just use an equals sign, otherwise known as an assignment operator, like so:

```
userName = "Rob Hawkes";
```

Notice we don't need to include the `var` keyword because the variable has already been declared. If we hadn't already declared the *userName* variable, everything would still work as JavaScript is pretty relaxed. But, although it is possible to declare a variable without a `var` keyword (by assigning a value to it), it's not very good practice and it can lead to all sorts of issues with scoping in JavaScript. In short, you should always properly declare a variable with the `var` keyword the first time you use it. Also notice that the name *Rob Hawkes* is enclosed within quotation marks, this is because text values (strings) require special treatment. We'll cover strings and other kinds of values in the data types section shortly.

> *Note: Variables declared outside of all functions are in the global scope, and can be accessed by all further code. Variables declared inside of functions are in the local scope, and can't be accessed outside of the function in which they were created. Issues with scoping are common, so it's worth checking where your variables are being declared if you're having trouble getting them to work.*

You can actually declare and assign variables all in one go, like this:

```
var userName = "Rob Hawkes";
var age = 34;
```

This is a much quicker way of doing things, and generally looks a bit neater.

You can also reassign a value to a variable by dropping the `var` keyword. In the previous example, my age is wrong, so to change it you would do this:

```
age = 24;
```

That's much better.

Accessing variables

It would be a bit pointless if you could assign a value to a variable and not be able to get that value back later on. The good thing is that accessing the value of a variable is ridiculously easy – you just use the name you gave it! For example, if you wanted to display the *userName* variable in an alert box, you'd do it like this:

```
var userName = "Rob Hawkes";
alert(userName);
```

If all went well you'd get an alert with the user's name in, as shown in Figure 2-2.

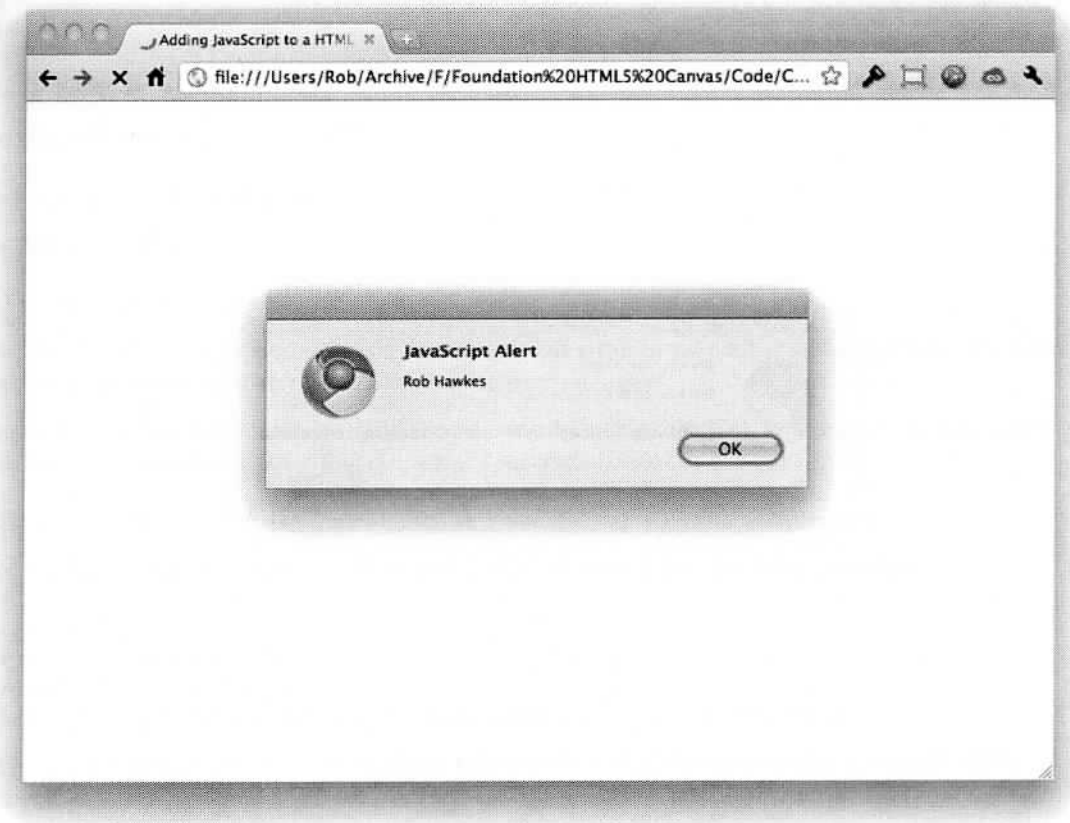

Figure 2-2. Outputting the user's name using variables

You could even display the user's age at the same time if you wanted:

```
var userName = "Rob Hawkes";
var age = 24;
alert(userName+" is "+age+" years old");
```

Which, by using something called string concatenation, will output the message shown in Figure 2-3.

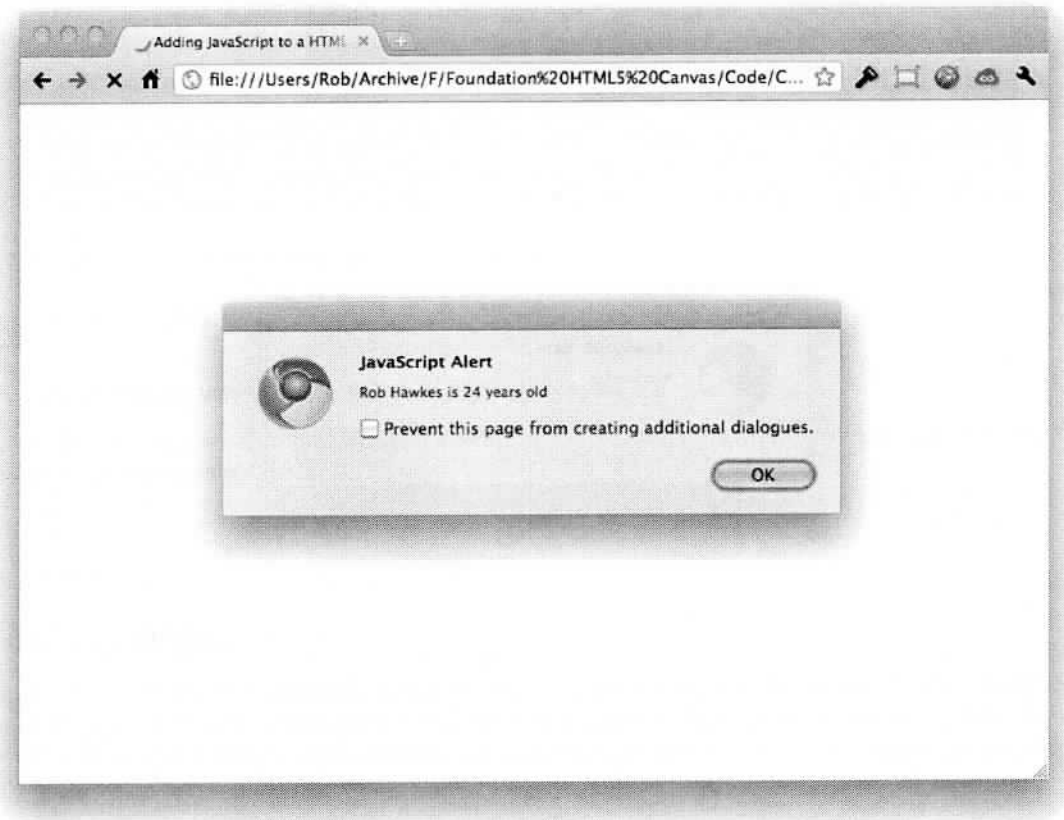

Figure 2-3. Outputting the user's name and age using variables

This is the beauty of variables; they let you access values at any point in your code, and let you combine them so you can display those values in practically any way you want.

Arithmetic

Another way to use variables is with arithmetic. The great thing about JavaScript is that it includes pretty much every kind of arithmetic you'll need. For example, to add two numbers together you'd do this:

```
var myNum = 24+6;
alert(myNum); // Outputs 30
```

Wait a minute, what's that double slash thing doing there? You've just experienced a JavaScript comment, a method of adding little notes within your code that don't run or get displayed. I'll be using them from now on to give the expected output of our examples. It's important to note that everything after and on the same line as a comment will be ignored by JavaScript and treated as a comment. This can sometimes be the cause of errors, so it's something to look out for.

To subtract two numbers, you'd do this (note that you can use `alert` on its own without the need for variables):

```
alert(24-4); // Outputs 20
```

Simple stuff. To multiply, you'd do this:

```
alert(2*5); // Outputs 10
```

And to divide, this:

```
alert(100/2); // Outputs 50
```

And so on. It's all very much common sense, and uses the same mathematical symbols that would be used on a computer calculator or another programming language.

To make arithmetic more readable, you should consider using variables to store the number values. For example, consider this equation that outputs the number of seconds in a day:

```
alert(24*60*60); // Outputs 86400
```

It works, but it's not very readable. Now consider the same equation, but using variables:

```
var hoursInDay = 24;
var minutesInHour = 60;
var secondsInMinute = 60;
alert (hoursInDay*minutesInHour*secondsInMinute); // Outputs 86400
```

It's actually readable. Anyone could walk up to that code and understand what is being calculated. This is why variables are so important, and is also why you should name them properly.

Data types

It's impossible to avoid data types; you'll use them in pretty much every piece of JavaScript you write. Data types define the various kinds of data you use, placing restrictions on what can be done with them. Having knowledge of the various data types will help you avoid unnecessary errors and problems later on. One quick example is the different way the String and Number data types are dealt with when adding things together:

```
var myString = "24"+"6";
var myNum = 24+6;
alert(myString); // Outputs "246"
alert(myNum); // Outputs 30
```

When strings are added together they are joined together, using a process known as concatenation. This is the same even when the strings are numbers, because JavaScript sees anything within quotation marks as a string, not a number.

Primitive data types

These are the building blocks of data, they are primitive because they cannot be broken down into other types of data. They include the following:

- **Numbers**: Integers (whole numbers), and floating point numbers (e.g., 2.04).
- **Booleans**: Truth values, can be either *true* or *false*.
- **Strings**: A number of characters enclosed within quotations marks (e.g., "Rob Hawkes").

Composite data types

These data types are slightly more advanced and contain values that are primitive, or composite data types. An example of a recursive composite data type is an array that contains more arrays, or an array that contains objects.

- **Arrays**: Commonly used for storing a list of related values.
- **Objects**: The base for all objects in JavaScript.

These data types are a little more confusing. I'll be covering both arrays and objects in much more detail later on in this chapter.

Other data types

As well as your standard data types, there are a couple of others that refer to non-existent and empty values.

- **Null**: Means nothing, or no value.
- **Undefined**: Given to a variable after it has been declared, but before a value has been assigned to it.

You'll often see these data types when something has gone wrong with a variable declaration, or a value has not been properly assigned to a variable, but they do have legitimate uses as well. One use would be to assign the value of `null` to a variable when you want to clear its original value, or you can use them in conditional statements to check whether a variable has been declared or has had a value assigned to it.

Conditional statements

Would you like a cup of tea: yes, or no? What I've done there is given you a choice. I've asked you a question and you have the option of saying, "Yes, a tea would be lovely Rob," or, "No, what are you on, you fool? I hate tea!" If *yes*, you get a tea; if *no*, you don't. It's as simple as that. Conditional statements in JavaScript are very much the same, as they deal with choices – they are basically just questions written in code.

Conditional statements are an incredibly useful part of your JavaScript toolkit because they allow you to deal with situations that aren't always going to have the same outcome. They also allow you to make decisions, like working out what a variable is, or if something is right (*true*) or wrong (*false*).

If statements

The most basic conditional statement is the `if` statement. It allows you to make a true or false decision about something; if *true*, then the code within the `if` statement runs, if *false*, then nothing happens.

Here is a really basic example:

```
var age = 24;
if (age == 24) {
        alert("You're 24 years old"); // Will output only if age equals 24
};
```

We set a variable called *age* and then create an `if` statement to perform a true or false decision on that variable. In our case, we're checking to see if *age* is equal to 24, which it is, so the alert will be displayed. If we changed the *age* variable to something that isn't 24, like 30, the alert won't be displayed because *age* (now 30) does not equal 24. Our code has become intelligent, sort of.

It's important to note that we're using a double equals symbol, which is referred to as a comparison operator. This double equals symbol is nothing like the single equals (the assignment operator) that you use to assign values to a variable. Getting the two confused can lead to all sorts of problems. For example, using a single equals symbol in the `if` statement in the previous example would lead to the statement always running:

```
var age = 24;
if (age = 24) {
        alert("You're 24 years old"); // Will output all the time
};
```

You could even change the age to something that you know is wrong, but the statement will still run because the condition is always going to be true because the age variable is being set inside the `if` statement. We'll cover this in more detail shortly.

Another common use of an `if` statement is to explicitly check a variable for truth by using the boolean data type, which can only ever contain the values *true* or *false*. Here is an example using boolean values:

```
var wouldLikeATea = true;
if (wouldLikeATea == true) {
        alert("Milk, 1 sugar. Thank you!"); // Will output if the user wants tea
};
```

In this example we have a variable that tells us if the user wants a cup of tea or not; *true* for yes, *false* for no – boolean values. We then construct an `if` statement that uses boolean values to see if the user would like a cup of tea and, if so, output an alert. We could actually take a shortcut and rewrite the example like this:

```
var wouldLikeATea = true;
```

```
if (wouldLikeATea) {
        alert("Milk, 1 sugar. Thank you!"); // Will output if the user wants tea
};
```

Notice we've dropped the double equals symbol and the word *true* in the `if` statement. We can do this because `if` statements check for truth by default. If the variable being checked is *true* then the code inside the statement will run. It's important to note that if you used this method to check a variable that isn't a boolean value, like a string or number, it would always return true, unless it's the number zero. This is because values that aren't boolean will return true if they contain any data at all. For example, the following code shows how a number can return true if it's non-zero, and false if it is zero:

```
var myInteger = 7;
if (myInteger) {
        alert("This code executes.");
};

myInteger = 0;
if (myInteger) {
        alert("This code does not execute.")
};
```

Comparison operators

You aren't just limited to checking for one thing to be equal to another thing, there are a whole range of other checks that can be performed. In JavaScript, you change the way a conditional statement checks something by changing the comparison operator that it uses. Table 2-1 shows a list of the most common operators available:

Table 2-1. List of comparison operators in JavaScript

Operator	Name	Notes
a == b	Equal	True if a is equal to b.
a === b	Identical	True if a is equal to b, and are of the same data type.
a != b	Not equal	True if a is not equal to b
a !== b	Not identical	True if a is not equal to b, or they are not of the same data type.
a < b	Less than	True if a is less than b.
a > b	Greater than	True if a is greater than b.
a <= b	Less than or equal to	True if a is less than or equal to b.
a >= b	Greater than or equal to	True if a is greater than or equal to b.

Note: Notice that in conditional statements two equals symbols (==), known as a comparison operator, are used to check if something is equal to another. An assignment operator (one equals symbol, =) is only ever used to assign a value to a variable. Getting the two mixed up is a really common error in code, so keep an eye out for it if things don't seem to be working the way they should.

Multiple truth checks within an if statement

As well as comparison operators, you also have logic operators. These operators allow you to check more than one thing in a single conditional statement. If we wanted to see if two checks were true, we'd use the *and* (&&) operator:

```
var age = "24";
var userName = "Rob Hawkes";
if (age == 24 && userName == "Rob Hawkes") {
        alert("You're definitely Rob Hawkes"); // Will output if both checks are
true
};
```

The `if` statement will now only work if both checks return true, that the *age* variable is 24, and that the *userName* variable is "Rob Hawkes." If either of those checks return false then the `if` statement won't run. To run the `if` statement if only one of the checks is true, you would use the *or* operator (||):

```
var age = "30";
var userName = "Rob Hawkes";
if (age == 24 || userName == "Rob Hawkes") {
        alert("You're either 24 or Rob Hawkes"); // Will output if either check is
true
};
```

Using these logic operators allows you to combine checks and build some pretty powerful conditional statements.

Else and else if statements

Now, `if` statements are great if you only care about the true outcome, but what if you want to do something else when the outcome is false? This is where the `else` and `else if` statements come in.

If you just want to do something else when the `if` statement is false, then you'd use the `else` statement:

```
var age = 30;
if (age == 24) {
        alert("You're 24 years old"); // Will output only if age equals 24
} else {
        alert("You're not 24 years old"); // Will output if age doesn't equal 24
};
```

You can take things further by performing a further check if the original `if` statement is false. For this, you'd use the `else if` statement:

```
var age = 30;
if (age == 24) {
        alert("You're 24 years old"); // Will output only if age equals 24
} else if (age == 30) {
        alert("You're 30 years old"); // Will output only if age equals 30
} else {
        alert("You're not 24 or 30 years old"); // Will output if age doesn't equal
24 or 30
};
```

The `else if` statement is another way of asking question after question: *Is it this?* No. *Well what about this?* No. *Well what about this?* And so on.

Functions

One problem you'll inevitably encounter when programming is code duplication. Sometimes it's because of the sheer quantity of code; other times it's the result of a lazy programmer. Either way, code duplication can cause a whole world of pain if not dealt with; from simply wasting your time creating it all, to completely ruining your day because you've got to edit 50 different lines of the same code.

Now, anyone with their head screwed on is probably wondering why you would ever duplicate code. What's the point? The simple answer is that some pieces of code will be used over and over again because you want to perform a similar action multiple times. For example, a piece of code that lets you output a full name by combining a first name and last name:

```
var firstName = "Rob";
var lastName = "Hawkes";
var fullName = firstName+" "+lastName;
alert(fullName);

var anotherFirstName = "John";
var anotherLastName = "Smith";
var anotherFullName = anotherFirstName+" "+anotherLastName;
alert(anotherFullName);
```

Notice how the code to combine the first name and last name is practically identical in both cases. The only difference is that they use different variables. It doesn't look like much in just two different places, but imagine if you needed to combine people's names in tens of places in your code. Now imagine if you needed to change the way those names were formatted so the last name came first, it would be mayhem. What if you missed one out? It's not worth thinking about! Wouldn't it be awesome if we had some special code that meant we only had to change the name formatting once and have it automatically updated in every place that it's used? This is what a function does.

Creating functions

Using functions is quite straightforward. You just need to understand how they work. The basic premise of functions is that they're like a little machines that perform the same actions over and over again, exactly the same way each time. You can also feed information into them, and get information back out of them. The easiest way to explain this is by showing you how a function works.

If you rewrote the name formatter we used earlier as a function, it would look like this:

```
function formatName(firstName, lastName) {
        return firstName+" "+lastName;
};
```

The `function` keyword declares our function and lets JavaScript know that the following word is the name of our function, in this case it's *formatName*. The words inside the parentheses are called arguments (or parameters) – they are variables that declare values that are being inputted into the function. If your function has no arguments, you still need to include the parentheses; but in such a case you just leave them empty. All the code that you want to run in the function is placed within the curly brackets.

It's quite common to require a function to output, or return some kind of value. To do this you use the `return` keyword, which returns whatever value comes after it. The `return` keyword also quits the function, so any code after the `return` will not run.

Calling functions

A function is pointless unless you can actually use it, or call it, in developer-speak. Calling a function allows you to run the code that has been packaged up inside of it, and allows you to access any values that are returned from the function. When calling a function you can also include values (arguments) that will be used as input. By putting this all together we can call our new *formatName* function like this:

```
formatName(firstName, lastName);
```

Or, together with the rest of the code:

```
var fullName = formatName("Rob", "Hawkes");
alert(fullName);

var anotherFullName = formatName("John", "Smith");
alert(anotherFullName);
```

Calling a function is a little like accessing a variable, you just use the name that you used when creating it in the first place. The difference is that when calling a function you need to include a pair of parentheses after the function name. Inside of the parentheses you place the input values for the function, in our case we use the first and last name variables. If we take a look at the function again we can see what's going on:

```
function formatName(firstName, lastName) {
        return firstName+" "+lastName;
};
```

In the first example we call the *formatName* function and send it the *first name* (Rob) and *last name* (Hawkes). The function takes these strings as input arguments, called *firstName* and *lastName*, which we can refer to as normal variables. Our function is really simple and returns the formatted name right back, no messing around. The returned value is then stored in the *fullName* variable, just like it was when we weren't using functions. To do the same for the second name we call the function in the same way, just replacing the input variables with a different *first name* and a different *last name*.

The benefits to using this method are huge. Firstly, we saving time by not rewriting the same code over and over again, which I'm sure you'll appreciate. Secondly, if we want to edit the way a name is formatted we now only have to change the code in one place, rather than multiple places. This is a really, *really* good thing. For example, you could easily reverse the function to output the last name, followed by the first name:

```
function formatName(firstName, lastName) {
        return lastName+" "+firstName;
};
```

Objects

Some people would argue that objects are a fairly advanced topic for a foundation book, but I tend to disagree. Objects will make things easier for us in the long run, so why should I teach you a bad habit when you can learn the right way from the start? We're going to use them extensively in the games and, in all honesty, they really aren't as complicated as people make out. You'll understand how to use them in no time.

What are objects?

Before we dive into how to create and use objects, it would be a good idea to make sure we know what they are and why they're awesome. A technical reference would describe objects as a collection of properties (variables) and methods (functions) that deal with related values and tasks. They're also a data type, which means we can assign them to variables, but we'll get on to that later.

The best way to think of objects in JavaScript is to imagine them like real-world objects, like a car, or a rocket – I like rockets. An object in JavaScript has properties and methods, as does our real-world rocket object. If we take this example further, the properties of a rocket would be things like the number of engines it has, the amount of thrust it has, the number of astronauts on board, or even something as simple as its color. A property is essentially a value that describes something about the object. On the other hand, the methods of a rocket would be things like turning the engines on and off, opening the door for the astronauts, or sending a message to ground control. A method is an action of some sort that is performed either on or by the object at hand.

So what is an object in a nutshell? And why are nutshells so great for explaining things? Well, I can't help with the nutshells, but I can certainly try and describe an object in one line. JavaScript objects are templates, they describe the features of something, and they define the actions that it can perform.

From personal experience, I've found that the best way to really understand objects is to start playing with them, so let's do just that.

Creating and using objects

There are a few ways to create an object in JavaScript, the simplest being:

```
var rocket = new Object();
```

What you've done here is create a new instance (version) of an `Object`, a blank object template, and assigned it to a variable. Technically we're using the `Object` object here, but I'm going to refer to it as plain old object so we don't get confused. Unless you're already confused? I do hope not.

Right now there's nothing in our rocket object apart from some built-in JavaScript methods. This isn't good enough, we want engines, thrust, and astronauts! Declaring these descriptive elements of an object (its properties) is no more complicated that creating variables:

```
var rocket = new Object();
rocket.engineCount = 2;
rocket.thrust = 5000;
rocket.astronautCount = 4;
rocket.colour = "red";
```

The pattern here is quite straightforward, you refer to the object variable, then use a dot, then type in the property name we'd like (just like a variable), then assign a value. Objects aren't scary, they just look a bit odd.

Unfortunately, there is a big problem with this method, being that our object is unique. We've created it from a blank object template and had to declare all the properties manually, it's a one off. If we wanted more than one rocket we'd have to duplicate this code over and over again, and code duplication is bad! It also means we can't trust each rocket object to have the same properties and methods, like you can see here:

```
var rocket = new Object();
rocket.engineCount = 2;

var anotherRocket = new Object();
anotherRocket.engineCount = 1;
anotherRocket.wings = 2;
```

Both rocket objects are seemingly identical, apart from the second rocket has a new property called *wings*. Because the first rocket doesn't have this, we encounter a pretty killer problem. For example, if we outputted the number of wings on the second rocket, we'd be returned "2", but if we did the same to the first rocket, we'd be returned "undefined". The first rocket has no wings, they were never created for it.

So how do we overcome this issue of objects being unique and unpredictable? By creating our own rocket object template with all the properties and methods of a rocket already declared inside, of course. It's easier to show why this is better with an example:

```
function Rocket(engineCount, thrust, wings) {
        this.engineCount = engineCount;
        this.thrust = thrust;
```

```
        this.wings = wings;
};
```

You'll probably recognize that as a function, and that's for a very good reason – functions are objects. This allows you to create a function as you'd normally do so, then use it in a way that treats it as an object. It's important to point out that the major difference between a regular function and our object template function is the way variables are declared. We want to declare properties, not variables, so we use the `this` keyword instead of `var`. The `this` keyword just means that the property is assigned to *this* instance of the object, allowing different instances to have different property values. It's a bit confusing, but bear with me.

Using the new rocket object template is really easy:

```
var rocket = new Rocket(2, 5000, 4);
var anotherRocket = new Rocket(1, 2000, 2);
```

Not only is this so much neater, but we've completely dropped the code duplication. The other benefit is that we can trust that our rocket objects have the same properties:

```
alert(rocket.wings); // Outputs 4
alert(anotherRocket.wings); // Outputs 2
```

Hooray! But properties are pretty static, how do we make our rocket object actually do something? For this we're going to need methods, which are surprisingly simple to implement. The following will add a method to turn the rocket's engines on:

```
function Rocket(engineCount, thrust, wings) {
        this.engineCount = engineCount;
        this.enginesOn = false;
        this.thrust = thrust;
        this.wings = wings;

        this.turnEnginesOn = function() {
                this.enginesOn = true;
                alert("Engines are now on");
        };
};
```

An object method is basically a function assigned to an object property. Remember, functions are objects so they're a data type, which means they can be assigned to variables (and properties). The name of the method will take the name of the property the method is assigned to, this will make sense when we come to calling the method later on.

For our purposes we have declared a new rocket property called *enginesOn* that tells us whether the engines are on (true), or off (false). By using this in the *turnEnginesOn* method we effectively turn the rocket's engines on, and to clarify matters we also pop up an alert box for good measure (remember, we're only using alert boxes for testing purposes).

Now we've set up the rocket *turnEnginesOn* method, it's a trivial task to actually use it:

```
var rocket = new Rocket(2, 5000, 4);
rocket.turnEnginesOn();
```

Object methods are functions, so they are called exactly as you would a function. That's really as complicated as methods get. They may take a while to get your head around, but they aren't scary at all.

I completely understand if you still don't quite get objects, they took me a little while to grasp as well. My hope is that this brief overview of them will at least give you the necessary knowledge to not get scared by our use of objects in the games. You'll have to trust me on this one, but objects will make much more sense when you start to use them in proper programming projects.

Arrays

At some point you're going to get the urge to be rebellious and store more than one value in a variable. I mean, those variables are just too damn restrictive, right? Fortunately, there is a perfect feature in JavaScript that allows you to continue your disruptive streak. That feature is the `Array` object.

Arrays (created using the `Array` object) are essentially containers that allow you to store multiple values. The great thing is that they're a data type, which means they can be assigned to variables, just like we've seen variables used with numbers and strings so far. So one variable with an `Array` object assigned to it can effectively hold countless values (known as elements) – pretty cool stuff!

The purpose of an array is to stop you having to write out a huge amount of variables when you want to store lots of related data, like a list of planets in the solar system. An array acts like a list, keeping a number of related values happily contained inside a single variable.

Creating arrays

There are a few methods available for creating an `Array` object, the most verbose of which looks like this:

```
var planets = new Array();
planets[0] = "Mercury";
```

The first line creates an empty `Array` object and assigns it to the variable *planets*. The second line assigns a planet name, as a string, to the first element of the array. It's important to note that the first element in an `Array` object has an index value of 0, not 1.

To add further elements to the same array you would do the following:

```
planets[1] = "Venus";
planets[2] = "Earth";
planets[3] = "Mars";
```

You can also create the `Array` object and assign some values in one go, like so:

```
var planets = new Array("Mercury", "Venus", "Earth", "Mars");
```

Or, if you really want to impress people you can use something called literal notation – a shortcut that uses square brackets:

```
var planets = ["Mercury", "Venus", "Earth", "Mars"];
```

Each method produces the same result, so feel free to use the method that you understand best when creating an Array object.

Accessing and modifying arrays

It's all well and good creating an Array object and packing all your values in a neat list, but how do you get those values back out again? It's easy. In fact, we've already seen the code that lets us do it.

This is how you can access the second element in the array of planets:

```
var planets = ["Mercury", "Venus", "Earth", "Mars"];
alert(planets[1]); // Outputs Venus
```

Notice again how the index number for elements in an array starts at 0, so the second value would be 1, not 2. And to access the fourth element in the array? Simple:

```
alert(planets[3]); // Outputs Mars
```

And that is really as complex as arrays need to get right now. There are many more things that you can do with them, but we'll cover those as we require them further on in the book.

Loops

We've already learnt that functions are great for automating chunks of code. However, there's one downside to functions, and that is that you have to call them every single time you want to use them. For example, want to run a function 5 times? Then call it 5 times. Want to run it 100 times? Call it 100 times. As you can see, writing 100 lines of code to call the same function is a little pointless. It would seem a bit odd for JavaScript to let us down now after being so awesome with functions and arrays. If you're doing the same thing 100 times, surely there's an easier way to do it? The good news is that JavaScript has not let us down (it's too good for that), it has a perfect set of features to deal with this – loops.

As a really simple example, imagine if we had a list of names in an array and wanted to run our good friend the *formatName* function 5 times, once for each name. Without loops it would look something like this:

```
var names = [["Rob", "Hawkes"], ["John", "Smith"], ["Jane", "Doe"], ["Queen",
"Elizabeth"],➥
 ["Steve", "Jobs"]];

var first = formatName(names[0][0], names[0][1]); // "Rob Hawkes"
var second = formatName(names[1][0], names[1][1]); // "John Smith"
var third = formatName(names[2][0], names[2][1]); // "Jane Doe"
var forth = formatName(names[3][0], names[3][1]); // "Queen Elizabeth"
var fifth = formatName(names[4][0], names[4][1]); // "Steve Jobs"
```

You're probably wondering what on earth is going on with that array. Fear not, it's just a multi-dimensional array, which basically means an array inside an array. The main array being a list of 5 other arrays, those 5

other arrays being the first and last name of a group of people. Don't fret about it too much if you don't get it; the focus is on loops so just go with the flow.

Now for only 5 iterations this isn't necessarily a bad thing, but it's certainly not pretty or efficient. So let's try it again, this time with a loop:

```
var names = [["Rob", "Hawkes"], ["John", "Smith"], ["Jane", "Doe"], ["Queen",
"Elizabeth"],➥
 ["Steve", "Jobs"]];

for (var i = 0; i < names.length; i++) {
        var fullName = formatName(names[i][0], names[i][1]);
};
```

There's no denying that this already looks much more efficient, let alone neater. The loop we're using here is a `for` loop, which is commonly used for running a block of code a specific number of times. Inside the parentheses we have three main areas of importance, each separated by a semi-colon. Let's look at each area in turn.

```
var i = 0;
```

In the first area we declare a variable (`i`) to be used as the counter for the loop, which we assign a value of 0 (remember: the first element of an array is 0).

```
i < names.length;
```

The second area is a check that is performed on our counter before each loop, in this case we're checking to see if our counter is less then the number of elements in the names array – its length, which is a property of every `Array` object. If this check returns true we move on to the third and final area, if not, we exit the loop.

```
i++
```

The final area is run at the end of each loop and, in our case, increases the counter by one by using the increment operator. This is the same as using `i = i+1`.

The result of all this is a counter variable that loops from 0 to 4, allowing us to access the 5 elements of the *names* array with just a single line:

```
var fullName = formatName(names[i][0], names[i][1]);
```

Notice how we place the *i* variable where we had the numbers 0 to 4 in the example without a loop. This is a much simpler way of doing things!

> *Note: There are other kinds of loops out there, like the `while` loop, but we won't be using them in this book. They aren't any more complex than `for` loops, so don't let me stop you learning them in your own time.*

Timers

As the name implies, timers are methods that allow you to run blocks of code once after a certain amount of time has passed (like a cooking timer), or many times after a repeating interval (like the indicator in your car). Both methods are incredibly useful, and will actually form an integral part of the games later in the book.

Setting one-off timers

The `setTimeout` method allows us to run a block of code after a specific delay in milliseconds. It's dead easy to set up, and it uses nothing that we haven't already seen before:

```
function onTimeout() {
        alert("Ding dong!");
};
var timer = setTimeout(onTimeout, 3000);
```

There are two arguments in a `setTimeout` method; the code or function you want to call when the timer runs, and the delay in milliseconds before running the timer. In our case we're calling a function in the timer, but notice how we left out the parentheses. This is because if we left the parentheses in, the function would be called immediately when setting up the timer – not what we want to happen. It's a bit counterintuitive, but bear with it. The second argument is the delay in milliseconds, which in our case is 3000, or 3 seconds. Remember that there are 1000 milliseconds in a second, forgetting that has caught me out a few times.

If all went well, then after three seconds you'll get an alert with the message "Ding dong!".

Unsetting one-off timers

Sometimes you'll want to stop a `setTimeout` method from running, usually when something has happened after setting it that means you don't need it any more. To unset a timer created using `setTimeout` you need to use the `clearTimeout` method. By placing the variable you assigned the timer to as an argument in the `clearTimeout` method, you'll stop it from running:

```
clearTimeout(timer);
```

Obviously, if you're too late clearing the timer then it will run as normal.

Setting repeating timers

A lot of the time a single-use timer isn't enough, you want something that repeats over and over again. For these situations you want to use the `setInterval` method, which is set in an identical way to `setTimout`:

```
function onInterval() {
        alert("Ding dong!");
};

var interval = setInterval(onInterval, 3000);
```

The only difference between this and the `setTimeout` method is that the function we pass to the `setInterval` method will be run every 3000 milliseconds, forever. So be warned, running this in your browser will cause an alert box to pop up every 3 seconds! I'd hope in a real-world situation that you'd be doing something other than annoying users with alert boxes.

Unsetting repeating timers

Just like the `clearTimeout` method, we have a similar way of clearing intervals, the `clearInterval` method. It's used in exactly the same way, except this time we pass the variable we assigned the `setInterval` method to as an argument:

```
clearInterval(interval);
```

To avoid things like infinite loops (when something never ends), you'll probably want to call the `clearInterval` method once your timer has achieved what you wanted from it. Leaving it running would just be a waste of resources, and would also be really annoying.

> *Note: It's important to note that both `setTimeout` and `setInterval` are methods of the DOM `window` object. They aren't truly JavaScript, although we do access and manipulate them through JavaScript. This will make more sense to you as we learn more about JavaScript and the DOM.*

The DOM

We encountered the DOM earlier in this chapter, its purpose is to represent the raw structure of our content and HTML elements in our web page. In other words, the DOM is where you'll be going if you want to access or manipulate your content with JavaScript. For the purposes of this book we won't be covering absolutely everything you can do with the DOM, but we'll certainly cover everything needed for games. Anything else can be found easily online or in other books, some dedicated entirely to teaching the DOM.

An example HTML web page

We'll be accessing HTML elements a lot in this section, so let's set up a cut down version of the HTML5 blog page we used in the first chapter. It's not a particularly useful web page, but it will serve the purpose for the coming examples:

```
<!DOCTYPE html>

<html>
        <head>
                <title>The DOM</title>
                <meta charset="utf-8">

                <script type="text/javascript" src="http://ajax.googleapis.com➥
/ajax/libs/jquery/1/jquery.min.js"></script>
```

```
            <script type="text/javascript">
                $(document).ready(function() {
                    // JavaScript will go in here
                });
            </script>
        </head>

        <body>
            <section id="blogArticles">
                <article>
                    <header>
                        <hgroup>
                            <h1><a href="/blog/first-post-
link/">Main➥
 heading of the first blog post</a></h1>
                            <h2>Sub-heading of the first blog
post</h2>
                        </hgroup>
                        <p>Posted on the <time pubdate datetime=➥
"2010-10-30T13:08">30 October 2010 at 1:08 PM</time></p>
                    </header>

                    <p>Summary of the first blog post.</p>
                </article>
            </section>
        </body>
</html>
```

Accessing the DOM using pure JavaScript

Before I show you the easier jQuery way, it's important to highlight how to access the DOM using pure JavaScript. The way we do this is by using the document object, which is the root of your web page that contains all of the HTML elements. As well as containing elements, the document object also provides a few methods that allow you to access specific elements, based on their attributes or tag name. For example, the following would give you access to the first HTML element with the id *blogArticles*:

```
document.getElementById("blogArticles");
```

The method we're using is very much self-explanatory, as are the rest of them. It's quite obvious that the method will get a HTML element by its id attribute. You can see why it's so important to name things well, it makes it crystal clear about what they do.

Another document object method allows you to access all the elements with a specific tag name. In our example, we can access the p elements like this:

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

The `getElementsByTagName` method returns an array containing all the matching HTML elements. In our case it returns an array containing one `p` element from the `header` element, and another from the `article` element.

Now, simply calling `getElementById` and `getElementsByTagName` doesn't achieve much. The really interesting stuff happens when you dig a little deeper into the HTML elements they both return. Once you have accessed a HTML element using the DOM, a whole range of properties becomes available to you. Some let you access the value of the elements' attributes, while one in particular lets you access the content contained within the element. This property is quite an interesting one, so let's take a quick look at how to access the content of a HTML element by using the `innerHTML` property:

```
var secondaryHeadings = document.getElementsByTagName("h2");
alert(secondaryHeadings[0].innerHTML);
```

The first thing we do here is search for all HTML `h2` elements and assign the resulting array to the *secondaryHeadings* variable. Now, remembering that arrays start at 0, we're able to access the `innerHTML` property of the first (in our case, the only) `h2` element on the page. The result is an alert box that outputs the text inside of the `h2` element. This is the point at which I started to get excited when I first learnt about the DOM, but maybe that's just me.

Accessing the DOM using jQuery

Using pure JavaScript to access the DOM isn't particularly scary, although it's a bit long-winded if anything. Still, jQuery allows us to access the DOM in a simple, yet ridiculously powerful way. For example, the jQuery equivalent of the `getElementById` method is:

```
$("#blogArticles");
```

Check out how short it is! But seriously, it's 21 characters shorter – that's really, *really* short. It's worth noticing the hash symbol (#) before the id name, this is because jQuery uses the same prefixes for matching elements as CSS – a hash for an id name, a dot for a class name, and nothing to match a tag name. It's also worth noting the dollar symbol ($) at the beginning of the code; this is a shortcut for accessing jQuery, and is the equivalent of calling `jQuery("#blogArticles")`.

So, by now you'd probably find it trivial to rewrite the `getElementsByTagName` example in jQuery, but I'll show you just in case. It's really easy:

```
$("p");
```

That is genuinely all there is to it. It goes beyond belief that so few characters can do so much, but do much those 7 characters can.

And what about the `innerHTML` property, is that easy in jQuery as well? You betcha!

```
var secondaryHeadings = $("h2");
alert(secondaryHeadings.html());
```

All you need to do is use the jQuery `html` method, it's does exactly the same as the `innerHTML` property really. The main difference with the `html` method is that is always returns the first HTML element, so you don't need to refer to the array index, like you do with `innerHTML`.

Manipulating the DOM

Being able to access content in the DOM is cool, but being able to edit that content is even cooler. The best part is that we already know the code to do it, it's all to do with the `html` method in jQuery, or the `innerHTML` property in pure JavaScript.

Let's jump in and change the `h2` element content in our cut-down HTML5 blog page:

```
var secondaryHeadings = $("h2");
secondaryHeadings.html("Now we've changed the content");
```

If you did everything right you should get something a little like Figure 2-4 (notice the heading has changed):

Figure 2-4. Manipulating the DOM using jQuery.

It's important to realize that we haven't edited the actual HTML file, we have merely changed the way it is being outputted in the browser. I really can't stress this enough. If you don't believe me, try disabling

JavaScript in your browser and looking at the page again – the heading will be back to what it was originally.

As I mentioned at the beginning of this section, we have only scratched the surface with what the DOM can really do. The scope of this book means I can't teach you everything, instead I must focus on the functionality we are actually going to use. This is the same for everything I'm teaching you about JavaScript, there is much that has been left out. If you want to learn more about the DOM or JavaScript then I definitely recommend picking up a book that focuses on them, or checking out the W3Schools website at `www.w3schools.com`.

Summary

We've covered an absolutely massive amount in this chapter, I commend you for making it this far. In just one chapter I've tried to explain everything I possibly can about JavaScript, something that really deserves a whole book dedicated to it! Still, we've learnt things like where JavaScript has come from, why libraries like jQuery make things easier for us, and how to add JavaScript to a HTML web page. We've also covered the fundamental features of JavaScript, like variables, data types, conditional statements, functions, arrays, loops, timers, the DOM, and objects. It sounds like quite a lot when you list it like that, doesn't it?

Perhaps you should take a little break now. Have a cup of tea, let your brain take in what we've covered, and then pick the book back up again. Next up, we're going to learn how to use HTML5 canvas (my not-so-secret favorite part of HTML5).