Pro JavaScript Techniques

REAL-WORLD JAVASCRIPT TECHNIQUES FOR TODAY'S DEVELOPER

John Resig, Russ Ferguson, and John Paxton

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

friendsof

Apress®

Contents at a Glance

v

■ ■ ■

Professional JavaScript Techniques

Welcome to *Pro JavaScript Techniques*. This book provides an overview of the current state of JavaScript, particularly as it applies to the professional programmer. Who is the professional programmer? Someone who has a firm grasp of the basics of JavaScript (and probably several other languages). You are interested in the breadth and depth of JavaScript. You want to look at the typical features like the Document Object Model (DOM), but also learn about what's going on with all this talk of Model-View-Controller (MVC) on the client side. Updated APIs, new features and functionality, and creative applications of code are what you are looking for here.

This is the second edition of this book. Much has changed since the first edition came out in 2006. At that time, JavaScript was going through a somewhat painful transition from being a toy scripting language to being a language that was useful and effective for several different tasks. It was, if you will, JavaScript's adolescence. Now, JavaScript is at the end of another transition: to continue the metaphor, from adolescence to adulthood. JavaScript usage is nearly ubiquitous, with anywhere from 85 to 95 percent of websites, depending on whose statistics you believe, having some JavaScript on their main page. Many people speak of JavaScript as the most popular programming language in the world (in the number of people who use it on a regular basis). But more important than mere usage are effectiveness and capability.

JavaScript has transitioned from a toy language (image rollovers! status bar text manipulations!) to an effective, if limited tool (think of client-side form validation), to its current position as a broad-featured programming language no longer limited to mere browsers. Programmers are writing JavaScript tools that provide MVC functionality, which was long the domain of the server, as well as complex data visualizations, template libraries, and more. The list goes on and on. Where in the past, designers would have relied on a .NET or Java Swing client to provide a full-featured, rich interface to server-side data, we can now realize that application in JavaScript with a browser. And, using Node.js, we have JavaScript's own version of a virtual machine, an executable that can run any number of different applications, all written in JavaScript and none requiring a browser.

This chapter will describe how we got here and where we are going. It will look at the various improvements in browser technology (and popularity) that have abetted the JavaScript Revolution. The state of JavaScript itself needs inspection, as we want to know where we are before we look at where we are going. Then, as we examine the chapters to come, you will see what the professional JavaScript programmer needs to know to live up to his or her title.

How Did We Get Here?

As of the first edition of the book, Google Chrome and Mozilla Firefox were relatively new kids on the block. Internet Explorer 6 and 7 ruled the roost, with version 8 gaining some popularity. Several factors combined to jump-start JavaScript development.

For most of its life, JavaScript was dependent upon the browser. The browser is the runtime environment for JavaScript, and a programmer's access to JavaScript functionality was highly dependent

upon the make, model, and version of browser visiting said programmer's website. By the mid-2000s, the browser wars of the 90s had been easily won by Internet Explorer, and browser development stagnated. Two browsers challenged this state of affairs: Mozilla Firefox and Google Chrome. Firefox was the descendant of Netscape, one of the earliest web browsers. Chrome had Google's backing, more than enough to make it an instant player on the scene.

But both of these browsers made a few design decisions that facilitated the JavaScript revolution. The first decision was to support the World Wide Web consortium's implementation of various standards. Whether dealing with the DOM, event handling, or Ajax, Chrome and Firefox generally followed the spec and implemented it as well as possible. For programmers, this meant that we didn't have to write separate code for Firefox and Chrome. We were already used to writing separate code for IE and something else, so having branching code in itself was not new. But making sure that the branching was not overly complex was a welcome relief.

Speaking of standards, Firefox and Chrome also put in a lot of work with the European Computer Manufacturer's Association (ECMA, now styled Ecma). Ecma is the standards body that oversees JavaScript. (To be technical, Ecma oversees the ECMAScript standard, since JavaScript is a trademark of Oracle and… well, we don't really care about those details, do we? We will use JavaScript to refer to the language and ECMAScript to refer to the specification to which a JavaScript implementation adheres.) ECMAScript standards had languished in much the same way as IE development. With the rise of real browser competition, the ECMAScript standard was taken up again. ECMAScript version 5 (2009) codified many of the changes that had been made in the ten years (!) since the previous version of the standard. The group itself was also energized, with version 5.1 coming out in 2011. The future is provided for, with significant work currently being done on both versions 6 and 7 of the standard.

To give credit where credit is due, Chrome pushed the updating of JavaScript as well. The Chrome JavaScript engine, called V8, was a very important part of Chrome's debut in 2008. The Chrome team built an engine that was much faster than most JavaScript engines, and it has kept that goal at the top of the list for subsequent versions. In fact, the V8 engine was so impressive that it became the core of Node.js, a browser-independent JavaScript interpreter. Originally intended as a server that would use JavaScript as its main application language, Node has become a flexible platform for running any number of JavaScript-based applications.

Back to Chrome: the other major innovation Google introduced to the land of browsers was the concept of the evergreen application. Instead of having to download a separate browser install for updates, Chrome's default is to automatically update the browser for you. While this approach is sometimes a pain in the corporate world, it is a great boon to the noncorporate consumer surfer (also known as a person!). If you use Chrome (and, for the last few years, Firefox), your browser is up-to-date, without your having to make any effort. While Microsoft has done this for a long time in pushing security updates via Windows Update, it does not introduce new features to Internet Explorer unless they are coupled to a new version of Windows. To put it another way, updates to IE are slow in coming. Chrome and Firefox always have the latest and greatest features, as well as being quite secure.

As Google pressed on with Chrome's features, the other browser makers played catch-up. Sometimes this came in sillier ways, such as when Firefox adapted Chrome's version numbering. But it also resulted in Mozilla and Microsoft taking a cold, hard look at JavaScript engines. Both browser makers have significantly overhauled their JS engines over the last few years, and Chrome's lead, while formidable, is no longer insurmountable.

Finally, Microsoft has (mostly) thrown in the towel on its classic "embrace and extend" philosophy, at least when it comes to JavaScript. With IE version 9, Microsoft implemented World Wide Web Consortium (W3C) event handling and standardized its DOM interfaces as well as its Ajax API. For most of the standard features of JavaScript, we no longer have to implement two versions of the same code! (Legacy code for legacy browsers is still a bit of an issue, of course…)

It seems almost a panacea. JavaScript is faster than ever before. It is easier to write code for a variety of different browsers. Standards documents both describe the real world and provide a useful roadmap to features to come. And most of our browsers are fully up-to-date. So what do we need to worry about now, and where are we going in the future?

Modern JavaScript

It has never been easier to develop serious applications with JavaScript. We have a clear, clean break with the bad old days of separate code for multiple browsers, poor standards poorly implemented, and slow JavaScript engines that were often an afterthought. Let's take a look at the state of the modern JavaScript environment. Specifically, we will look at two areas: the modern browser and the modern toolkit.

Modern JavaScript depends on the idea of the modern browser. What is the modern browser? Different organizations describe it in different ways. Google says that their applications support the current and previous major versions of browsers. (Fascinating, as Gmail still works on IE9, as far as we can tell!) In an interesting article, the people behind the British Broadcasting Company (BBC) website revealed that they define a modern browser as one that supports the following capabilities:

1. `document.querySelector()` / `document.querySelectorAll()`

2. `window.addEventListener()`

3. The Storage API (`localStorage` and `sessionStorage`)

jQuery, probably the most popular JavaScript library on the web, split its versions into the 1.x line, which supports IE 6 and later, and the 2.x line, which supports "modern" browsers like IE 9 and later. And make no mistake, IE is the dividing line between the modern and the ancient. The other two major browsers are evergreen. And while Safari and Opera are not evergreen, they update on a faster schedule than IE and don't have nearly the market share it does.

So where is the borderline for the modern browser? Alas, the border seems to wander between Internet Explorer versions 9 through 11. But IE 8 is definitely on the far side of browser history. It does not support most of the features of ECMAScript 5. It does not include the API for W3C event handling. The list goes on and on. So when we discuss modern browsers, we will refer to at least Internet Explorer 9. And our coverage will not endeavor to support ancient browsers. Where relevant and simple, we will point out polyfills for older versions of IE, but in general, our floor is Internet Explorer 9.

The Rise of Libraries

In addition to the modern browser, there is another important aspect of the current environment for JavaScript we need to discuss: libraries. Over the past 8 years, there has been an explosion in the number and variety of JavaScript libraries. There are more than 800,000 GitHub repositories for JavaScript; of these, almost 900 have more than 1,000 stars. From its humble beginnings as collections of utility functions, the JavaScript library ecosystem has evolved (somewhat chaotically) into a vast landscape of possibilities.

How does this affect us as JavaScript developers? Well, of course, there is the model of "library as expansion," where a library provides additional functionality. Think of the MVC libraries like Backbone and Angular (which we will be looking at in a later chapter), or the data visualization libraries like d3 or Highcharts. But JavaScript is in an interesting position, as libraries can also provide a level interface to features that are standard on some browsers but not on others.

For a long time, the standard example of a variably implemented feature in JavaScript was event handling. Internet Explorer had its own event-handling API. Other browsers generally followed the W3C's API. Various libraries provided unified implementations for event handling, including the best of both worlds. Some of these libraries stood alone, but the successful ones also normalized functionality for Ajax, the DOM, and a number of other features that were differently implemented across browsers.

The most popular of these libraries has been jQuery. Since its inception, jQuery has been the go-to library for using new JavaScript features without worrying about the browser's support for those features. So instead of using IE's event handling or the W3C's, you could simply use jQuery's `.on()` function, which wrapped around the variance, providing a unified interface. Several other libraries provided similar functionality: Dojo, Ext JS, Prototype, YUI, MooTools, and so on. These toolkit libraries aimed to standardize APIs for developers.

The standardization goes further than providing simple branching code. These libraries often ameliorate buggy implementations. The official API for a function may not change much between versions, but there will be bugs; sometimes those bugs will be fixed, sometimes not, and sometimes the fixes will introduce new bugs. Where libraries could fix or work around these bugs, they did. For example, jQuery 1.11 contains more than a half-dozen fixes for problems with the event-handling API.

Some libraries (jQuery in particular) also provided new or different interpretations of certain capabilities. The jQuery selector function, the core of the library, predates the now-standard `querySelector()` and `querySelectorAll()` functions, and it was a driver for including those functions in JavaScript. Other libraries provide access to functionality despite very different underlying implementations. Later in the book, we will look at Ajax's new Cross Origin Resource Sharing (CORS) protocol, which allows for Ajax requests to servers other than the one that originally served the page. Some libraries have already implemented a version of this that uses CORS but falls back to JSON with padding (JSON-P) where needed.

Because of their utility, some libraries have become part of a professional JavaScript programmer's standard development toolkit. Their features may not be standardized into JavaScript (yet), but they are an accumulation of knowledge and functionality that simply makes it easier to realize designs quickly. In recent years, though, you could get quite a few hits to your blog by asking whether jQuery (or another library) was really necessary for development on a modern browser. Consider the BBC's requirements; you can certainly realize a large degree of jQuery-like functionality if you have those three methods available to you. But jQuery also includes a simplified yet expanded DOM interface, it handles bugs for a variety of different edge cases, and if you need support for IE 8 or earlier, jQuery is your major option. Accordingly, the professional JavaScript programmer must look at the requirements for a project and consider whether it pays to risk reinventing the wheel that jQuery (or another similar library) provides.

More Than a Note about Mobile

In older JavaScript and web development books, you would reliably see a section, maybe a whole *chapter*, on what to do about mobile browsing. Mobile browsing was a small enough share of total browsing, and the market was so fractured, that it seemed only specialists would be interested in mobile development. That's not the case anymore. Since the first edition of this book, mobile web browsing has exploded, and it is a very different beast from desktop development. Consider some statistics: according to a variety of sources, mobile browsing represents between 20 and 30 percent of all browsing. By the time you are reading this, it may well represent more, as it has consistently increased since the debut of the iPhone. Speaking of which, well over 40 percent of mobile browsing is done with iOS Safari, although Android's browser and Chrome for Android are gaining ground (and may have overtaken Safari, depending on whose stats you believe). Safari on iOS is not the same as Safari on the desktop, and the same goes for Android Chrome vs. desktop Chrome and mobile Firefox vs. desktop Firefox. Mobile is mainstream.

The browsers on mobile devices provide a new set of challenges and opportunities. Mobile devices are often more limited than desktops (though that's another gap that is rapidly closing). Conversely, mobile devices offer new features (swipe events, more accurate geolocation, and so on) and new interaction idioms (using the hand instead of the mouse, swiping for scrolling). Depending on your development requirements, you may have to build an app that looks good on mobile as well as the desktop, or reimplement existing functionality for a mobile platform.

The JavaScript landscape has changed extensively over the last few years. Despite some standardization of APIs, there are also many new challenges. How will this affect us as professional JavaScript programmers?

Where Do We Go from Here?

We should set down some standards for ourselves. We have already set one: IE9 as the floor of the modern browser experience. The other browsers are evergreen, and not to worry about. What about mobile, then? While the issue is complex, iOS 6 and Android 4.1 (Jelly Bean) will, in general, serve as our floors. Mobile computing updates faster and more frequently than desktops do, so we are confident in using these more recent versions of mobile operating systems.

That said, let us digress for a moment to discuss not browser versions, operating systems, or platforms, but your audience. While we can quote statistics all day long, the valuable statistics tell you about *your* audience, not the audience in general. Perhaps you are designing for your employer, who has standardized on IE 10. Or maybe your idea for an application depends heavily on features that only Chrome provides. Or maybe there isn't even a desktop version, but you're aiming for a roll-out to iPads and Android tablets. Consider your target audience. This book is written to be broadly applicable, and your application may be as well. But it would be folly to spend time worrying about bugs in IE9 for that previously mentioned tablet-only application, wouldn't it? Now, back to our standards.

For screenshots and testing, this book will generally prefer Google Chrome. Occasionally, we will demonstrate code on Firefox or Internet Explorer where it is relevant. Chrome, for developers, is the gold standard—not necessarily in user-friendliness, but certainly in the information exposed to the programmer. In a later chapter, we will look at the various developer tools available, scrutinizing not only Chrome, but Firefox (with and without Firebug) and IE as well.

As a standard library, we will refer to jQuery. There are many alternatives, of course, but jQuery wins for two reasons: first, it is the most popular general-use JavaScript library on the web. Second, at least one of the authors (John Resig) has a little bit of history with jQuery, which predisposed the other author (John Paxton) to concede the point of working with it. In updating this book, we have replaced many of the techniques from the previous version with jQuery's library of functionality. In these cases, we are disinclined to reinvent the wheel. As needed, we will refer to the appropriate jQuery functionality. We will, of course, discuss new and exciting techniques, as well!

JavaScript IDEs have updated significantly in the last few years, driven by JavaScript's own rise. The possibilities are too numerous to list here, but there are a few applications of note. John Resig uses a highly customized version of vim for his development environment. John Paxton is a little bit lazier, and has elected to use JetBrains' excellent WebStorm (`http://www.jetbrains.com/webstorm/`) as his IDE. Adobe offers the open source, free Brackets IDE (`http://brackets.io/`), currently at version 1.3. Eclipse is also available, and many people have reported positive results by customizing SublimeText or Emacs to do their bidding. As always, use what you feel most comfortable with.

There are other tools that can assist in JavaScript development. Rather than list them here, we will dedicate a chapter to them later in the book. Which means it's a good time to give an outline of what's to come.

Coming Up Next

Starting with Chapter 2, we will look at the latest and greatest in the JavaScript language. This means looking at new features like those available through the `Object` type, but also reexamining some older concepts like references, functions, scope, and closures. We will lump all of this under the heading of **Features**, **Functions**, **and Objects**, but it covers a bit more than that.

Chapter 3 discusses **Creating Reusable Code**. Chapter 2 skips over one of the biggest new features of JavaScript, the `Object.create()` method, and its implications for object-oriented JavaScript code. So in this chapter we will spend time with `Object.create()`, functional constructors, prototypes, and object-oriented concepts as implemented in JavaScript.

Having spent two chapters developing code, we should start thinking about how to manage it. Chapter 4 shows you tools for **Debugging JavaScript Code**. We start by examining browsers and their developer tools.

Chapter 5 begins a sequence discussing some high-usage areas of JavaScript functionality. Here we look at the **Document Object Model**. The DOM API has increased in complexity and has not really become more straightforward since the last edition. But there are new features that we should familiarize ourselves with.

In Chapter 6, we attempt to master Events. The big news here is the standardization of the events API along the lines of the W3C style. This provides us the opportunity to move away from utility libraries and finally go deep into the events API without worrying about large variations between browsers.

One of the first non-toy applications for JavaScript was client-side form validation. Amazingly, it took browser makers over a decade to think about adding functionality to form validation beyond capturing the submit event. When looking in Chapter 7 at **JavaScript and Form Validation**, we will discover that there is a whole new set of functionality for form validation provided by both HTML and JavaScript.

Everyone who develops with JavaScript has spent some time **Introduction to Ajax**. With the introduction of Cross-Origin Resource Sharing (CORS), Ajax functionality has finally moved past the silliest of its restrictions.

Command line tools like Yeoman, Bower, Git and Grunt are covered in **Web Production Tools**. These tools will show us how to quickly add all the files and folders needed. This way we can focus on development.

Chapter 10 covers **AngularJS and Testing**. Using the knowledge gained in the previous chapter, we now start to look at what makes Angular work and how to implement both unit and end to end testing.

Last, Chapter 11 discusses the **Future of JavaScript**. ECMAScript 6 will be settled, more or less, by the time this book goes to press. ECMAScript 7 is in active development. Beyond the basics of where JavaScript is going, we will look at what features you can use right now.

Summary

We spent a lot of this chapter on everything around JavaScript: the platform(s), the history, the IDEs, and so on. We believe that history informs the present. We wanted to explain where we were, and how we got here, to help you understand why JavaScript is where it is, and is what it is, today. Of course, we plan to spend the bulk of this book talking about how JavaScript works, particularly for the professional programmer. We feel quite strongly that this book covers the techniques and APIs that every professional JavaScript programmer should be familiar with. So without further ado…

CHAPTER 2

■ ■ ■

Features, Functions, and Objects

Objects are the fundamental units of JavaScript. Virtually everything in JavaScript is an object and interacts on an object-oriented level. To build up this solid object-oriented language, JavaScript includes an arsenal of features that make it unique in both its foundation and its capabilities.

This chapter covers some of the most important aspects of the JavaScript language, such as references, scope, closures, and context. These are not necessarily the cornerstones of the language, but the elegant arches, which both support and refine JavaScript. We will delve into the tools available for working with objects as data structures. A dive into the nature of object-oriented JavaScript follows, including a discussion of classes vs. prototypes. Finally, the chapter explores the use of object-oriented JavaScript, including exactly how objects behave and how to create new ones. This is quite possibly the most important chapter in this book if taken to heart, as it will completely change the way you look at JavaScript as a language.

Language Features

JavaScript has a number of features that are fundamental to making the language what it is. There are very few other languages like it. We find the combination of features to fit just right, contributing to a deceptively powerful language.

References and Values

JavaScript variables hold data in one of two ways: by copies and references. Anything that is a primitive value is *copied* into a variable. Primitives are strings, numbers, Booleans, null, and undefined. The most important characteristic of primitives is that they are assigned, copied, and passed to and returned from functions by *value*.

The rest of JavaScript relies on references. Any variable that does not hold one of the aforementioned **primitive** values holds a **reference** to an object. A reference is a pointer to the location in memory of an object (or array, or date, or what-have-you). The actual object (array, date, or whatever) is called the **referent**. This is an incredibly powerful feature, present in many languages. It allows for certain efficiencies: two (or more!) variables do not have their own copies of an object; they simply refer to the same object. Updates to the referent made via one reference are reflected in the other reference. By maintaining sets of references to objects, JavaScript affords you much more flexibility. An example of this is shown in Listing 2-1, where two variables point to the same object, and the modification of the object's contents via one reference is reflected in the other reference.

7

Listing 2-1. Example of Multiple Variables Referring to a Single Object

```
// Set obj to an empty object
// (Using {} is shorter than 'new Object()')
var obj = {};

// objRef now refers to the other object
var refToObj = obj;

// Modify a property in the original object
obj.oneProperty = true;

// We now see that the change is represented in both variables
// (Since they both refer to the same object)
console.log( obj.oneProperty === refToObj.oneProperty );

// This change goes both ways, since obj and refToObj are both references
refToObj.anotherProperty = 1;
console.log( obj.anotherProperty === refToObj.anotherProperty );
```

Objects have two features: properties and methods. These are often referred to collectively as the *members* of an object. Properties contain the data of an object. Properties can be primitives or objects themselves. Methods are functions that act upon the data of an object. In some discussions of JavaScript, methods are included in the set of properties. But the distinction is often useful.

Self-modifying objects are very rare in JavaScript. Let's look at one popular instance where this occurs. The Array object is able to add additional items to itself using the push method. Since, at the core of an Array object, the values are stored as object properties, the result is a situation similar to that shown in Listing 2-1, where an object becomes globally modified (resulting in multiple variables' contents being simultaneously changed). An example of this situation can be found in Listing 2-2.

Listing 2-2. Example of a Self-Modifying Object

```
// Create an array of items
// (Similar to 2-1, using [] is shorter than 'new Array()')
var items = [ 'one', 'two', 'three' ];

// Create a reference to the array of items
var itemsRef = items;

// Add an item to the original array
items.push( 'four' );

// The length of each array should be the same,
// since they both point to the same array object
console.log( items.length == itemsRef.length );
```

It's important to remember that references point only to the referent object, not to another reference. In Perl, for example, it's possible to have a reference point to another variable that also is a reference. In JavaScript, however, it traverses down the reference chain and only points to the core object. An example of this situation can be seen in Listing 2-3, where the physical object is changed but the reference continues to point back to the old object.

Listing 2-3. Changing the Reference of an Object While Maintaining Integrity

```
// Set items to an array (object) of strings
var items = [ 'one', 'two', 'three' ];
// Set itemsRef to a reference to items
var itemsRef = items;

// Set items to equal a new object
items = [ 'new', 'array' ];

// items and itemsRef now point to different objects.
// items points to [ 'new', 'array' ]
// itemsRef points to [ 'one', 'two', 'three' ]
console.log( items !== itemsRef );
```

Finally, let's look at a strange instance that you might think would involve references but does not. When performing string concatenation, the result is always a new string object rather than a modified version of the original string. Because strings (like numbers and Booleans) are primitives, they are not actually referents, and the variables that contain them are not references. This can be seen in Listing 2-4.

Listing 2-4. Example of Object Modification Resulting in a New Object, Not a Self-Modified Object

```
// Set item equal to a new string object
var item = 'test';

// itemRef now refers to the same string object
var itemRef = item;

// Concatenate some new text onto the string object
// NOTE: This creates a new object and does not modify
// the original object.
item += 'ing';

// The values of item and itemRef are NOT equal, as a whole
// new string object has been created
console.log( item != itemRef );
```

Strings are often particularly confusing because they act like objects. You can create instances of strings via a call to new String. Strings have properties like length. Strings also have methods like indexOf and toUpperCase. But when interacting with variables or functions, strings are very much primitives.

References can be a tricky subject to wrap your mind around, if you are new to them. Nonetheless, understanding how references work is paramount to writing good, clean JavaScript code. In the next couple of sections we're going to look at features that aren't necessarily new or exciting but are important for writing good, clean code.

Scope

Scope is a tricky feature of JavaScript. Most programming languages have some form of scope; the differences lie in the duration of that scope. There are only two scopes in JavaScript: functional scope and global scope. This is deceptively simple. Functions have their own scope, but blocks (such as while, if, and for statements) do not. This may seem strange if you are coming from a block-scoped language. Listing 2-5 shows an example of the implications of function-scoped code.

Listing 2-5. Example of How the Variable Scope in JavaScript Works

```
// Set a global variable, foo, equal to test
var foo = 'test';

// Within an if block
if ( true ) {
    // Set foo equal to 'new test'
    // NOTE: This still belongs to the global scope!
    var foo = 'new test';
}

// As we can see here, as foo is now equal to 'new test'
console.log( foo === 'new test' );

// Create a function that will modify the variable foo
function test() {
    var foo = 'old test';
}

// However, when called, 'foo' remains within the scope
// of the function
test();

// Which is confirmed, as foo is still equal to 'new test'
console.log( foo === 'new test' );
```

You'll notice that in Listing 2-5, the variables are within the global scope. All globally scoped variables are actually visible as properties of the window object in browser-based JavaScript. In other environments, there will be a global context to which all globally-scoped variables belong.

In Listing 2-6 a value is assigned to a variable, foo, within the scope of the test() function. However, nowhere in Listing 2-6 is the scope of the variable actually declared (using var foo). When the foo variable isn't explicitly scoped, it will become defined globally, even though it is only intended to be used within the context of the function.

Listing 2-6. Example of Implicit Globally Scoped Variable Declaration

```
// A function in which the value of foo is set
function test() {
    foo = 'test';
}

// Call the function to set the value of foo
test();

// We see that foo is now globally scoped
console.log( window.foo === 'test' );
```

JavaScript's scoping is often a source of confusion. If you are coming from a block-scoped language, this confusion can lead to accidentally global variables, as shown here. Often, this confusion is compounded by imprecise usage of the var keyword. For simplicity's sake, the pro JavaScript programmer should always initialize variables with var, regardless of scope. This way, your variables will have the scope you expected, and you can avoid accidental globals.

When declaring variables within a function, be aware of the issue of hoisting. Any variable declared within a function has its declaration (not the value it is initialized with) hoisted to the top of the scope. JavaScript does this to ensure that the variable's name is available throughout the scope.

Especially when we combine scope with the concept of context and closures, discussed in the next two sections, JavaScript reveals itself as a powerful scripting language.

Context

Your code will always have some form of context (a scope within which the code is operating). Context can be a powerful tool and is essential for object-oriented code. It is a common feature of other languages, but JavaScript, as is often the case, has a subtly different take on it.

You access context through the variable this, which will always refer to the context that the code is running inside. Recall that global objects are actually properties of the window object. This means that even in a global context, this will still refer to an object. Listing 2-7 shows some simple examples of working with context.

Listing 2-7. Examples of Using Functions Within Context and Then Switching Context to Another Variable

```
function setFoo(fooInput) {
    this.foo = fooInput;
}

var foo = 5;
console.log( 'foo at the window level is set to: ' + foo );

var obj = {
    foo : 10
};

console.log( 'foo inside of obj is set to: ' + obj.foo );

// This will change window-level foo
setFoo( 15 );
console.log( 'foo at the window level is now set to: ' + foo );

// This will change the foo inside the object
obj.setFoo = setFoo;
obj.setFoo( 20 );
console.log( 'foo inside of obj is now set to: ' + obj.foo );
```

In Listing 2-7, our setFoo function looks a bit odd. We do not typically use this inside a generic utility function. Knowing that we were eventually going to attach setFoo to obj, we used this so we could access the context of obj. However, this approach is not strictly necessary. JavaScript has two methods that allow you to run a function in an arbitrary, specified context. Listing 2-8 shows the two methods, call and apply, that can be used to achieve just that.

Listing 2-8. Examples of Changing the Context of Functions

```
// A simple function that sets the color style of its context
function changeColor( color ) {
    this.style.color = color;
}

// Calling it on the window object, which fails, since it doesn't
// have a style object
changeColor('white' );

// Create a new div element, which will have a style object
var main = document.createElement('div');

// Set its color to black, using the call method
// The call method sets the context with the first argument
// and passes all the other arguments as arguments to the function
changeColor.call( main, 'black' );

//Check results using console.log
//The output should say 'black'
console.log(main.style.color);

// A function that sets the color on the body element
function setBodyColor() {
    // The apply method sets the context to the body element
    // with the first argument, and the second argument is an array
    // of arguments that gets passed to the function
    changeColor.apply( document.body, arguments );
}

// Set the background color of the body to black

setBodyColor('black' );
```

While the usefulness of context may not be immediately apparent, it will become clearer when we look at object orientation soon.

Closures

Closures are a means through which an inner function can refer to the variables present in its outer enclosing function after its parent functions have already terminated. That's the technical definition, anyway. Perhaps it is more useful to think of closures tied to contexts. Up to this point, when we have defined an object literal, that object was open for modification. We have seen that we can add properties and functions to the object at any time. But what if we wanted a context that was locked? A context that "saved" values as defaults. What about a context that could not be accessed without the API we provide? This is what a closure provides: a context that is accessible only in the manner we choose.

This topic can be very powerful and very complex. We highly recommend referring to the sites mentioned at the end of this section, as they have some excellent information about closures.

Let's begin by looking at two simple examples of closures, shown in Listing 2-9.

Listing 2-9. Two Examples of How Closures Can Improve the Clarity of Your Code

```
// Find the element with an ID of 'main'
var obj = document.getElementById('main');

// Change its border styling
obj.style.border = '1px solid red';

// Initialize a callback that will occur in one second
setTimeout(function(){
    // Which will hide the object
    obj.style.display = 'none';
}, 1000);

// A generic function for displaying a delayed alert message
function delayedAlert( msg, time ) {
    // Initialize an enclosed callback
    setTimeout(function(){
        // Which utilizes the msg passed in from the enclosing function
        console.log( msg );
    }, time );
}
// Call the delayedAlert function with two arguments
delayedAlert('Welcome!', 2000 );
```

The first function call to setTimeout shows an instance where new JavaScript developers often have problems. It's not uncommon to see code like this in a new developer's program:

```
setTimeout('otherFunction()', 1000);
```

or even...

```
setTimeout('otherFunction(' + num + ',' + num2 + ')', 1000);
```

In both examples, the functions being called are expressed as strings. This can cause problems with the minification process when you are about to move your code into production. By using closures, you can call functions, use variables, and pass parameters as originally intended.

Using the concept of closures, it's entirely possible to circumnavigate this mess of code. The first example in Listing 2-9 is simple; there is a setTimeout callback being called 1,000 milliseconds after it is first called, but still referring to the obj variable (which is defined globally as the element with an ID of main). The second function defined, delayedAlert, shows a solution to the setTimeout mess that occurs, along with the ability to have closures within function scope.

You should find that when using simple closures such as these in your code, the clarity of what you're writing increases instead of turning into a syntactical soup.

Let's look at a fun side effect of what's possible with closures. In some functional programming languages, there's the concept of *currying*, a way to prefill a number of arguments to a function, creating a new, simpler function. Listing 2-10 has a simple example of currying, creating a new function that prefills an argument to another function.

Listing 2-10. Example of Function Currying Using Closures

```
// A function that generates a new function for adding numbers
function addGenerator( num ) {

    // Return a simple function for adding two numbers
    // with the first number borrowed from the generator
    return function( toAdd ) {
        return num + toAdd
    };

}
// addFive now contains a function that takes one argument,
// adds five to it, and returns the resulting number
var addFive = addGenerator( 5 );

// We can see here that the result of the addFive function is 9,
// when passed an argument of 4
console.log( addFive( 4 ) == 9 );
```

There's another common JavaScript-coding problem that closures can solve. New JavaScript developers often accidentally leave a lot of extra variables sitting in the global scope. This is generally considered bad practice, as those extra variables could quietly interfere with other libraries, causing confusing problems to occur. Using a self-executing anonymous function, you can essentially hide all normally global variables from being seen by other code, as shown in Listing 2-11.

Listing 2-11. Example of Using Anonymous Functions to Hide Variables from the Global Scope

```
// Create a new anonymous function, to use as a wrapper
(function(){
    // The variable that would normally be global
    var msg = 'Thanks for visiting! ';

    // Binding a new function to a global object
    window.onload = function(){
        // Which uses the 'hidden' variable
        console.log( msg );
    };

// Close off the anonymous function and execute it
})();
```

Finally, let's look at one problem that occurs with closures. Remember that a closure allows you to reference variables that exist within the parent function. However, it does not provide the value of the variable at the time it is created; it provides the last value of the variable within the parent function. You'll most commonly see this occur during a for loop. There is one variable being used as the iterator (i). Inside the for loop, new functions are being created that utilize the closure to reference the iterator again. The problem is that by the time the new closured functions are called, they will reference the last value of the iterator (that is, the last position in an array), not the value that you would expect. Listing 2-12 shows an example of using anonymous functions to induce scope, to create an instance where expected closure is possible.

Listing 2-12. Example of Using Anonymous Functions to Induce the Scope Needed to Create Multiple Closure-Using Functions

```
// An element with an ID of main
var obj = document.getElementById('main');

// An array of items to bind to
var items = ['click', 'keypress' ];

// Iterate through each of the items
for ( var i = 0; i < items.length; i++ ) {
    // Use a self-executed anonymous function to induce scope
    (function(){
        // Remember the value within this scope
       // Each 'item' is unique.
      //Not relying on variables created in the parent context.
        var item = items[i];
        // Bind a function to the element
        obj['on' + item ] = function() {
            // item refers to a parent variable that has been successfully
            // scoped within the context of this for loop
            console.log('Thanks for your ' + item );
        };
    })();
}
```

We will return to closures in our section on object-oriented code, where they will help us to implement private properties.

The concept of closures is not a simple one to grasp; it took us a lot of time and effort to truly wrap our minds around how powerful closures are. Luckily, there are some excellent resources explaining how closures work in JavaScript: "JavaScript Closures" by Richard Cornford, at http://jibbering.com/faq/faq_notes/closures.html, and another explanation at the Mozilla Developer Network, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures.

Function Overloading and Type-Checking

A common feature in other object-oriented languages is the ability to *overload* functions to perform different behaviors depending on the type or number of arguments passed in. While this ability isn't a language feature in JavaScript, we can use existing capabilities to implement overloading of functions.

Our overloaded functions need to know two things: how many arguments have been passed in and what type of arguments have been passed. Let's start by looking at the number of arguments provided.

Inside every function in JavaScript there exists a contextual variable named arguments that acts as an array-like object containing all the, well, arguments passed into the function. The arguments object isn't a true array; it does not share a prototype with Array, and it does not have array-processing functions like push or indexOf. It does have positional array access (for example, arguments[2] returns the third argument), and there is a length property. There are two examples of this in Listing 2-13.

Listing 2-13. Two Examples of Function Overloading in JavaScript

```
// A simple function for sending a message
function sendMessage( msg, obj ) {
    // If both a message and an object are provided
    if ( arguments.length === 2 ) {
        // Send the message to the object
        // (Assumes that obj has a log property!)
        obj.log( msg );
    } else {
        // Otherwise, assume that only a message was provided
        // So just display the default error message
        console.log( msg );
    }
}

// Both of these function calls work
sendMessage( 'Hello, World!' );
sendMessage( 'How are you?', console );
```

You may wonder if there is a way to have the full functionality of an array available to the `arguments` object. It is not possible with `arguments` itself, but it is possible to create a copy of `arguments` that is an array. By invoking the `slice` method from the Array prototype, we can quickly copy the `arguments` object into an array, as in Listing 2-14.

Listing 2-14. Converting Arguments to an Array

```
function aFunction(x, y, z) {
    var argsArray = Array.prototype.slice.call( arguments, 0 );
    console.log( 'The last argument is: ' + argsArray.pop() );
}

// Will output 'The last argument is 3'.
aFunction( 1, 2, 3 );
```

We will learn more about the `prototype` property very soon. For the moment, suffice it to say that the prototype allows us to access object methods in a static manner.

What if the message were not defined? We need to be able to check not just for the presence of an argument, but also its absence. We can take advantage of the fact that any argument that isn't provided has a value of `undefined`. Listing 2-15 shows a simple function for displaying an error message and providing a default message if a particular argument is not provided. (Note that we must use `typeof` here, because otherwise, an argument with the literal string "undefined" would indicate an error.)

Listing 2-15. Displaying an Error Message and a Default Message

```
function displayError( msg ) {
    // Check and make sure that msg is not undefined
    if ( typeof msg === 'undefined' ) {
        // If it is, set a default message
        msg = 'An error occurred.';
    }
```

```
    // Display the message
    console.log( msg );
}

displayError();
```

The use of the typeof statement helps to lead us into the topic of type-checking. Because JavaScript is a dynamically typed language, this proves to be a very useful and important topic. There are a number of ways to check the type of a variable; we're going to look at two that are particularly useful.

The first way of checking the type of an object is by using the obvious-sounding typeof operator. This utility gives us a string name representing the type of the contents of a variable. An example of this method can be seen in Listing 2-16.

Listing 2-16. Example of Using typeof to Determine the Type of an Object

```
var num = '50';
var arr = 'apples,oranges,pears';

// Check to see if our number is actually a string
if ( typeof num === 'string' ) {
    // If it is, then parse a number out of it
    num = parseInt( num );
}

// Check to see if our array is actually a string
if ( typeof arr == 'string' ) {
    // If that's the case, make an array, splitting on commas
    arr = arr.split( ',' );
}
```

The advantage of typeof is that you do not have to know what the actual type of the tested variable is. This would be the perfect solution except that for variables of type Object or Array, or a custom object such as User, typeof only returns "object", making it hard to differentiate between specific object types. The next two ways to figure out the type of a variable require you to test against a specific existing type.

The second way to check the type of an object is to use the instanceof operator. This operator checks the left operand against the constructor of the right operand, which may sound a bit more complex than it actually is! Take a look at Listing 2-17, showing an example of using instanceof.

Listing 2-17. Example of Using instanceof

```
var today = new Date();
var re = /[a-z]+/i;

// These don't give us enough details
console.log('typeof today: ' + typeof today);
console.log('typeof re: ' + typeof re);

// Let's find out if the variables are of a more specific type
if (today instanceof Date) {
    console.log('today is an instance of a Date.');
}
```

```
if (re instanceof RegExp) {
    console.log( 're is an instance of a RegExp object.' );
}
```

In the next chapter, when we look at object-oriented JavaScript, we will discuss the `Object.isPrototypeOf()` function, which also helps in type determination.

Type-checking variables and verifying the length of argument arrays are simple concepts at heart but can be used to provide complex methods that can adapt and provide a better experience to both the developer and code users. When you need specific type-checking (is this an Array? Is it a Date? A specific type of custom object?), we advise creating a custom function for determining the type. Many frameworks have convenience functions for determining Arrays, Dates, and so on. Encapsulating this code into a function ensures that you have one and only one place to check for that specific type, instead of having checking code scattered throughout your codebase.

New Object Tools

One of the more exciting developments in JavaScript the language has been the expansion of tools for managing objects. As we will see, these tools can be used on object literals (which are more like data structures) and on object instances.

Objects

Objects are the foundation of JavaScript. Virtually everything within the language is an object. Much of the power of the language is derived from this fact. At their most basic level, objects exist as a collection of properties, almost like a hash construct that you see in other languages. Listing 2-18 shows two basic examples of the creation of an object with a set of properties.

Listing 2-18. Two Examples of Creating a Simple Object and Setting Properties

```
// Creates a new Object object and stores it in 'obj'
var obj = new Object();

// Set some properties of the object to different values
obj.val = 5;
obj.click = function(){
    console.log('hello');
};

// Here is some equivalent code, using the {...} shorthand
// along with key-value pairs for defining properties
var obj = {

    // Set the property names and values using key/value pairs
    val: 5,
    click: function(){
        console.log('hello');
    }
};
```

In reality there isn't much more to objects than that. Where things get tricky, however, is in the creation of new objects, especially ones that inherit the properties of other objects.

Modifying Objects

JavaScript now has three methods that can help you control whether an object can be modified. We will look at them on a scale of restrictiveness, from least to greatest.

An object in JavaScript by default can be modified at any time. By using `Object.preventExtensions()`, you can prevent new properties from being added to the object. When this happens, all current properties can be used but no new ones can be added. Trying to add a new property will result in a `TypeError`—or will fail silently; you are more likely to see the error when running in strict mode. Listing 2-19 shows an example.

Listing 2-19. An example of using `Object.preventExtensions()`

```
// Creates a new object and stores it in 'obj'
var obj = {};

// Creates a new Object object using preventExtensions
var obj2 = Object.preventExtensions(obj);

// Generates TypeError when trying to define a new property
function makeTypeError(){
'use strict';

//Generates TypeError when trying to define a new property
Object.defineProperty(obj2, 'greeting',{value: 'Hello World'});
}

makeTypeError();
```

Using `Object.seal()`, you can restrict the ability of an object, similar to what you did with `Object.preventExtensions()`. Unlike our previous example, however, properties cannot be deleted or converted into accessors (getter methods). Trying to delete or add properties will also result in a `TypeError`. Existing writable properties can be updated without resulting in an error. Listing 2-20 shows an example.

Listing 2-20. An example of using `Object.seal()`

```
// Creates a new object and uses object.seal to restrict it
var obj = {};
obj.greeting = 'Welcome';
Object.seal(obj);

//Updating the existing writable property
//Cannot convert existing property to accessor, throws TypeErrors
obj.greeting = 'Hello World';
Object.defineProperty(obj, 'greeting', {get:function(){return 'Hello World'; } });

// Cannot delete property, fails silently
delete obj.greeting;
```

```
function makeTypeError(){
  'use strict';

  //Generates TypeError when trying to delete a property
  delete obj.greeting;

  //Can still update property
  obj.greeting = 'Welcome';
  console.log(obj.greeting);
}

makeTypeError();
```

Object.freeze(), demonstrated in Listing 2-21, is the most restrictive of the three methods. Once it is used, an object is considered immutable. Properties cannot be added, deleted or updated. Any attempts will result in a TypeError. If a property is itself an object, that can be updated. This is called a *shallow freeze*. In order to make an object fully immutable, all properties whose values contain objects must also be frozen.

Listing 2-21. An example of using Object.freeze()

```
//Creates a new object with two properties. Second property is an object
var obj = {
  greeting: "Welcome",
  innerObj: {}
};

//Freeezes our obj
Object.freeze(obj);

//silently fails
obj.greeting = 'Hello World';

//innerObj can still be updated
obj.innerObj.greeting = 'Hello World';
console.log('obj.innerObj.greeting = ' + obj.innerObj.greeting);

//Cannot convert existing property to accessor
//Throws TypeError
Object.defineProperty(obj, 'greeting', {get:function(){return 'Hello World'; } });

// Cannot delete property, fails silently
delete obj.greeting;

function makeTypeError(){
  'use strict';
}

//Generates TypeError when trying to delete a property
delete obj.greeting;
```

```
//Freeze inner object
Object.freeze(obj.innerObj);

//innerObj is now frozen. Fails silently
obj.innerObj.greeting = 'Worked so far...';

function makeTypeError(){
     'use strict';
    //all attempts will throw TypeErrors

    delete obj.greeting;
    obj.innerObj.greeting = 'Worked so far...';
    obj.greeting = "Welcome";

};

makeTypeError();
```

By understanding how you can control the mutability of an object, you can create a level of consistency. For example, if you have an object named `User`, you can be sure that every new object based on that will have the same properties as the first. Any properties that could be added at runtime would fail.

Summary

The importance of understanding the concepts outlined in this chapter cannot be understated. The first half of the chapter, giving you a good understanding of how JavaScript behaves and how it can be best used, is the starting point for fully grasping how to use JavaScript professionally. Simply understanding how objects act, references are handled, and scope is decided can unquestionably change how you write JavaScript code.

Building on these skills, advanced techniques provide us with additional ways to solve problems with JavaScript. Understanding scope and context led to using closures. Looking into how to determine types in JavaScript allowed us to add function overloading to a language that doesn't have it as a native feature. And then we spent time with one of the foundational types in JavaScript: the Object. The various new features in the Object type allow us much greater control over the object literals we create. This will lead naturally into the next chapter, where we start building our own object-oriented JavaScript.

■ ■ ■

Creating Reusable Code

In the introduction to the last chapter, we discussed objects as the fundamental unit of JavaScript. Having addressed JavaScript object literals, we will use a large portion of this chapter to examine how those objects interact with object-oriented programming. Here, JavaScript exists in a state of tension between classical programming and JavaScript's own, nearly unique capabilities.

Moving outward from organizing our code into objects, we will look at other patterns for managing our code. We will want to ensure that we don't pollute the global namespace, or (overly) rely on global variables. That means we will start with a discussion of namespaces, but namespaces are only the tip of the iceberg, and some newer invocation patterns are available to help us properly fence in our code: modules and, later, immediately invoked function expressions (IIFEs or "iffies").

Once we have organized our code well within an individual file, it makes sense to look at the tools available for managing multiple JavaScript files. Certainly, we can rely on content delivery networks for some libraries we might use. But we should also think about the best way to load our own JavaScript files, lest we end up with HTML files that contain script tag after script tag after script tag.

Object-Oriented JavaScript

JavaScript is a prototypal language, not a classical language. Let's get that out of the way up front. Java is a classical language, as everything in Java requires a class. In JavaScript, on the other hand, everything has a prototype; thus it is prototypal. But it is, as Douglas Crockford and others have said, "conflicted" about its prototypal nature. Like some reluctant superhero, JavaScript sometimes doesn't want to stand out from the crowd of other programming languages and let its abilities shine. Well, let's give it a cape and see what happens!

First, let's reiterate, JavaScript is not a classical language. There are many books, blog posts, guides, slide decks, and libraries that will try to impose class-based language structures on JavaScript. You are welcome to examine them in great depth, but keep in mind that in doing so, despite good intentions, their authors are trying to hammer a square peg into a round hole. We are not trying to do that here. This chapter will not discuss how to make JavaScript act as if it were Java. Instead, we will focus on JavaScript's intersections with capabilities outlined in object-oriented theory, and how it sometimes falls short and at other times exceeds expectations.

Ultimately, why do we want to use object-oriented programming? It provides patterns of usage that allow for simplified code reuse, eliminating duplication of effort. Also, programming in an object-oriented style helps us to think more deeply about the code that we're working with. It provides an outline, a map, which we can follow to successful implementations. But it is not the only map. JavaScript's prototypes are a similar but distinct way to reach our destination.

Start with the prototype itself. Every type (an Object, a Function, a Date, and so on) in JavaScript has a prototype. The ECMAScript standard specifies that this property is hidden and is referred to as [[Prototype]]. Until now, you could access this property in one of two ways: the nonstandard __proto__ property and the prototype property. At first, exposing __proto__ was not reliably available across browsers, and even when available was not always implemented the same way. [Footnote: Shocking, that browsers would implement critical parts of JavaScript differently!] With ECMAScript 6 (coming soon to a browser near you!), __proto__ will become an official property of types and will be available to any conforming implementation. But the future is not yet now.

You can also access the prototype property of certain types. All of the core JavaScript types (Date, String, Array, and so on) have a public prototype property. And any JavaScript type that is created from a function constructor also has a public prototype property. But instances of those types, be they strings, dates, or whatever, do *not* have a prototype property. That is because the prototype property is unavailable on instances. We will not be using the prototype property here either, because we will not use functions as constructors. We will use objects as constructors.

That's right; we will use an object literal as the basis for other objects. If that sounds a lot like classes and instances, there are some similarities but, as you might expect, also some differences. Consider a Person object like that shown in Listing 3-1.

Listing 3-1. A Person Object

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date('1964-09-05'),
    gender: 'male',
    getAge : function() {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor(diff / year);
    }
};
```

Nothing remarkable here: a person has a first name, a last name, a gender, a birth date and a way to calculate their age. This Person is an object literal, not really anything we'd recognize as being class-like. But we want to use this Person as if it were a class. We want to create more objects that conform to the structure that Person has set forth. To preserve the distinction between classless JavaScript and object-oriented languages that have classes, we will refer to Person as a type (similar to the way Date, Array, and RegExp are all types). We want to create instances of the Person type: to do so, we can use Object.create (Listing 3-2).

Listing 3-2. Creating People

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },
```

```
    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    }
};

var bob = Object.create( Person );
bob.firstName = 'Bob';
bob.lastName = 'Sabatelli';
bob.birthDate = new Date( '1969-06-07' );
console.log( bob.toString() );
```

An instance has been created from the Person object. We are storing an instance of Person in the variable bob. No classes. But there is a link between the Person objects we created and the Person type. This link is over the [[Prototype]] property. If you are running a sufficiently modern browser (at the time of this writing, this worked in IE11, Firefox 27, and Chrome 33), you can open the console in the developer tools and look at the __proto__ property on bob. You'll note that it points to the Person object. In fact, you can test this by checking that bob.__proto__ === Person.

Object.create was added to JavaScript with ECMAScript 5, ostensibly to simplify and clarify the relationship between objects, particularly which objects were related by their prototype. But in doing so, it allowed for a simple, one-step creation of that relationship between objects. This relationship feels very much like the object-oriented idea of the class and the instance. But because JavaScript has no classes, we simply have objects with a relationship between each other.

This relationship is often referred to as the prototype chain. In JavaScript, the prototype chain is one of two places that are examined to resolve the value of a member of an object. That is, when you refer to foo.bar or foo[bar], the JavaScript engine looks up the value of bar in two potential places: on foo itself, or on foo's prototype chain.

In his three-part essay on object-oriented JavaScript (http://davidwalsh.name/javascript-objects), Kyle Simpson makes an elegant point about how we should look at this process. Instead of seeing bob's relationship to Person as that of an instance to a class, or a child to a parent, we should see it as a case of behavior delegation. The bob object has its own firstName and lastName, but it does not have any getAge functionality. That is delegated to Person. The delegate relationship is established through the use of Object.create. The prototype chain is the mechanism of this delegation, allowing us to delegate behavior to something further along the chain. Viewed from bob's perspective, functionality accumulates as we successively invoke Object.create, layering on additional capabilities.

By the way, you might be concerned that you have a browser that doesn't support ECMAScript 5 or at least doesn't have its version of Object.create. This isn't a problem; Object.create can be polyfilled quite easily across any browser with a JavaScript engine, as shown in Listing 3-3.

Listing 3-3. An Object.create Polyfill

```
if ( typeof Object.create !== 'function' ) {
    Object.create = function ( o ) {
        function F() {
        }
        F.prototype = o;
        return new F();
    };
}
```

Finally, some people don't like the idea of constantly using `Object.create` to, well, create objects. They feel more at home with the typical phrasing of `someInstance = new Type();` If that's the case, consider the quick modification to the Person object in Listing 3-4, which provides a factory method for generating more Persons.

Listing 3-4. The Person Object with a Factory Method

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },

    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    },

    extend : function ( config ) {
        var tmp = Object.create( this );
        for ( var key in config ) {
            if ( config.hasOwnProperty( key ) ) {
                tmp[key] = config[key];
            }
        }
        return tmp;
    }
};

var bob = Person.extend( {
    firstName : 'Bob',
    lastName : 'Sabatelli',
    birthDate : new Date( '1969-06-07' )
} );

console.log( bob.toString() );
```

Here, the extend function encapsulates the call to `Object.create`. When extend is called, it invokes `Object.create` internally. Presumably, extend is invoked with a configuration object passed in, a fairly typical JavaScript usage pattern. By looping over the properties in tmp, the extend function also ensures that only the properties of the config already present on the tmp object are extended onto the newly created tmp object. Once we've copied the properties from config to tmp, we can return tmp, our instance of a Person.

Now that we've looked at the new style of setting up relationships between objects in JavaScript, let us see how it affects JavaScript's interactions with typical object-oriented concepts.

Inheritance

By far, the biggest question mark has to be inheritance. Much of the point of object-oriented code is to reuse functionality by working from general parent classes to more specific child classes. We have already seen that it is easy to create a relationship between two objects with Object.create. We can simply extend that usage to create whatever sort of inheritance hierarchy we prefer. (OK, whatever sort of single-inheritance hierarchy we prefer. Object.create does not allow multiple inheritance.) Remember the idea that we are delegating behavior; as we create subclasses with Object.create, they are delegating some of their behavior to types further up the prototype chain. Inheritance with Object.create tends to be more of a bottom-up affair, rather than the typically top-down object oriented style.

Inheritance is actually quite simple: use Object.create. To elaborate, use Object.create to create a relationship between the "parent" type and the "child" type. The child type can add functionality, delete functionality, or override existing functionality. Call Object.create with an argument of whatever object you decide is your "parent" type, and the returned value will be whatever you decide your "child" type is. Then repeat the pattern from Listing 3-4 and use the extend method (or reuse Object.create!) to create instances of that child type (Listing 3-5).

Listing 3-5. Person Is the Parent of Teacher

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },

    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    },

    extend : function ( config ) {
        var tmp = Object.create( this );
        for ( var key in config ) {
            if ( config.hasOwnProperty( key ) ) {
                tmp[key] = config[key];
            }
        }
        return tmp;
    }
};
```

```
var Teacher = Person.extend( {
    job : 'teacher',
    subject : 'English Literature',
    yearsExp : 5,
    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender + ' ' + this.subject + ' teacher.';
    }
} );

var patty = Teacher.extend( {
    firstName : 'Patricia',
    lastName : 'Hannon',
    subject: 'chemistry',
    yearsExp : 20,
    gender : 'female'
} );

console.log( patty.toString() );
```

Object.create established a link between the [[Prototype]] of Teacher and the [[Prototype]] of Person. If you have one of the modern browsers mentioned earlier, you should be able to look at the __proto__ property of Teacher and see that it points to Person.

In Chapter 2, we talked about instanceof as a way to find out whether an object is an instance of a type. The instanceof operator will not work here. It relies on the explicit prototype property to trace the relationship of an object to a type. Put more simply, the right-hand operand of instanceof must be a function (though most likely a function constructor). The left-hand operand must be something that was created from a function constructor (though not necessarily the function constructor on the right). So how can we tell if an object is an instance of a type? Enter the isPrototypeOf function.

The isPrototypeOf function can be invoked on any object. It is present on all JavaScript objects, much like toString. Invoke it on the object that is fulfilling the role of the type (Person or Teacher, in our examples so far) and pass it an argument of the object that is fulfilling the role of an instance (bob or patty). Therefore, Teacher.isPrototypeOf(patty) will return true, as you would expect. Listing 3-6 provides the code that looks at combinations of Teachers, Persons, bob, and patty and invocations of isPrototypeOf.

Listing 3-6. The isPrototypeOf() Function

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },
```

```
    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    },

    extend : function ( config ) {
        var tmp = Object.create( this );
        for ( var key in config ) {
            if ( config.hasOwnProperty( key ) ) {
                tmp[key] = config[key];
            }
        }
        return tmp;
    }
};

var Teacher = Person.extend( {
    job : 'teacher',
    subject : 'English Literature',
    yearsExp : 5,
    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender + ' ' + this.subject + ' teacher.';
    }
} );

var bob = Person.extend( {
    firstName : 'Bob',
    lastName : 'Sabatelli',
    birthDate : new Date( '1969-06-07' )
} );

var patty = Teacher.extend( {
    firstName : 'Patricia',
    lastName : 'Hannon',
    subject: 'chemistry',
    yearsExp : 20,
    gender : 'female'
} );

console.log( 'Is bob an instance of Person? ' + Person.isPrototypeOf(bob) );       // true
console.log( 'Is bob an instance of Teacher? ' + Teacher.isPrototypeOf( bob ) );     // false
console.log( 'Is patty an instance of Teacher? ' + Teacher.isPrototypeOf( patty ) ); // true
console.log( 'Is patty an instance of Person? ' + Person.isPrototypeOf( patty ) );   // true
```

There is a companion function to isPrototypeOf; it's named getPrototypeOf. Called as
Object.getPrototypeOf(obj), it returns a reference to the type that was the basis for the current object.
As noted, you can also look at the (currently nonstandard but soon to be standard) __proto__ property for
the same information (Listing 3-7).

Listing 3-7. getPrototypeOf

```
console.log( 'The prototype of bob is Person' + Object.getPrototypeOf( bob ) );
```

What about accessing overridden methods? It is, of course, possible to override a method from the parent object in the child object. There is nothing special about this capability, and it's expected in any object-oriented system. But in most object-oriented systems, an overridden method has access to the parent method via a property or accessor called something like super. That is, when you are overriding a method, you can usually call the method you are overriding via a special keyword.

We do not have that available here. JavaScript's prototype-based object-oriented code simply does not have a super() feature. There are, generally, three ways to solve this problem. First, you could write some code to reimplement super. This would involve traversing back up the prototype chain, probably with getPrototypeOf, to find the object in the inheritance chain that had the previous edition of the method you're overriding. (Remember, you aren't always overriding something in the parent; it could be something from the "grandparent" class, or something further up the prototype chain.) Then you would need some way to access that method and call it with the same set of arguments passed to your overriding method. This is certainly possible, but it tends to be ugly and quite inefficient at the same time.

As a second solution, you could explicitly call the parent's method as shown in Listing 3-8.

Listing 3-8. Reproducing the Effect of the super Function

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },

    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    },

    extend : function ( config ) {
        var tmp = Object.create( this );
        for ( var key in config ) {
            if ( config.hasOwnProperty( key ) ) {
                tmp[key] = config[key];
            }
        }
        return tmp;
    }
};
```

```
var Teacher = Person.extend( {
    job : 'teacher',
    subject : 'English Literature',
    yearsExp : 5,
    toString : function () {
        var originalStr = Person.toString.call(this);
        return originalStr + ' ' + this.subject + ' teacher.';
    }
} );

var patty = Teacher.extend( {
    firstName : 'Patricia',
    lastName : 'Hannon',
    subject: 'chemistry',
    yearsExp : 20,
    gender : 'female'
} );

console.log( patty.toString() );
```

Pay particular attention to the `toString` method in Teacher. You will note that Teacher's `toString` function makes an explicit call to Person's `toString` function. Many object-oriented designers would argue that we should not have to hard-code the relationship between Person and Teacher. But as a simple means to an end, doing so does solve the problem quickly, neatly, and efficiently. On the other hand, it's not portable. This approach will only work for objects that are somehow related to the Parent object.

The third possibility is that we could simply not worry about whether we have `super` at all. Yes, JavaScript the language lacks the `super` feature, which is present in many other object-oriented languages. But that feature is not the be-all, end-all of object-oriented code. Perhaps in the future, JavaScript will have a `super` keyword with the appropriate functionality. (Actually, it is known that in ECMAScript 6, there is a `super` property for objects.) But for now, we can get along quite well without it.

Member Visibility

In object-oriented code, we often want to control the visibility of our objects' data. Most of our members, whether functions or properties, are public, in keeping with JavaScript's implementation. But what if we need private functions or private properties? JavaScript does not have easy, straightforward visibility modifiers (like "private" or "protected" or "public") that control who can access a member of a property. But you can have the effect of private members. Further, you can provide special access to those private members through what Douglas Crockford calls a privileged function.

Recall that JavaScript has only two scopes: global scope and the scope of the currently executing function. We took advantage of this in the previous chapter with closures, a critical part of implementing privileged access to private members. It works this way: create private members using `var` inside the function that builds your object. (Whether those private members are functions or properties is up to you.) In the same scope, create a function; it will have implied access to the private data, because both the function and the private data belong to that same scope. Add this new function to the object itself, making the function (but not the private data) public. Because the function comes from the same scope, it can still access that data indirectly. Look at Listing 3-9 for details.

Listing 3-9. Private Members

```
var Person = {
    firstName : 'John',
    lastName : 'Connolly',
    birthDate : new Date( '1964-09-05' ),
    gender : 'male',
    getAge : function () {
        var today = new Date();
        var diff = today.getTime() - this.birthDate.getTime();
        var year = 1000 * 60 * 60 * 24 * 365.25;
        return Math.floor( diff / year );
    },

    toString : function () {
        return this.firstName + ' ' + this.lastName + ' is a ' + this.getAge() +
            ' year-old ' + this.gender;
    },

    extend : function ( config ) {
        var tmp = Object.create( this );

        for ( var key in config ) {
            if ( config.hasOwnProperty( key ) ) {
                tmp[key] = config[key];
            }
        }

        // When was this object created?
        var creationTime = new Date();

        // An accessor, at the moment, it's private
        var getCreationTime = function() {
            return creationTime;
        };

        tmp.getCreationTime = getCreationTime;
        return tmp;
    }
};

var Teacher = Person.extend( {
    job : 'teacher',
    subject : 'English Literature',
    yearsExp : 5,
    toString : function () {
        var originalStr = Person.toString.call(this);
        return originalStr + ' ' + this.subject + ' teacher.';
    }
} );
```

```
var patty = Teacher.extend( {
    firstName : 'Patricia',
    lastName : 'Hannon',
    subject: 'chemistry',
    yearsExp : 20,
    gender : 'female'
} );

console.log( patty.toString() );
console.log( 'The Teacher object was created at %s', patty.getCreationTime() );
```

As you can see, the `creationTime` variable is local to the `extend` function. It is not available outside that function. If you were to examine Person on the console with, say, `console.dir`, you would not see `creationTime` listed as a public property of Person. Initially, the same is true for `getCreationTime`. It is a function that was created at the same scope as `creationTime`, so the function has access to `creationTime`. Using simple assignment, we attach `getCreationTime` to the object instance we are returning. Now, `getCreationTime` is a public method, with privileged access to the private data in `creationTime`.

A minor caveat: this is not the most efficient of patterns. Every time you create an instance of Person, or any of its child types, you will be creating a brand-new function with access to the execution context of the call to `extend` that created the instance of Person. By contrast, when we use `Object.create`, our public functions are references to those on the type we pass into `Object.create`. Privileged functions are not particularly inefficient at the small scale we are dealing with here. But if you added more privileged methods, they would each retain a reference to that execution context, and each would be its own instance of that privileged method. The memory costs can multiply quickly. Use privileged methods sparingly, reserving them for data that needs strict access control. Otherwise, become comfortable with the notion that most data in JavaScript is public anyway.

The Future of Object-Oriented JavaScript

We would be remiss in overlooking the fact that there are some changes coming to object-oriented JavaScript with ECMAScript 6. The most important of these changes is the introduction of a working `class` keyword. The `class` keyword will be used to define JavaScript types (not classes, as JavaScript still won't have classes!). It will also include provisos for the use of the keyword `extends` to create an inheritance relationship. Finally, when overriding functions in a child type, ECMAScript 6 sets aside the `super` keyword to refer to the version of the function on the prototype chain.

All of this is syntactic sugar, though. When these structures are desugared by the JavaScript engine, they are revealed to be uses of functional constructors. These new features do not actually establish new functionality: they simply introduce an idiom more palatable to programmers from other object-oriented languages. Worse, they continue to obscure some of the best features of JavaScript by trying to have it conform to these other languages' notions of what a "true" object-oriented language should look like. It appears that sometimes, JavaScript is still a little shy about putting on the cape and tights before using its powers for good.

Packaging JavaScript

Moving outward from object-oriented JavaScript, we should consider how to organize our code for broad reuse. We want a set of tools for properly encapsulating our code, preventing accidental use of the global context, as well as ways to make our code reusable and redistributable. Let's tackle the various requirements in order.

Namespaces

So far, we have declared our types (and earlier, our functions and variables) to be part of the global context. We have not done this explicitly, but by virtue of the fact that we have not declared these objects and variables to be part of any other context. We would like to encapsulate functions, variables, objects, types, and so on into a separate context, so as not to rely on the global context. To do so, we will rely (initially) on namespaces.

Namespaces are not unique to JavaScript, but, as is the case with so many things in JavaScript, they are a little different from what you might expect. A namespace provides a context for variables and functions. The namespace itself is likely to be global, of course. This is a lesser-of-two-evils approach. Instead of having numerous variables and functions belonging to the window, we can have one variable belong to the window, and then a variety of data and functionality belong to that one variable. The implementation is simple: use an object literal to encapsulate the code that you want to hide from the global context (Listing 3-10).

Listing 3-10. Namespaces

```
// Namespaces example

var FOO = {};

// Add a variable
FOO.x = 10;

// Add a function
FOO.addEmUp = function(x, y) {
    return x + y;
};
```

Namespaces are best used as ad-hoc solutions to the encapsulation of otherwise unaffiliated code. If we try to use namespaces for all our code, they can quickly become unwieldy, as they accrete more and more functionality and data. You might be tempted to set up namespaces within namespaces, emulating something of the way packages work with Java. The Ext JS library uses this technique well, for what it's worth. But they also have spent a lot of time thinking about how to organize their functionality, what code belongs to what namespace or sub-namespace, and so on. There are trade-offs with extensive use of namespaces.

Also, namespace names are hard-coded: FOO in the example, Ext in the case of the aforementioned library Ext JS, YAHOO in the case of the similarly popular YUI library. These namespaces are effectively reserved words for those libraries. What happens if two or more libraries settle on the same namespace (as with jQuery's use of $ as a namespace)? Potential conflicts. JQuery has added explicit code to deal with this possibility, should it arrive. Although this issue is potentially less likely with your own code, it is a possibility that has to be considered. This is especially true in a team environment where multiple programmers have access to the namespace, raising the possibility of accidentally overwriting or deleting another coder's namespace.

The Module Pattern

We have some tools for improving the way we use namespaces. We can work with the module pattern, which encapsulates generation of the namespace within a function. This allows for a variety of improvements, including establishing a baseline for what functions and data the namespace contains, use of private variables within the generator function, which might make implementation of some functionality easier, and simply having a function generate the namespace, which means that we can have JavaScript dynamically generate part or all of the namespace at runtime instead of at compile-time.

Modules can be as simple or as complex as you prefer. Listing 3-11 provides a very simple example of creating a module.

Listing 3-11. Creating a Module

```
function getModule() {
    // Namespaces example
    var FOO = {};

    // Add a variable
    FOO.x = 10;

    // Add a function
    FOO.addEmUp = function ( x, y ) {
        return x + y;
    };

    return FOO;
}

var myNamespace = getModule();
```

We have encapsulated our namespace code inside a function. Thus, when we initially set up the FOO object, it is private to the getModule function. We can then return FOO to anyone who invokes getModule, and they can use the encapsulated structure as they see fit, including naming it whatever they want.

Another advantage to this pattern is that we can once again utilize our friend the closure to set up data that is private only to the namespace. If our namespace, our encapsulated code, needs to have internal data or internal functions, we can add them without worrying about making them public (Listing 3-12).

Listing 3-12. Modules with Private Data

```
function getModule() {
    // Namespaces example
    var FOO = {};

    // Add a variable
    FOO.x = 10;

    // Add a function
    FOO.addEmUp = function ( x, y ) {
        return x + y;
    };

    // A private variable
    var events = [];

    FOO.addEvent = function(eventName, target, fn) {
        events.push({eventName: eventName, target: target, fn: fn});
    };
```

35

```
    FOO.listEvents = function(eventName) {
        return events.filter(function(evtObj) {
            return evtObj.eventName === eventName
        });
    };

    return FOO;
}

var myNamespace = getModule();
```

In this example, we have implemented a public interface for adding some sort of event tracking with `addEvents`. Later, we might want to get back event references by their names via `listEvents`. But the actual `events` collection is private, managed by the public API we provide to it, but hidden from direct access.

Modules, like namespaces, have the same problem of being a lesser-of-two-evils approach. We have traded a global variable for our namespace for a global function `getModule`. Wouldn't it be nice if we could have full control over what winds up in the global namespace, without necessarily using globally scoped objects or functions to do so? Luckily, we are about to see a tool that can help us do exactly that.

Immediately Invoked Function Expressions

If we want to avoid polluting the global namespace, functions are the logical solution. Functions create their own execution context when they are running, which is subordinate to but insulated from the global namespace. When the function finishes running, the execution context is available for garbage collection and the resources dedicated to it can be reclaimed. But all of our functions have been either global or part of a namespace, which is itself global. We would like to have a function that can immediately execute, without having to be named and without having to be part of a namespace or context, global or otherwise. Then, within that function, we could build the module that we need. We could return such an object, export it, and make it otherwise available, but we would not have to have a public function around to take up resources generating it. This is the idea behind the immediately invoked function expression (IIFE).

All of the functions we have worked with to this point have been *function declarations*. Whether we define them as `function funcName { ... }` or `var funcName = function() { ... }`, we are declaring functions, reserving their usage for later. Can we instead create a *function expression*, which would be a function that is created and executed in one fell swoop? The answer is yes, but doing so will require a degree of syntactical intrepidity.

How do we execute functions? Typically, with a named function, we print the name of the function, and then append some parentheses afterwards, indicating we want to execute the code associated with that name. We cannot do the same with a function definition, in and of itself. The result would be a SyntaxError, obviously not what we want.

But we can put the function declaration inside a set of parentheses, a hint to the parser that this is not a statement but an expression. Parentheses cannot contain statements, but only code to be evaluated as an expression, which is what we want out of our function. We need one more bit of syntax to make this work, another set of parentheses, usually at the end of the function declaration itself. Listing 3-13 will illuminate the full syntax.

Listing 3-13. An Immediately Invoked Function Expression

```
// A regular function
function foo() {
    console.log( 'Called foo!' );
}
```

```
// Function assignment
var bar = function () {
    console.log( 'Called bar!' );
};

// Function expression
(function () {
    console.log( 'This function was invoked immediately!' )
})();

// Alternate syntax
(function () {
    console.log( 'This function was ALSO invoked immediately!' )
}());
```

Compare and contrast the first two functions, which are function declarations, with the latter two, which are function expressions. The expressions are wrapped in parentheses to "expressionize" them (or, if you prefer: "de-declarify" them) and then use a second set of parentheses to invoke the expression. Nifty!

As an aside, there are a variety of JavaScript syntactical particles that will result in IIFEs: functions as components of a logical evaluation, unary operators prefixed to a function declaration, and so on. "Cowboy" Ben Alman's article on IIFEs (`http://benalman.com/news/2010/11/immediately-invoked-function-expression/`) contains terrific detail on valid syntaxes and goes deep into the guts of how IIFEs work and how they came to be.

Now that we know how to create an IIFE, how do we use it? There are many applications of IIFEs, but the one we're concerned with here is the generation of a module. Can we capture the result of an IIFE into a variable? Of course! So we can wrap our module generator in an IIFE and have it return the module (Listing 3-14).

Listing 3-14. An IIFE Module Generator

```
var myModule = (function () {
    // A private variable
    var events = [];

    return {
        x : 10,
        addEmUp : function ( x, y ) {
            return x + y;
        },
        addEvent : function ( eventName, target, fn ) {
            events.push( {eventName : eventName, target : target, fn : fn} );
        },
        listEvents : function ( eventName ) {
            return events.filter( function ( evtObj ) {
                return evtObj.eventName === eventName
            } );
        }
    };

})();
```

We have changed a few things in this last example. First, and simplest, we are now capturing the output of our factory IIFE in `myModule` instead of `myNamespace`. Second, instead of creating an object and then returning it, we are returning the object directly. This simplifies our code, cutting down on reserving a space for an object we ultimately never use.

The IIFE pattern opens up many new possibilities, including the use of libraries or other tools as needed. The parentheses at the end of our function expression are the same parentheses we expect on a regular function invocation. Therefore, we can pass arguments into our IIFE and use them within. Imagine an IIFE that had access to jQuery functionality (Listing 3-15).

Listing 3-15. Passing Arguments to an IIFE

```
// Here, the $ refers to jQuery and jQuery only for the entire
// scope of the module
var myModule = (function ($) {
    // A private variable
    var events = [];

    return {
        x : 10,
        addEmUp : function ( x, y ) {
            return x + y;
        },
        addEvent : function ( eventName, target, fn ) {
            events.push( {eventName : eventName, target : target, fn : fn} );
            $( target ).on( eventName, fn );
        },
        listEvents : function ( eventName ) {
            return events.filter( function ( evtObj ) {
                return evtObj.eventName === eventName
            } );
        }
    };

})(jQuery); // Assumes that we had included jQuery earlier
```

We pass jQuery into our IIFE, and then refer to it as $ throughout the IIFE. Internally, it's used within the `addEvent` function to add an event handler to the DOM. (Don't worry if the syntax does not make sense; it isn't the core of the example!)

Based on this code, you can probably imagine a system where modules generated by IIFEs talk to each other, passing arguments back and forth and using libraries, all without necessarily interacting at the global level. In fact, that is part of what the next chapter is about.

Summary

The problem before us at the start of this chapter was one of code management. How can we write code in such a way as to follow good object-oriented guidelines, and how can we encapsulate that code for reusability? In the former case, we concentrated on JavaScript's prototypal nature, using it to generate something similar to classes and instances, but with a unique JavaScript spin on it. And the implementation was a lot simpler than attempting to force JavaScript to act like C# or Java. For the latter requirement, we worked our way through a variety of solutions that enable us to encapsulate our code: namespaces, modules, and immediately invoked function expressions. Ultimately, a combination of all three provided us with the best-case solution for least use of the global context.

■ ■ ■

Debugging JavaScript Code

Sometimes it's not the writing of code, but the management of it that gets to us, that drives us up a wall and back to our favorite video game. Why does it work on this machine, not that one? What do you mean, double-equals (==) is bad and triple-equals (===) is good? Why is running tests such a hassle? How should I package this code for distribution? We are plagued by questions, distracted by questions that do not directly bear on the code we are writing.

Of course, we should not ignore these issues. We want to write code of the highest quality, and when we fall short, we want access to easy-to-use debugging tools. We want good test coverage, both for now and for future refactorings. And we should think about how our code will be distributed down the line. That is what this chapter is all about.

We will start by looking at how to solve problems with our code. We would love to be perfect programmers, writing everything correctly the first time. But we all know that does not happen in the real world. So let's start with debugging tools.

Debugging Tools

All of the modern browsers have some form of developer's toolkit. Even benighted Internet Explorer 8 had a rudimentary debugger, although you needed administrator access to install it. What we have now is a far cry from the days of development with various alert() statements or the occasional logging to a DOM element as our only recourse.

In general, a developer's toolkit will have the following utilities:

- The console: A combination JavaScript scratch pad and logging location for our applications.

- A debugger: The tool that eluded JavaScript developers for so long.

- A DOM inspector: Much of our work concentrates on manipulating the DOM, and right-clicking to choose View Source won't cut it. The inspector should reflect the current state of the DOM (not the original source). Most DOM inspectors go with a tree-based view, with an option to select a DOM element by clicking it in either the inspector or the page itself.

- A network analyzer: Show me what files were requested, which files were actually found, and how long it took to download them.

- A profiler: These are often somewhat crude, but they're better than wrapping a call in a pair of calls to new Date().getTime().

There are also extensions that can be added to browsers to give you extra debugging capability that goes beyond what is built into the browser. For example, Postman (http://getpostman.com) is an extension for Chrome that will let you create any HTTP request and see what the response is. Another popular extension is Firebug (http://getfirebug.com), an open source project that adds all the developer tools to Firefox and can also have its own set of extensions.

In this chapter we will refer to the common set of tools as the developer's tools or the developer's toolkit, unless discussing a specific browser's toolset.

The Console

The console is where we spend a lot of our time as developers. The console interface was modeled after the familiar logging levels on most applications: debug, info, warn, error, and log. Often, we first encounter it as a replacement for alert() statements in our code, especially when debugging. On some older versions of IE, only log is supported, but as of IE 11, all five functions are supported. Additionally, the console has a dir() function, which will give you a recursive, tree-based interface to an object. On the off-chance that the console is not present on your platform of choice, try Listing 4-1 as a polyfill.

Listing 4-1. A Console Polyfill

```
if (!window.console) {
  window.console = {
    log : alert
  }
}
```

(Obviously, this is only a polyfill for the log function. Were you to use others, you would have to add them individually.)

The output of the various levels varies little. On Chrome or Firefox, console.error includes an automatic stack trace. The other browsers (and native Firefox) simply add an icon and change the text color to differentiate the various levels. Perhaps the main reason to use the various function levels is that they can be filtered out on all three major browsers. Listing 4-2 provides some test code, followed by screen shots from each of the major browsers: Chrome, Firefox, and Internet Explorer (Figures 4-1 through 4-3).

Listing 4-2. Console Levels

```
console.log( 'A basic log message.' );
console.debug( 'Debug level.' );
console.info( 'Info level.' );
console.warn( 'Warn level.' );
console.error( 'Error level (possibly with a stacktrace).' );

var person = {
  name : 'John Connelly',
  age : 56,
  title : 'Teacher',
  toString: function() {
    return this.name + ' is a ' + this.age + '-year-old ' + this.title + '.';
  }
};
```

```
console.log( 'A person: ' );
console.dir( person );

console.log( 'Person object (implicit call to toString()): ' + person );
console.log( 'Person object as argument, similar to console.dir: ', person );
```
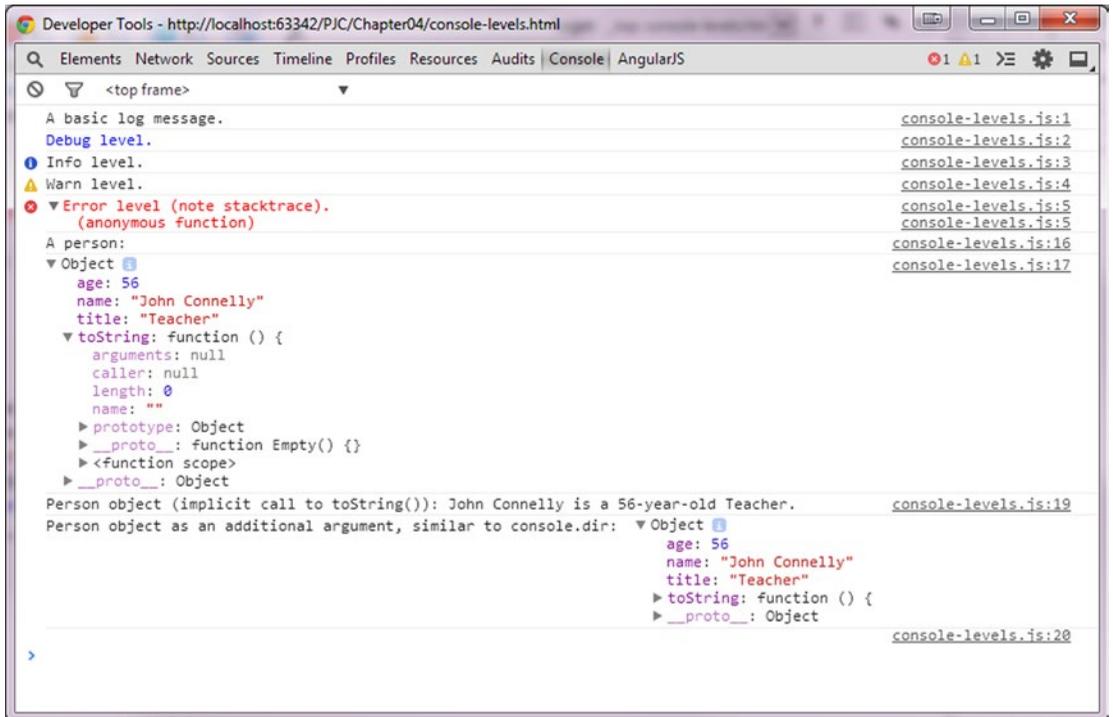

Figure 4-1. *Test code viewed in Chrome 40.0*
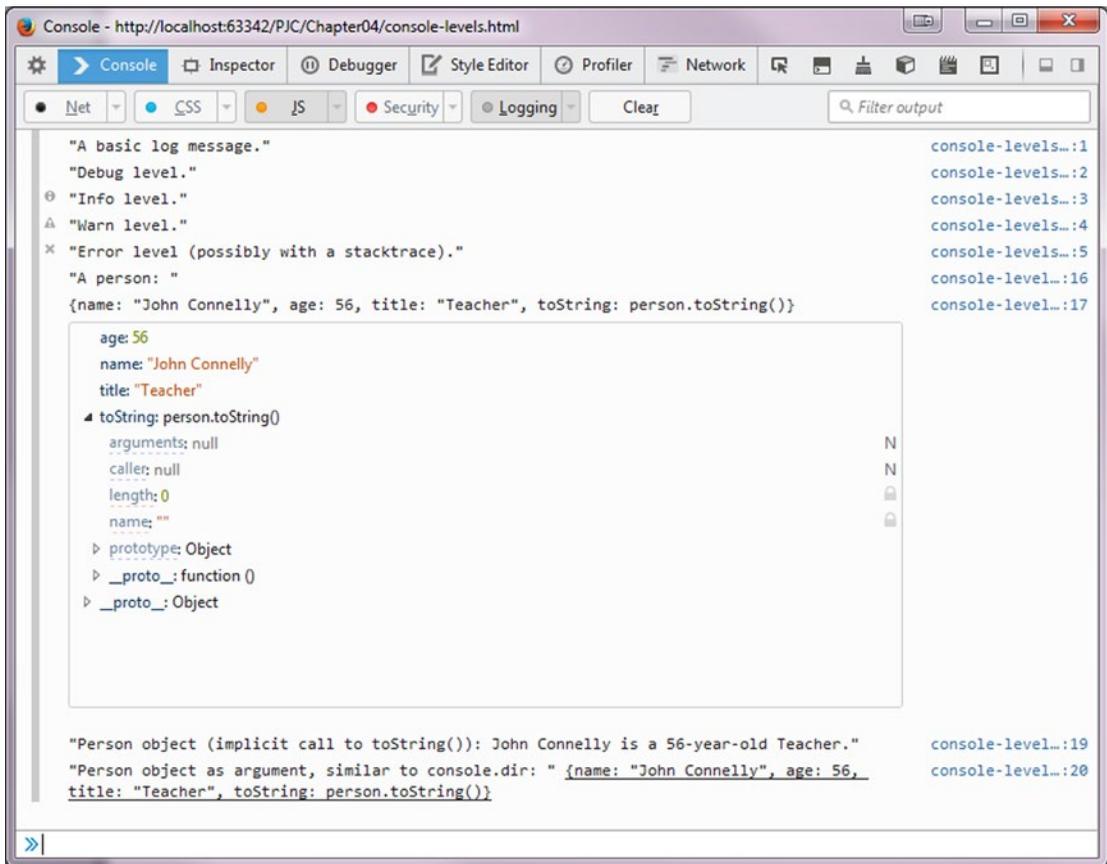
Figure 4-2. *Test code viewed in Firefox 35.0.1*
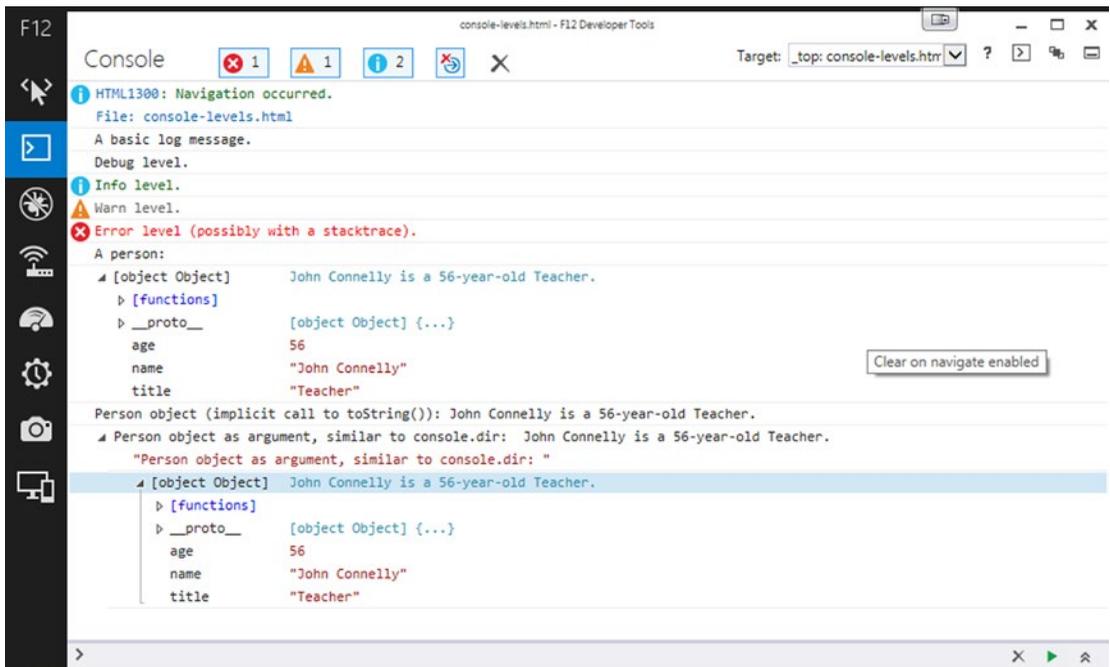
Figure 4-3. *Test code viewed in Internet Explorer 11.0*

Leveraging the Console Features

So what's the best way to use these console functions? As with many features of JavaScript, the key is consistency. You and your team should agree on usage patterns, keeping a few things in mind: First and most important is that all of your console statements should be removed by the time you deploy your code for the world to see. There is no need for production code to include console statements, and it is trivially easy to remove invocations of console functions (as you will see later in this chapter). Also remember that debugging, which we will look at soon, can replace logging for one-off needs. In general, use console logging for information about the state of an application: Has it started? Could it find data? What do various complicated objects look like? And so on. Your logging will give you a chronicle of the life of the application, a view into the application's changing state. If your application is a highway, good logging acts as a sort of mile marker—an indication of progress and a general indicator of where to start searching when problems inevitably arise.

The console is also much more than a logging utility. It is a JavaScript scratch pad. Consoles start in single-line mode, where you can enter JavaScript line-by line. Should you want to enter multiple lines of code, you can switch to multiline mode (enabled via icons in Firefox and IE; in Chrome, simply terminate your lines with Shift+Enter). In single-line mode, you can enter various JavaScript statements, enjoying auto-complete, by hitting either Tab or the right-arrow key. The console also includes a simple history, through which you can move backward and forward with the up- and down-arrow keys. The console maintains state, so variables defined on a previous line (or run of the multiline mode) hang around until you reload the page.

This last feature bears further examination. The console has the entire current state of the JavaScript interpreter available to it. This is incredibly powerful. Have you loaded up jQuery? Then you can enter commands according to its API. Want to check the state of a variable at the end of the page? Or maybe you need to look at what's going on with a particular animation? The console is your friend here. You can call functions, examine variables, manipulate the DOM, and so on. Think of any commands you enter as being added to the just-completed script and having access to all of its state.

The console also has an extended command-line API. Originally created by the fine folks at Firebug, elements of it have been ported to other browsers as well. It is now supported by Chrome and native Firefox, but not by Internet Explorer. There are numerous useful applications of this API, and we wholeheartedly recommend checking out the details at https://getfirebug.com/wiki/index.php/Command_Line_API. Here are a few of the highlights:

- `debug(`*`functionName`*`)`: When *functionName* is invoked, the debugger will automatically start before the first line of code in the function.

- `undebug(`*`functionName`*`)`: Stops debugging the named function.

- `include(`*`url`*`)`: Pulls a remote script into the page. Very handy if you want to pull in another debugging library, or something that manipulates the DOM differently, or what-have-you.

- `monitor(`*`functionName`*`)`: Turns on logging for all calls to the named function; does not affect `console.*` calls, but rather inserts a custom call to `console.log` for each invocation of the function. This logs the function name, its arguments and their values.

- `unmonitor(`*`functionName`*`)`: Turns off logging enabled via `monitor()` for all calls to the function.

- `profile([`*`title`*`])`: Turns on the JavaScript profiler; you can pass in an optional title for this profile.

- `profileEnd()`: Ends the currently running profile and prints a report, possibly with the title specified in the call to profile.

- `getEventListeners(element)`: Gets the event listeners for the provided element.

Thanks to the console, we developers have a full-featured tool for interacting with our code. We can record snapshots of the state of an application, and we can interact with it once it has completed loading. The console will also figure prominently in our next tool, the debugger.

The Debugger

For years, one of the knocks against JavaScript was that it couldn't be a "real" language because it lacked tools like a debugger. Fast-forward to now, and a debugger is standard equipment with all of the developer toolkits. All current browsers have a developer tools that lets you inspect your application and debug your work. Let's look at how these tools work, starting with the debugger.

The idea behind the debugger is simple: as a developer, you need to pause the execution of your application and examine its current state. Although we could accomplish the latter part with judiciously applied `console.log` statements, we cannot take care of the former without a debugger. Once we have paused our application, there are a few tools we need access to. We need a way to tell the debugger to activate. Within the code itself, we can add the simple statement `debugger;` to activate the debugger at that line. As mentioned earlier, we could also invoke the `debug` command from the console, passing it the name of a function that, when invoked, will start up the debugger. But the easiest way to pick when the debugger starts is to set a breakpoint.

Breakpoints allow us to run the JavaScript code up to a certain point, and then freeze the application there. When we hit the breakpoint we can then start to understand the current state of the application. From here we can see the content of variables, the scope of these variables, and so on. Also, we have a navigation menu, which includes at least four options: step into the current function (going a layer deeper into the stack), step out of the current function (running the current stack frame to completion and resuming debugging at the point the frame returns to), step over the current function (no need to dive into the function in the first place) and resume execution (run until completion or the next breakpoint).

DOM Inspector

Many JavaScript applications make extensive changes to the state of the DOM—changes so extensive, in fact, that it is often useless to refer to the actual HTML source code mere moments after loading a page. The DOM inspector reflects the current state of the DOM (instead of the state of the DOM when the page was loaded). It should dynamically and instantly update whenever there are changes made to the DOM. Developer tools have included a DOM inspector as a standard feature.

Network Analyzer

Since the previous edition of this book, Ajax has moved from an exotic feature of JavaScript to a standard-issue tool in the professional JavaScript programmer's bag of tricks. It took a while for debugging tools to catch up. Now, developer tools provide several ways to track Ajax requests. Generally, you should be able to get information on Ajax requests at either the console or the network analyzer. The latter has the more detailed interface. You should be able to sort on specific types of requests (XHR/Ajax, scripts, images, HTML, and so on). Each request should get its own entry, which will usually give you information about the state of the request (both the response code and the response message), where it went to (full URL), how much data was exchanged, and how long the request took. Diving into an individual request, you can see the request and response headers, a preview of processed data and, depending on the data type, a raw view of the data. For example, if your application makes a request for JSON-formatted data, the network analyzer will both tell you about the raw data (a plain string) and, potentially, pass that string through a JSON formatter, so it can show you the end result of the request. Figure 4-4 shows the Network Analyzer in Chrome 40.0, and Figure 4-5 shows it in Firefox 35.0.1.
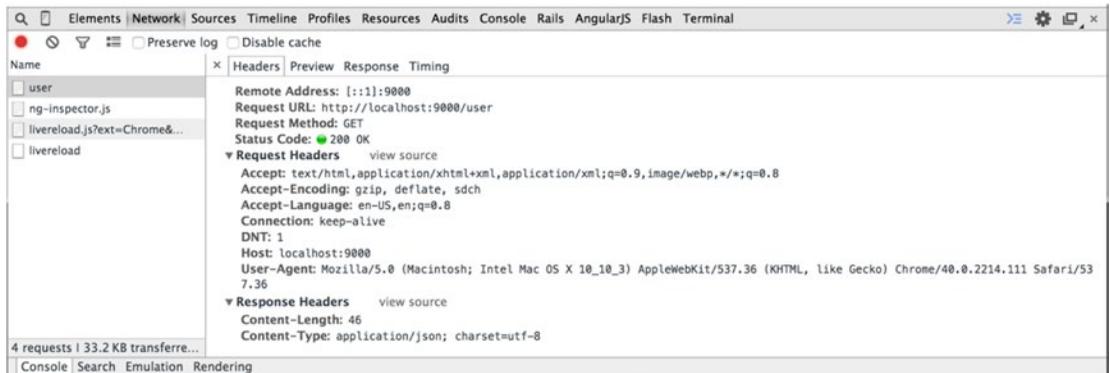
Figure 4-4. *Network Analyzer in Chrome 40.0*
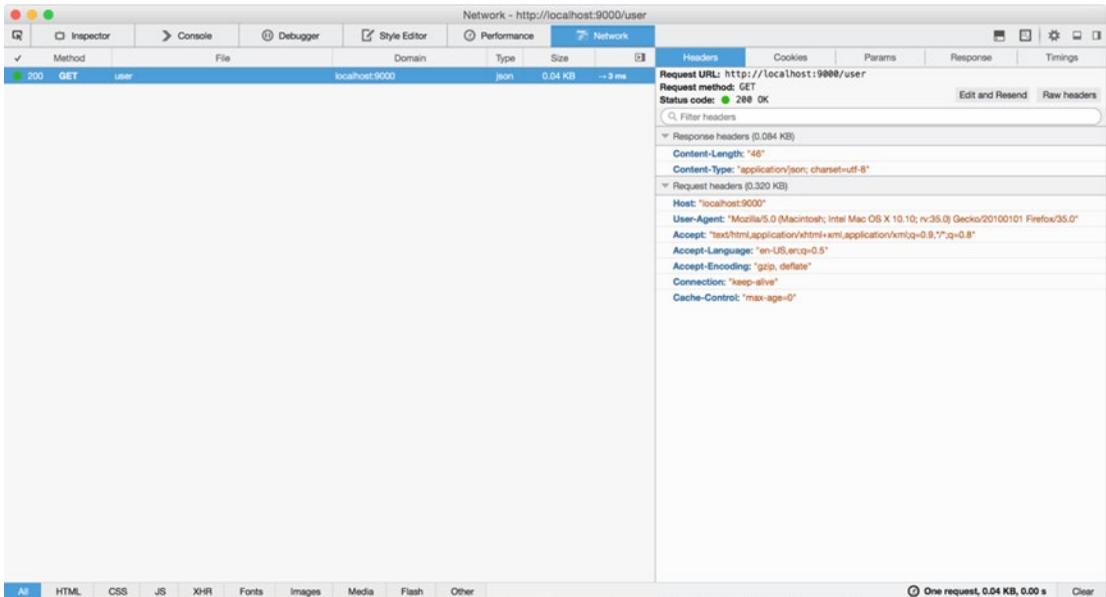
Figure 4-5. *Network Analyzer in Firefox 35.0.1*

Using both the heap profiler and the timeline, you can detect memory leaks on both the desktop and mobile devices. First let's look at the timeline.

Timeline

When you first notice your page getting slow, the timeline can quickly help you see how much memory you are using over time. The features in the timeline are very similar in all modern browsers, so to keep this short we are going to focus on Chrome.

Go to the Timeline panel and check off the Memory checkbox. Once there you can click the Record button on the left side. This will start to record the memory consumption of your application. While recording, use your application in a way to expose the memory leak. Stop recording, and the graph will show you how much memory you have been using over time.

If you find that over time your application is using memory and the level is never dropping with garbage collection, then you have a memory leak.

Profiler

If you find that you do have a memory leak, the next step is to look at the profiler and try to understand what is going on.

It's helpful to understand how memory works in the browser and how it is cleaned up or garbage-collected. Garbage collection is handled automatically in the browser. It is the process in which the browser looks at all the objects that have been created. Objects that are no longer referenced are removed and the memory is reclaimed.

All browsers now have profiling tools built in. These will let you see which objects are using more memory over time.

Figure 4-6 shows the Profiles panel in Chrome 40.0.
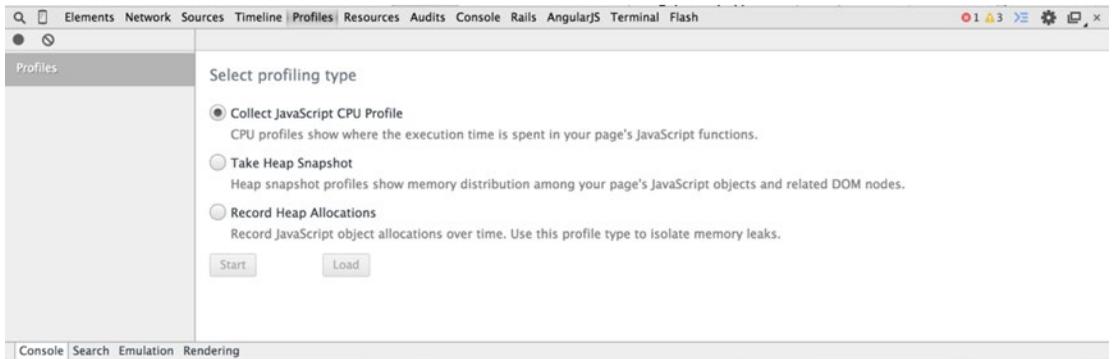
Figure 4-6. *Profiles panel in Chrome 40.0*

Figure 4-7 shows the equivalent panel in Firefox 35.0.1, the Performance tab.
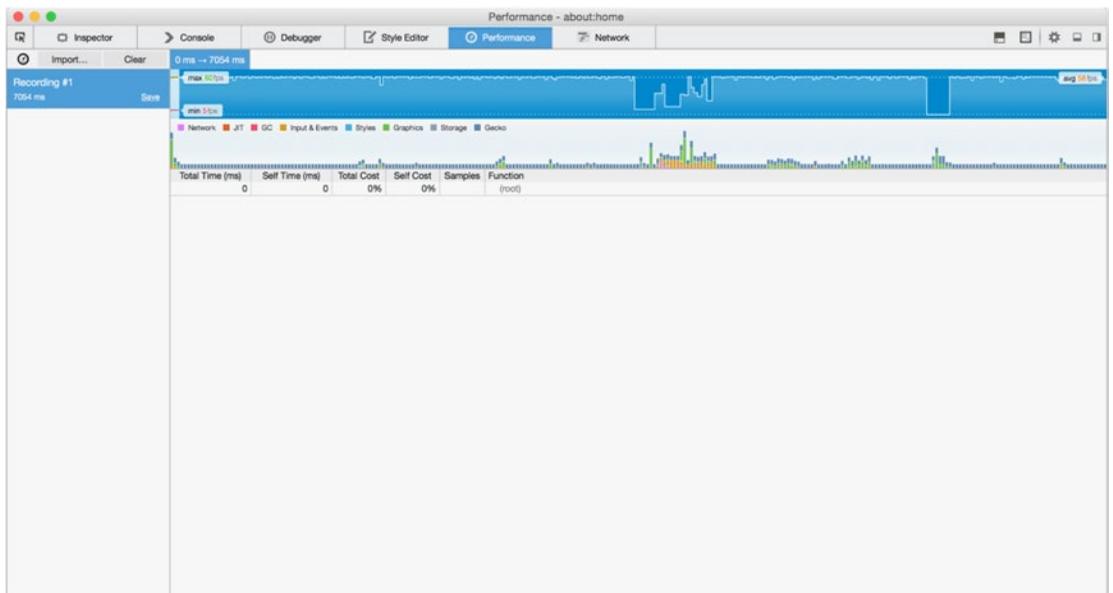
Figure 4-7. *Profile Panel in Firefox 35.0.1*

Using the profiler is similar in that you need the browser to record the application in action. In this case you take what's called a snapshot. The Gmail team recommends taking three, in the following order:

1. Take a snapshot.

2. Perform the actions where you think the leak is coming from.

3. Take the second snapshot.

4. Perform the same actions.

5. Take the third snapshot.

6. Filter the objects from Snapshot 1 and 2 in the Summary view of Snapshot 3.

At this point you can start to see all the objects that are still around and could be taking up memory. You should now be able to see which references are remaining and dispose of them.

So what are references? Generally a reference happens when an object has a property whose value is another object. Listing 4-3 shows an example.

Listing 4-3. Creating Object References

```
var myObject = {};
myObject.property = document.createElement('div');
mainDiv.appendChild(myObject.property);
```

Here `myObject.property` now has a reference to the newly created `div` object. The `appendChild` method can use it with no problem. If at some point you remove that `div` from the DOM, `myObject` will still have a reference to the `div` and will not be garbage-collected. When objects no longer hold on to references, they are automatically garbage-collected.

One way to remove the reference is by using the `delete` keyword, as illustrated in Listing 4-4.

Listing 4-4. Deleting Object References

```
delete myObject.property;
```

Summary

As you can see, modern browsers have the tools to give you an environment that helps you fully understand your application. If you do see areas where you can make improvements, the timeline can show how much memory is being used over time. The debugger can help you see the values of your variables at any given time. Using the profiler can help you see where you are leaking memory and how you can fix it.

CHAPTER 5

■ ■ ■

The Document Object Model

Working with the Document Object Model (the DOM) is a critical component of the professional JavaScript programmer's toolkit. A comprehensive understanding of DOM scripting yields benefits not only in the range of applications we can build, but also in the quality of those applications. Like most features of JavaScript, the DOM has a somewhat checkered history. But with modern browsers, it is easier than ever to manipulate and interact with the DOM unobtrusively. Understanding how to use this technology and how best to wield it can give you a head start toward developing your next web application.

In this chapter we discuss a number of topics related to the DOM. For readers new to the DOM, we will start out with the basics and move through all the important concepts. For those of you already familiar with the DOM, we provide a number of cool techniques that we are sure you will enjoy and start using in your own web pages.

The DOM is also at a crossroads. Historically, because DOM interface updates were not in sync with browser or JavaScript updates, there was a disconnect between browsers and DOM support. This disconnect was only exacerbated by buggy implementations. Popular libraries like jQuery and Dojo have arisen to address these problems. But with modern browsers, the DOM has normalized and the interface has settled quite a bit. We will need to address the issue of whether to use libraries to help in our access of the DOM or to do everything with the standard DOM interface.

An Introduction to the Document Object Model

Initially, the DOM was created as a way to represent *parts* of an HTML document within a browser. Using JavaScript, a developer could look at forms, anchors, images and other components of the page, but not necessarily the entire page. This is sometimes referred to as the "legacy DOM" or DOM Level 0. Eventually, the DOM changed into an interface, overseen by the W3C. From humble beginnings, the DOM has become the official interface not only to HTML documents but also to XML documents. It is not necessarily the fastest, lightest, or easiest-to-use interface, but it is the most ubiquitous, with an implementation existing in most programming languages (such as Java, Perl, PHP, Ruby, Python, and, of course, JavaScript). As we work with the DOM interface, remember that just about everything you learn can be applied to HTML and XML, even though most of the time we will refer exclusively to HTML.

The World Wide Web Consortium oversees the DOM specification. For a variety of historical reasons, versions of the DOM specification are identified as DOM Level *n*. The current specification (as of publication time) is DOM Level 4. This can sometimes be confusing, as a DOM tree itself can have levels. We will endeavor to refer to versions of the DOM specification as DOM Level (with a capital L) and then DOM tree levels with a lowercase l.

Before anything else, we should quickly discuss the structure of HTML documents. Because this is a book on JavaScript, not HTML, we will focus on the effects of an HTML document on our JavaScript. Let us set forth a few simple principles:

1. Our HTML documents should start with an HTML 5 doctype. They are very simple: `<!DOCTYPE html>`. Including the doctype prevents browsers from falling into quirks mode, where the behavior of the browser is less consistent.

2. We should prefer including separate JavaScript files via `<script>` tags over script blocks or in-line scripts. This makes for easier development (separating our JavaScript from our HTML) and easier management. There are rare cases where script blocks make more sense than included files. But as a general rule, prefer included files.

3. Our script tags should appear at the bottom of the HTML document, immediately before the closing `</body>` tag.

The third item requires some explanation. By having our `<script>` includes at the bottom of the page, we gain several advantages. The majority (if not all) of our HTML should have loaded (as well as associated files: images, audio, video, CSS, and so on). Why is this important? Processing JavaScript code locks up the rendering of a page! Browsers cannot render other page elements while JavaScript code is being parsed (and sometimes when JavaScript code is running!). Therefore, we should wait until the last minute to load our JavaScript code wherever possible. Also, in mobile or slow-connection scenarios, fetching and loading JavaScript can be slower than on a desktop browser. Early loading of the rest of the page means your users are not stuck looking at a blank page with nothing but a spinner. The principle here is that the user should be able to see feedback that some of the page has loaded as soon as possible.

Right. So what does this ideal HTML page look like? Check out Listing 5-1.

Listing 5-1. A Sample HTML File

```
<!DOCTYPE html>
<html>
<head>
  <title>Introduction to the DOM</title>
</head>
<body>
<h1>Introduction to the DOM</h1>

<p id="intro" class="test">There are a number of reasons why the DOM is awesome; here are
some:</p>
<ul id="items">
  <li id="everywhere">It can be found everywhere.</li>
  <li class="test">It's easy to use.</li>
  <li class="test">It can help you to find what you want, really quickly.</li>
</ul>
<script src="01-sample.js"></script>
</body>
</html>
```

Sometimes, in our examples, we will have an in-line script block. If so, it will appear within the page as appropriate to its functionality. If its location is irrelevant to its functionality, we will have the script block at the bottom of the page, much like our script includes.

DOM Structure

The structure of an HTML document is represented in the DOM as a navigable tree. All the terminology used is akin to that of a genealogical tree (parents, children, siblings, and so on). For our purposes, the trunk of the tree is the Document node, also known as the document element. This element contains pointers to its children and, in turn, each child node then contains pointers back to its parent, its fellow siblings, and its children.

The DOM uses particular terminology to refer to the different objects within the HTML tree. Just about everything in a DOM tree is a node: HTML elements are nodes, the text within elements is a node, comments are nodes, the DOCTYPE is a node, and even attributes are nodes! Obviously, we will need to be able to differentiate among these nodes, so each node has a node type property called, appropriately, nodeType (Table 5-1). We can query this property to figure out which type of node we are looking at. If you get a reference to a node, it will be an instance of a Node type, implementing all of the methods and having all of the properties of that type.

Table 5-1. *Node Types and Their Constant Values*

Node Name	Node Type Value
ELEMENT_NODE	1
ATTRIBUTE_NOTE (deprecated)	2
TEXT_NODE	3
CDATA_SECTION_NODE (deprecated)	4
ENTITY_REFERENCE_NODE (deprecated)	5
ENTITY_NODE (deprecated)	6
PROCESSING_INSTRUCTION_NODE	7
COMMENT_NODE	8
DOCUMENT_NODE	9
DOCUMENT_TYPE_NODE	10
DOCUMENT_FRAGMENT_NODE	11
NOTATION_NODE (deprecated)	12

Node types marked deprecated are superseded and may be removed, but it's very unlikely. They probably still work, as they have been in use for a few years now.

As you can see in the table, nodes have various specializations, which correspond to interfaces in the DOM specification. Of particular interest are documents, elements, attributes, and text. Each of these has its own implementing type: Document, Element, Attr, and Text, respectively.

■ **Note** Attributes are a special case. Under DOM Levels 1, 2, and 3, the Attr interface implemented the Node interface. This is no longer true for DOM Level 4. Thankfully, this is more of a common-sense change than anything else. More details can be found in the section on attributes.

In general, the Document is concerned with managing the HTML document as a whole. Each tag within that document is an Element, which is itself specialized into specific HTML element types (for example, `HTMLLIElement`, `HTMLFormElement`). The attributes of an element are represented as instances of Attr. Any plain text within an Element is a text node, represented by the Text type. These subtypes are not all of the types that inherit from Node, but they are the ones we are most likely to interact with.

Given our listing, let's look at the structure: the entire document, from `<!DOCTYPE html>` to `</html>` is the Document. The doctype is itself an instance of the Doctype type. The `<html>...</html>` element is our first and main Element. It contains child Elements for the `<head>` and `<body>` tags. Diving a little more deeply, we can see that the `<p>` element within the `<body>` has two attributes, `id` and `class`. That same `<p>` element has a single Text node for its content. The hierarchical structure of the document is duplicated in the relationships between the instances of the various DOM types. We should look at these relationships in greater detail.

DOM Relationships

Let's examine a very simple document fragment, to show the various relationships between nodes:

```
<p><strong>Hello</strong> how are you doing?</p>
```

Each portion of this snippet breaks down into a DOM node with pointers from each node pointing to its direct relatives (parents, children, siblings). If you were to completely map out the relationships that exist, it would look something like Figure 5-1. Each portion of the snippet (rounded boxes represent elements, regular boxes represent text nodes) is displayed along with its available references.
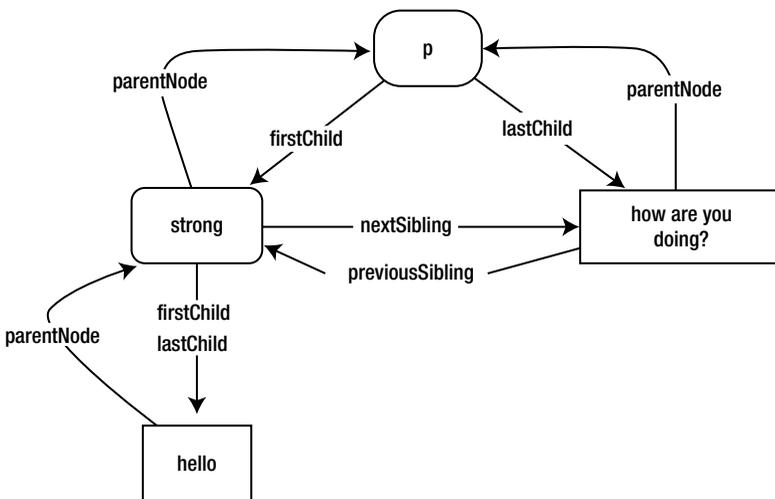
Figure 5-1. *Relationships between nodes*

Every single DOM node contains a collection of pointers that it can use to refer to its relatives. You'll be using these pointers to learn how to navigate the DOM. All the available pointers are displayed in Figure 5-2. Each of these properties, available on every DOM node, is a pointer to another Node or subclass thereof. The only exception is `childNodes` (a collection of all of the child nodes of the current node). And, of course, if one of these relationships is undefined, the value of the property will be null (think of an `<img>` tag, which will have neither `firstChild` nor `lastChild` defined).
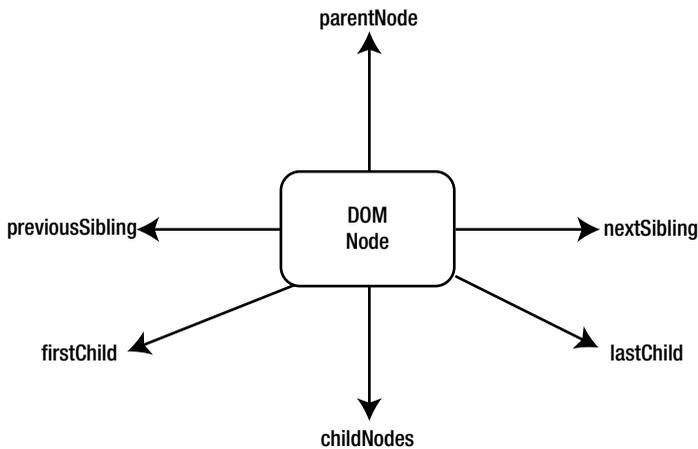
Figure 5-2. *Navigating the DOM tree using pointers*

Using nothing but the different pointers, it's possible to navigate to any element or text block on a page. Recall Listing 5-1, which showed a typical HTML page. Before, we looked at it from the perspective of JavaScript types. Now we will look at it from the perspective of the DOM.

In the example document, the document node is the <html> element. Accessing this element is trivial in JavaScript: document.documentElement refers directly to the <html> element. The root node has all the pointers used for navigation, just like any other DOM node. Using these pointers you have the ability to start browsing the entire document, navigating to any element that you desire. For example, to get the <h1> element, you could use the following:

```
// Does not work!
document.documentElement.firstChild.nextSibling.firstChild
```

But we have just hit a major snag: The DOM pointers can point to both text nodes and elements. Our JavaScript statement doesn't actually point to the <h1> element; it points to the <title> element instead. Why did this happen? It happened because of one of the stickiest and most-debated aspects of XML: white space. If you will notice in Listing 5-1, between the <html> and <head> elements there is actually an end line, which is considered white space, which means that there's actually a text node first, not the <head> element. We can learn four things from this:

- Writing nice, clean HTML markup can actually make things very confusing when attempting to browse the DOM using nothing but pointers.

- Using nothing but DOM pointers to navigate a document can be very verbose and impractical.

- In fact, DOM pointers are clearly quite brittle, as they tie your JavaScript logic to your HTML entirely too closely.

- Frequently, you don't need to access text nodes directly, only the elements that surround them.

This leads us to a question: Is there a better way to find elements in a document? Yes, there is! More accurately: there are! We have two major approaches available for accessing elements within the page. On the one hand, we could continue down the line of relative access, sometimes known as DOM traversal. For

the reasons just listed, we will avoid this approach for general DOM access. We will revisit DOM traversal later on, though, when we have a better handle on accessing specific elements. Instead, we will take a second path, focusing on the various element retrieval functions provided with the modern DOM interface.

Accessing DOM Elements

All modern DOM implementations contain several methods that make it easy to find elements within the page. Using these methods together with some custom functions can make navigating the DOM a much smoother experience. To start with, let's look at how we can access a single element:

> `document.getElementById('everywhere')`: This method, which can only be run on the document object, finds all elements that have an ID equal to *everywhere*. This is a very powerful function and is the fastest way to access an element immediately.

The `getElementById` method returns a reference to the HTML element with the supplied ID, or null otherwise. The returned object is specifically an instance of the Element type. We will discuss what we can do with this Element soon.

■ **Caution**   `getElementById` works as you would imagine with HTML documents: it looks through all elements and finds the one single element that has an attribute named `id` with the specified value. However, if you are loading in a remote XML document and using `getElementById` (or using a DOM implementation in any language other than JavaScript), it doesn't use the `id` attribute by default. This is by design; an XML document must explicitly specify what the `id` attribute is, generally using an XML definition or a schema.

Let's continue our tour of element-accessing functions. The next two functions provide access to collections of elements:

> `getElementsByTagName('li')`: This method, which can be run on any element, finds all descendent elements that have a tag name of `li` and returns them as a live `NodeList` (which is nearly identical to an array).

> `getElementsByClassName('test')`: Similar to getElementsByTag *name*, this method can be run from any instance of Element. It returns a live `HTMLCollection` of matching elements.

These two functions allow us to access multiple elements at once. Putting aside the difference in return type for a moment, the collection returned is *live*. This means that if the DOM is modified, and those modifications would be included in the collection (or would remove elements from the collection), the collection will automatically update with those changes. Very powerful!

It is odd that these two methods, similar in function, return two different types. First, let's consider the simple parts: Both types have array-like positional access. That is, for the following:

```
var lis = document.getElementsByTagName('li');
```

you can access the second list item in the `lis` collection via `lis[1]`. Both collections have a `length` property, which tells you how many items are in the collection. They also have an `item` method, which takes as its argument the position to access and returns the element at that position. The `item` method is a functional way to access elements positionally. Finally, neither collection has any of the higher-order Array methods, like `push`, `pop`, `map`, or `filter`.

If you would like to use Array methods on your `HTMLCollection` or `NodeList`, you can always use them as shown in Listing 5-2.

Listing 5-2. Array Functions on `NodeLists`/`HTMLCollections`

```
// A simple filtering function
// An Element's nodeName property is always the name of the underlying tag.
function filterForListItems(el) {
    return el.nodeName === 'LI';
}

var testElements = document.getElementsByClassName( 'test' );
console.log( 'There are ' + testElements.length + ' elements in testElements.');

// Generating an array from the elements gathered from testElements
// based on whether they pass the filtering proccess set up by filterForListItems
var liElements = Array.prototype.filter.call(testElements, filterForListItems);
console.log( 'There are ' +  liElements.length + ' elements in liElements.');
```

The difference between methods in the return type is caused by the vagaries of DOM implementation in browsers. In the future, both should return `HTMLCollection` instances, but that future is not yet here. Because the access patterns for `NodeLists` and `HTMLCollections` are virtually identical, we do not have to concern ourselves too much with which method returns which type.

When using either `getElementsByClassName` or `getElementsByTagName`, it is worth remembering that they belong not only to Document instances, but also Element instances. When called from the document, they will conduct searches over the entire document. Consider that your `<head>` section will be searched for `<li>` tags or that you will be looking there for elements with the class `foo`. This is, as you can imagine, somewhat inefficient. Imagine that you are searching through your house for your keys. You would probably not search in the refrigerator, or in the shower, as they are not likely spots to have left your keys. So you will look in the bedroom, the living room, the entryway, and so on. Wherever possible, limit the scope of your search to the appropriate containing element. Take a look at Listing 5-3, which gets the same results as Listing 5-2 but limits its scope to a specific parent element.

Listing 5-3. Limiting Search Scope

```
var ul = document.getElementById( 'items' );
var liElements = ul.getElementsByClassName( 'test' );
console.log( 'There are ' +  liElements.length + ' elements in liElements.');
```

■ **Note**   `document.getElementById`, unlike `getElementsByClassName` or `getElementsByTagName`, is *not* available on instances of the Element type. It is *only* available on the document or an instance of the Document type.

These three methods are available in all modern browsers and can be immensely helpful for locating specific elements. Going back to the earlier example where we tried to find the `<h1>` element, we can now do the following:

```
document.getElementsByTagName('h1')[0];
```

This code is guaranteed to work and will always return the first `<h1>` element in the document.

Finding Elements by CSS Selector

As a web developer, you already know of an alternative way to select HTML elements: CSS selectors. A CSS selector is the expression used to apply CSS styles to a set of elements. With each revision of the CSS standard (1, 2, and 3, also sometimes referred to as CSS Level 1, Level 2 or Level 3, respectively) more features have been added to the selector specification, so that developers can more easily locate the exact elements they desire. Browsers were occasionally slow to provide full implementations of CSS 2 and 3 selectors, and so you may not know of some of the cool new features that they provide. This has largely been resolved in modern browsers. If you're interested in all the cool new features in CSS, we recommend exploring the W3C's pages on the subject:

- CSS 1 selectors: http://www.w3.org/TR/CSS1/#basic-concepts

- CSS 2.1 selectors: http://www.w3.org/TR/CSS21/selector.html

- CSS 3 selectors: http://www.w3.org/TR/css3-selectors/

The features that are available from each CSS selector specification are generally similar, in that each subsequent release contains all the features from the past ones, too. However, with each release a number of new features are added. As an example, CSS 2.1 contains attribute and child selectors, while CSS 3 provides additional language support, selecting by attribute type, and negation. For modern browsers, all of these are valid CSS selectors:

> #main <div> p: This expression finds an element with an ID of main, all <div> element descendants, and then all <p> element descendants. All of this is a proper CSS 1 selector.

> div.items > p: This expression finds all <div> elements that have a class of items, then locates all child <p> elements. This is a valid CSS 2 selector.

> div:not(.items): This locates all <div> elements that do not have a class of items. This is a valid CSS 3 selector.

There are two methods that provide access to elements via CSS selectors: querySelector and querySelectorAll. Give querySelector a valid CSS selector, and it will return a reference to the first element it finds matching that selector. The only thing that changes when using querySelectorAll is that you get back a non-live NodeList of matching elements. (The list is not live, because a live list would be resource-intensive). As with getElementsByTagName and getElementsByClassName, you can call querySelector and querySelectorAll from any instance of Element. Where possible, prefer limiting the scope of searches this way for greater efficiency and faster returns.

We now have four ways to access elements. Which should we use? First, for single-element access, document.getElementById should always be the fastest. But for multiple-element access, or if the element you want doesn't have an ID, consider using getElementsByTagName, then getElementsByClassName, then querySelectorAll. But keep in mind that this only takes into account speed. Sometimes, ease of querying, or accuracy of matched elements, or even the need for a live collection matters more than speed. Use the method that suits your needs best.

Waiting for the HTML DOM to Load

One of the difficulties when working with HTML DOM documents is that your JavaScript code is able to execute before the DOM is completely loaded, potentially causing a number of problems in your code. The order of operation inside a browser looks something like this:

1. HTML is parsed.

2. External style sheets are loaded.

3. Scripts are executed as they are parsed in the document.

4. HTML DOM is fully constructed.

5. Images and external content are loaded.

6. The page is finished loading.

Of course, all of this is largely dependent on the structure of your HTML. If you have a `<script>` tag before the `<link>` tag that loads your CSS, then the JavaScript will load before the CSS does. (By the way, do not do this. It is inefficient.) Scripts that are in the head and loaded from an external file are executed before the HTML DOM is actually constructed. As mentioned previously, this is a significant problem because all scripts executed in those two places won't have access to the DOM. That is part of why we have avoided putting our script tags in the `<head>` section. But even when we follow best practices and include our `<script>` tags just before the closing `<body>` tag, there is the possibility that the DOM is not ready for processing by our JavaScript. Thankfully, there exist a number of workarounds for this problem.

Waiting for the Page to Load

By far the most common technique is simply waiting for the entire page to load before performing any DOM operations. This technique can be utilized by simply attaching a function, to be fired on page load, to the load event of the window object. We will discuss events in greater detail in Chapter 6. Listing 5-4 shows an example of executing DOM-related code after the page has finished loading.

Listing 5-4. The addEventListener Function for Attaching a Callback to the Window load Property

```
// Wait until the page is loaded
// (Uses addEventListener, described in the next chapter)
window.addEventListener('load', function() {
    // Perform HTML DOM operations
    var theSquare = document.getElementById('square');
        theSquare.style.background = 'blue';
});
```

While this operation may be the simplest, it will always be the slowest. From the order of loading operations, you will notice that the page being loaded is the last step taken. The load event does not fire until all elements with src attributes have had their files downloaded. This means that if your page has a significant number of images, videos, and so on, your users might be waiting quite a while until the JavaScript finally executes. On the other hand, this is the most backward-compatible solution.

Waiting for the Right Event

If you have a more modern browser, you can check for the `DOMContentLoaded` event. This event fires when the document has been completely loaded and parsed. In our list, this matches roughly to "HTML DOM is fully constructed." But keep in mind that images, stylesheets, videos, audio, and the like may not have completely loaded by the time this event fires. If you need your code to fire after a particular image or video file has loaded, consider using the load event for that particular tag. If you need to wait until all elements with a `src` attribute have downloaded their files, fall back to using the window load event. Look at Listing 5-5 for details.

Listing 5-5. Using `DOMContentLoaded`

```
document.addEventListener('DOMContentLoaded' functionHandler);
```

Internet Explorer 8 does not support `DOMContentLoaded`, but you can check to see if the ready state has changed on the document. Listing 5-6 shows how to detect whether the DOM has loaded in a cross-browser–compatible fashion.

Listing 5-6. Cross-browser `DOMContentLoaded`

```
if(document.addEventListener){
     document.addEventListener('DOMContentLoaded', function(){
      document.removeEventListner('DOMContenLoded',arguments.callee);
})else if(document.attachEvent){
     document.attachEvent('onreadystatechange', function(){
     document.detachEvent('onreadystatechange', arguments.callee,);
}
```

Getting the Contents of an Element

All DOM elements can contain one of three things: text, more elements, or a mixture of text and elements. Generally speaking, the most common situations are the first and last. In this section you're going to see the common ways available for retrieving the contents of an element.

Getting the Text of an Element

Getting the text inside an element is probably the most confusing task for those who are new to the DOM. However, it is also a task that works in both HTML DOM documents and XML DOM documents, so knowing how to do this will serve you well. In the example DOM structure shown in Figure 5-3, there is a root `<p>` element that contains a `<strong>` element and a block of text. The `<strong>` element itself also contains a block of text.
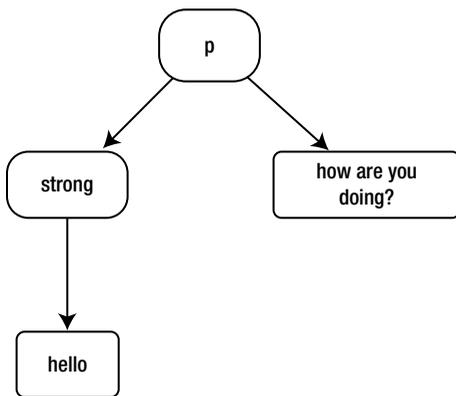
Figure 5-3. *A sample DOM structure containing both elements and text*

Let's look at how to get the text of each of these elements. The <strong> element is the easiest to start with, since it contains only one text node and nothing else.

It should be noted that there exists a property called innerText that captures the text inside an element in all non–Mozilla-based browsers. It's incredibly handy in that respect. Unfortunately, because it doesn't work in a noticeable portion of the browser market, and it doesn't work in XML DOM documents, you still need to explore viable alternatives.

The trick with getting the text contents of an element is that you need to remember that text is not contained within the element directly; it's contained within the child text node, which may seem a bit strange. For example, Listing 5-7 shows how to extract text from inside an element using the DOM; it is assumed that the variable strongElem contains a reference to the <strong> element.

Listing 5-7. Getting the Text Contents of the <strong> Element

```
// Non-Mozilla Browsers:
strongElem.innerText

// All platforms including Non-Mozilla browsers:
strongElem.firstChild.nodeValue
```

Now that you know how to get the text contents of a single element, you need to look at how to get the combined text contents of the <p> element. In doing so, you might as well develop a generic function to get the text contents of any element, regardless of what it actually contain, as shown in Listing 5-8. Calling text(*Element*) will return a string containing the combined text contents of the element and all child elements that it contains.

Listing 5-8. A Generic Function for Retreiving the Text Contents of an Element

```
function text(e) {
    var t = '' ;
    // If an element was passed, get its children,
    // otherwise assume it's an array
    e = e.childNodes || e;
```

```
    // Look through all child nodes
    for ( var j = 0; j < e.length; j++ ) {
        // If it's not an element, append its text value
        // Otherwise, recurse through all the element's children
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }

    // Return the matched text
    return t;
}
```

With a function that can be used to get the text contents of any element, you can retrieve the text contents of the <p> element, used in the previous example. The code to do so would look something like this:

```
// Get the text contents of the <p> Element
var pElm = document.getElementsByTagName ('p');
console.log(text( pElem ));
```

The particularly nice thing about this function is that it's guaranteed to work in both HTML and XML DOM documents, which means you now have a consistent way of retrieving the text contents of any element.

Getting the HTML of an Element

Unlike getting the text inside an element, getting an element's HTML is one of the easiest DOM tasks that can be performed. Thanks to a feature developed by the Internet Explorer team, all modern browsers now include an extra property on every HTML DOM element: innerHTML. With this property you can get all the HTML and text inside of an element. Additionally, using the innerHTML property is very fast—often much faster than doing a recursive search to find all the text contents of an element. However, it isn't all roses. It's up to the browser to figure out how to implement the innerHTML property, and because there's no true standard for that, the browser can return whatever content it deems worthy. For example, here are some of the weird bugs you can look forward to when using the innerHTML property:

- Mozilla-based browsers don't return the <style> elements in an innerHTML statement.

- Internet Explorer 8 and lower returns its elements in all caps, which if you're looking for consistency can be frustrating.

- The innerHTML property is consistently available only as a property on elements of HTML DOM documents; trying to use it on XML DOM documents will result in retrieving null values.

Using the innerHTML property is straightforward; accessing the property gives you a string containing the HTML contents of the element. If the element doesn't contain any subelements and only text, the returned string will contain only the text. To look at how it works, we're going to examine the two elements shown in Figure 5-2:

```
// Get the innerHTML of the <strong> element
// Should return "Hello"
strongElem.innerHTML
// Get the innerHTML of  the <p> element
// Should return "<strong>Hello</strong> how are you doing?"
pElem.innerHTML
```

If you're certain that your element contains nothing but text, this method could serve as a very simple replacement to the complexities of getting the element text. On the other hand, being able to retrieve the HTML contents of an element means that you can build some cool dynamic applications that take advantage of in-place editing; more on this topic can be found in Chapter 10.

Working with Element Attributes

Next to retrieving the contents of an element, getting and setting the value of an element's attribute is one of the most frequent operations. Typically, the list of attributes that an element has is preloaded with information collected from the XML representation of the element itself and stored in an associative array for later access, as in this example of an HTML snippet inside a web page:

```
<form name="myForm" action="/test.cgi" method="POST">
    ...
</form>
```

Once loaded into the DOM and the variable formElem, the HTML form element would have an associative array from which you could collect name/value attribute pairs. The result would look something like this:

```
formElem.attributes = {
    name: 'myForm',
    action: '/test.cgi',
    method: 'POST'
};
```

Figuring out whether an element's attribute exists should be absolutely trivial using the attributes array, but there's one problem: for whatever reason Safari doesn't support this feature. Internet Explorer version 8 and above support it, as long as IE8 is in standards mode. So how are you supposed to find out if an attribute exists? One possible way is to use the getAttribute function (covered in the next section) and test to see whether the return value is null, as shown in Listing 5-9.

Listing 5-9. Determining Whether an Element Has a Certain Attribute

```
function hasAttribute( elem, name ) {
    return elem.getAttribute(name) != null;
}
```

With this function in hand, and knowing how attributes are used, you are now ready to begin retrieving and setting attribute values.

Getting and Setting an Attribute Value

There are two methods to retrieve attribute data from an element, depending on the type of DOM document you're using. If you want to be safe and always use generic XML DOM–compatible methods, there are getAttribute() and setAttribute(). They can be used in this manner:

```
// Get an attribute
document.getElementById('everywhere').getAttribute('id');
// Set an attribute value
document.getElementsByTagName('input')[0].setAttribute('value', 'Your Name');
```

61

In addition to this standard getAttribute/setAttribute pair, HTML DOM documents have an extra set of properties that act as quick getters/setters for your attributes. These are universally available in modern DOM implementations (but only guaranteed for HTML DOM documents), so using them can give you a big advantage when writing short code. The following example shows how you can use DOM properties to both access and set DOM attributes:

```
// Quickly get an attribute
document.getElementsByTagName('input')[0].value;

// Quickly set an attribute
document.getElementsByTagName('div')[0].id = 'main';
```

There are a couple of strange cases with attributes that you should be aware of. The one that's most frequently encountered is that of accessing the class name attribute. If you are referencing the name of a class directly, elem.className will let you set and get the name. However, if you're using the get/setAttribute method, you can refer to it as getAttribute('class'). To work with class names consistently in all browsers you must access the className attribute using elem.className, instead of using the more appropriately named getAttribute('class'). This problem also occurs with the for attribute, which gets renamed to htmlFor. Additionally, it is also the case with a couple of CSS attributes: cssFloat and cssText. This particular naming convention arose because words such as class, for, float, and text are all reserved words in JavaScript.

To work around all these strange cases and simplify the process of dealing with getting and setting the right attributes, you should use a function that will take care of all those particulars for you. Listing 5-10 shows a function for getting and setting the values of element attributes. Calling the function with two parameters, such as attr(element, id), returns that value of that attribute. Calling the function with three parameters, such as attr(element, class, test), will set the value of the attribute and return its new value.

Listing 5-10. Getting and Setting the Values of Element Attributes

```
function attr(elem, name, value) {
    // Make sure that a valid name was provided
    if ( !name || name.constructor != String ) return '' ;

    // Figure out if the name is one of the weird naming cases
    name = { 'for': 'htmlFor', 'className': 'class' }[name] || name;

    // If the user is setting a value, also
    if ( typeof value != 'undefined' ) {
        // Set the quick way first
        elem[name] = value;

        // If we can, use setAttribute
        if ( elem.setAttribute )
            elem.setAttribute(name,value);
    }

    // Return the value of the attribute
    return elem[name] || elem.getAttribute(name) || '';
}
```

Having a standard way to both access and change attributes, regardless of their implementation, is a powerful tool. Listing 5-11 shows some examples of how you could use the attr function in a number of common situations to simplify the process of dealing with attributes.

Listing 5-11. Using the attr Function to Set and Retreive Attribute Values from DOM Elements

```
// Set the class for the first <h1> Element
attr( document.getElementByTagName('h1')[0], 'class', 'header' );

// Set the value for each <input> element
var input = document.getElementByTagName('input');
for ( var i = 0; i < input.length; i++ ) {
    attr( input[i], 'value', ''  );
}

// Add a border to the <input> Element that has a name of 'invalid'
var input = document.getElementByTagName('input');
for ( var i = 0; i < input.length; i++ ) {
    if ( attr( input[i], 'name' ) == 'invalid' ) {
        input[i].style.border = '2px solid red';
    }
}
```

Up until now, we've only discussed getting/setting attributes that are commonly used in the DOM (ID, class, name, and so on). However, a very handy technique is to set and get nontraditional attributes. For example, you could add a new attribute (which can only be seen by accessing the DOM version of an element) and then retrieve it again later, all without modifying the physical properties of the document. For example, let's say that you want to have a definition list of items, and whenever a term is clicked have the definition expand. The HTML for this setup would look something like Listing 5-12.

Listing 5-12. An HTML Document with a Definition List, with the Definitions Hidden

```
<html>
<head>
    <title>Expandable Definition List</title>
    <style>dd { display: none; }</style>
</head>
<body>
    <h1>Expandable Definition List</h1>

    <dl>
        <dt>Cats</dt>
        <dd>A furry, friendly, creature.</dd>
        <dt>Dog</dt>
        <dd>Like to play and run around.</dd>
        <dt>Mice</dt>
        <dd>Cats like to eat them.</dd>
    </dl>
</body>
</html>
```

We'll talk more about the particulars of events in Chapter 6, but for now we'll try to keep our event code simple enough. What follows is a quick script that allows you to click the terms and show (or hide) their definitions. Listing 5-13 shows the code required to build an expandable definition list.

Listing 5-13. Allowing for Dynamic Toggling to the Definitions

```
// Wait until the DOM is Ready
document.addEventListener('DOMContentLoaded', addEventClickToTerms);

// Watch for a user click on the term
function addEventClickToTerms(){
    var dt = document.getElementsByTagName('dt');
    for ( var i = 0; i < dt.length; i++ ) {
        dt[i].addEventListener('click', checkIfOpen);
    }
}

// See if the definition is already open or not
//Need two nextSiblings because the first sibling is a text node (the words that were
clicked on).
//If it's never been clicked, the style will be blank ''. F it has been, the style will be
'none', so we check for both with an if statement.
function checkIfOpen(e){
    if(e.target.nextSibling.nextSibling.style.display == '' || e.target.nextSibling.
nextSibling.style.display == 'none'){
        e.target.nextSibling.nextSibling.style.display = 'block';
    }else{
        e.target.nextSibling.nextSibling.style.display = 'none';
    }
}
```

Now that you know how to traverse the DOM and how to examine and modify attributes, you need to learn how to create new DOM elements, insert them where they are needed, and remove elements that you no longer need.

Modifying the DOM

By knowing how to modify the DOM, you can do anything from creating custom XML documents on the fly to building dynamic forms that adapt to user input; the possibilities are nearly limitless. Modifying the DOM comes in three steps: first you need to learn how to create a new element, then you need to learn how to insert it into the DOM, then you need to learn how to remove it again.

Creating Nodes Using the DOM

The primary method of modifying the DOM is the createElement function, which gives you the ability to create new elements on the fly. However, this new element is not immediately inserted into the DOM when you create it (a common point of confusion for people just starting with the DOM). First, we'll focus on creating a DOM element.

The createElement method takes one parameter, the tag name of the element, and returns the virtual DOM representation of that element—no attributes or styling included. If you're developing applications that use XSLT-generated XHTML pages (or if the applications are XHTML pages served with an accurate content type), you have to remember that you're actually using an XML document and that your elements need to have the correct XML namespace associated with them. To work around this seamlessly, you can have a simple function that quietly tests to see whether the HTML DOM document you're using has the ability to create new elements with a namespace (a feature of XHTML DOM documents). If this is the case, you must create a new DOM element with the correct XHTML namespace, as shown in Listing 5-14.

Listing 5-14. A Generic Function for Creating a New DOM Element

```
function create( elem ) {
    return document.createElementNS ?
        document.createElementNS('http://www.w3.org/1999/xhtml', elem ) :
        document.createElement( elem );
}
```

For example, using the previous function you can create a simple `<div>` element and attach some additional information to it:

```
var div = create('div');
div.className = 'items';
div.id = 'all';
```

Additionally, it should be noted that there is a DOM method for creating new text nodes, called createTextNode. It takes a single argument, the text that you want inside the node, and it returns the created text node.

Using the newly created DOM elements and text nodes, you can now insert them into your DOM document right where you need them.

Inserting into the DOM

Inserting into the DOM is confusing and can feel clumsy at times, even for those experienced with the DOM. You have two functions in your arsenal that you can use to get the job done.

The first function, insertBefore, allows you to insert an element before another child element. When you use the function, it looks something like this:

```
parentOfBeforeNode.insertBefore( nodeToInsert, beforeNode );
```

The mnemonic that we use to remember the order of the arguments is the phrase "You're inserting the first element, before the second."

Now that you have a function to insert nodes (including both elements and text nodes) before other nodes, you should be asking yourself: "How do I insert a node as the last child of a parent?" There is another function you can use, called appendChild, that allows you to do just that. appendChild is called on an element, appending the specified node to the end of the list of child nodes. Using the function looks something like this:

```
parentElem.appendChild( nodeToInsert );
```

Listing 5-15 is an example of how you can use both insertBefore and appendChild in your application.

Listing 5-15. A Function for Inserting an Element Before Another Element

```
document.addEventListener(DOMContentLoaded, 'addElement');

function addElement(){
 //Grab the ordered list that is in the document
 //Remember that getElementById returns an array like object

   var orderedList = document.getElementById('myList');

 //Create an <li>, add a text node then append it to <li>
 var li = document.createElement('li');
     li.appendChild(document.createTextNode('Thanks for visiting'));

 //element [0] is how we access what is inside the orderedList
   orderedList.insertBefore(li, orderedList[0]);
}
```

The instant you "insert" this information into the DOM (with either insertBefore or appendChild) it will be immediately rendered and seen by the user. Because of this, you can use it to provide instantaneous feedback. This is especially helpful in interactive applications that require user input.

Now that you've seen how to create and insert nodes using nothing but DOM-based methods, it should be especially beneficial to look at alternative methods of injecting content into the DOM.

Injecting HTML into the DOM

A technique that is even more popular than creating normal DOM elements and inserting them into the DOM is that of injecting HTML straight into the document. The simplest method for achieving this is by using the previously discussed innerHTML method. Besides being a way to retrieve the HTML inside an element, it is also a way to set the HTML inside an element. As an example of its simplicity, let's assume that you have an empty <ol> element and you want to add some s to it; the code to do so would look like this:

```
// Add some LIs to an OL element
document.getElementsByTagName('ol')[0].innerHTML = "<li>Cats.</li><li>Dogs.</li><li>Mice.</li>";
```

Isn't that so much simpler than obsessively creating a number of DOM elements and their associated text nodes? You'll be happy to know that (according to http://www.quirksmode.org) it's much faster than using the DOM methods, too. It's not all perfect, however—there are a number of tricky problems that exist with using the innerHTML injection method:

- As mentioned previously, the innerHTML method doesn't exist in XML DOM documents, meaning that you'll have to continue to use the traditional DOM creation methods.

- XHTML documents that are created using client-side XSLT don't have an innerHTML method, as they, too, are pure XML documents.

- innerHTML completely removes any nodes that already exist inside the element, meaning that there's no way to conveniently append or insert before, as we can with the pure DOM methods.

The last point is especially troublesome, as inserting before another element or appending onto the end of a child list is a particularly useful feature. Let's look at how it can be done in Listing 5-16 using the same methods we were using before.

Listing 5-16. Adding New DOM Nodes to an Existing Ordered List

```
document.addEventListener('DOMContentLoaded', activateButtons);

function activateButtons(){
    //ad event listeners to buttons
    var appendBtn = document.querySelector('#appendButon');
        appendBtn.addEventListener('click', appendNode);

    var addBtn = document.querySelector('#addButton');
        addBtn.addEventListener('click', addNode);
}

function appendNode(e){

    //get the <li>s that exist and make a new one.
    var listItems = document.getElementsByTagName('li');
    var newListItem = document.createElement('li');
        //append a new text node
        newListItem.appendChild(document.createTextNode('Mouse trap.'));

        //append to existing list as the new 4th item
        listItems[2].appendChild(newListItem);
}

function addNode(e){

    //get the whole list
     var orderedList = document.getElementById('myList');

     //get all the <li>s
    var listItems = document.getElementsByTagName('li');
    //make a new <li> and attach text node
    var newListItem = document.createElement('li');
        newListItem.appendChild(document.createTextNode('Zebra.'));
        //add to list, pushing the 2nd one down to 3rd
        orderedList.insertBefore(newListItem,listItems[1]);
}
```

Using this example you can see that it is not very difficult to make changes to an existing document. However, what if you want to move the other way and remove nodes from the DOM? As always, there's another method to handle that, too.

Removing Nodes from the DOM

Removing nodes from the DOM occurs nearly as frequently as creating and inserting them. When you're creating a dynamic form asking for an unlimited number of items, for example, it becomes important to allow the user to be able to remove portions of the page that they no longer wish to deal with. The ability to remove a node is encapsulated into one function: removeChild. It's used just like appendChild, but it has the opposite effect. The function in action looks something like this:

```
NodeParent.removeChild( NodeToRemove );
```

With this in mind, you can create two separate functions to quickly remove nodes. The first removes a single node, as shown in Listing 5-17.

Listing 5-17. Function for Removing a Node from the DOM

```
// Remove a single Node from the DOM
function remove( elem ) {
    if ( elem ) elem.parentNode.removeChild( elem );
}
```

Listing 5-18 shows a function for removing all child nodes from an element, using only a reference to the DOM element.

Listing 5-18. A Function for Removing All Child Nodes from an Element

```
// Remove all of an Element's children from the DOM
function empty( elem ) {
    while ( elem.firstChild )
        remove( elem.firstChild );
}
```

As an example, let's say you want to remove an <li> that you added in a previous section, assuming that you've already given the user enough time to view the <li> and that it can be removed without implication. Listing 5-19 shows the JavaScript code that you can use to perform such an action, creating a desirable result.

Listing 5-19. Removing Either a Single Element or All Elements from the DOM

```
// Remove the last <li> from an <ol>
var listItems = document.getElementsByTagName('li');
remove(listItems[2]);

// The preceding will convert this:
<ol>
    <li>Learn Javascript.</li>
    <li>???</li>
    <li>Profit!</li>
</ol>
// Into this:
<ol>

    <li>Learn Javascript.</li>
    <li>???</li>
</ol>
```

68

```
// If we were to run the empty() function instead of remove()
var orderedList = document.getElementById('myList');
empty(orderedList);
// It would simply empty out our <ol>, leaving:
<ol></ol>
```

Handling White Space in the DOM

Let's go back to our example HTML document. Previously, you attempted to locate the single `<h1>` element and had difficulty because of the extraneous text nodes. That may be acceptable for one single element, but what if you want to find the next element after the `<h1>` element? You still hit the infamous white space bug, and you'll need to do `.nextSibling.nextSibling` to skip past the end lines between the `<h1>` and the `<p>` elements. All is not lost, though. There is one technique that can act as a workaround for the white-space bug, shown in Listing 5-20. This particular technique removes all white-space–only text nodes from a DOM document, making it easier to traverse. Doing this will have no noticeable effect on how your HTML renders, but it will make it easier for you to navigate by hand. It should be noted that the results of this function are not permanent and will need to be rerun every time the HTML document is loaded.

Listing 5-20. A Workaround for the White-Space Bug in XML Documents

```
function cleanWhitespace( element ) {
    // If no element is provided, do the whole HTML document
    element = element || document;
    // Use the first child as a starting point
    var cur = element.firstChild;
    // Go until there are no more child nodes
    while ( cur != null ) {
        // If the node is a text node, and it contains nothing but whitespace
        if ( cur.nodeType == 3 && ! /\S/.test(cur.nodeValue) ) {
            // Remove the text node
            element.removeChild( cur );
        // Otherwise, if it's an element
        } else if ( cur.nodeType == 1 ) {
            // Recurse down through the document
            cleanWhitespace( cur );
        }
        cur = cur.nextSibling; // Move through the child nodes
    }
}
```

Let's say that you want to use this function in your example document to find the element after the first `<h1>` element. The code to do so would look something like this:

```
cleanWhitespace();
// Find the H1 Element
document.documentElement
    .firstChild        // Find the Head Element
    .nextSibling       // Find the <body> Element
    .firstChild        // Get the H1 Element
    .nextSibling       // Get the adjacent Paragraph
```

This technique has both advantages and disadvantages. The greatest advantage is that you get to maintain some level of sanity when trying to navigate your DOM document. However, this technique is particularly slow, considering that you have to traverse every single DOM element and text node looking for the text nodes that contain nothing but white space. If you have a document with a lot of content in it, it could significantly slow down the loading of your site. Additionally, every time you inject new HTML into your document, you'll need to rescan that portion of the DOM, making sure that no additional space-filled text nodes were added.

One important aspect of this function is the use of node types. A node's type can be determined by checking its `nodeType` property for a particular value. We have a list of types at the beginning of this chapter. So you can see are a number of possible values, but the three that you'll encounter the most are the following:

> Element (`nodeType = 1`): This matches most elements in an XML file.
> For example, `<li>`, `<a>`, `<p>`, and `<body>` elements all have a `nodeType` of 1.
>
> Text (`nodeType = 3`): This matches all text segments within your document.
> When navigating through a DOM structure using `previousSibling` and
> `nextSibling`, you'll frequently encounter pieces of text inside and between
> elements.
>
> Document (`nodeType = 9`): This matches the root element of a document.
> For example, in an HTML document it's the `<html>` element.

Additionally, you can use constants to refer to the different DOM node types (in version 9 of IE and later). For example, instead of having to remember 1, 3, or 9, you could just use `document.ELEMENT_NODE`, `document.TEXT_NODE`, or `document.DOCUMENT_NODE`. Since constantly cleaning the DOM's white space has the potential to be cumbersome, you should explore other ways to navigate a DOM structure.

Simple DOM Navigation

Using the principle of pure DOM navigation (having pointers in every navigable direction), you can develop functions that might better suit you when navigating an HTML DOM document. This particular principle reflects the fact that most web developers only need to navigate around DOM elements and very rarely navigate through sibling text nodes. To aid you, there are a number of helpful functions that can be used in place of the standard `previousSibling`, `nextSibling`, `firstChild`, `lastChild`, and `parentNode`. Listing 5-21 shows a function that returns the element previous to the current element, or null if no previous element is found, similar to the `previousSibling` element property.

Listing 5-21. A Function for Finding the Previous Sibling Element in Relation to an Element

```
function prev( elem ) {
    do {
        elem = elem.previousSibling;
    } while ( elem && elem.nodeType != 1 );
    return elem;
}
```

Listing 5-22 shows a function that returns the element next to the current element, or null if no next element is found, similar to the `nextSibling` element property.

Listing 5-22. A Function for Finding the Next Sibling Element in Relation to an Element

```
function next( elem ) {
    do {
        elem = elem.nextSibling;
    } while ( elem && elem.nodeType != 1 );
    return elem;
}
```

Listing 5-23 shows a function that returns the first element child of an element, similar to the `firstChild` element property.

Listing 5-23. A Function for Finding the First Child Element of an Element

```
function first( elem ) {
    elem = elem.firstChild;
    return elem && elem.nodeType != 1 ?
        next ( elem ) : elem;
}
```

Listing 5-24 shows a function that returns the last element child of an element, similar to the `lastChild` element property.

Listing 5-24. A Function for Finding the Last Child Element of an Element

```
function last( elem ) {
    elem = elem.lastChild;
    return elem && elem.nodeType != 1 ?
        prev ( elem ) : elem;
}
```

Listing 5-25 shows a function that returns the parent element of an element, similar to the `parentNode` element property. You can optionally provide a number to navigate up multiple parents at a time—for example, `parent(elem,2)` is equivalent to `parent(parent(elem))`.

Listing 5-25. A Function for Finding the Parent of an Element

```
function parent( elem, num ) {
    num = num || 1;
    for ( var i = 0; i < num; i++ )
        if ( elem != null ) elem = elem.parentNode;
    return elem;
}
```

Using these new functions you can quickly browse through a DOM document without having to worry about the text in between each element. For example, to find the element next to the <h1> element, as before, you can now do the following:

```
// Find the Element next to the <h1> Element
next( first( document.body ) )
```

You should notice two things with this code. First, there is a new reference: `document.body`. All modern browsers provide a reference to the `<body>` element inside the body parameter of an HTML DOM document. You can use this to make your code shorter and more understandable. The other thing you might notice is that the way the functions are written is very counterintuitive. Normally, when you think of navigating you might say, "Start at the `<body>` element, get the first element, and then get the next element," but with the way it's physically written, it seems backward.

Summary

In this chapter we talked a lot about what the DOM is and how it's structured. We also covered relationships between nodes, node types and how to access elements using JavaScript. When we have access to these elements we can change the attributes of them by using `element.get/setAttribute()`. We also discussed creating and adding new nodes to the DOM, handing white space, and navigating though the DOM. In the next chapter we will talk about events with JavaScript.

CHAPTER 6

■ ■ ■

Events

It was the best of times; it was the worst of times. It was the age of Netscape; it was the age of Internet Explorer. The new event handlers before us made it the spring of hope. The fact that browsers implemented event handling differently left us in a winter of despair. But in recent years, the sun has shone clear and bright, and the event-handling API has standardized across browsers (most aspects of the API, anyway). The ultimate goal of writing usable JavaScript code has always been to have a web page that will work for the users, no matter what browser they use or what platform they are on. For too long, this has meant writing event-handling code that managed two different event-handling models. But with the advent of modern browsers, we developers never have to worry about that again.

The concept of events in JavaScript has advanced through the years to the reliable, usable plateau where we now stand. Once Internet Explorer implemented the W3C model for event handling in version 8, we could stop writing libraries for managing differences between browsers and instead focus on doing interesting and amazing things with events. Eventually, this leads us toward the powerful Model-View-Controller (MVC) model for JavaScript, which we will discuss in a later chapter.

In this chapter we will start with an introduction to how events work in JavaScript. Following this theory with a practical application, we will look at how to bind events to elements. Then we will examine the information the event model provides and how you can best control it. Of course, we also need to cover the types of events available to us. We conclude with event delegation and a few bits of advice about events and best practices.

Introduction to JavaScript Events

If you look at the core of any JavaScript code, you'll see that events are the glue that holds everything together. Whether using a full MVC-based single-page application or simply using JavaScript to add some functionality to a page or two, event handlers are how the user communicates with our code. Our data will be bound in JavaScript, probably as object literals. We will represent this data in the DOM, using it as our view. Events, raised from the DOM, handled by JavaScript code, capture user interactions and guide the flow of our application. The combination of using the DOM and JavaScript events is the fundamental union that makes all modern web applications possible.

The Stack, the Queue, and the Event Loop

In many programming languages, including JavaScript, there are metaphors which describe the flow of control, elements in memory, and planning for what happens next. The code we run, whether from the global context, directly as a function, or as a function called from (or within!) another function, is known as the *stack*. If you are running a function foo, which calls a function bar, then the stack is three *frames* deep (global, foo, and then bar). What's up after this code runs? That's the province of the *queue*, which manages

the next set of code to run after the current stack is resolved. Any time the stack empties, it goes to the queue and picks up a new bit of code to run. These are the elements that are critical to our understanding of events. There is a third element, though: the *heap*. This is where variables and functions and other named objects live. When JavaScript needs to access an object, a function, or a variable, it goes to the heap to get access to the information. For us, the heap is less relevant, as it does not play as big a role in event handling as the stack and the queue do.

How do the stack and the queue factor into event handling? To answer this question, we need to introduce the event loop. This is a collaboration between two threads in your browser: the event-tracking thread, and the JavaScript thread.

■ **Note**   Remember that, except for web workers, JavaScript is single-threaded.

These threads work together to capture user events and then sort them according to the events for which we have registered events handlers. This process is collectively known as the event loop. Each time it runs, user events are checked to see if there are event handlers registered against them. If not, then nothing happens. If there are event handlers, the loop pushes them onto the top of JavaScript's queue, so that the handler is executed at JavaScript's earliest convenience.

And there's the rub. The queue manages the notion of "earliest convenience." Generally, this means after the current stack has been resolved. This may give event handling an asynchronous feel, particularly if the stack is many frames deep or contains long-running code. Events are allowed to jump to the head of the queue, but they may not interrupt the stack. Most of the time, the distinction is immaterial to developers, because the duration between when an event fires, a stack frame resolves, and event-handling code runs may not be perceptible in human terms. Nonetheless, it is important for us to understand that the event loop only jumps events to the front of the line; it does not push currently running code out of the way.

We now understand how the browser, the queue, and the stack work together to determine when an event handler will run. Soon, we will look at the mechanics of binding events to event handlers. But there is one architectural issue we need to cover first. Consider this: If you click a link in a list item in an unordered list in a paragraph in a div in the body of your HTML document, which of those elements handles that event? Could more than one element handle the event? If so, which element gets the event first? To answer this question, we'll need to look at event phases.

Event Phases

JavaScript events are executed in two phases, called *capturing* and *bubbling*. What this means is that when an event is fired from an element (for example, the user clicking a link, causing the click event to fire), the elements that are allowed to handle it, and in what order, vary. You can see an example of the execution order in Figure 6-1. It shows which event handlers are fired and in what order, whenever a user clicks the first <a> element on the page.
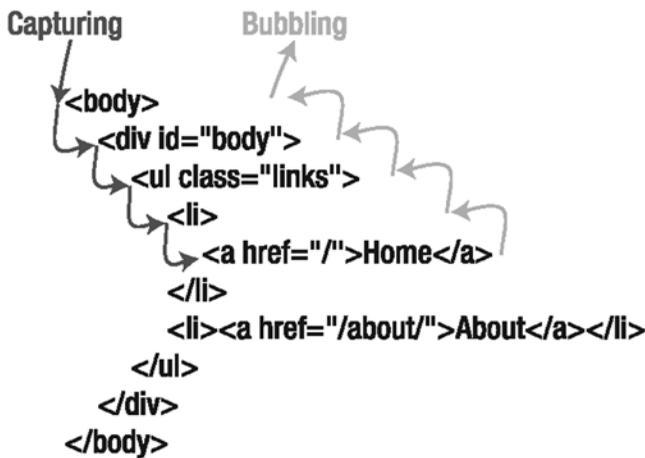
Figure 6-1. *The two phases of event handling*

Looking at this simple example of someone clicking a link, you can see the order of execution for an event. Imagine that the user clicked the `<a>` element; the click handler for the document is fired first, then the `<body>`'s handler, then the `<div>`'s handler, and so on, down to the `<a>` element, a cycle called the capturing phase. Once that finishes, it moves back up the tree again, and the `<li>`, `<ul>`, `<div>`, `<body>`, and document event handlers are all fired, in that order.

There are very specific historical reasons why event handling is built this way. When Netscape introduced event handling, it decided that event capturing should be used. When Internet Explorer caught up with its own version of event handling, it went with event bubbling. It was the time of the browser wars, and diametrically opposed architectural choices like this were commonplace. They hampered development of JavaScript for years, as programmers had to waste time maintaining libraries that normalized event handling (and some of the DOM, and Ajax, and a few other things!).

The good news is that we now live in the future. Modern browsers allow users to choose at which phase to capture events. In fact, you can assign event handlers at both phases, if you so choose. It's a brave new world.

Regardless of at which phase you bind events, two things should be immediately apparent. First, we discussed the idea that you might click on an anchor tag within a list item. Shouldn't that send you off to wherever the `href` attribute for the link points? Maybe there is some way to override that behavior. Additionally, consider the general premise of event phases: whether capturing or bubbling, an event is communicated through the DOM hierarchy. What if we do not want that event to be communicated? Can we prevent an event from being passed up (or down) the hierarchy?

But we are getting ahead of ourselves. We have not even discussed how to bind event listeners yet! Let's take care of that right now.

Binding Event Listeners

The best way to bind event handlers to elements has been a constantly evolving quest in JavaScript. It began with browsers forcing users to write their event-handler code inline, in the HTML document. First efforts are thought of as drafts or alpha code for a reason! It turns out that, later on, when we want to follow established best practices like separating logic from presentation, using in-line event handlers is, let's say suboptimal. OK, it's severely problematic. Try to imagine managing a codebase where half of your critical paths depend on code embedded in your presentation layer. Not something that the professional JavaScript programmer wants to do! Thankfully, that technique has been obviated by evolving browser APIs, as well as evolving standards of best practices.

When Netscape and Internet Explorer were actively competing with each other, they each developed separate, but very similar, event registration models. In the end, Netscape's model was modified to become a W3C standard, and Internet Explorer's stayed the same. Until Internet Explorer 9, that is, when Microsoft finally caved and implemented what is generally called W3C event handling. In fact, it went further and deprecated its older API for event handling. This was a boon for developers, as now we no longer had to write and maintain libraries dealing with the quibbling difficulties between browsers.

Today, there are two ways of reliably registering events. The traditional method is an offshoot of the old inline way of attaching event handlers, but it's reliable and works consistently, even on older browsers. The other method is to use the W3C standard for registering events. We will, of course, look at both, as you are likely to encounter both.

Traditional Binding

The traditional way of binding events is the simplest way of binding event handlers. This takes advantage of the fact that event handlers are properties of DOM elements. To use this method, you attach a function as a property to the DOM element that you wish to watch. Retrieve an element with `document.getElementById` (or any of the other element-retrieving functions we discussed in Chapter 5). Let's assume that you want to watch for `click` events. Simply assign a function to the `onclick` property of the retrieved element. Done!

For the examples in this chapter, we will use a standard HTML page with many targetable elements. The content for the page is presented in Listing 6-1.

Listing 6-1. Example HTML Code Used for Event Handling

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title>Event Handling</title>
  <link rel="stylesheet" href="school.css"/>
</head>
<body>

<div id="main">
  <nav id="navbar">
    <ul>
      <li>Students
        <ul>
          <li id="Academics">Academics</li>
          <li id="Athletics">Athletics</li>
          <li id="Extracurriculars">Extracurriculars</li>
        </ul>
      </li>
      <li>Faculty
        <ul>
          <li id="Frank Walsh">Frank Walsh</li>
          <li id="Diane Walsh">Diane Walsh</li>
          <li id="John Mullin">John Mullin</li>
          <li id="Lou Garaventa">Lou Garaventa</li>
```

```
      <li id="Dan Tully">Dan Tully</li>
       <li id="Emily Su">Emily Su</li>
      </ul>
    </li>
  </ul>
 </nav>
 <div id="welcome">
   <h1>Welcome to the School of JavaScript</h1>
   <h3 id="welcome-header">Click here for a welcome message!</h3>
   <p id="welcome-content">Welcome to the School of JavaScript. Here, you will find many
    <a href="/examples" id="examples-link">examples</a> of JavaScript,
    taught by our most esteemed <a href="/faculty">faculty</a>. <span
    id="disclaimer">Please note that these are only examples, and are not
    necessarily <a href="/production-ready">production-ready code</a>.</span></p>
 </div>
 <hr/>
 <div id="form-container">
   <h2>Contact Form</h2>

   <p>Thank you for your interest in the School of JavaScript. Please fill out the form
   below so we can send you even more materials!</p>

   <form id="main-form">
     <ul>
       <li><label for="firstName">First Name: </label><input id="firstName" type="text"/></li>
       <li><label for="lastName">Last Name: </label><input id="lastName" type="text"/></li>
       <li><label for="city">City: </label><input id="city" type="text"/></li>
       <li><label for="state">State: </label><input id="state" type="text"/></li>
       <li><label for="postCode">Postal Code: </label><input id="postCode" type="text"/></li>
       <li><label for="comments">Comments: </label><br/>
         <textarea name="" id="comments" cols="30" rows="10"></textarea>
       </li>
       <li><input type="submit"/> <input type="reset"/></li>
     </ul>
   </form>
 </div>
</div>

</body>
</html>
```

As you can see, there are many elements in the front page for our imaginary School of JavaScript. The navbar will eventually have appropriate event handling to function as a menu, we will add event handling to the form for simple validation (with more complex validations coming in Chapter 8), and we also plan to have a bit of interactivity in the welcome message.

For now, let's do something simple. When we click into the firstName field, let's record that on the console, and then set a yellow background for our element. Obviously, we are planning to do more soon, but baby steps first! Let's bind this event with traditional event handling (Listing 6-2).

Listing 6-2. Binding a Click Event the Traditional Way

```
// Retrieve the firstName element
var firstName = document.getElementById('firstName');

// Attach the event handler
firstName.onclick = function() {
    console.log('You clicked in the first name field!');
    firstName.style.background = 'yellow';
};
```

Terrific! It works. But it lacks a certain something. That something is flexibility. At this rate, we would have to write a separate event-handling function for each of the form fields. Tedious! Could there be a way to get a reference to the element that fired the event?

In fact, there are two ways! The first, and most straightforward, is to provide an argument in your event-handling function, as shown in Listing 6-3. This argument is the event object, which contains information about the event that just fired. We will be looking at the event object in greater detail shortly. For now, know that the target property of that event object refers to the DOM element that emitted the event in the first place.

Listing 6-3. Event Binding with an Argument

```
// Retrieve the firstName element
var firstName = document.getElementById('firstName');

// Attach the event handler
firstName.onclick = function(e) {
    console.log('You clicked in the ' + e.target.id + ' field!');
    e.target.style.background = 'yellow';
};
```

Since e.target points to the firstName field, and is, in fact, a reference to the DOM element for the firstName field, we can check its id property to see what field we clicked in. More importantly, we can change its style property as well! This means we can broaden this event handler to work on just about any of the text fields in the form.

There is an alternative to using the event object explicitly. We could also use the this keyword in the function, as shown in Listing 6-4. In the context of an event-handling function, this refers to the emitter of the event. Put another way, event.target and this are synonymous, or, at least, they point to the same thing.

Listing 6-4. Event Binding Using the this Keyword

```
// Retrieve the firstName element
var firstName = document.getElementById('firstName');

// Attach the event handler
firstName.onclick = function() {
    console.log('You clicked in the ' + this.id + ' field!');
    this.style.background = 'yellow';
};
```

Which should you use? The event object gives you all of the information you need, while the this object is somewhat limited, as it only points to the DOM element that emitted the event. There is no cost in using one over the other, so, in general, we recommend preferring the event object, as you will always have all the details of the event immediately available. However, there are some cases where the this object is still useful. The target always refers to the closest element emitting the event. Look at the <div> with the ID welcome back in Listing 6-1. Let's say we added a mouseover event handler to change the background color when we hovered over the element, and a mouseout event handler to change the background color back when the mouse leaves the <div>. If you make the style change on e.target, the event will fire for each of the subelements (welcome-header, welcome-content, and more)! On the other hand, if you make the style change on this, the change is made only on the welcome <div>. When we discuss event delegation, we will cover this difference in greater detail.

Advantages of Traditional Binding

Traditional binding has the following advantages:

- The biggest advantage of using the traditional method is that it is incredibly simple and consistent, in that you're pretty much guaranteed that it will work the same no matter what browser you use it in.

- When handling an event, the this keyword refers to the current element, which can be useful (as demonstrated in Listing 6-4).

Disadvantages of Traditional Binding

It also has some disadvantages, however:

- The traditional method allows no control over event capturing or bubbling. All events bubble, and there is no possibility of changing to event capturing.

- It's only possible to bind one event handler to an element at a time. This has the potential to cause confusing results when working with the popular window.onload property (effectively overwriting other pieces of code that have used the same method of binding events). An example of this problem is shown in Listing 6-5, where an event handler overwrites an earlier event handler.

Listing 6-5. Event Handlers Overwriting Each Other

```
// Bind your initial load handler
window.onload = myFirstHandler;

// somewhere, in another library that you've included,
// your first handler is overwritten
// only 'mySecondHandler' is called when the page finishes loading
window.onload = mySecondHandler;
```

- The event object argument is not available in Internet Explorer 8 and older. Instead, you would have to use window.event.

Knowing that it's possible to blindly override other events, you should probably opt to use the traditional means of event binding only in simple situations, where you can trust all the other code that is running alongside yours. One way to get around this troublesome mess, however, is to use the W3C event-binding method implemented by modern browsers.

DOM Binding: W3C

The W3C's method of binding event handlers to DOM elements is the only truly standardized means of doing so. With that in mind, every modern browser supports this way of attaching events. Internet Explorer 8 and older do not, but old versions of Internet Explorer are hardly modern browsers. If you must design for those, consider using traditional binding.

The code for attaching a new handler function is simple. It exists as a function available on every DOM element. The function is named addEventListener and takes three parameters: the name of the event (such as click; note the lack of the prefix on), the function that will handle the event, and a Boolean flag to enable or disable event capturing. An example of addEventListener in use is shown in Listing 6-6.

Listing 6-6. Sample Code That Uses the W3C Way of Binding Event Handlers

```
// Retrieve the firstName element
var firstName = document.getElementById( 'firstName' );

// Attach the event handler
firstName.addEventListener( 'click', function ( e ) {
  console.log( 'You clicked in the ' + e.target.id + ' field!' );
  e.target.style.background = 'yellow';
} );
```

Note that in this example, we are not passing a third argument to addEventListener. In this case, the third argument defaults to false, meaning that event bubbling will be used. If we had wanted to use event capturing, we could have passed a true value explicitly.

Advantages of W3C Binding

The advantages to the W3C event-binding method are the following:

- This method supports both the capturing and bubbling phases of event handling. The event phase is toggled by setting the last parameter of addEventListener to false (the default, for bubbling) or true (for capturing).

- Inside the event-handler function, the this keyword refers to the current element, as it did in traditional event handling.

- The event object is always available in the first argument of the handling function.

- You can bind as many events to an element as you wish, with no overwriting previously bound handlers. Handlers are stacked internally by JavaScript and run in the order that they were registered.

Disadvantage of W3C Binding

The W3C event-binding method has only one disadvantage:

- It does not work in Internet Explorer 8 and older. IE uses attachEvent with a similar syntax.

Unbinding Events

Now that we have bound events, what if we want to unbind events? Perhaps that button we tied a `click` event handler to is now disabled. Or we no longer need to highlight that div when hovering over it. Disconnecting an event and its handler is relatively simple.

For traditional event handling, simply assign an empty string or null to the event handler, as shown here:

```
document.getElementById('welcome-content').onclick = null;
```

Not too difficult, right?

The situation with W3C event handling is somewhat more complex. The relevant function is `removeEventListener`. Its three arguments are the same: the type of event to remove, the associated handler, and a true/false value for capture or bubble mode. There is a catch, though. First and foremost, the function must be a reference to the same function that was assigned with `addEventListener`. Not just the same lines of code, but the same reference. So if you assigned an anonymous, in-line function with `addEventListener`, you cannot remove it.

■ **Tip**   You should always use a named function for an event handler if you think you might need to remove that handler later on.

In a similar vein, if you set the third argument when you originally invoked `addEventListener`, you must set it again in `removeEventListener`. If you leave the argument off, or give it the wrong value, `removeEventListener` will silently fail. Listing 6-7 has an example of unbinding an event handler.

Listing 6-7. Unbinding an Event Handler

```javascript
// Assume we have two buttons 'foo' and 'bar'
var foo = document.getElementById( 'foo' );
var bar = document.getElementById( 'bar' );

// When we click on foo, we want to log to the console "Clicked on foo!"
function fooHandler() {
  console.log( 'Clicked on the foo button!' );
}

foo.addEventListener( 'click', fooHandler );

// When we click on bar, we want to _remove_ the event handler for foo.
function barHandler() {
  console.log( 'Removing event handler for foo....' );
  foo.removeEventListener( 'click', fooHandler );
}

bar.addEventListener( 'click', barHandler );
```

Common Event Features

JavaScript events have a number of relatively consistent features that give you more power and control when developing. The simplest and oldest concept is that of the event object, which provides you with a set of metadata and contextual functions so you can deal with things such as mouse events and key presses. Additionally, there are functions that can be used to modify the normal capture/bubbling flow of an event. Learning these features inside and out can make your life much simpler.

The Event Object

One standard feature of event handlers is some way to access an event object, which contains contextual information about the current event. This object serves as a very valuable resource for certain events. For example, when handling key presses you can access the keyCode property of the object to get the specific key that is pressed. There are some subtle differences between event objects, but we will address those later in the chapter. For now, let us address two dangling issues: event propagation and default behavior.

Canceling Event Bubbling

You know how event capturing/bubbling works, so let's explore how you can take control of it. An important point brought up in the previous example is that if you want an event to occur only on its target and not on its parent elements, you have no way to stop it. Stopping the flow of an event bubble would cause an occurrence similar to what is shown in Figure 6-2, where the result of an event is captured by the first <a> element and the subsequent bubbling is canceled.
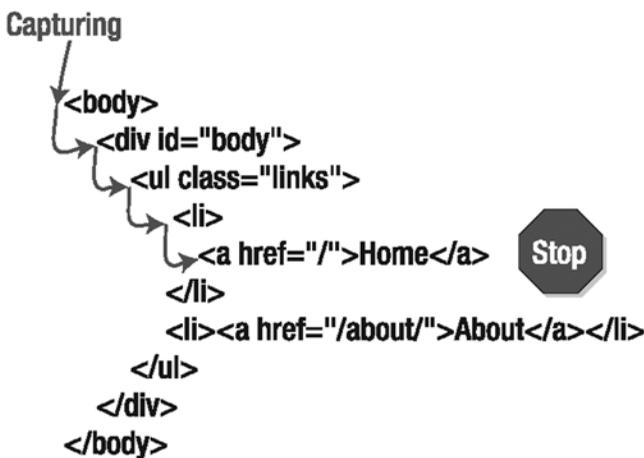
Figure 6-2. *The result of an event being captured by the first* <a> *element*

Stopping the bubbling (or capturing) of an event can prove immensely useful in complex applications. And it's simple to implement. Invoke the event object's stopPropagation method to prevent the event from traversing further up (or down) the hierarchy. Listing 6-8 shows an example.

Listing 6-8. An Example of Stopping Event Bubbling

```
document.getElementById( 'disclaimer' ).addEventListener( 'click', function ( e ) {

  // When clicking on the disclaimer, highlight it by making it bold
  e.target.style.fontWeight = 'bold';

  // The parent element wants to hide itself if this element is clicked on. We need to
prevent that behavior
  e.stopPropagation();
} );

document.getElementById( 'welcome-content' ).addEventListener( 'click', function ( e ) {
  e.target.style.visibility = 'hidden';
} );
```

Listing 6-9 shows a brief snippet that adds a red border around the element that the user is hovering over. You do this by adding a mouseover and a mouseout event handler to every DOM element. If you don't stop the event bubbling, every time the mouse is moved over an element, the element and all of its parent elements will have the red border, which isn't what you want.

Listing 6-9. Using stopPropagation to Keep All the Elements from Changing Color

```
// Event handling functions
function mouseOverHandler( e ) {
  e.target.style.border = '1px solid red';
  e.stopPropagation();
}

function mouseOutHandler( e ) {
  this.style.border = '0px';
  e.stopPropagation();
}

// Locate, and traverse, all the elements in the DOM
var all = document.getElementsByTagName( '*' );
for ( var i = 0; i < all.length; i++ ) {

  // Watch for when the user moves the mouse over the element
  // and add a red border around the element
  all[i].addEventListener( 'mouseover', mouseOverHandler );

  // Watch for when the user moves back out of the element
  // and remove the border that we added
  all[i].addEventListener( 'mouseout', mouseOutHandler );

}
```

With the ability to stop the event bubbling, you now have complete control over which elements can see and handle an event. This is a fundamental tool necessary for exploring the development of dynamic web applications. The final aspect is to cancel the default action of the browser, allowing you to completely override what the browser does and implement new functionality instead.

Overriding the Browser's Default Action

For most events that take place, the browser has some default action that will always occur. For example, clicking an `<a>` element will take you to its associated web page; this is a default action in the browser. This action will always occur after both the capturing and the bubbling event phases, as shown in Figure 6-3, which illustrates the results of a user clicking an `<a>` element in a web page. The event begins by traveling through the DOM in both a capturing and bubbling phase (as discussed previously). However, once the event has finished traversing, the browser attempts to execute the default action for that event and element. In this case, it's visiting the / web page.

Figure 6-3. *The full life cycle of an event*

Default actions can be summarized as anything the browser does that you did not explicitly tell it to do. Here's a sampling of the different types of default actions that occur, and on what events:

- Clicking an `<a>` element will redirect you to a URL provided in its `href` attribute.

- Using your keyboard and pressing Ctrl+S, the browser will attempt to save a physical representation of the site.

- Submitting an HTML `<form>` will submit the query data to the specified URL and redirect the browser to that location.

- Moving your mouse over an `<img>` with an `alt` or a `title` attribute (depending on the browser) will cause a tool tip to appear, providing the value of the attribute.

All of the previous actions are executed by the browser even if you stop the event bubbling or if you have no event handler bound at all. This can lead to significant problems in your scripts. What if you want your submitted forms to behave differently? Or what if you want `<a>` elements to behave differently than their intended purpose? Because canceling event bubbling isn't enough to prevent the default action, you need some specific code to handle that directly. The W3C event handling API provides this functionality with the `preventDefault` method of the event object (Listing 6-10). With many browsers you can choose to simply return false from your event handler as an alternative, and you may see this behavior coded in some examples and libraries. Using `preventDefault` is preferred, though, as it is self-documenting—unlike the somewhat obscure technique of occasionally returning false from an event handler.

Listing 6-10. A Generic Function for Preventing the Default Browser Action from Occurring

```
document.getElementById('examples-link').addEventListener('click', function(e) {
  e.preventDefault();
    console.log("examples-link clicked");
});
```

Using the `preventDefault` function, you can now stop any default action presented by the browser. For example, this allows you to take advantage of `mouseover` events for a link, without worrying about the user accidentally clicking on the link and sending the browser elsewhere. And you can override the default behavior of showing where the link goes in the status bar. Or consider a Submit button that you want to use to kick off form validation. You can now hold off on submitting the form (the default behavior) if that validation fails.

Event Delegation

We have almost all the tools in place to manipulate event handlers. The one lingering problem is a matter of technique. Presume that we have an unordered list with 20 items in it. We want to add an event handler for each list item. More accurately, we want to be able to handle clicks from each list item differently. We could grab all of the elements with `document.querySelectorAll`, iterate over the results, and attach individual event handlers. This is inefficient both as a process and in the browser. We are setting up 20 event handlers (even if they all point to the same handling function) when we could set up just one.

All of the list items are contained within an unordered list tag, so why not take advantage of the fact that we can capture click events at the `<ul>` level? The only thing we need is some way to differentiate between the various list items. Back in the section on traditional event binding, when we discussed the `this` object, we noted that `this` refers to the element where the event is captured, while `event.target` refers to the element that actually emits the event in question. Clearly, we could use a combination of `this` and `event.target`. But the event-handling specification provides the `event.currentTarget` property to solve this problem.

In our list item scenario, we attach a `click` event handler to the unordered list. In the event handler, the `<ul>` is `event.currentTarget`. Each list item will be the `event.target` property. Therefore, we can check `event.target` to see which list item was clicked and dispatch to the appropriate function. Listing 6-11 shows an example of event delegation in action.

Listing 6-11. Event Delegation

```
function clickHandler(e) {
  console.log( 'Handled at ' + e.currentTarget.id );
  console.log( 'Emitted by ' + e.target.id );
}

var navbar = document.getElementById('navbar');
navbar.addEventListener( 'click', clickHandler );
```

The `clickHandler` function handles events at the `<nav>` level, but it receives events emitted from a variety of list items under the `<nav>` element.

The Event Object

The event object is provided, or is available, inside every event-handler function. In general, the properties of the event object cover the details you might want to know about an event: what kind of event it was, where it originated from, what coordinates were clicked, or maybe what keys were pressed. There are some subtle differences between the ways different browsers communicate this information, though.

General Properties

A number of properties exist on the event object for every type of event being captured. All of these event object properties relate directly to the event itself, and nothing is browser-specific. What follows is a list of all the event object properties with explanations and example code.

type

This property contains the name of the event currently being fired (such as click or mouseover). It can be used to provide a generic event-handler function, which then deterministically executes related code. Listing 6-12 shows an example of using this property to make a handler have different effects depending on the event type.

Listing 6-12. Using the type Property to Provide Hoverlike Functionality for an Element

```
function mouseHandler(e){
  // Toggle the background color of the <div>, depending on the
  // type of mouse event that occurred
  this.style.background = (e.type === 'mouseover') ? '#EEE' : '#FFF';
}

// Locate the <div> that we want to hover over
var div = document.getElementById('welcome');

// Bind a single function to both the mouseover and mouseout events
div.addEventListener( 'mouseover', mouseHandler );
div.addEventListener( 'mouseout', mouseHandler );
```

target

This property contains a reference to the element that fired to the event. For example, binding a click handler to an <a> element would have a target property equal to the <a> element itself.

stopPropagation

The stopPropagation method stops the event bubbling (or capturing) process, making the current element the last one to receive the particular event.

preventDefault / returnValue = false

Calling the preventDefault method stops the browser's default action from occurring in all modern W3C-compliant browsers.

Mouse Properties

Mouse properties exist within the event object only when a mouse-related event is initiated (such as click, mousedown, mouseup, mouseover, mousemove, mouseout, mouseenter, and mouseleave). At any other time, you can assume that the values being returned do not exist or are not reliably present. This section lists all the properties that exist on the event object during a mouse event.

pageX and pageY

These properties contain the x- and y-coordinates of the mouse cursor relative to the absolute upper-left corner of the browser window. They will be the same regardless of scrolling.

clientXand clientY

These properties contain the x- and y-coordinates of the mouse cursor relative to the browser window. Therefore, if you have scrolled the document down (or across), the numbers are relative to the edges of the browser window. These numbers change as you scroll through your document.

layerX/layerY and offsetX/offsetY

These properties should contain the x- and y-coordinates of the mouse cursor relative to the event's target element. The offset* properties work in Chrome and IE, but not in Firefox. Firefox supports layerX and layerY, but they don't contain the same information. Instead, the layer* properties seem to be equivalent to the appropriate page* property.

button

This property, available only on the click, mousedown, and mouseup events, is a number representing the mouse button that's currently being clicked. Left clicks are 0 (zero), middle clicks are 1, right clicks are 2.

relatedTarget

This event property contains a reference to the element that the mouse has just left. More often than not, relatedTarget is used in situations where you need to use mouseover/mouseout, but you also need to know where the mouse just was, or where it is going. Listing 6-13 shows a variation on a tree menu (<ol> elements containing other <ol> elements) in which the subtrees display only the first time the user moves the mouse over the <li> subelement.

Listing 6-13. Using the relatedTarget Property to Create a Navigable Tree

```
// When DOMContent is ready, get the references to the elements.
document.addEventListener('DOMContentLoaded', init);

function init(){
    var top = document.getElementById("top");
    var bottom = document.getElementById("bottom");

    top.addEventListener("mouseover", onMouseOver);
    top.addEventListener("mouseout", onMouseOut);
```

```
    bottom.addEventListener("mouseover", onMouseOver);
    bottom.addEventListener("mouseout", onMouseOut);
}

function onMouseOut(event) {
  console.log("exited " + event.target.id + " for " + event.relatedTarget.id);
}

function onMouseOver(event) {
  console.log("entered " + event.target.id + " from " + event.relatedTarget.id);
}

// Sample HTML:
<style>
div > div {
  height: 128px;
  width: 128px;
}
#top    { background-color: red; }
#bottom { background-color: blue; }
</style>
<title>Untitled Document</title>
</head>

<body>


<div id="outer">
  <div id="top"></div>
  <div id="bottom"></div>
</div>
```

Keyboard Properties

Keyboard properties generally only exist within the event object when a keyboard-related event is initiated (such as keydown, keyup, and keypress). The exception to this rule is for the ctrlKey and shiftKey properties, which are available during mouse events (allowing the user to Ctrl+click an element). At any other time, you can assume that the values contained within a property do not exist or are not reliably present.

ctrlKey

This property returns a Boolean value representing whether the keyboard Ctrl key is being held down. The ctrlKey property is available for both keyboard and mouse events.

keyCode

This property contains a number corresponding to the different keys on a keyboard. The availability of certain keys (such as PageUp and Home) can vary, but generally speaking, all other keys work reliably. Table 6-1 is a reference for all of the commonly used keyboard keys and their associated key codes.

Table 6-1. *Commonly Used Key Codes*

Key	Key Code
Backspace	8
Tab	9
Enter	13
Space	32
Left arrow	37
Up arrow	38
Right arrow	39
Down arrow	40
0–9	48–57
A–Z	65–90

shiftKey

This property returns a Boolean value representing whether the keyboard Shift key is being held down. The `shiftKey` property is available for both keyboard and mouse events.

Types of Events

Common JavaScript events can be grouped into several categories. Probably the most commonly used category is that of mouse interaction, followed closely by keyboard and form events. The following list provides a broad overview of the different classes of events that exist and can be handled in a web application.

> **Loading and error events:** Events of this class relate to the page itself, observing its load state. They occur when the user first loads the page (the `load` event) and when the user finally leaves the page (the `unload` and `beforeunload` events). Additionally, JavaScript errors are tracked using the `error` event, giving you the ability to handle errors individually.

> **UI events:** These are used to track when users are interacting with one aspect of a page over another. With these tools you can reliably know when a user has begun input into a form element, for example. The two events used to track this are `focus` and `blur` (for when an object loses focus).

> **Mouse events:** These fall into two categories: events that track where the mouse is currently located (`mouseover`, `mouseout`), and events that track where the mouse is clicking (`mouseup`, `mousedown`, `click`).

Keyboard events: These are responsible for tracking when keyboard keys are pressed and within what context—for example, tracking keystrokes inside form elements as opposed to keystrokes that occur within the entire page. As with the mouse, three event types are used to track the keyboard: keyup, keydown, and keypress.

Form events: These relate directly to interactions that occur only with forms and form input elements. The submit event is used to track when a form is submitted; the change event watches for user input into an element; and the select event fires when a `<select>` element has been updated.

Page Events

All page events deal specifically with the function and status of the entire page. The majority of the event types handle the loading and unloading of a page (whenever a user visits the page and then leaves again).

load

The load event is fired once the page has completely finished loading; this event includes all images, external JavaScript files, and external CSS files. It's also available on most elements with a src attribute (img, script, audio, video, and so on). Load events do not bubble.

beforeunload

This event is something of an oddity, as it's completely nonstandard but widely supported. It behaves very similarly to the unload event, with an important difference. Within your event handler for the beforeunload event, if you return a string, that string will be shown in a confirmation message asking users if they wish to leave the current page. If they decline, they will be able to stay on the current page. Dynamic web applications, such as Gmail, use this to prevent users from potentially losing any unsaved data.

error

The error event is fired every time an error occurs within your JavaScript code. It can serve as a way to capture error messages and display or handle them gracefully. This event handler behaves differently than others, in that instead of passing in an event object, it includes a message explaining the specific error that occurred.

resize

Page events: The resize event occurs every time the user resizes the browser window. When the user adjusts the size of the browser window, the resize event will only fire once the resize is complete, not at every step of the way.

scroll

The scroll event occurs when the user moves the position of the document within the browser window. This can occur from keyboard presses (such as using the arrow keys, Page Up/Down, or the spacebar) or by using the scrollbar.

unload

This event fires whenever the user leaves the current page (for example, by clicking a link, hitting the Back button, or even closing the browser window). Preventing the default action does not work for this event (the next best thing is the beforeunload event).

UI Events

User interface events deal with how the user is interacting with the browser or page elements. The UI events can help you determine what elements on the page the user is currently interacting with and provide them with more context (such as highlighting or help menus).

focus

The focus event is a way of determining where the page cursor is currently located. By default, the focus is within the entire document; however, whenever a link or a form input element is clicked or tabbed into using the keyboard, it moves to that instead. (An example of this event is shown in Listing 6-14).

blur

The blur event occurs when the user shifts focus from one element to another (within the context of links, input elements, or the page itself). (An example of this event is shown in Listing 6-14).

Mouse Events

Mouse events occur when the user either moves the mouse pointer or clicks one of the mouse buttons.

click

A click event occurs when a user presses the left mouse button down on an element (see the mousedown event) and releases the mouse button (see the mouseup event) on the same element.

dblclick

The dblclick event occurs after the user has completed two click events in rapid succession. The rate of the double-click depends upon the settings of the operating system.

mousedown

The mousedown event occurs when the user presses down a mouse button. Unlike the keydown event, this event will only fire once while the mouse is down.

mouseup

The mouseup event occurs when the user releases the pressed mouse button. If the button is released on the same element that the button is pressed on, a click event also occurs.

mousemove

A mousemove event occurs whenever the user moves the mouse pointer at least one pixel on the page. The number of mousemove events fired (for a full movement of the mouse) depends on how fast the user is moving the mouse and how quickly the browser can keep up with the updates.

mouseover

The mouseover event occurs whenever the user moves the mouse into an element from another. To find which element the user has come from, use the relatedTarget property. This event is resource-intensive, as it can fire once for each pixel or subelement that it goes over. Prefer mouseenter, described shortly.

mouseout

The mouseout event occurs whenever the user moves the mouse outside an element. This includes moving the mouse from a parent element to a child element (which may seem unintuitive at first). To find which element the user is going to, use the relatedTarget property. This event is resource-intensive, as, paired with mouseover, it can fire many, many times. Prefer mouseleave, described shortly.

mouseenter

Functions like mouseover, but intelligently pays attention to where it is within an element. Will not fire again until it leaves the element's box.

mouseleave

Functions like mouseout, but intelligently pays attention to when it leaves an element.

Listing 6-14 shows an example of attaching pairs of events to elements to allow for keyboard-accessible (and mouse-accessible) web page use. Whenever the user moves the mouse over a link or uses the keyboard to navigate to it, the link will receive some additional color highlighting.

Listing 6-14. Creating a Hover Effect by Using the mouseover and mouseout Events

```
// mouseEnter handler
function mouseEnterHandler() {
  this.style.backgroundColor = 'blue';
}

// mouseLeave handler
function mouseLeaveHandler() {
  this.style.backgroundColor = 'white';
}

// Find all the <a> elements, to attach the event handlers to them
var a = document.getElementsByTagName('a');
for ( var i = 0; i < a.length; i++ ) {
```

```
    // Attach a mouseover and focus event handler to the <a> element,
    // which changes the <a>s background to blue when the user either
    // mouses over the link, or focuses on it (using the keyboard)
    a[i].addEventListener('mouseenter', mouseEnterHandler);
    a[i].addEventListener('focus', mouseEnterHandler);

    // Attach a mouseout and blur event handler to the <a> element
    // which changes the <a>s background back to its default white
    // when the user moves away from the link
    a[i].addEventListener('mouseleave', mouseLeaveHandler);
    a[i].addEventListener('blur', mouseLeaveHandler);

}
```

Keyboard Events

Keyboard events handle all instances of a user pressing keys on the keyboard, whether inside or outside a text input area.

keydown/keypress

The keydown event is the first keyboard event to occur when a key is pressed. If the user continues to hold down the key, the keydown event will continue to fire. The keypress event is a common synonym for the keydown event; they behave virtually identically, with one exception: if you wish to prevent the default action of a key being pressed, you must do it on the keypress event.

keyup

The keyup event is the last keyboard event to occur (after the keydown event). Unlike the keydown event, this event will only fire once when released (since you can't release a key for a long period of time).

Form Events

Form events deal primarily with <form>, <input>, <select>, <button>, and <textarea> elements, the staples of HTML forms.

select

The select event fires every time a user selects a different block of text within an input area, using the mouse. With this event, you can redefine the way a user interacts with a form.

change

The change event occurs when the value of an input element (this includes <select> and <textarea> elements) is modified by a user. The event fires only after the user has already left the element, letting it lose focus.

submit

The submit event occurs only in forms and only when a user clicks a Submit button (located within the form) or hits Enter/Return within one of the input elements. By binding a Submit handler to the form, and not a click handler to the Submit button, you'll be sure to capture all attempts to submit the form by the user.

reset

The reset event only occurs when a user clicks a Reset button inside a form (as opposed to the Submit button, which can be duplicated by hitting the Enter key).

Event Accessibility

The final piece to take into consideration when developing a purely unobtrusive web application is to make sure that your events will work even without the use of a mouse. By doing this, you help two groups of people: those in need of accessibility assistance (vision-impaired users), and people who don't like to use a mouse. (Sit down one day, disconnect your mouse from your computer, and learn how to navigate the Web using only the keyboard. It's a real eye-opening experience).

To make your JavaScript events more accessible, any time you use the click, mouseover, and mouseout events, you should strongly consider providing alternative nonmouse bindings. Thankfully, there are easy ways to remedy this situation quickly:

> **The click event:** One smart move on the part of browser developers was to make the click event work whenever the Enter key is pressed. This completely removes the need to provide an alternative to this event. One point to note, however, is that some developers like to bind click handlers to Submit buttons in forms to watch for when a user submits a web page. Instead of using that event, the developer should bind to the submit event on the form object, a smart alternative that works reliably.

> **The mouseover event:** When navigating a web page using a keyboard, you're actually changing the focus to different elements. By attaching event handlers to both the mouseover and focus events, you can make sure that you'll have an equivalent solution for both keyboard and mouse users.

> **The mouseout event:** Like the focus event for the mouseover event, the blur event occurs whenever the user's focus moves away from an element. You can then use the blur event as a way to simulate the mouseout event with the keyboard.

In reality, adding the ability to handle keyboard events, in addition to typical mouse events, is completely trivial. If nothing else, doing so can help keyboard-dependent users better use your site, which is a huge win for everyone.

Summary

In this chapter we started with an introduction to how events work in JavaScript and compared them to event models in other languages. Then you saw what information the event model provides and how you can best control it. We then explored binding events to DOM elements, and the different types of events that are available. We concluded with a discussion of event object properties, event types, and how to code for accessibility.

CHAPTER 7

■ ■ ■

JavaScript and Form Validation

It is inevitable that, when encountering a form, one considers the fate of the data for that form. One of the first practical applications of JavaScript was providing a way to validate data on the client side, instead of having to endure a round trip to the server. Form validation was a bit ad hoc at the time, with no practical API and no real integration with the browser. Instead, programmers bound together events and basic text manipulation to provide a handy user interface enhancement.

Fast-forward to the present day, and form validation is in much better shape. With modern browsers, we have an integrated validation API, which works with both HTML and CSS to provide an extensive set of form validation features. We also have regular expressions, which—for all their complications—are much better for data validation than, say, iterating over a string character by character.

Our concern in this chapter is JavaScript and forms. While we will focus on form validation, we will also look at general improvements in the ways JavaScript interacts with forms, as well as some newly available form-related APIs.

HTML and CSS Form Validation

As mentioned, form validation has come a long way since the early days of JavaScript. To really dive into the state of form validation, we will need to look at not only JavaScript, but also HTML5 and CSS. Let's start with the HTML side of things. In the last few years, HTML has evolved and added many new features, thanks to the hard work of the Web Hypertext Application Technology Working Group (WHATWG). This organization has pushed for the evolution and updating of HTML into what has come to be known as HTML5. Although the scope of the HTML5 specification means that we can't discuss the details here, you can find more information in *HTML5 Programmer's Reference* by Jonathan Reid (Apress, 2015).

Of particular note in HTML5 are the advances to the set of form controls. These changes have broadly fallen into two categories: addition of new controls or styles of controls (URL fields, date pickers, and the like), and form validation. Initially, our focus will be on the latter. Simple form validation has moved into HTML, without the need of any JavaScript whatsoever. This validation is available through the addition of certain attributes to form controls. A simple example is the required attribute, which pairs with input elements and forces the field to have value before the form can be submitted. Listing 7-1 is a basic example.

Listing 7-1. A Simple Form

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
</head>
<body>
<h2>A basic form</h2>
```

```
<p>Please provide your first and last names.</p>

<form>
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName"/><br/>
  </fieldset>
  <input type="submit" value="Submit the form"/> <input type="reset" value="Reset the
form"/>
</form>

</body>
</html>
```

Notice that in the form, we have an input field with an ID of firstName, which has added the aforementioned required attribute. Were we to attempt to submit the form without filling out this field, we would see a result something like Figure 7-1.
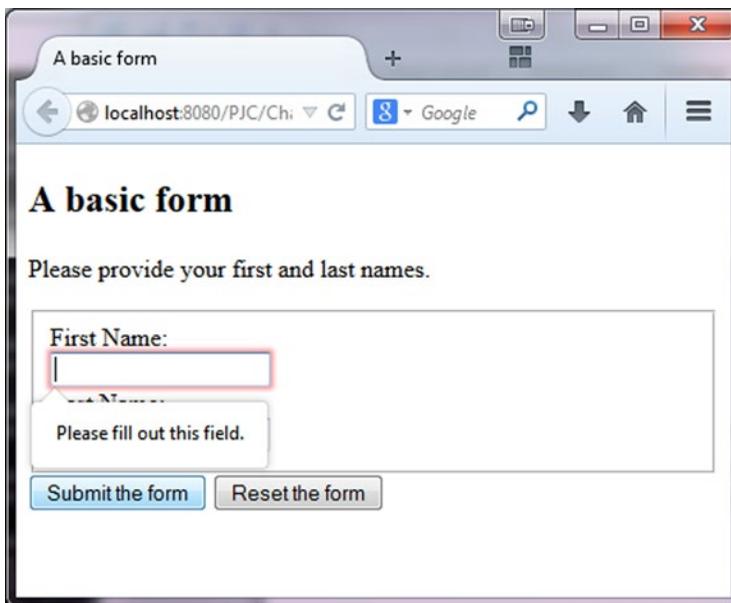
Figure 7-1. *First name is missing from this basic form*

The display looks roughly the same on Chrome and IE 11 (Chrome does not surround the field with a red border, but IE has a blockier, more assertive red border). If you were to make both the firstName and lastName fields required, the border would appear on each of the fields, but the popup tooltip would only be associated with the first field that had a problem. What about customizing the popup? We will deal with that soon, but it will require JavaScript.

There are several other types of validation that can be activated via HTML attributes. They are

- `pattern`: This attribute takes a regular expression as an argument. The regular expression does not need to be surrounded by slashes. The regular expression language engine is the same as JavaScript's (with the same issues as well). This is attached to the input element. Note that input types of `email` and `url` imply pattern values appropriate to valid email addresses and URLs, respectively. Pattern validation does not work with Safari 8, iOS Safari 8.1, or Opera Mini.

- `step`: Forces the value to match a multiple of the specified step value. Constrained to input types of `number`, `range`, or one of the date-times. Step validation works in Chrome 6.0, Firefox 16.0, IE 10, Opera 10.62, and Safari 5.0.

- `min` / `max`: Minimum or maximum values, appropriate to not only numbers but also date-times. This method works in Chrome 41, Opera 27, and Chrome for Android 41.

- `maxlength`: Maximum length, in characters, of the data in the field. Only valid for input types of `text`, `email`, `search`, `password`, `tel`, or `url`. This method usually does not validate so much as prevent a user from entering too much data in the field it is attached to. It works on all modern browsers.

At the form level, you can turn off validation as a whole one of two ways. Either you can add the `formnovalidate` attribute to the Submit button for the form, or you can add the `novalidate` attribute to the form element itself.

CSS

Not content to have HTML5 do all the work, the CSS specification has been updated to address form validation as well. Form elements that are in an invalid state can be accessed via the `:invalid` pseudoclass. Unfortunately, the implementation of this pseudoclass leaves a bit to be desired. First, form elements are checked for their validity at page load. Thus, if you had styling like the following:

```
:invalid { background-color: yellow }
```

when the page loaded, many of your fields would have a yellow background. Second, Chrome and IE apply `:invalid` only to form elements. Firefox will apply it to the entire form, if any element in the form is invalid. Consider Listing 7-2.

Listing 7-2. Using the `:invalid` Pseudoclass

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
  <style>
    :invalid {
      background-color: yellow
    }
  </style>
</head>
<body>
<h2>A basic form</h2>

<p>Please provide your first and last names.</p>
```

97

```
<form>
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName"/><br/>
  </fieldset>
  <input type="submit" value="Submit the form"/> <input type="reset" value="Reset the
form"/>
</form>

</body>
</html>:
```

In this listing, Firefox displays the entire form with a yellow background, since one element of the form is in an invalid state. Fix this by changing the styling for `:invalid` to `input:invalid`, which gives you consistent behavior across browsers.

CSS also provides a few other pseudoclasses:

- `:valid` covers elements that are in a valid state.

- `:required` gets elements that have their `required` attribute set to true.

- `:optional` gets elements that do not have the `required` attribute set.

- `:in-range` is used for elements that are within their min/max boundaries; it is not supported by IE.

- `:out-of-range` is used for those that are outside those bounds; it is not supported by IE.

Finally, let's talk about the red glow and the popup message. There is a red glow effect around an invalid element after submission in Firefox. (In Internet Explorer, it's a straightforward red border, without a glow effect.) Firefox exposes the effect as the `:-moz-ui-invalid` pseudoclass. You can override it as follows:

```
:-moz-ui-invalid { box-shadow: none }
```

Internet Explorer, alas, does not expose its effect as a pseudoclass. This means we have reached the limit of what we can do with HTML and CSS alone. There are features we would like to change and features we would like to implement. This is where JavaScript comes back into play. >:

JavaScript Form Validation

Thanks in large part to the HTML5 living standard, JavaScript now has a coherent API for form validation. This rests on a relatively simple lifecycle for validation checking: Does this form element have a validation routine? If so, does it pass? If it fails, why did it fail? Interweaved with this process are logical access points for JavaScript, either through method calls or capturing events. It's a good system, although that's not to say it could not bear a bit of improvement. But let's not get ahead of ourselves.

The simplest way to check a form element's validity is to call `checkValidity` on it. The JavaScript object backing every form element now has this `checkValidity` method available. This method accesses the validation constraint set in the HTML for the element. Each constraint is tested against the element's current value. If any of the constraints fail, `checkValidity` returns false. If all pass, `checkValidity` returns true. Calls to `checkValidity` are not limited to individual elements. They can also be made against a form tag. If this is the case, the `checkValidity` call will be delegated to each of the form elements within the form. If all of the subcalls come back true (that is, all of the form elements are valid), then the form is valid as a whole. Conversely, if any of the subcalls come back false, then the form is invalid.

In addition to getting a simple Boolean answer on the validity of an element, we can find out why the element failed validity. Any element's validity property is an object containing all the possible reasons it could fail validation, known as a ValidityState object. We can iterate over its properties and, if one is true, we know that this is one of the reasons that the element failed its validity check. The properties are shown in Table 7-1.

Table 7-1. *Validity State Properties*

Property	Explanation
valid	Whether or not the element's value is valid. Start with this property first.
valueMissing	A required element without a value.
patternMismatch	Failed a check against a regexp specified by pattern.
rangeUnderflow	Value is lower than min value.
rangeOverflow	Value is higher than max value.
stepMismatch	Value is not a valid step value.
tooLong	Value is greater (in characters) than maxlength allows.
typeMismatch	Value fails a check for the email or url input types.
customError	True if a custom error has been thrown.
badInput	A sort of catch-all for when the browser thinks the value is invalid but not for one of the reasons already listed; not implemented in Internet Explorer.

The act of checking an element's validity property runs the validity checks. It is not necessary to invoke *element*.checkValidity first.

Let's take a look at validity checks in action. First, Listing 7-3 shows the relevant section of our HTML.

Listing 7-3. Our HTML Form

```
<body>
<h2>A basic form</h2>

<p>Please fill in the requested information.</p>

<form id="nameForm">
  <div id="fields">
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" class="foo" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName" required/><br/>
    <label for="phone">Phone</label><br/>
    <input type="tel" id="phone"/><br/>

    <label for="age">Age (must be over 13):</label><br/>
    <input type="number" name="age" id="age" step="2" min="14" max="100"/><br/>
    <label for="email">Email</label><br/>
    <input type="email" id="email"/><br/>
    <label for="url">Website</label><br/>
    <input type="url" id="url"/><br/>
  </div>
```

```
  <div id="buttons">
    <input id="overallBtn" value="Check overall validity" type="button"/>
    <input id="validBtn" type="button" value="Display validity"/>
    <input id="submitBtn" type="submit" value="Submit the form"/>
    <input type="reset" id="resetBtn" value="Reset the form"/>
  </div>
</form>

<div>
  <h2>Validation results</h2>
  <div id="vResults"></div>
  <div id="vDetails"></div>
</div>

</body>
```

Note that the submit, reset, and validity checking buttons are in their own div. This makes it easier to use document.querySelectorAll to retrieve only the relevant form fields, which are in a separate div. Now, on to our JavaScript (Listing 7-4).

Listing 7-4. Form Validation and Validity

```
window.addEventListener( 'DOMContentLoaded', function () {
  var validBtn = document.getElementById( 'validBtn' );
  var overAllBtn = document.getElementById( 'overallBtn' );
  var form = document.getElementById( 'nameForm' ); // Or document.forms[0]
  var vDetails = document.getElementById( 'vDetails' );
  var vResults = document.getElementById( 'vResults' );

  overallBtn.addEventListener( 'click', function () {
    var formValid = form.checkValidity();
    vResults.innerHTML = 'Is the form valid? ' + formValid;
  } );

  validBtn.addEventListener( 'click', function () {
    var output = '';

    var inputs = form.querySelectorAll( '#fields > input' );

    for ( var x = 0; x < inputs.length; x++ ) {
      var el = inputs[x];
      output += el.id + ' : ' + el.validity.valid;
      if (! el.validity.valid) {
        output += ' [';
        for (var reason in el.validity) {
          if (el.validity[reason]) {
            output += reason
          }
        }
        output += ']';
      }
```

```
    output += '<br/>'
  }

  vDetails.innerHTML = output;
} );
} );
```

The entire code block is an event tied to when the DOM has loaded. Recall that we don't want to try to add event handlers to elements that may not have been created yet. First, we will retrieve relevant elements within the page: the two validity checking buttons, the output divs, and the form. Next, we will set up event handling for the overall validity check. Note that in this case, we are checking the validity of the entire form for simplicity's sake. We display the results of this check in the `vResults` div.

The second event handler covers checking the individual validity state of each of the form elements. We capture the appropriate elements by using `querySelectorAll` to grab all the input fields under the `div` with ID `fields`. (This winds up being simpler than writing an extended CSS selector to find input types that do not include submits, resets, and buttons.) After obtaining the elements we want, it's simple enough to iterate over the elements and check the `valid` subproperty of their `validity` property. If they are invalid (`valid` is false), then we print out the reason the field is invalid. We encourage you to try this out with a variety of different input values.

This demonstration reveals some interesting things. First, if you load up the page and click the "Display validity" button, the `firstName` and `lastName` fields are invalid (as you would expect, since they are empty), but the `phone`, `age`, `email`, and `url` fields (also empty) are valid! If the field is not required, an empty value is valid. Also, note that the email field has two validations, the implied validation of email, as well as a pattern requirement. Try entering an email that does not contain something like "@foo.com" and you will see it is possible to fail multiple validations at once. Firefox will also tell you that the value fails for `typeMismatchbadInput` if you enter an incomplete email address (say, just a username). You might be inclined to rely on only the `valid` property, but knowing which reason(s) the field failed validation is important information to convey to the user, who will not, after all, be able to submit the form successfully without passing the various validation constraints.

Validation and Users

So far, we have spent most of our time on the technical aspects of form validation. We should also discuss when form validation should be performed. We have a number of options. Simply using the form validation API means that we have automatic form validation at submission time. And we have the capability to invoke validation on any given element when we want to, thanks to the `checkValidity` method. As a best practice, we should perform form validation as early as possible. What this means in practice depends on the field being validated. Start with the idea that we should validate on a change in the form field. Attach a change event handler to your form control and have it call `checkValidity` on that control. Working within the form validation API, this is a fairly straightforward answer to the question of when to validate.

But what if we aren't working within the form validation API? One of the more significant limitations of working within the form validation API is that it has no facility for custom validations. You cannot add a chunk of custom code, bound in a function, say, to run as a validation routine. But you will doubtless want to do that at some point. When you find yourself in this case, it still makes general and practical sense to tie the validation to the change event handler. There are possible exceptions. Consider a field that requires an Ajax call to validate its value, perhaps based on the first few characters entered into the field. In this case, you would tie validation to a keypress event, possibly integrating autosuggest functionality as well. In the next chapter, covering Ajax, we will look at an example of this.

At whatever stage you choose to validate, keep your users in mind. It is very frustrating to fill out a form and then find out that much of the data is invalid for whatever reasons. Users tend to be more accepting of in-line fixes, rather than being presented with a list of errors at submission time.

Validation Events

Another addition to the form validation API is the fact that invalid form elements now throw an invalid event. This event is only thrown in response to a call to checkValidity. The checkValidity call can be made on either the element itself or the form that contains the element. The invalid event does not bubble. Forms do not have an invalid event themselves, despite the fact that forms can be invalid.

You can capture the event with the usual call to addEventListener on the emitting control. Once inside the event handler, the event object itself does not provide any validation-related information. You will have to retrieve the element via the event.target property, and then query its validity property to find out exactly why the element is invalid. But you can do something quite interesting with the preventDefault method of the event. When you invoke preventDefault, the browser's styling behavior for invalid elements will not be applied. Keep in mind that styling changes are only consistently applied when the form is submitted. (Firefox will apply styling changes if you change the value of the form control and blur away from it.) This means different things for different browsers:

- Chrome, which does not style invalid elements but does give them a popup message, will suppress the popup for that element.

- Firefox, which has both popup and styling, will suppress the popup but will not suppress or prevent the red glow effect around the element.

- Internet Explorer, which has both popup and a red border around the element, will suppress both the popup and the border around the element.

Let's look at an example that shows this behavior in action. Start with a relatively familiar HTML form in Listing 7-5.

Listing 7-5. Validity Events Form

```
<!DOCTYPE html>
<html>
<head>
  <title>A basic form</title>
  <style>
    input:invalid {
      background-color: yellow
    }
  </style>
</head>
<body>
<h2>A basic form</h2>

<p>Please provide your first and last names.</p>

<form id="nameForm">
  <fieldset>
    <label for="firstName">First Name:</label><br/>
    <input id="firstName" name="firstName" type="text" required/><br/>
    <label for="lastName">Last Name:</label><br/>
    <input type="text" name="lastName" id="lastName" required/><br/>
  </fieldset>
  <div>
    <input type="submit" value="Submit the form"/> <input type="reset" value="Reset the
form"/>
```

```
    </div>
    <div>
      <input id="firstNameBtn" type="button" value="Check first name validity."/>
      <input id="formBtn" type="button" value="Check form validity"/>
      <input id="preventBtn" type="button" value="Prevent default behavior"/>
      <input id="restoreBtn" type="button" value="Restore default behavior"/>
    </div>
</form>

<div id="vResults"></div>

<script src="listing_7_5.js"></script>
</body>
</html> .
```

Note that we have added styling for invalid input elements. This styling is not associated with the default behavior for an invalid event. Let's look at the backing code (Listing 7-6).

Listing 7-6. Validity Events in JavaScript

```
window.addEventListener( 'DOMContentLoaded', function () {
  var outputDiv = document.getElementById( 'vResults' );
  var firstName = document.getElementById( 'firstName' );

   firstName.addEventListener("focus", function(){
    outputDiv.innerHTML = '';
   });

  function preventDefaultHandler( evt ) {
    evt.preventDefault();
  }

  firstName.addEventListener( 'invalid', function ( event ) {
    outputDiv.innerHTML = 'firstName is invalid';
  } );

  document.getElementById( 'firstNameBtn' ).addEventListener( 'click', function () {
    firstName.checkValidity();
  } );

  document.getElementById( 'formBtn' ).addEventListener( 'click', function () {
    document.getElementById( 'nameForm' ).checkValidity();
  } );

  document.getElementById( 'preventBtn' ).addEventListener( 'click', function () {
    firstName.addEventListener( 'invalid', preventDefaultHandler );
  } );

  document.getElementById( 'restoreBtn' ).addEventListener( 'click', function () {
    firstName.removeEventListener( 'invalid', preventDefaultHandler );
  } );

} ); .
```

As usual, all of our code is activated after the DOMContentLoaded event has fired. We have a basic event handler for invalid events on the firstName field, which outputs to the vResults div. Then we add event handlers for the specialized buttons. First we create two convenience buttons: one for checking the validity of the firstName field, the other for checking the validity of the entire form. Then we add behavior for overriding or restoring the default behavior associated with invalid events. Try it out!

Customizing Validation

We now have almost all the tools available to us to control form validation comprehensively. We can choose which validations to activate. We can control when validation is performed. And we can capture invalid events and prevent the default behavior (particularly as regards styling) from firing. As discussed before, we cannot customize the actual validation routines (alas). So what is left for us to work with? We might like to control the message in the validation popup users see on submitting a form. (The look and feel of the popup is also not customizable. Remember that we mentioned there were some shortcomings with the API and its implementation?)

To change the validation message that appears when a form field is invalid, use the setCustomValidity function associated with the form control. Pass setCustomValidity a string as an argument, and that string will appear as the text of the popup. This does have some other side effects, though. In Firefox, the field will show, at page load time, as invalid, with the red glow effect. Using setCustomValidity with either Internet Explorer or Chrome has no effect at page load time. The Firefox styling can be turned off, as mentioned earlier, by overriding the :-moz-ui-invalid pseudoclass. But more problematic is that when setCustomValidity is used, the customError property of the validityState of the form control in question is set to true. This means that the valid property of the validityState is false, which means it reads as invalid. All for simply changing the message associated with the validity check! This is unfortunate and renders setCustomValidity nearly useless.

An alternative would be to use a polyfill. There is a long list of polyfills, not only for form validation but also for other HTML5 items that may not have support on every browser you need to work on. You can find them here:

https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills

Preventing Form Validation

There is one aspect of form validation we have not explored: turning it off. The majority of this chapter has focused on using the new form validation API, and exploring its limits. But what if there were a deal-breaker with the API, some feature (or bug, or inconsistency) that prevented our wholesale use of the API? Were this the case, we would want to do two things: discontinue the automatic form validation, and substitute our own. It is the former which interests us, as the latter is simply the case of reimplementing an API according to our whims. To turn off form validation behavior, add the novalidate attribute to the form element. You can prevent behavior per submit-button click by adding the formnovalidate attribute to a submit button, but this does not turn off form validation for the form as a whole. Since we might want to substitute our own customized API for form validation, we want to use the novalidate attribute to deactivate form validation (for the parent element) entirely.

Summary

In this chapter we spent the bulk of our time looking at the new JavaScript form validation API. Within its constraints, it is a powerful tool for automatically validating user data. Validation happens automatically on form submission, and we can validate at any time (or times) of our own choosing as well. Validation can be turned off if needed. We can customize the appearance of valid and invalid elements, and can even customize the message displayed when an element is invalid.

The API is not without problems. it lacks some critical customization capabilities, like allowing styling for error messages, or custom validation routines. There are small implementation differences between the major browsers that can be a bit of a pain to clean up after. Some official parts of the API (think the `willValidate` property) are not currently implemented and other parts (`setCustomValidity`) have crippling problems.

Overall the API is a big step forward for JavaScript, HTML, CSS, and the browser. We look forward to seeing how it will be refined in the future.

CHAPTER 8

■ ■ ■

Introduction to Ajax

Ajax is a term coined by Jesse James Garrett of Adaptive Path to describe the asynchronous client-to-server communication that is made possible using the XMLHttpRequest object, which is provided by all modern browsers. An acronym for *A*synchronous *J*avaScript *a*nd *X*ML, Ajax has evolved into a term used to encapsulate the techniques necessary to create a dynamic web application. Additionally, the individual components of the Ajax technique are completely interchangeable—using JSON instead of XML (for example) is perfectly valid.

Since the first edition of this book, usage of Ajax has changed significantly. Once an exotic API, Ajax is now a standard part of the professional JavaScript programmer's toolbox. The W3C has overhauled the XMLHttpRequest object, the foundation of Ajax, by the, adding new features and clarifying the behavior of other features. One of the core rules of Ajax: no connections to outside domains; this is enforced by using the Cross-Origin Resource Sharing standard, also known as CORS.

In this chapter, you're going to see the details that make up the full Ajax process. We will concentrate on the XMLHttpRequest object API, but we'll also discuss ancillary issues like handling responses, managing HTTP status codes, and so on. The intent is to provide you with a comprehensive understanding of what goes on within an Ajax request/response cycle.

We will not provide an API for Ajax interactions. Writing code to the various specifications governing Ajax is a straightforward affair, but writing a complete Ajax library for the real world is most assuredly not. Consider the Ajax portion of the jQuery library, which has over a dozen edge cases that handle various oddities about the API in Internet Explorer, Firefox, Chrome, and other browsers. Also, because jQuery, Dojo, Ext JS, and several other smaller libraries already have Ajax implementations, we see no reason to reinvent that particular wheel here. Instead, we will provide examples of Ajax interactions, written according to the current (as of publishing) specifications. These examples are intended to be instructive and demonstrative, not final. We encourage you when using Ajax to look into utility libraries like jQuery, Zepto, Dojo, Ext JS, and MooTools, or Ajax-focused libraries like Fermata and reqwest.

That leaves quite a bit to discuss! This chapter will cover the following:

- Examining the different types of HTTP requests

- Determining the best way to send data to a server

- Looking at the entire HTTP response and thinking about how to handle not only a successful response, but one that goes wrong (in some fashion) as well

- Reading, traversing, and manipulating the data result that comes from the server in its response

- Handling asynchronous responses

- Making requests across domains, enabled by CORS

Using Ajax

It doesn't take much code to create a simple Ajax implementation; however, what the implementation affords you is great. For example, instead of having to force the user to request an entirely new web page after a form submission, your code can handle the submission process asynchronously, loading a small portion of desired results upon completion. In fact, when we tie Ajax requests to handlers for events like a keypress, there is no need to wait for the form submission at all. This is what is behind the "magic" of Google's autosuggest search function. When you start to enter a search term, Google fires an Ajax request based on your entry. As you refine your search, it sends out other Ajax requests. Google will display not only suggestions but, based on the first possible option, even a first page of results. Figure 8-1 shows an example of this process.
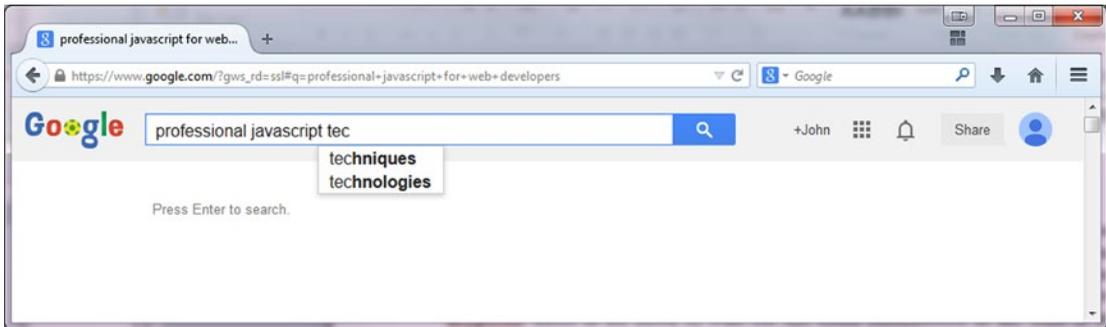
Figure 8-1. *An example of Instant Domain Search looking for domain names as you type*

HTTP Requests

The most important and probably most consistent aspect of the Ajax process is the HTTP request portion. The Hypertext Transfer Protocol (HTTP) was designed simply to transfer HTML documents and associated files. Thankfully, all modern browsers support a means of establishing HTTP connections dynamically and asynchronously, using JavaScript. This proves to be incredibly useful in developing more responsive web applications.

Asynchronously sending data to the server and receiving additional data back is the ultimate purpose of Ajax. How the data is formatted ultimately depends on your specific requirements.

In the following sections, you will see how to format data to be transferred to a server using the different HTTP requests. You will then look at how to establish basic connections with the server, and you will see the details needed to make this happen in a cross-browser environment.

Establishing a Connection

All Ajax processes start with a connection to the server. Connections to the server are generally organized through the XMLHttpRequest object. (The lone exception is in older versions of Internet Explorer when making cross-domain requests. But we will cover that later on. For now, we shall rely upon the XMLHttpRequest object.)

Communication with the XMLHttpRequest object follows a lifecycle:

1.   Create an instance of XMLHttpRequest.

2.   Configure the object with appropriate settings.

3. Open the request via a specific HTTP verb and destination.

4. Send the request.

Listing 8-1 shows how to establish a basic GET request with the server.

Listing 8-1. A Cross-Browser Means of Establishing an HTTP GET Request with the Server

```
// Create the request object
var xml = new XMLHttpRequest();

// If we needed to do any special configuration, we would do it here

// Open the socket
xml.open('GET', '/some/url.cgi', true);

// Establish the connection to the server and send any additional data
xml.send();
```

The code needed to establish a connection with a server, as you can see, is quite simple; there really isn't much to it. One set of difficulties arises when you want advanced features (such as checking for time-outs or modified data); we will cover those details in the "HTTP Response" section of this chapter. Another set of difficulties comes into play when you want to transfer data from the client (your browser) to the server. This is one of the most important features of the whole Ajax methodology. Will we send simple data on the URL? What about POSTed data? What about more complicated formats? With these questions (and others, of course) in mind, let's look at the details needed to package some data and send it to a server.

Serializing Data

The first step of sending a set of data to a server is to format it so that the server can easily read it; this process is called *serialization*. We need to ask a few questions before serializing data. First:

1. What data are we sending? Are we sending variable-and-value pairs? Large sets of data? Files?

2. How are we sending this data, GET? POST? Another HTTP verb?

3. What format of data are we using? There are two: `application/x-www-form-urlencoded` and `multipart/form-data`. The former is sometimes called query string encoding and takes the familiar form of var1=val1&var2=val2...

From a JavaScript perspective, the third question is the most important. The first and second questions are issues of design. They will have an effect on our application but will not necessarily require different code. But the question of which data format we use has a strong effect on our application.

In modern browsers, it is actually easier to deal with multipart/form-data information. Thanks to the FormData object, we can very easily serialize data into an object that our browser will automatically convert to the multipart/form-data format. Unfortunately, not all browsers support every option that is in the specification yet. But there is a lot we can do right now.

FormData Objects

FormData objects are a relatively new proposal covered by HTML5. The WHATWG and the W3C intended to give a more object-oriented, map-like approach to information sent as part of an Ajax (or really any HTTP) request. Accordingly, a FormData object can be either initialized as empty or associated with a form. If you are initializing with a form, get a reference to the containing form DOM element (usually via getElementById) and pass it into the FormData constructor. Otherwise, as stated, the FormData object will be empty. Either way, you can choose to add new data via the append function (Listing 8-2).

Listing 8-2. An Example Using the append method with FormData

```
// Create the formData object
var  formDataObj= new FormData();

//append name/values to be sent to the server
formDataObj.append('first',  'Yakko');
formDataObj.append('second', 'Wakko');
formDataObj.append('third', 'Dot');

// Create the request object
var xml = new XMLHttpRequest();

// Set up a POST to the server
xml.open('POST', '/some/url.cgi');

// Send over the formData
xml.send(formDataObj);
```

There are some differences in the specifications, though. The WHATWG specification also includes functions for deleting, getting, and setting values on the object. None of the modern browsers implement these functions. In part, that's because the W3C version of the specification has only the append function. The modern browsers follow this W3C spec, at least at the moment. This means that a FormData object is one-way: data goes in, but is only accessible on the other side of an HTTP request.

The alternative to FormData objects is to serialize in JavaScript. That is, take the data you intend to transfer to the server, URL-encode it, and send it to the server as part of the request. This is not too difficult, although there are some caveats to be cautious of.

Let's take a look at some examples of the type of data that you can send to the server, along with their resulting server-friendly, serialized output, shown in Listing 8-3.

Listing 8-3. Examples of Raw JavaScript Objects Converted to Serialized Form

```
// A simple object holding key/value pairs
{
    name: 'John',
    last: 'Resig',
    city: 'Cambridge',
    zip: 02140
}
// Serialized form
name=John&last=Resig&city=Cambridge&zip=02140
```

```
// Another set of data, with multiple values
[
    { name: 'name', value: 'John' },
    { name: 'last', value: 'Resig' },
    { name: 'lang', value: 'JavaScript' },
    { name: 'lang', value: 'Perl' },
    { name: 'lang', value: 'Java' }
]

// And the serialized form of that data
name=John&last=Resig&lang=JavaScript&lang=Perl&lang=Java

// Finally, let's find some input elements
[
    document.getElementById( 'name' ),
    document.getElementById( 'last' ),
    document.getElementById( 'username' ),
    document.getElementById( 'password' )
]
// And serialize them into a data string
name=John&last=Resig&username=jeresig&password=test
```

The format that you're using to serialize the data is the standard format for passing additional parameters in an HTTP request. You're likely to have seen them in a standard HTTP GET request looking like this:

http://someurl.com/?name=John&last=Resig

This data can also be passed to a POST request (and in a much greater quantity than a simple GET). We will look at those differences in an upcoming section.

For now, let's build a standard means of serializing the data structures presented in Listing 8-3. A function to do just that can be found in Listing 8-4. This function is capable of serializing most form input elements, with the exception of multiple-select inputs.

Listing 8-4. A Standard Function for Serializing Data Structures to an HTTP-Compatible Parameter Scheme

```
// Serialize a set of data. It can take two different types of objects:
// - An array of input elements.
// - A hash of key/value pairs
// The function returns a serialized string
function serialize(a) {
    // The set of serialize results
    var s = [];

    // If an array was passed in, assume that it is an array
    // of form elements
    if ( a.constructor === Array ) {

        // Serialize the form elements
        for ( var i = 0; i < a.length; i++ )
            s.push( a[i].name + '=' + encodeURIComponent( a[i].value ) );
```

```
    // Otherwise, assume that it's an object of key/value pairs
    } else {

        // Serialize the key/values
        for ( var j in a )
            s.push( j + '=' + encodeURIComponent( a[j] ) );
    }

    // Return the resulting serialization
    return s.join('&');
}
```

Now that there is a serialized form of your data (in a simple string), you can look at how to send that data to the server using a GET or a POST request.

Establishing a GET Request

Let's revisit establishing an HTTP GET request with a server, using XMLHttpRequest, but this time sending along additional serialized data. Listing 8-5 shows a simple example of this.

Listing 8-5. A Cross-Browser Means of Establishing an HTTP GET Request with the Server (and Not Reading Any Resulting Data)

```
// Create the request object
var xml = new XMLHttpRequest();

// Open the asynchronous GET request
xml.open('GET', '/some/url.cgi?' + serialize( data ), true);

// Establish the connection to the server
xml.send();
```

The important part to note is that the serialized data is appended to the server URL (separated by a ? character). All web servers and application frameworks know to interpret the data included after the ? as a serialized set of key/value pairs.

Establishing a POST Request

The other way to establish an HTTP request with a server, using XMLHttpRequest, is with a POST, which involves a fundamentally different way of sending data to the server. Primarily, a POST request is capable of sending data of any format and of any length (not just limited to your serialized string of data).

The serialization format that you've been using for your data is generally given the content type application/x-www-form-urlencoded when passed to the server. This means that you could also send pure XML to the server (with a content type of text/xml or application/xml) or even a JavaScript object (using the content type application/json).

A simple example of establishing the request and sending additional serialized data appears in Listing 8-6.

Listing 8-6. A Cross-Browser Means of Establishing an HTTP POST Request with the Server (and Not Reading Any Resulting Data)

```
// Create the request object
var xml = new XMLHttpRequest();

// Open the asynchronous POST request
xml.open('POST', '/some/url.cgi', true);

// Set the content-type header, so that the server
// knows how to interpret the data that we're sending
xml.setRequestHeader(
    'Content-Type', 'application/x-www-form-urlencoded');

// Establish the connection to the server and send the serialized data
xml.send( serialize( data ) );
```

To expand on the previous point, let's look at a case of sending data that is not in your serialized format to the server. Listing 8-7 shows an example.

Listing 8-7. An Example of POSTing XML Data to a Server

```
// Create the request object
var xml = new XMLHttpRequest();

// Open the asynchronous POST request
xml.open('POST', '/some/url.cgi', true);

// Set the content-type header, so that the server
// knows how to interpret the XML data that we're sending
xml.setRequestHeader( 'Content-Type', 'text/xml');

// Establish the connection to the server and send the serialized data
xml.send( '<items><item id='one'/><item id='two'/></items>' );
```

The ability to send bulk amounts of data (there is no limit on the amount of data that you can send; by contrast, a GET request maxes out at just a couple KB of data, depending on the browser) is extremely important. With it, you can create implementations of different communication protocols, such as XML-RPC or SOAP.

This discussion, however, for simplicity is limited to some of the most common and useful data formats that can be made available as an HTTP response.

HTTP Response

Level 2 of the XMLHttpRequest class now provides better control over telling the browser how we want our data back. We do this by setting the responseType property and receive the requested data using the response property.

To start, let's look at a very naïve example of processing the data from a server response, as shown in Listing 8-8.

Listing 8-8. Establishing a Connection with a Server and Reading the Resulting Data

```
// Create the request object
var request = new XMLHttpRequest();

// Open the asynchronous POST request
request.open('GET', '/some/image.png', true);

//Blob is a Binary Large Object
request.responseType = 'blob';

request.addEventListener('load', downloadFinished, false);

function downloadFinished(evt){
    if(this.status == 200){
        var blob = new Blob([this.response], {type: 'img/png'});
    }
}
```

In this example you can see how to receive binary data and convert it into a PNG file. The responseType property can be set to any of the following:

- Text: Results return as a string of text

- ArrayBuffer: Results return as an array of binary data

- Document: Results are assumed to be a XML document, but it could be an HTML document

- Blob: Results return as a file like object of raw data

- JSON: Results return as a JSON document

Now that we know how to set the responseType, we can look at how to monitor the progress of our request.

Monitoring Progress

As we have seen before, using addEventListener makes our code easy to read and very flexible. Here we use the same technique on our request object. No matter whether you are downloading data from the server or uploading to it, you can listen for these events as shown in Listing 8-9.

Listing 8-9. Using addEventListener to Listen for Progress on a Request from the Server

```
var myRequest = new XMLHttpRequest();

myRequest.addEventListener('loadstart', onLoadStart, false);
myRequest.addEventListener('progress', onProgress, false);
myRequest.addEventListener('load', onLoad, false);
myRequest.addEventListener('error', onError, false);
myRequest.addEventListener('abort', onAbort, false);

//Must add eventListeners before running a send or open method

myRequest.open('GET', '/fileOnServer');
```

```
function onLoadStart(evt){
    console.log('starting the request');
}

function onProgress(evt){
     var currentPercent = (evt.loaded / evt.total) * 100;
     console.log(currentPercent);
}

function onLoad(evt){
    console.log('transfer is complete');
}

function onError(evt){
  console.log('error during transfer');
}

function onAbort(evt){
  console.log('the user aborted the transfer');
}
```

You can now understand a lot more about what is going on with your file than you could before. Using the loaded and total properties you might work out the percentage of the file being downloaded. If for some reason the user decided to stop the download, you would receive the abort event. If there was something wrong with the file, or if it had finished loading, you would receive either the error or the load event. Finally, when you first make the request of the server, you would receive the loadstart event. Now let's take a quick look at timeouts.

Checking for Time-Outs and Cross-Origin Resource Sharing

Simply put, time-outs let you set a time for how long the application should wait until it stops looking for a response from the server. It is easy to set a time-out and listen for it.

Listing 8-10 shows how you would go about checking for a time-out in an application of your own.

Listing 8-10. An Example of Checking for a Request Time-Out

```
// Create the request object
var xml = new XMLHttpRequest();

// We're going to wait for a request for 5 seconds, before giving up
xml.timeout = 5000;

//Listen for the timeout event
xml.addEventListener('timeout', onTimeOut, false);

// Open the asynchronous POST request
xml.open('GET', '/some/url.cgi', true);

// Establish the connection to the server
xml.send();
```

By default, browsers will not allow applications to make request to servers other than the one the site originated from. This protects users from cross-site scripting attacks. The server must allow requests; otherwise, an INVALID_ACCESS error is given. The header given by the server would look like this:

```
Access-Control-Allow-Origin:*
   //Using a wild card (*) to allow access from anyone.
Access-Control-Allow_origin:http://spaceappleyoshi.com
   //Allowing from a certain domain
```

Summary

We now have a solid foundation to work with data on the server. We can tell the server what kind of results we expect back. We can also listen for events that will tell us things like the progress of the file transfer or if there was an error during the transfer. Finally, we discussed time-outs and cross-origin resource sharing or (CORS). In the next chapter we'll take a look a few development tools for web production.

■ ■ ■

Web Production Tools

The tools for developing websites have matured over the years. We went from using simple editors like Notepad to full-scale development environments like WebStorm. We also have libraries like JQuery. We can use Handlebars as a templating engine and AngularJS as a full MVC framework. There are also unit testing frameworks and version control systems to help us do our jobs better and faster. So now that we have all of these things available, how do we keep them all organized?

To answer that, we are going to break this down into two parts. First, we'll look at tools for the creation of a site, then at tools for keeping track of the changes to that site. In order to create a site we will explore Yeoman, Grunt, Bower, and Node Package Manager (NPM). To track changes we will use Git.

All of these tools work together, so let's break down what each one does:

- Bower is a package management system. Its purpose is to make sure that all the client-side code that your project depends on has been downloaded and installed. Site: http://bower.io/

- Grunt is what's called a build tool. It lets you automate many types of tasks, including unit testing, linting (checking JavaScript for errors), and adding your code to version control. It can also be used in deploying your code to a server. Site: http://gruntjs.com/

- Yeoman is what's called the scaffolding tool. It creates the files and folders needed to make a bare-bones version of the project. It then uses Bower to gather all the code that the project is dependent on. Finally it uses a build tool (like Grunt) to automate tasks. It does this by way of using generators. Site: http://yeoman.io/

- Node Package Manager (NPM), as you can see from its name, manages packages. These packages run on top of Node.js. As Node became more popular, some of these packages were developed for client-side development instead of just server-side, where Node.js is used. Site: https://nodejs.org/

- Git is a version control system. If you have heard of tools like Subversion or Perforce, Git is similar. It will keep track of all the files you work with and can tell you the difference between the files. Site: http://git-scm.com/

Scaffolding Your Projects

Computers are great at doing tasks that people don't want to do, and they can do them over and over again without getting bored. No one wants to make folders for the images, CSS, and JavaScript files every time you create a project. There are plenty of small tasks that we take for granted that can now be automated. Wouldn't it be nice to start a project that just has all the folders worked out for you with a single command?

That's the idea behind scaffolding. Because most websites are organized the same way, there isn't a need to work out the structure manually. Yeoman lets you scaffold just about any type of web project you want. Using best practices from the community, Yeoman uses generators to set up our project quickly and easily.

Generators are really templates that anyone can make. There are teams that sponsor projects to create "official" generators, but if that one doesn't do the thing you want, someone else may have made one that does. Generators are also open source so you can look under one's hood and see how it was made. In order to work with Yeoman, we first need to install Node.js.

NPM is the Foundation for Everything

Node Package Manager (NPM) gives you the ability to manage dependencies in an application. This means that if you need code for your project (say JQuery), NPM makes it easy to add to your project. It is also what is running behind the scenes with most of the tools we are about to install. NPM is part of Node.js, which is an open source cross-platform environment for making server-side applications with JavaScript. Even though we are not going to create a Node.js project here, we do need to install node. There are a few ways to do this; for our example, we will try to make it as simple as possible.

When you go to nodejs.org, the site will figure out what operating system you have. Click the Install button to download and run the installer.

Once it is installed, you can go to terminal mode (Mac, Linux) or the command prompt (Windows) and type **node -version**. You should see the current version of node displayed in the window.

With node installed, now we can get everything else we need. To install Yeoman, type **npm install -g yo**, for Grunt type **npm install -g grunt-cli**, and for Bower type **npm install -g bower**. Using -g means that the install will happen globally; you can run these utilities in any folder as you create new projects. The cli stands for command line interface. For our exercise, we will be spending time at the command line. It's a good thing to get used to it, and worth the effort. Now we can install a generator and start looking at the other tools.

Generators

As we talked about before, generators are really templates that describe the structure of the site. You can adjust these templates by passing different parameters to Yeoman. At Yeoman.io you can find a list of generators and links back to the GitHub repositories. The repositories have all the instructions on how to use the generators. For example, if you wanted to make a site using AngularJS, you would enter

```
npm install -g generator-angular
```

This will install the AngularJS generator. If you wanted an AngularJS site and also wanted to add Karma (a JavaScript test runner) to help run your unit tests, the install would look like this:

```
npm install -g generator-angular generator-karma
```

Now that you have a generator installed, by typing **yo** at the command line you can look at a list of generators that are installed and update them. From here you can also install new generators.

At this point, you should make a folder for your project and, while you are in that folder, the next command should be

```
yo angular
```

Yeoman will start to ask you questions about your app. For example, it will ask whether you would like to use Sass, as shown in Figure 9-1.

```
Last login: Tue Apr 21 07:10:46 on ttys001
Russs-MBP:yeomanAngular asciibn$ yo angular


      _-----_
     |       |
     |--(o)--|    .--------------------------.
    `---------´   |    Welcome to Yeoman,     |
     ( _´U`_ )    |    ladies and gentlemen!   |
     /___A___\   '--------------------------'
      |  ~  |
    __'.___.'__
  ´   `  |° ´ Y `

Out of the box I include Bootstrap and some AngularJS recommended modules.

? Would you like to use Sass (with Compass)? (Y/n)
```

Figure 9-1. *Yeoman setting up an AngularJS site*

You will be asked a few other questions about how you want the project set up. Once it is finished, Bower will grab the latest versions of all the libraries you need from GitHub and scaffold your project together for you. Once everything is installed, type the following to see the site in action:

```
grunt serve
```

This time we are using Grunt to create a local web server in your current folder and serve the home page (Figure 9-2).

Figure 9-2.   Yeoman running AngularJS on a local server

That's all you need to do. You now have a site that is ready to go. This would be a good time to put our code in version control.

Version Control

Change is constant. Our files are updated over and over again. As we work, things sometimes break. In some cases, a simple undo is fine. At other times we may need to revert to a previous state, especially when working with a team and there are many changes. One change can break the whole site, and it can be difficult to track down the problem. This is where version control can be very helpful.

Git is the version control system we are going to use. It's popular and has GUI clients as well. Just as we did with the Node.js example, we are going to take the quickest way to get it installed. To do that, go to git-scm.com to download and install Git.

Once it is installed, you can configure it by using the command line. To add your identity, type **git config -global user.name "*your name*"** and **git config -global user.email your@email.com**.

Now that you have Git installed and configured, we are going to quickly add files and then commit them locally. Make sure you are in a project folder. Type **git init** inside that folder. That creates a .Git folder that will hold all the information about the project. This folder is usually invisible, so you may need to change some settings in your operating system if you want to see it. Next, let's check the status of our commit. Type **git status** and you can see that at this point, no files have been added to version control (Figure 9-3).

```
Russs-MBP:yeomanAngular asciibn$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .bowerrc
        .editorconfig
        .gitattributes
        .gitignore
        .jshintrc
        .travis.yml
        .yo-rc.json
        Gruntfile.js
        README.md
        app/
        bower.json
        package.json
        test/

nothing added to commit but untracked files present (use "git add" to track)
Russs-MBP:yeomanAngular asciibn$ 
```

Figure 9-3. *Files have not yet been added to Git*

Adding Files, Updates, and the First Commit

Next we add files so they can be tracked by Git. Adding files to the repository is as simple as typing **git add *file name/folder***. Git will then start to track the files. In Figure 9-4 we added the app folder by typing **git add app/**. You can check the status again and see the results.

Figure 9-4. *The app folder has been added to Git*

Continue adding files, especially `bower.json` and `package.json`. These files keep track of your dependent modules and the versions of those modules. `Gruntfile.js` will have all the tasks you can run. We ran a task before by typing `grunt serve`. That task ran a local server for us.

As a best practice, the `node_modules` and `bower_components` folders are not added to version control. You can reinstall them later by using the `npm install` and `bower install` commands.

Files that you do not want to add to Git are defined in the `.gitignore` file. You can modify this file to include file types or anything else you would like Git to ignore.

We talked about how files change over time, so you may ask how Git knows when a file is changed. First, the file needs to be added to the repository. We did that with the `git add` command. Then, when a change happens, you can ask for the status again. This will list any files that have changed since the last time it's been added. Just remember that you need Git to know about the file before you make the change, so that can track the changes over time. In Figure 9-5 we can change the `README.md` file to illustrate our point.

```
 ● ● ●                                    1. bash
  (use "git rm --cached <file>..." to unstage)

        new file:    README.md
        new file:    app/.buildignore
        new file:    app/.htaccess
        new file:    app/404.html
        new file:    app/favicon.ico
        new file:    app/images/yeoman.png
        new file:    app/index.html
        new file:    app/robots.txt
        new file:    app/scripts/app.js
        new file:    app/scripts/controllers/about.js
        new file:    app/scripts/controllers/main.js
        new file:    app/styles/main.scss
        new file:    app/views/about.html
        new file:    app/views/main.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:    README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .bowerrc
        .editorconfig
        .gitattributes
        .gitignore
        .jshintrc
        .travis.yml
        .yo-rc.json
        Gruntfile.js
        bower.json
        package.json
        test/

Russs-MBP:yeomanAngular asciibn$ █
```

Figure 9-5. Git can see when a file has been modified

When a file has been changed you can add it again just as you did before. The next time you check the status it will be back on the list of tracked items.

Now that you have all the files being tracked by Git, you can commit all the files. Think of a commit as a snapshot of the project at its current state (Figure 9-6). If anything were to happen to the project, you can always revert back to the last state.

123

```
Russs-MacBook-Pro:yeomanAngular asciibn$ git commit -m "My First Commit"
[master (root-commit) 9e29ab1] My First Commit
 21 files changed, 1689 insertions(+)
 create mode 100644 Gruntfile.js
 create mode 100644 README.md
 create mode 100644 app/.buildignore
 create mode 100644 app/.htaccess
 create mode 100644 app/404.html
 create mode 100644 app/favicon.ico
 create mode 100644 app/images/yeoman.png
 create mode 100644 app/index.html
 create mode 100644 app/robots.txt
 create mode 100644 app/scripts/app.js
 create mode 100644 app/scripts/controllers/about.js
 create mode 100644 app/scripts/controllers/main.js
 create mode 100644 app/styles/main.scss
 create mode 100644 app/views/about.html
 create mode 100644 app/views/main.html
 create mode 100644 bower.json
 create mode 100644 package.json
 create mode 100644 test/.jshintrc
 create mode 100644 test/karma.conf.js
 create mode 100644 test/spec/controllers/about.js
 create mode 100644 test/spec/controllers/main.js
Russs-MacBook-Pro:yeomanAngular asciibn$ git log
commit 9e29ab14dc88d176c9578d8546d950a8387b8ee8
Author: Russ Ferguson <rferguson@technologycoach.com>
Date:   Sat May 16 18:25:07 2015 -0400

    My First Commit
Russs-MacBook-Pro:yeomanAngular asciibn$ 
```

Figure 9-6. *Files committed and shown in the log*

To commit, type git **commit -m "*notes about the commit*"**. The –m flag stands for message. Alternatively, you can use a text editor to make your message. If you want to see the history of messages, you can type **git log**.

We now have a way of keeping track of changes on our local machine. This will work just fine if you are a lone developer.

If you want to share the code or work with a team, however, you will need to add a server-side component. Two of the most popular options are GitHub (https://github.com/) and BitBucket (https://bitbucket.org/). Using either of these options, you can have a remote repository in addition to the files on your local machine.

Summary

We hope that after reading this chapter, you see the large amount of resources available to you. The ability to put a site together quickly, using Yeoman generators, will save you time and effort as you put new projects together. Yeoman can also be used as a learning tool to understand how different frameworks work.

We gave just a basic overview of Git. It's a subject that could be a book by itself. Fortunately, there is *Pro Git* by Scott Chacon and Ben Straub (Apress, 2nd edition 2014). A link to it can also be found on the Git home page (http://git-scm.com/). It's online or as a digital file. Now that we can quickly put a site together, let's take a look in Chapter 10 at a very popular framework, AngularJS.

CHAPTER 10

■ ■ ■

AngularJS and Testing

In the previous chapter, you learned how to use the current set of tools to quickly put a site together and use version control to keep track of all the files you work with and the difference between them. In this chapter, we'll dig in and understand how frameworks like Angular work.

Briefly, frameworks help you build large applications in a way that is more organized and easier to maintain. One of the other benefits to using a framework is a shorter learning curve for the team. Once a new member learns the framework, they have a better understanding of how the entire site works. With our example we will take a high-level look at AngularJS.

At the time of this writing, the current version of Angular is 1.4.1. Information can be found at `https://angularjs.org/`. Information about Angular 2 can be found at `https://angular.io/`.

One of the problems Angular tries to solve is to make it easy to develop dynamic applications. HTML on its own isn't designed to make single-page applications. Angular provides a way to develop modern web applications in a way that is quick to learn. It does this by keeping each part of the application separate, so that each part can be updated independently of the others. This architectural pattern is called Model-View-Controller (MVC). You will find that other frameworks, like Backbone and Ember, work in a similar way.

In Chapter 9 we introduced some development tools that can help us be more productive. Yeoman (`http://yeomanio/`.) uses community-built generators to quickly develop all the files and folders needed to get a basic site working. Grunt (`http://gruntjs.com/`) is used to automate tasks like unit testing and optimizing files for a production-ready site. This chapter assumes that both tools are installed. Please refer to the previous chapter or the sites listed for information on installing them.

To create a new Angular project, type **your angular**. In response, Yeoman asks some questions about how you want to set up the project:

- **Would you like to use Sass (with Compass)?** Sass (`http://sass-lang.com/`)stands for Syntactically Awesome Style Sheets. Sass gives you features like nesting selectors and using variables to develop style sheets. Compass is a tool written in Ruby that uses Sass files and adds features like generating sprite sheets out of a series of images.

  By agreeing to this option, you will get a SCSS file using Twitter Bootstrap's styles as default. If you choose no, Yeoman will give you a regular CSS file with the same CSS selectors.

- **Would you like to include Bootstrap?** Twitter Bootstrap (`http://getbootstrap.com/`) helps you develop the user interface for your website. Bootstrap can help make your site responsive so it can look good on multiple devices and gives you items like buttons, an image carousel, and other user interface components.

If you choose to use this tool, Yeoman then asks whether to use the Sass version of Bootstrap.

- **Which modules would you like to include?** Modules give Angular extra abilities. For example, the `angular-aria` module provides accessibility features, while `angular-route` gives you the ability to add deep linking features. You can choose to add or remove any of the modules. Modules can also be added later manually.

Once you have answered the questions, all of the files needed will be downloaded, and Grunt will then launch a local server with the default browser loading `http://localhost:9000`, as shown in Figure 10-1.
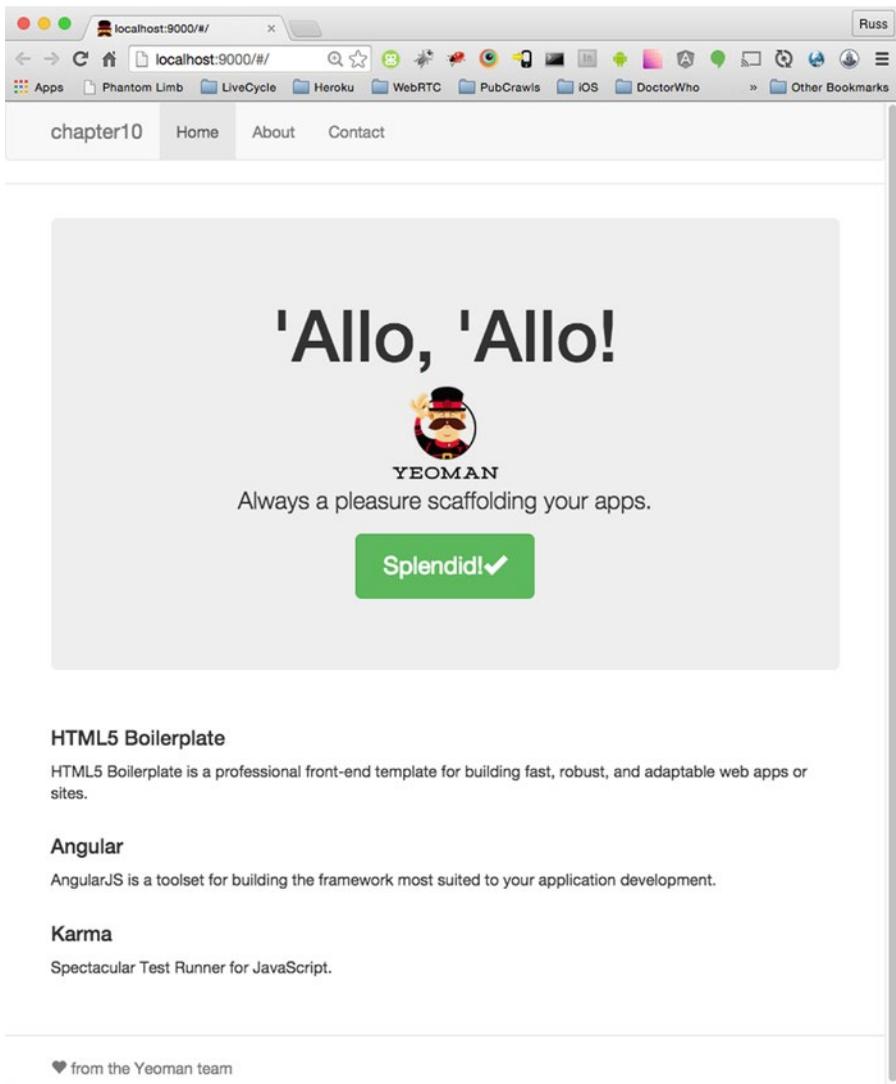
Figure 10-1. AngularJS running on port 9000

From here we can look at the folders that make up this project. The app folder contains our main application, and it is where we are going to start.

The folders are very standard for what you would find in an HTML site. Inside the scripts folder is where it starts to get interesting.

At the root of the scripts folder is app.js. This is the main application file for Angular. Open this file to see how Angular is being bootstrapped. In Figure 10-2, you find the name of the application chapter10app (because that was the name of the folder the app was created in). This works with the ng-app directive in index.html. A directive gives added ability to DOM elements. In this case, it tells Angular that the application starts here and is called chapter10app.
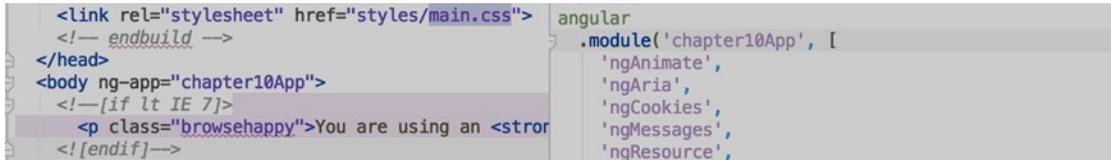
```
<link rel="stylesheet" href="styles/main.css">     angular
<!-- endbuild -->                                    .module('chapter10App', [
</head>                                                'ngAnimate',
<body ng-app="chapter10App">                           'ngAria',
<!--[if lt IE 7]>                                      'ngCookies',
    <p class="browsehappy">You are using an <stron     'ngMessages',
<![endif]-->                                           'ngResource',
```

Figure 10-2. Using the ng-app *directive tells Angular where the root element of the application is.*

As you look at app.js, you see after the name of the application that numerous modules are loaded, which give Angular extra ability. One of these modules is named ngRoute; it will let you handle URLs for the application. The .config method uses $routeProvider to understand the URL and load the correct template with the controller by using a series of when statements.

The when statement enables you to customize the URLs for the application. In this example, if you were to type **/about**, Angular would know to load the about.html template and use AboutCtrl as the controller. Let's explain in detail what that means.

Views and Controllers

We've discussed that Angular, like other frameworks, use the MVC pattern. This means that the application is broken up into three distinct parts.

- Model: Stores the data for the application.

- View: Creates a representation of the model data; for example, generating a chart to represent data.

- Controller: Sends commands to the model to update data. Also sends commands to the view to update the presentation of the model's data.

The views folder contains the HTML templates that can be updated with the data coming from the model. The controllers folder contains the JavaScript files that will communicate with both the model and the view files.

Let's look at the about.html file. Here we will add a button and have the controller work with it.

Open about.html, found in the views folder. Inside it, add a button tag. In this button tag we are going to use another directive, which will let Angular know when the button is clicked.

We need to add the ng-click directive to the button. Type **ng-click='onButtonClick()'** as shown in Listing 10-1. This will be resolved by our controller, giving us separation between the visual parts of the application and the business logic.

Listing 10-1. Defining a Method Using the `ng-click` Directive

```
<button ng-click="onButtonClick()">Button</button>
```

Open `about.js` in the `controllers` folder. Controllers let you add all the business logic you need for this part of the application to work. In the `controller` method you see the name `AboutCtrl`, which matches what we saw in the `app.js` file. The other thing you see in the controller method is the `$scope` property.

`$scope` lets you add methods or properties to the view that you are working with. Here we will deal with a function that was declared in the HTML file. Because `ng-click="onButtonClick()"` was defined in the HTML code, it is part of the scope of this controller.

Listing 10-2 shows how we get our controller and HTML to work together.

Listing 10-2. Using the Controller to Define the Function Declared in the HTML File

```
angular.module('chapter10App')
.controller('AboutCtrl', function ($scope, $http) {
    $scope.awesomeThings = [
      'HTML5 Boilerplate',
      'AngularJS',
      'Karma'
    ];
  $scope.onButtonClick = function(){
    console.log('hello world');
  };
  });
```

If the app is currently running in the browser, it should see that the JavaScript file has been updated and recompile everything. Once that is done, you can click the button, look at the console, and see the message.

Otherwise, go to the command line and in the main folder of your application type **grunt serve**.

This is just the beginning of being able to separate the application's view from its business logic. For example, what if you wanted display a list of items?

Going back to the controller, we will add a JSON object to our scope; Listing 10-3 shows the code.

Listing 10-3. Data Assigned to the Scope of the about Controller

```
$scope.bands = [
{'name':"Information Society", 'album':"_hello world"},
{'name':"The Cure", 'album':"Wish"},
{'name':"Depeche Mode", 'album':"Delta Machine"}];
```

Now that we have data we need, the next step is to pass the data to the HTML template. Back in the `about.html` file, we will use the `ng-repeat` directive (Listing 10-4).

Listing 10-4. The `ng-repeat` Directive Loops through the Data Provided by the Controller to Display the Correct Number of Items in the List

```
<ul>
<li ng-repeat="band in bands">{{band.name}}<p>{{band.album}}</p></li>
</ul>
```

At this point your page should look something like Figure 10-3.
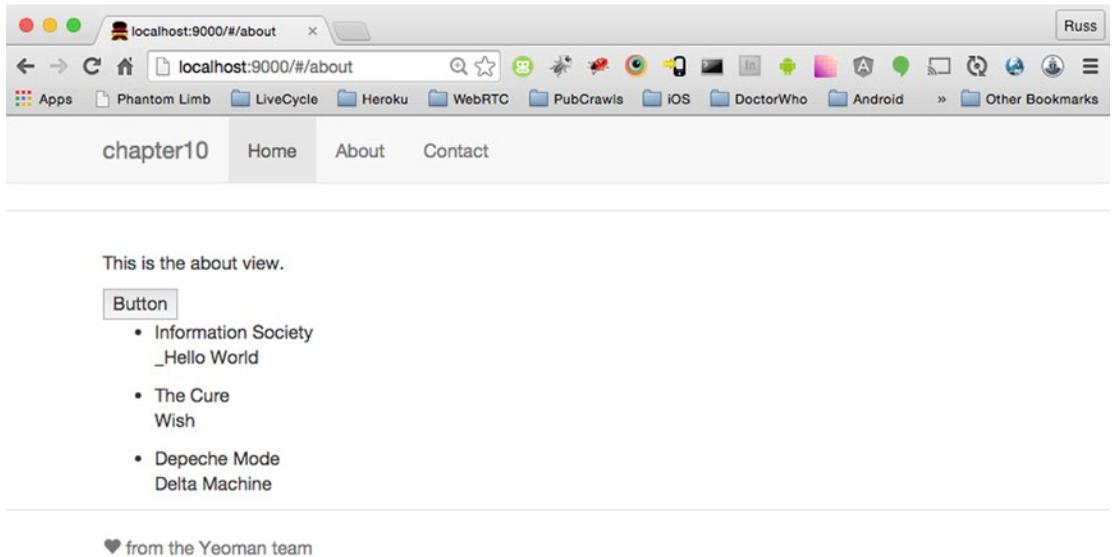
Figure 10-3. *The data from the controller rendered on the page*

So far, we have been able to connect the controller to items that are defined in the HTML view using directives. Our data has also been defined in the controller. So, you may ask, suppose we wanted to get the data from an outside source; how do we make a call to a remote server and display the results? We will take that up in the next section.

Remote Data Sources

Let's take our button method and use it as a way to get some remote data.

We are going to use the $http service, which will handle the remote call for us. This is similar to the AJAX method in JQuery. In order to take advantage of the service, we first need to add it to the controller method.

In the controller method add the $http service. It should now look something like Listing 10-5.

Listing 10-5. Adding the $http Service to the AboutCtrl Controller

```
.controller('AboutCtrl', function($scope, $http){
$scope.awesomeThings = [
'HTML5 Boilerplate',
'AngularJS',
'Karma'
];
});
```

Now that the service is available to the controller, we can use it with the onButtonClick method. We are going to use the get method, which matches the REST get verb. For more information about REST methods, look at this Wikipedia article:

http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods

We'll also use JSONPlaceholder, a service that lets you test your REST requests and return fake data. In the onButtonClick method, remove the existing code and add the code shown in Listing 10-6.

Listing 10-6. Making an HTTP Call and Assigning the Results to the results Property

```
$http.get('http://jsonplaceholder.typicode.com/photos').success(function(data){
    $scope.results = data;
});
```

Looking at this code we see that we are using the get method but we are also listening for the success event. If the call is successful, we use an anonymous function to assign the result to a property called results.

Now that we have the results, we can update the template to display not just the text but also the thumbnail images that return from the service. We do that by using another directive for the image tag.

Open about.html and update the existing code. Remove the previous list and add the fcode shown in Listing 10-7 to the template.

Listing 10-7. Creating an Unordered List Based on the Results That Came Back from the REST Service

```
<ul>
    <li ng-repeat="result in results">
    <p>ID: {{result.id}}</p>
    <p>{{result.title}}</p>
    <p><img ng-src="{{result.thumbnailUrl}}"/></p>
</li>
</ul>
```

Here we are using the ng-src directive to tell the image tag where to find each image from the list. Just as in the previous example, this will loop though the whole list of results and display them all on screen. One of the nice things about using this directive is that the image tag will not try to display an image until the data has been received. So we just need to get results back from the REST service.

At this point the page should look something like Figure 10-4.
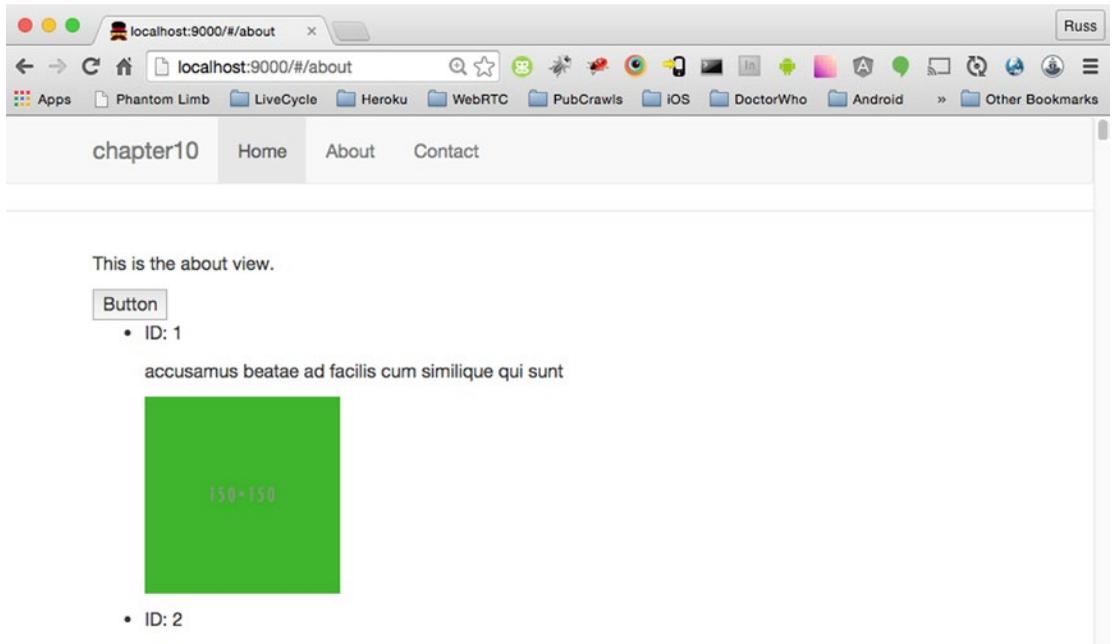
Figure 10-4. *Displaying the results from a REST service*

With a few steps we have a site that is able to retrieve data from a remote source and render it to the browser when we need it. Angular has many features, but we are going to cover one more. Our next lesson will cover routes.

Routes

Routes allow us to create custom URLs for our application and give us the ability to bookmark a page so we can go directly to it later.

We saw an example of this in the app.js file. The .config method uses the $routeProvider API. Within the series of when statements, we are able to work out what HTML template will be loaded and which controller will be used with that template.

Now that we have a good feel for how this works, what if you wanted to pass parameters to the application? For example, suppose we only wanted to show one post from the previous example. Here we will create a route that will do exactly that.

If you created the application with Yeoman as in the previous lessons, then go over to the command line and type:

```
yo angular:route post
```

Using this command, Yeoman will update the app.js file by adding a new a new route. It will also create a new JavaScript file for your controller and an HTML file for the view. Not bad for one command.

If the application is running, you can go to the browser and type:

```
http://localhost:9000/#/post
```

You should see that the post view is ready to go.

The goal here is to load a post based on the number added to our URL. So, for example, if the URL looks like this:

```
http://localhost:9000/#/post/4
```

You should see the fourth post in the list based on the service we used in the last example.

Route Parameters

In order to get this to work we need to make our routing function a little more flexible. Open app.js; here we are going to update the post route so it can take variables in the URL.

It should go from:

```
/post
```

to:

```
 /post/:postId
```

By adding :postId we create a variable that we can use in the controller. This variable will represent the number in the URL. To illustrate that, let's update the controller.

Open post.js, where you'll see that inside the controller method you have an anonymous function using $scope. In our other examples we saw that $scope gives us the ability to control our HTML template. We will add an extra parameter called $routeParams so that we can access our variable in the URL.

Now we can grab the variable from the URL and assign it to $scope. This will enable us to display it once we update the template.

Type the following in the controller method:

```
$scope.postId = $routeParams.postId;
```

To see our number on screen, we can quickly update the template.

Open the post.html file in the views folder. Here we can quickly update this template. First remove the copy between the paragraph tags, so that it looks like this:

```
<p>{{postId}}</p>
```

With that done, you can type a number into the url and see it displayed on screen. At this point, the browser should look like Figure 10-5.
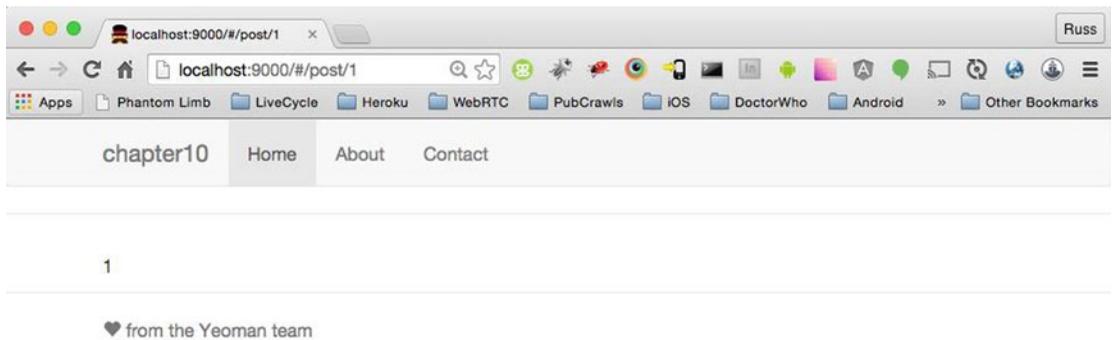
Figure 10-5. *Displaying a variable on screen based on the URL*

That wasn't so bad. So now we just need to connect it to a GET request and display the results. The code here will be very similar to what we did before. In the post control we need to add the $HTTP service so we can make the REST call. Listing 10-8 shows the code.

Listing 10-8. The Complete PostCtrl with Both the $routeParams and $http Services Added

```
.controller('PostCtrl', function($scope, $routeParams, $http){
$scope.awesomeThings = [
'HTML5 Boilerplate',
'AngularJS',
'Karma'
];
});
```

Right under this we will make the same REST call as before but this time add the postId variable, as shown in Listing 10-9.

Listing 10-9. Using the $http Service and routeParam to Get a Single Result

```
$http.get('http://jsonplaceholder.typicode.com/photos/'+$routeParams.postId).
success(function(data){
  $scope.results = data;
});
```

As for the HTML template (Listing 10-10), we'll use the same code as the about example; the only difference is that we will remove the ng-repeat directive and make sure we use the word results.

Listing 10-10. HTML Template for Displaying the Post

```
<ul>
    <li>
    <p>ID: {{results.id}}</p>
    <p>{{results.title}}</p>
    <p><img ng-src='{{results.thumbnailUrl}}'/></p>
</li>
</ul>
```

Now as you update the number in the address bar, you should see a new post (Figure 10-6).
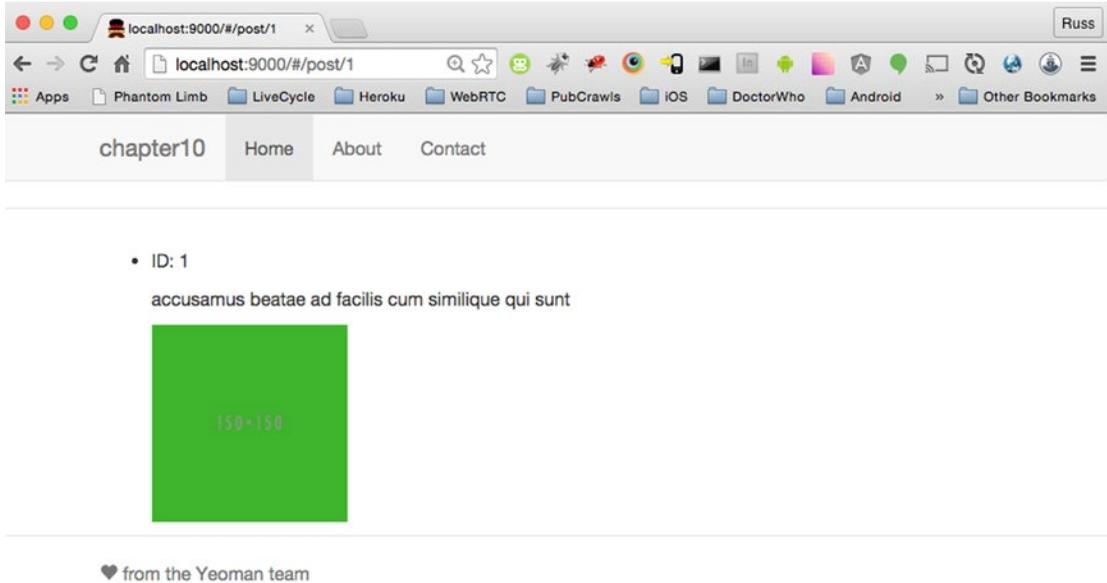
Figure 10-6. *Single post being displayed based on the URL variable*

Application Testing

As you read the documentation for Angular, one of the things you will see is how to write tests that cover the different parts of the application you are building.

Testing helps you make sure that the code you write is stable all the time. For example, if you need to make a change to a function, it should still give you the same result, even if you change how the function works. If that change creates an unexpected result somewhere else in the application, the test should let you know.

So with that out of the way, how do you test an Angular application? There are two types of tests we will consider: *unit testing* and *end-to-end (E2E) testing*.

Unit Testing

When you write a function, you consider what parameters it should receive, what it does with that information, and what the result should be. Having a test for that unit of code will assure you that it behaves the way you expect it to.

If you have been using the Yeoman-generated version of Angular from the previous chapters, we need to install a few extra items to get the tests working. Go to the command line and type

```
npm install grunt-karma --save-dev
```

This will install Karma.

Karma is described in the Angular developer guide as "a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests." In short, Karma will launch browsers, run the code against the tests that you have written, and give you the results. If you don't have a browser installed that you want to test (for example, IE if you are using a Mac), you can use a

service like BrowserStack (https://www.browserstack.com/) or Sauce Labs (https://saucelabs.com/). For the most up-to-date information about Karma, go to http://karma-runner.github.io/.

Next we need to install PhantomJS. Type

```
npm install karma-phantomjs-launcher --save-dev
```

at the command line.

PhantomJS is a *headless* browser—a browser without a user interface. By including it, you can run your application in a browser, and all the commands will be executed from the command line. For the latest information about PhantomJS, go to http://phantomjs.org/.

Finally, type

```
npm install karma-jasmine --save-dev
```

at the command line. This will install Jasmine, the testing framework we'll use to test our application. You'll find documentation at http://jasmine.github.io/.

At this point, let's make sure that everything works, by typing **grunt test**. This should run the tests that are in the test folder.

Adding New Tests

One of the benefits of using Yeoman is that when you create new controllers, it also creates corresponding files for unit tests.

Look at the main folder. There you will find a test folder. Inside that folder will be a spec folder, containing a controllers folder, which contains files that will unit-test every controller that has been created.

Open about.js so we can see how to test a controller.

Before we start writing tests, let us first look at the code that exists to get an idea of what is going on.

At the top there is a describe method, which is used to talk about the tests that are about to be written. The describe method can be used to describe at a high level everything that is about to be tested.

Next there is a beforeEach method, which will run before each test. This gives us access to the entire application by loading it as a module.

Two variables are created, AboutCtrl and scope; then we create another beforeEach method, which assigns the variables the values of the controller and the scope inside that controller, as if we were using the controller directly.

Finally we can write our tests, which we describe in a series of it methods. This helps to make the tests easy to document. Here you describe how the function should work.

The default test has the message "should attach a list of awesomeThings to the scope"; then it runs a function with an expect method. This method is important because it give you a chance to test the expected result. In this case, we check the length of the array awesomeThings and expect it to be 3. If it isn't, the test will fail.

We can now test the length of the array bands that we created before. Add a new it method as shown in Listing 10-11.

Listing 10-11. Unit Test for the number of Bands That Should Be In the about Controller

```
it('should have at least 3 bands', function(){
    expect(scope.bands.length).toBe(3);
});
```

If you go to the command line and type **grunt test**, this test should pass. If you had a different number in the expect method, for example 2, the test would fail. You can see that in Figure 10-7.

Figure 10-7. *Test expected two items and received three*

We can also check the value of an item in the array (Listing 10-12).

Listing 10-12. Unit Test to Check the Value in the Array

```
it('should have the second album be Wish', function(){
    expect(scope.bands[1].album).toEqual('Wish');
});
```

This is a very quick overview of how to test the controller. From here, you can test methods that have been written and evaluate the result. Jasmine gives you many ways to make sure the code you write is solid. Let's take another look at testing, by opening post.js and testing our HTTP request.

Testing HTTP Requests with $httpBackend

In our previous example, we tested some of the data that was associated with the controller. In that case the data originated inside the controller. In most applications, you will get data from a remote source. So how do you write a test where you do not have control of the data source? In this case you use $httpBckend to make sure that the requests you create work independently of the service.

This test will recreate everything we did with $routeParams. It will be self-contained and will not actually make a call to the server.

First, we will add a few variables. In addition to PostCtrl and scope, add httpBackend and routeParams. In this case we are not referring to the directives, so you don't need to add the dollar ($) sign.

The second beforeEach method is where we are currently initializing the controller. This is where we will add directives, just as in the real controller; here add $httpBackend and $routeParams.

Now we assign values to the variables we created earlier. In the browser we were able to get a single post by assigning a value to postId from the URL. We simulate it as shown in Listing 10-13.

Listing 10-13. Assigning Values So We Can Simulate Getting a Value from the Browser (Part 1)

```
beforeEach(inject(function($controller,$rootScope,$httpBackend,$routeParams){
  scope = $routeScope.$new();
  routeParams = $routeParams;
  routeParams.postId = 1;
  httpBackend = $httpBackend;
  httpBackend.expectGet('http://jsonplaceholder.typicode.com/photos/'+routeParams.postId).
  respond({id:'1', title:'title 1', thumbnailUrl:'myImage.png'});
PostCtrl = $controller('PostCtrl', {
  $scope: scope
});
httpBackend.flush();
});
```

Since we are not loading this in a browser to make sure it works, we will hard-code the value of `postId` to 1. Then, using `httpBackend`, we simulate the call the same way it's done in the controller. In this case we use the method `expectGet`. This will simulate an HTTP `GET` request. If you wanted to do a `POST` request, you would use the `expectPost` method.

The `response` method gives us the payload that we can test against. Here we pass back a simple object that is just like what the API delivers.

After the scope is assigned, we see the `httpBackend` object once again using the `flush` method. This will make the response available for testing as if you had made the HTTP call.

Now on to the tests. Just as in the other example, a series of `it` methods describe what you are expecting.

Let us first make sure that we are only getting one result back from the server (Listing 10-14).

Listing 10-14. Assigning Values So We Can Simulate Getting a Value from the Browser (Part 2)

```
it('should be a single post', function(){
    expect(scope.results).not.toBeGreaterThan(1);
});
```

Jasmine makes the tests easy to read. We have results and just want to make sure that we have only one object.

If we wanted to make sure there was an ID property on this object, we'd use the code shown in Listing 10-15.

Listing 10-15. Checking the ID Property

```
it('should have an id', function(){
    expect(scope.results.id).toBeDefined();
}
```

Just as before, we can add tests that will allow us to understand the controller as we add more functionality to it. The process of writing tests first as you develop your code is called *test-driven development*. In this approach you first write a test knowing that it will fail, and then go back and write a minimum amount of code to get the test to work. After that, you can refactor as needed. Jasmine is used to test units of code. It is also used to test integration with the browser. So how do you simulate button clicks on multiple browsers? After all, from the history of web development, we know even things that seem simple sometimes don't work in certain browsers. This is where Protractor comes in.

End to End Testing with Protractor

Protractor (http://angular.github.io/protractor) is a tool that lets you run real browsers and run tests in them. For example, you can make sure that when a button is clicked, it submits a form. Unlike unit testing where you are testing small units of code, end to end (E2E) testing lets you test whole sections of the application against an HTTP server. It is similar to you opening the site up in a browser and making sure things work. The nice part about this is that it's automated.

Tasks like this should be automated. Protractor is able to do this because it is built on top of WebDriver, a tool used to automate testing inside a browser. Protractor also has features that support Angular, so there is very little configuration on your part.

Let's install Protractor and run a few tests. At the command line, type:

```
npm install –g protractor --save-dev
```

As it did with other node packages, this line will install Protractor globally so you can use it with other projects.

There is some configuration we need to do to get this working. Let's create the config file.

Create a new file, which we will call protractor.conf.js, and save it in the test folder right next to the karma.config.js file.

In this file, we will give Protractor some information about where to find the Selenium server, what browsers to run, and where the test files are. Listing 10-16 shows the code.

Listing 10-16. Basic Configuration File for Protractor

```
export.config = {
    seleniumAddress: 'http://localhost:4444/wd/hub',
    multiCapabilities: [{browserName: 'firefox'},{browserName: 'chrome'}],
    baseUrl: 'http://localhost.9000',
    framework: 'jasmine',
    specs: ['protractor/*.js']
};
```

There are a few things to unpack here, so let's take a look.

The seleniumAddress property tells Protractor where the Selenium server is running. Next, the multiCapabilities property tells Protractor which browsers to run the tests on. As you can see, it's an array of objects listing the name of each browser.

Because we are testing locally, we can only test browsers that are installed on the machine. So you can't test IE if you are running a Mac. If you need to test browsers like IE or mobile browsers, you can add properties that will allow you to connect to either SauceLabs or BrowserStack.

Next we have the baseUrl property, which tells Protractor what server is hosting the application that is being tested. It's important that the site is running on a local server when you run the tests. The framework property is set to Jasmine, because that is the framework we are using for our tests.

The specs property is important because it is where we tell Protractor what folder has the tests in it. In our case, it is in the protractor folder, and we use the wild card to tell it to look at any JavaScript file in that folder.

We now have Protractor set up. It's time to write some tests. Within the tests folder create a Protractor folder. Here we will write a basic test.

Create a file called app-spec.js in the Protractor folder. The format will be very similar to what we did in the previous examples.

We start with the describe method for the suite of tests we are about to run. Right after that is our set of it statements (Listing 10-17). To make this simple we will use examples right from the Protractor site.

Listing 10-17. It Method Checking Whether the Site Has a Title

```
it('should have a title', function(){
    browser.get('http://juliemr.github.io/protractor-demo');
    expect(browser.getTitle()).toEqual('Super Calculator');
});
```

We now have everything we need. We are going to run this from the command line. If you are not already running the serve task and looking at the site locally, type:

```
grunt serve
```

This will let you run the site from the local server on port 9000, which is where we told Protractor to look when it runs the tests.

Now type

```
protractor test/protractor.conf.js
```

This will look in the test folder, run the configuration file, and launch the browser so it can run the tests. As shown in Figure 10-8, we should be getting a passing result.

```
Russs-MacBook-Pro:angularSite asciibn$ protractor test/protractor.conf.js
[launcher] Running 2 instances of WebDriver
.
-----------------------------------
[chrome #2] PID: 16221
[chrome #2] Specs: /Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/protractor/app-spec.js
[chrome #2]
[chrome #2] Using the selenium server at http://localhost:4444/wd/hub
[chrome #2] .
[chrome #2]
[chrome #2] Finished in 1.301 seconds
[chrome #2] 1 test, 1 assertion, 0 failures
[chrome #2]

[launcher] 1 instance(s) of WebDriver still running
.
-----------------------------------
[firefox #1] PID: 16220
[firefox #1] Specs: /Users/asciibn/Documents/Projects/apress/proJavaScript/chapter12/angularSite/test/protractor/app-spec.js
[firefox #1]
[firefox #1] Using the selenium server at http://localhost:4444/wd/hub
[firefox #1] .
[firefox #1]
[firefox #1] Finished in 1.535 seconds
[firefox #1] 1 test, 1 assertion, 0 failures
[firefox #1]

[launcher] 0 instance(s) of WebDriver still running
[launcher] chrome #2 passed
[launcher] firefox #1 passed
Russs-MacBook-Pro:angularSite asciibn$ ▌
```

Figure 10-8. *Protractor test passing in both Firefox and Chrome*

This will direct the browser to the URL and check the title. Pretty simple.

Now let's write a test where we can use the same calculator to add two values together and then check the result. Listing 10-18 shows the code.

Listing 10-18. Typing in Two Text Fields and Then Testing the Result

```
describe('Protractor Demo App', function() {
  it('should add one and two', function() {
    browser.get('http://juliemr.github.io/protractor-demo/');
    element(by.model('first')).sendKeys(1);
    element(by.model('second')).sendKeys(2);

    element(by.id('gobutton')).click();

    expect(element(by.binding('latest')).getText()).
      toEqual('3');
  });
});
```

Here we are able to look right into Angular's ng-model directive to access the text fields and give them values. Then we can find the button by its ID and click on it. That click triggers the method doAddition. Finally, we are able to look at and check the value that is being updated by the result of the method.

Summary

This was a very high-level look at AngularJS and writing both unit and end-to-end tests. Both of these topics can be books own their own.

As your projects become larger and more involved, having a framework can help you keep everything organized. In addition, being able to test your code gives you more confidence in the code that you write.

Unit tests lets you know that your front-end code is working as expected. Integration tests lets you know if that same code works with different browsers.

The site Year of Moo (www.yearofmoo.com/2013/09/advanced-testing-and-debugging-in-angularjs.html) has an excellent breakdown of testing and debugging with Angular. It covers topics like when you should write tests, testing in older browsers, and what to test and what not to test.

Now you can refactor with confidence, knowing that what you wrote will not break the app; and if it does, you will know about it as soon as possible.

■ ■ ■

The Future of JavaScript

We have taken quite the tour of JavaScript in this book. It is clear that JavaScript is a language in transition. From its humble beginnings as something of a toy language, JavaScript has ascended to the level of an enterprise-critical language. In the process, the seams have begun to show and, frankly, come somewhat loose. When developers from more mature languages come to JavaScript, they often marvel at what we have accomplished, given JavaScript's limitations. They are given to wonder at how the language got so far, and whether it will improve in the future.

JavaScript's "graduating class" of 1995 has some of the brightest lights of programming: Java, Ruby, PHP, and even ColdFusion are all thriving languages to this very day. Many developers would say that JavaScript's classmates are far ahead of JavaScript. Yet many of them are taking their cues from JavaScript, looking at its use of prototypes, its implementation of functions as first-class citizens, its flexible style and more as inspiration for their own new features.

What does the future hold for JavaScript? Where does it go from here? Thankfully, unlike our own dim and obscured futures, the future of JavaScript has a road map and even some evolving specifications. ECMAScript 6 will probably be a fully adopted (if not quite implemented) standard by the time this book is published. ECMAScript 7 is already in development and under debate. To deploy a cliché, JavaScript's future looks bright indeed.

Let's take a look at what lies ahead for JavaScript. We will discuss a little bit of how we got here and where we are going, looking at the standards process. Then we will look at what we need to do to use JavaScript with our current set of tools. But the bulk of this chapter is concerned with the details of ECMAScript 6: the language features you can expect to be working with over the next few years. We will even hint at some of the distant future, which may or may not come to pass...

Of course, we do not have the space to go over the entirety of the ECMAScript 6 specification. We have worked to pick and choose the more useful, better defined, and most interesting features to look at.

The Once and Future ECMAScript

We should start with what we know. The European Computer Manufacturers Association, now known as Ecma International, is the body that oversees the standard to which JavaScript adheres. One could write a book on how this came to pass, and it's not really important for our jobs as JavaScript programmers. What is important is that a group within Ecma, Technical Committee 39 (TC39), has taken up the banner of JavaScript standards and is promulgating updates. Perhaps more important is the fact that various computer manufacturers, software companies, and other interested parties are vested in the success of this standard. We, as the JavaScript community, have a viable system for directing the future of the language. This should help to head off some of the rancorous differences we have endured in the past, as well as to streamline the process of positioning JavaScript as an effective, enterprise-class language.

The process of wrangling JavaScript into an effective and effectively governed standard has been a long one. Most of the features that we think of as "standard" JavaScript came from ECMAScript version 3. This version comes from the era when Ecma was still playing catch-up with the browser makers. Although there were efforts to create a fourth version of the ECMAScript standard, they were ultimately abandoned. Ten years after version 3, version 5 of the standard was offered in 2009. This aimed to set a new status quo, catch up to the intervening changes in the landscape, and clarify the many ambiguities from version 3. The standard was widely accepted and helped to clear the path for Ecma's resumption of its duties managing the JavaScript standard.

Even now, wide acceptance of ECMAScript version 5 is not a given. Internet Explorer 9 was the first version of IE to implement the standard. A significant portion of the world still uses IE 8 and earlier. The challenge for Ecma International and TC39 has not so much been setting the standard, but getting the audience to upgrade to the standards once they are offered.

There was some confusion about what the next step would be after version 5. In the end, two tracks were taken: there was a 5.1 standard, which brought ECMAScript in line with the International Organization for Standardization's specification for ECMAScript (a long and boring story in itself). And there would be a new version of the ECMAScript standard, version 6, often referred to as ECMAScript Harmony (the name comes from a variety of proposals over time to harmonize various standards of JavaScript, ECMAScript, JScript and so on, as well as the original code name for ECMAScript version 4).

As programmers, we are most interested in the new standard and what it will enable us to do. The ES6/Harmony proposal should be finalized by the middle of 2015. So let's leave the world of standards behind and talk about how we can work with Harmony today.

Using ECMAScript Harmony

Unlike ECMAScript 5, which had a long life as a *de facto* standard before it became a *de jure* standard, Harmony is leading, rather than following. This means that current (as of the writing this chapter) implementation of Harmony is spotty. We need a few tools to manage Harmony's various states of implementation. First, we will need resources to tell us which browsers implement which aspects of Harmony. Second, we will look at how to enable Harmony in those browsers, and third, we will examine some software tools that can transpile our ES6 code into ES5-compliant code. After that, we should be able to dive into some of the features of the standard that are either in wide use or widely expected to make it into the final standard.

Harmony Resources

TC39 maintains a wiki that allows the public to track the state of the Harmony proposal, at http://wiki.ecmascript.org/. Two pages are of particular interest: the requirements/goals/means page and the proposals page. The requirements page lists the methodology that guided the development of ES6. While not critical in our understanding of the language, it does inform us about why certain decisions were made, in terms of the goals and means defined for the Harmony project. For example, Harmony has a proposal for proper block scope, but implements it through the addition of a keyword (let) instead of simply redefining the way JavaScript interpreters work. The proposal itself follows the first two subpoints of the first goal: Be a better language for writing complex applications and libraries. But the implementation follows the fourth goal (keep versioning as simple and linear as possible) as well as the first means (minimize the additional semantic state needed beyond ES5). You can find the requirements, goals, and means of the Harmony project at http://wiki.ecmascript.org/doku.php?id=harmony:harmony.

The other important page for the Harmony process is the proposals page. This tracks the various proposals made for ES6 and the state of each one. The call for proposals was closed in 2011, so the page should not see additions. For the most part, you should see the existing proposals, and, occasionally, proposals that have been removed from the specification. You can find the proposals page at http://wiki.ecmascript.org/doku.php?id=harmony:proposals.

Specifications are wonderful as reference documents, but they are sometimes lacking in implementation details. We would like to have a reference that tells us the state of ES6 implementation across browsers and other JavaScript engines. Luckily, we have two such pages. The best page is by the noted JavaScript developer who calls himself kangax (Juriy Zaytsev), and is known as the ECMAScript 6 compatibility table. You can find it here: `http://kangax.github.io/compat-table/es6/`. The compatibility table breaks down ES6 implementation by feature and checks it against most modern browsers, desktop and mobile, as well as other JavaScript implementations like Node.js. The tests are somewhat simplistic, usually focusing on existence of a proposal, not the functionality thereof and not the implementation's conformance with the proposal. Nonetheless, it's a great starting point. kangax also maintains compatibility tables for ES5, the coming ECMAScript 7 specification, as well as nonstandard features like `__defineGetter__` or the `caller` property on functions.

Thomas Lahn also maintains the ECMAScript matrix, which tracks implementation of ECMAScript standards across current versions of JavaScript engines. You can find his efforts at `http://pointedears.de/es-matrix/`. Lahn's approach is somewhat different from kangax's. Lahn is interested in current JavaScript engines, so he only tracks JavaScript, V8 (Google Chrome's engine), Opera, and a few others. Contrast this with kangax, who is more browser- and software-oriented. Also, Lahn's matrix tracks all of ECMAScript, at least through version 6, so you can see the `let` keyword evaluated alongside arrays and `for` loops. His approach is more thorough, but also leads to a larger table (and a somewhat more slowly loading page). Nonetheless, this, like kangax's compatibility table, is an indispensable resource for the professional JavaScript developer.

Working with Harmony

There are four states in which a browser can work with the features of Harmony.

- The browser, particularly evergreen Chrome and Firefox, may have a ready-to-go implementation of a Harmony feature with no special effort needed on the part of the programmer. Very few features work this way as of publication time, though.

- Most browsers will require you to "opt-in" to using Harmony features. We will talk about this in detail shortly.

- If your browser does not have a feature enabled or implemented (or it is not implemented properly!), you may consider using a transpiler, which will let you write ES6-level code and then output ES5 that can run on the engine of your choice.

- Alternatively, you could use a polyfill for a particular feature from ECMAScript 6 you would like to use.

Obviously, the first state requires little explanation, so let's talk about the second state. Both Chrome and Firefox have their own quirks in working with ECMAScript 6.

In Chrome, you will have to go to the `chrome://flags` URL, which allows you to enable experimental features. Specifically, you will need to enable `chrome://flags/#enable-javascript-harmony`. Now, keep in mind that doing so may change Chrome's behavior in many circumstances, and could have odd results in the way Chrome renders certain pages. And the change to the state of `enable-javascript-harmony` is persistent. If you would prefer, instead, to change only at startup of a specific session, run Chrome from the command line with the `--javascript-harmony` switch. Additionally, some examples require running JavaScript in strict mode, which can be managed at the code level.

For Firefox, you do not have to change any settings when you start it up, but you may have to change your code. In general, Firefox will require you to label your code as being different from standard JavaScript. Add the type attribute to your script tags, and set the type to `application/javascript;version=1.7`, This should enable most Harmony features. If there are any additional changes needed to run Harmony code, we will note them with the specific feature they enable.

Interestingly, Internet Explorer requires the least configuration to run ECMAScript 6 code—"least" in the sense of "none." On the other hand, IE 10 has only four items in the specification implemented. Internet Explorer 11 has a total of 12 items in the specification implemented, but it lags far behind Firefox and Chrome. It can be said that while IE 11 has not implemented much, what it has implemented, it has done so simply.

Transpilers

The third option for working with ECMAScript 6 is a *transpiler*. A transpiler takes code written for ECMAScript 6 and cross-compiles it into ECMAScript 5-compatible code. There are a number of different transpiling tools. Addy Osmani maintains a list at GitHub of transpilers and polyfills. You can view the list at https://github.com/addyosmani/es6-tools. You can see from that link that there are a number of transpilers and polyfills. We will demonstrate using the Traceur transpiler to take some ECMAScript 6 code and run it in a browser using ECMAScript 5. Traceur is among the more popular transpilers, and it is also among the more frequently updated.

The easiest way to use Traceur is to load it via Node.js. Similar to the way we used Node as our own JavaScript VM in the chapter on JavaScript tools, we will use Node here to load up additional code via NPM. Start by loading Traceur

```
npm install traceur
```

This installs the Traceur transpiler, currently on version 0.0.72. Recall that you can use the -g option if you want Traceur to be globally available. Either way, you will probably need to update your PATH variable to include Traceur. On Windows, in your node_modules folder you will find the following file: .bin\traceur. cmd, which is a Windows batch file wrapped around running Traceur with Node.js. You should be able to run Traceur directly if you add the node_modules\.bin directory to your PATH. Check that Traceur is on your path by running traceur--version, which should return the version number or an error message if Traceur can't be found.

You can run Traceur against existing ES6 code. Invoke traceur from the command line and pass it, as an argument, a JavaScript file that contains ES6 code. (You could, of course, pass it a file that only has ES5 code in it, but where's the fun in that?) Traceur will run your code and output anything it prints to the console.

Consider an example of the new class syntax in JavaScript. Quickly, ES6 will allow you to create classes, though the code is just a syntactic wrapper around the functional style of type. The syntax is easy to read and figure out, so let's use it for our Traceur example, shown in Listing 11-1.

Listing 11-1. ECMAScript 6 Classes with Traceur

```
class Car {

  constructor( make, model ) {
    this.make = make;
    this.model = model;
    this.speed = 0;
  }

  drive( newSpeed ) {
    console.log( 'DEBUG: Speed was previously %d', this.speed );
    this.speed = newSpeed;
    console.log( 'DEBUG: Speed is now %d', this.speed );
  }
```

```
  brake() {
    this.speed = 0;
    console.log( 'DEBUG: Setting speed to 0' );
  }

  getSpeed() {
    return this.speed;
  }

  toString() {
    return this.make + ' ' + this.model;
  }
}
var honda = new Car( 'Honda', 'Civic' );
console.log( 'honda.toString(): %s', honda.toString() );
honda.drive( 55 );
console.log( 'The Honda is going %d mph', honda.getSpeed() );
```

As you can see, we create a type, Car, and define three properties (make, model, and speed) as well as a few methods which are wrappers around a property (brake, drive, getSpeed) or conveniences (toString). Nothing too crazy.

This code will not run in any modern browser. We know, because we tested it. Also, if you look at kangax's compatibility tables (as of publication time), you will see that classes are not implemented by any of the major browsers. So this code is a good choice for experimentation with Traceur.

If you were to save the code in a file (classes.js in the folder for this chapter, as a matter of fact), you could run it with Traceur:

```
traceur classes.js
```

Your output would look something like this:

```
C:\Projects\PJT\FutureOfJS
>traceur classes.js
honda.toString(): Honda Civic
DEBUG: Speed was previously 0
DEBUG: Speed is now 55
The Honda is going 55 mph
```

As you can see, Traceur handles the code just fine. Behind the scenes, Traceur compiled the code down to ES5, and then simply ran the code using Node.js itself. No big deal.

But what about browsers? Well, we have two different options. We can use Traceur to generate output files that run in browsers which support ES5. Alternatively, we can use ES6 code directly in the browser and have Traceur transpile it live. Start with transpiling output. Use the --out option to tell Traceur to generate output to a file of your choosing. The output file is not a standalone file. You can run it with Traceur, obviously, but you cannot simply include it into an HTML page. You will need to load up the Traceur runtime first, and then the script you want to run. Listing 11-2 is an example of a Traceur HTML shell.

Listing 11-2. HTML Shell for Traceur

```
<!DOCTYPE html>
<html>
<head>
  <title>Traceur and classes</title>
</head>
<body>
<h2>Running Traceur output in the browser</h2>

<script src="../node_modules/traceur/bin/traceur-runtime.js"></script>
<script src="classes-es5.js"></script>
</body>
</html>
```

Note that we are loading the `traceur-runtime.js` file from the bin file of Traceur. This file is a distillation of the code needed to run Traceur-transpiled files. (The other, considerably larger, file in the directory is the code for Traceur itself.) Loading Listing 11-2 (available as `classes-es5.html` in the chapter folder) yields results on the console as expected. Perhaps more importantly, it works great for current versions of Firefox, Chrome, and Internet Explorer.

If you wanted to work with ECMAScript Harmony code directly, you could always have Traceur transpile your code live. We can use the original Harmony code from Listing 11-1, and modify the HTML shell from Listing 11-2 as follows:

```
<script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
<script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>
<script src="classes.js" type="module"></script>
```

We have switched to using Google's GitHub repository for Traceur simply because it's the easiest way to access the second file: `bootstrap.js`. This file is not included in the NPM install of Traceur. It is also not included in the Bower install of Traceur. So we will refer to it directly. Bootstrap allows you to run Traceur from within a JavaScript context. Also, we are now referring to the file `classes.js` as being of type `module`. This is a convention of `bootstrap.js`, which loads `classes.js`, via Ajax, as an explicitly ES6 file. Additionally, the `type` attribute has the side effect of not loading error-provoking code into the browser. As an alternative, you can simply include the JavaScript ES6 code in an inline script block, though you will still need the `type` attribute set to `module`.

While in-line transpiling is a fun experiment, we cannot recommend it as a regular way to develop or deploy. It's extra work that has to be done every time a page loads, work that can be done once by transpiling to an output file. Not to mention that `traceur.js` plus `bootstrap.js` plus your code is a much more significant download than `traceur-runtime.js` plus your code.

Polyfills

Finally, for some aspects of ECMAScript 6, you could load a polyfill into your page. The use case for this solution is a bit narrower, but the application is much broader. Instead of the whole set of ECMAScript 6, which you get with something like Traceur, you can focus on exactly the features you want by using polyfills. But this comes at a cost. It is easy and logical for a polyfill to provide the new methods on the prototypes for String, Number, and Array, or to implement the WeakMap and the Set. On the other hand, polyfills cannot, by their very nature, substitute language features like `let`, `const`, or arrow functions. So the set of Harmony features you can access with polyfills is limited.

All that said, for the implementable polyfills, there are quite a number of high-quality implementations. Addy Osmani's directory of ES6 tools contains a section on ES6 polyfills. Of note is Paul Miller's ES6-Shim (`https://github.com/paulmillr/es6-shim/`), which has polyfills for most aspects of Harmony that can be polyfilled. When we go over the features list for ECMAScript 6 later in this chapter, we will note those features that are provided by ES6-Shim.

ECMAScript Harmony Language Features

ECMAScript 6 introduces a large number of new language features. These are features that fill in what have been glaring blind spots in JavaScript (block scope), trim down the syntax to focus on the core of functionality (arrow functions), and expand JavaScript's ability to handle more complex code patterns (classes, modules, and promises).

Take block scope as an opening example. JavaScript's odd approach to scope and hoisting (the practice of lifting variable and function definitions to the top of their local scope) has been a stumbling block for new JavaScript programmers for years. It is also a cultural impediment: programmers from "real" languages (whatever that might mean) scoff at JavaScript because it lacks block scope (or classes, or this, or that). Regardless of the substance of this criticism, it is preventing some people from coming to experience the good parts of JavaScript. So let's address it and move forward.

The `let` keyword allows you to scope variables to an arbitrary block. Variables scoped with `let` are not hoisted. Those are the two critical differences between `let` and `var`. It could be said that variables scoped with `let` act the way you would expect most local variables to act (whereas variables scoped with `var` have some interesting features/capabilities). If you try to access a `let`-scoped variable outside its block, you get a ReferenceError, much as you would if you tried to access an inaccessible `var`-scoped variable. It isn't more complicated than that. Use `let` in place of `var` when you want block scoping.

As a companion to `let`, there is also `const`, which allows you to declare a constant-value variable. For variables declared with `const`, you should initialize at declaration time because otherwise, what's the point? You will not be able to change the value of the variable later on! Like `let`, `const` variables are not hoisted, and they are scoped to the block they are declared in. The semantics of trying to access `const` variables are a little different from browser to browser. If you try to modify a variable declared `const`, your browser will either fail silently or may throw an error of some type. Constants enable certain bits of code to be compiled into faster code, as the JavaScript engine knows that they will never change. Also, when paired with some of the collections we will see later in the chapter, you can use constants to store private data for a class. See `http://fitzgeraldnick.com/weblog/53/` for details.

Arrow Functions

Another cateogry of improvements in ECMAScript 6 is the introduction of syntactic changes that simplify certain declarations. One of these changes is the arrow function. Arrow functions serve to give you a shorter syntax for defining functions, particularly in-line functions. Consider this code:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach(function(num) {
  // Do something with the number here
});
```

It's not unwieldy, but it is a touch verbose. JavaScript has a unique challenge: try to minimize literal code length, while not compiling code. Having to define functions using the keyword `function` is not light on bandwidth. And the more functions we use, the more times we have to use the word `function`. It would be nice to have shorter function syntax.

Inspired by CoffeeScript, TC39 introduced a proposal for arrow functions or, more clearly, arrow-defined functions. The previous example can be written like so:

```
var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];
numbers.forEach(num => {
  // Do something with the number here
});
```

Somewhat more elegant, no? Here are the critical parts of the syntax:

```
arguments => { code }
()         => { code } // No arguments
i          => { code } // One argument
(i, j)     => { code } // multiple arguments
```

For specifying the body of the function, you may wrap multiple lines of code within a set of curly braces or, if you have only one line of code, you may leave it bare. Thus

```
x => x * 2
```

Is the equivalent of

```
function(x) {
  return x * 2;
}
```

There are a few differences between arrow functions and regular or standard functions. First, arrow functions are always bound to the context in which they are defined. This sounds complicated. You may have run into this problem:

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    courses.forEach( function ( course ) {
      console.log( 'Ask %s if he can teach %s', this.teacher, course );
    });
  }
};

courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

When you run this code, you will see output like this:

```
Ask undefined if he can teach Greek
```

Clearly not what we want. The context and value of this within the forEach function refers back to the canTeach() function, not to the courseAssignments object. Usually the problem is repaired this way:

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    var that = this; // Store the context of canTeach
```

```
    courses.forEach( function ( course ) {
      console.log( 'Ask %s if he can teach %s', that.teacher, course );
    });
  }
};
```

```
courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

Note the code in bold, which highlights a strategy for storing the context of this at the canTeach level, instead of inside the forEach.

Arrow functions simplify this situation dramatically. They are automatically bound to the proper context. You could rewrite the previous code as

```
var courseAssignments = {
  teacher : 'Stephen Duffy',
  canTeach: function(courses) {
    courses.forEach( course => console.log( 'Ask %s if he can teach %s', this.teacher,
course ) );
  }
};
```

```
courseAssignments.canTeach(['Greek', 'Latin', 'Theology', 'History']);
```

With the arrow function, this.teacher is automatically bound to the context of this in canTeach, which is what we wanted all along. Terrific!

There are two other differences with arrow functions. Arrow functions cannot be used as constructors. They lack the internal code to act as such. Arrow functions also do not support the arguments object. This is fine because ECMAScript 6 also defines and permits parameter default values and rest parameters, so we will no longer need to rely on the arguments object in functions in general.

Classes

One of the biggest syntactical changes in ECMAScript 6 is the introduction of classes. The keyword class had been reserved in JavaScript since its inception. But there was no implementation behind it. With the rise of object-oriented JavaScript, TC39 acknowledged that JavaScript needed a more syntactically clear way to implement classes and inheritance. At the same time, there was no desire to add yet another way to implement types, classes, or something like classes. The idea was to simplify and to relieve confusion, not create more confusion.

Classes, ECMAScript 6:The ECMAScript Harmony specification settled on using the class keyword as syntactic sugar to implement function-based types. If that reads a bit strange, here's what actually happens:

You write a JavaScript class, based on the new ES6 syntax.

The JavaScript engine compiles it down to a function which defines a type.

You interact with that class the same way you would a type.

So the semantics of interacting with the class/type and its instances have not changed. And many of the semantics of defining a class/type are the same as well. Let's go back to our previous example for more detail. Recall how we defined a class in JavaScript:

```
class [classname] {
  constructor(arg1, arg2) { ... }
  [other methods]
}
```

Define the class with the `class` keyword. Include a function called `constructor`. This is the function that will be called when invoking `new [classname]` later on. Supply other methods as needed. The other methods will be copied onto the prototype of the function defined as the constructor. Pretty nifty!

Classes will also have two critical features long desired for the object-oriented side of JavaScript: easy inheritance and a super accessor. Inheritance is available via the `extends` keyword:

```
class Car extends Vehicle
```

Within a subclass, you can use `super` as an accessor to the superclass's methods and properties. Unlike some languages (Java for instance), ECMAScript 6 doesn't let you invoke `super` as a method itself. Rather, it holds a reference to the superclass, similar to the way `this` holds a reference to the current instance.

Unfortunately, classes are not implemented at all in any of the major current browsers. The class specification will be part of the final specification of ECMAScript 6 released in 2015, and it seems that browser makers are waiting until the final specification to make sure their implementations don't get anything wrong. In the meantime, the Traceur transpiler handles classes quite well, including `extends` and `super`.

Promises

Back when we were talking about Ajax, we discussed a bit of the problem with Ajax-based systems: code flow. When involved with a function that returns results asynchronously, it is difficult to program dependent code. For a long time, there was no concrete, straightforward solution. Most programmers either wrote very long callbacks (which may have, in turn, called their own asynchronous functions, complicating the matter!) or used named functions, but bounced around the call stack. (Not to mention that this method introduces another complication; there are many named functions. Do you manage them with a module? Just a namespace?) Asynchronously running code is at the heart of JavaScript's feature list, so we needed a better way to manage asynchronous interactions. Enter promises.

A *promise* is a pattern that helps to manage asynchronously-returning code. It encapsulates that code and adds a layer of management that tastes of event-handling. We can register code with the promise that should run if it returns successfully or unsuccessfully. When the promise does complete, the appropriate code is run. We can register any number of functions to run on success or on failure (very much like event handling), and we can register a handler at any time (whether the promise has completed or not; very much NOT like event handling).

Let's take a technical walkthrough of a promise to clarify the feature's operation, as demonstrated in Listing 11-3. Broadly, the promise has two states: pending or settled. The promise is pending until the asynchronous call it wraps around returns or times out or otherwise finishes. At that point, the promise is settled. The state of being settled has two flavors (if you will): resolved and rejected. A resolved promise settled successfully, a rejected promise settled unsuccessfully. As promises are arbitrary (technically, they do not even require asynchronous code!), the definition of rejection or resolution is really up to you.

Listing 11-3. Promises

```
var p = new Promise( function(resolve, reject) {
  // Do some things here, maybe some Ajax work?

  setTimeout(function() {
    var result = 10 * 5;

    if (result === 50) {
      resolve(50);
    } else {
      reject( new Error( 'Bad math, mate' ) );
    }
  }, 1000);

});

p.then(function(result) {
  console.log( 'Resolved with a value of %d', result );

});

p.catch(function(err) {
  console.error( 'Something went horribly wrong.' );
});
```

Admittedly, this is a bit contrived, but it is nonetheless a good simple example of some asynchronous code. At the heart of the matter is the call to setTimeout. Of course, the function we are scheduling for later execution does nothing other than run a mathematical equation. The critical parts are the calls to resolve and reject. The resolve function tells a promise consumer that the promise has resolved (completed successfully). And the reject function does the same for promises that settled unsuccessfully. Note that in either case, we are passing along a value as well. That value will be what the consumer receives in the function that handles the resolution or rejection.

We also have some code here that consumes the promise. Note the use of then and of catch. Think of these as the "onSuccess" and "onFailure" event handlers. To be strict, the signature for then is Promise. then(onSuccess, onFailure). So you could use a call to then directly. In terms of style, though, it is much clearer to have a separate call to catch to handle any errors that come along.

The really clever bit is that we have disconnected the handling of the promise from the state of the promise. That is, we could call p.then (or p.catch) as many times as we want, regardless of the state of the promise. If we are still inside the 1000 millisecond time it took to settle the promise, code registered with p.then will be added to the stack of code to be called when the promise settles. If we are after that 1000 millisecond time period (as we are likely to be), the code will execute immediately. You could imagine some pseudocode inside the promise implementation that works like this:

```
function then(onSuccess, onFailure) {
  if (this.state === "pending") {
    addSuccessStack(onSuccess);
    addFailureStack(onFailure);
  } else if (this.state === "settled") {
```

```
    if (this.settledState === "success") {
      onSuccess();
    } else {
      onFailure();
    }
  }
}
```

The actual implementation has a few more details which we won't go into here. Nonetheless, this is a useful approximation of what goes on inside a promise when you register code to run when that promise settles.

Promises are the standard, ECMAScript 6 way of managing asynchronous code. Indeed, many JavaScript libraries are already using promises to manage Ajax, animation, and other typically asynchronous interactions. As we will see in the very next section, we will need to have and understand promises to use the Harmony implementation of modules.

If you want to work with promises in the meantime, there are a variety of options. Traceur understands promises and uses them internally to implement modules (and a few other features). Libraries like jQuery, Angular, and a few others have their own implementations of promises, though there are some differences from the official specification. This is particularly the case with jQuery, so be careful. ES6-Shim contains a promise implementation, and the excellent Q library by Kris Kowal has a Q.Promise type which implements the ES6 API.

Our lightweight discussion here has only touched on the promise of promises). There is more to the API and there is more to the type. There are a number of terrific articles on promises on the web, but if you are going to start somewhere, consider this HTML5Rocks article by Jake Archibald: http://www.html5rocks.com/en/tutorials/es6/promises/.

Modules

Wait, haven't we seen modules before? Well, yes, when speaking of, say, RequireJS, you can talk about AMD (Asynchronous Module Definition) modules. And if you are in the world of Node.js, you might think of CommonJS modules. The ECMAScript 6 standard seeks to resolve these two different styles of modules into one syntax. It is somewhat successful. Inevitably, there will be complaints from both sides. But in striking a middle ground, using aspects of both standards, TC39 has provided programmers with a clear path to follow. If you want to stick with AMD styled modules, or prefer CommonJS style modules, someone will eventually write a transpiler... possibly you.

That having been said, let's look at the implementation of ES6 modules. We will not spend time comparing and contrasting ES6 modules with AMD or CommonJS modules. That is an exercise for another chapter of another book (or for an excellent blog post by Alex Rauschmeyer: http://www.2ality.com/2014/09/es6-modules-final.html). Instead, we will work with modules to look at what they provide us.

The idea behind the module is simple: we want a safe, encapsulated namespace inside of which we can define data and functionality. Then, at our discretion, we can make some or all of that data and functionality available. Reusability is the main use of a module, allowing us to define functionality once and use it anywhere we need to. Listing 11-4 shows how modules implement these requirements.

Listing 11-4. A Module Defined

```
export const schoolName = 'Mickey Kullen Memorial High School';

export const firstSport = 'Basketball';

export function getPrincipal() {
```

```
  // Probably a call to a server in the real world
  return 'Jason Franzke';
}

export function fgPct(shots, baskets) {
  return baskets / shots;
}
```

As you can see, we export the public API for the module via the `export` keyword. When using `export` on a member-by-member basis, we can export constants, regular variables, and functions. If we prefer, we can list all of the exports as the last line of the file (even changing their names in the process!):

```
export {schoolName, firstSport, getPrincipal, fgPct};
export {schoolName as name, firstSport as sport ... };
```

You can also set a single member of the module to be the default export. This member is marked, unsurprisingly, with the `default` keyword, like so:

```
export default function () { ... };
export default 'foo';
```

You can mix both a default and named members, though that will complicate things when you later import these members.

With all these options, which should we choose? Choose the one that works! But if you are looking for expert guidance, Dave Herman, who is an integral part of the ES6 design process, suggested a preference for single export modules, using defaults. There was much discussion on this point, with many people in opposition, or at least expressing alternative preferences. It's JavaScript: go with a style that works for your needs.

When using a module, we have two ways to interact with it: declarative and programmatic. The declarative syntax, shown in Listing 11-5, is simple and straightforward.

Listing 11-5. Using a Module Declaratively

```
import * as school from 'test-module';

console.log('The principal of %s is %s.', school.schoolName, school.getPrincipal());
```

The `import` keyword allows you to define which parts of the module are imported into what namespace. The module name is the same as the file name, minus the `.js` extension. Paths are allowed, so `foo/bar` would find `bar.js` under the directory `foo` relative to the current file location.

The `import` command is quite flexible, as you can import as much or as little of the module as you would like:

```
// Imports 'schoolName' only
import schoolName from 'test-module';

// Imports both of these named members
import {schoolName, getPrincipal} from 'test-module;

// Imports the default exports
import someDefault from 'test-module';
```

```
// Imports the default, plus a named member; DOES NOT WORK IN TRACEUR
import someDef {schoolName} from 'test-module';

// Imports the entire module, the default function/variable/whatever is
// available as default
import * as school from 'test-module';
console.log(s.default()); // Executes the function exported as a default
```

Modules are loaded asynchronously. Your browser will wait until all modules have loaded before executing any code. If you want greater control over this process (or prefer a different syntax) you can use the programmatic style of module import (Listing 11-6).

Listing 11-6. Programmatic Imports

```
System.import( 'test-module' )
  .then( school => {
    console.log('The principal of %s is %s.', school.schoolName, school.getPrincipal());
  } )
  .catch( error => {
    console.error( 'Something has gone horribly wrong.' );
  } );
```

The syntax of the underlying module does not change. Instead, we have a promise-based syntax for loading up modules if you want to tie specific code to the loading or execution of specific modules. (We threw in a use of the new arrow function syntax just for fun!) The details of what else you can do with this syntax are more complex than space here allows. Suffice it to say, you will have extensive control over the loading and running of your modules, should you want or need it.

Several polyfills for modules are available. Traceur does a reasonable job with modules, with one or two exceptions. The ES6-Tools page keeps up to date on a few others, including a package named, simply, ES6 Module Loader Polyfill (https://github.com/ModuleLoader/es6-module-loader). Note that ES6-Shim does not have a module polyfill.

Type Extensions

The last category of changes to ECMAScript is improvements to existing types. Some of these changes formalize features long available in JavaScript, like the HTML functions on String types and so on. Some of these are self-explanatory, but others could use some clarification, which is what we're here to provide!

Strings

This set of String functions should be called from an instance of String. Put another way, they are available on String.prototype. First there are the HTML functions: anchor, big, bold, fixed, fontcolor, fontsize, italics, link, small, strike, sub, and sup. Each of these takes the String instance and returns a copy wrapped in the appropriate tag. Various browsers may add or subtract from this group (Chrome, for instance, has a blink function).

There are also some String utility functions:

`String.prototype.startsWith(str)`	Does the String start with the provided substring?
`String.prototype.endsWith(str)`	Does the String end with the provided substring?
`String.prototype.contains(str, [startPos])`	Does the String contain the provided substring?
`String.prototype.repeat(count)`	Generates a new string, which is a repetition of `String` for count times.

Numbers

The Number type gains several static methods, most of which are used to determine characteristics of the passed argument.

`Number.isNaN(num)`	Is *num* a Number? Replaces the global `isNaN function`, which had some issues.
`Number.isFinite(num)`	Is num finite? Positive and negative Infinity, NaN, and non-numbers are not finite.
`Number.isInteger(num)`	Is the number an integer? NaN is not an integer, nor is anything non-numeric.
`Number.isSafeInteger(num)`	Can this number be safely represented as an IEEE-754 double-precision number, and does no other IEEE-754 double-precision number round to this number?
`Number.parseInt(string, [radix])`	Replaces global `parseInt()`; prefer supplying a radix, as this can ameliorate implementation differences.
`Number.parseFloat(string)`	Replaces global `parseFloat()`.

Math

The Math utility library expands with a few useful functions. Most of these are more esoteric than we have time to go into here, but here is a brief list:

`Math.imul(x, y)`	Returns the result of the C-like 32-bit multiplication of the two parameters.
`Math.clz32(num)`	Returns the number of leading zero bits in the 32-bit binary representation of a number.
`Math.fround(num)`	Returns the nearest single-precision float representation of a number.
`Math.log10(num)`	Returns the base 10 logarithm of a number.
`Math.log2(num)`	Returns the base 2 logarithm of a number.
`Math.log1p(num)`	Returns the natural logarithm (base e) of 1 + a number.
`Math.expm1(x)`	Returns ex - 1, where *x* is the argument, and e the base of the natural logarithms.
`Math.cosh(num)`	Returns the hyperbolic cosine of a number.

(*continued*)

`Math.sinh(`*num*`)`	Returns the hyperbolic sine of a number.
`Math.tanh(`*num*`)`	Returns the hyperbolic tangent of a number.
`Math.acosh(`*num*`)`	Returns the hyperbolic arc-cosine of a number.
`Math.asinh(`*num*`)`	Returns the hyperbolic arc-sine of a number.
`Math.atanh(`*num*`)`	Returns the hyperbolic arc-tangent of a number.
`Math.hypot([`*num*`,` *num2*`,` *num3*`, ...])`	Returns the square root of the sum of squares of its arguments.
`Math.trunc(`*num*`)`	Returns the integral part of a number by removing any fractional digits. It does not round any numbers.
`Math.sign(`*num*`)`	Returns the sign of a number, which may be positive, negative, or zero.
`Math.cbrt(`*num*`)`	Returns the cube root of a number.

Arrays

Many new things have happened in the world of Arrays. Let's look at functions first. The first is a simple utility function: `Array.from()`, statically available on the Array type. This takes array-like objects, or iterable objects, and converts them to Arrays (giving you all the functions and features of Arrays). Think of `arguments` inside a function, or the return value of `document.querySelectorAll`, which are both like Arrays but lack many of the features of Arrays. Now you can convert iterables on-the-fly to Arrays.

Arrays also gain three new tools for iterating over their contents: `keys`, `values`, and `entries`. You can probably deduce that `keys` gives you the indices of an array, `values` the indices of the values, and `entries` an array consisting of the key and the value for each array entry. The part that's a little different is that this is done via an iterator. That is, instead of getting all the values, you see the individual elements as you are passing over them. This can be useful for dynamic arrays, better memory management, array searches, and so on.

For searching through arrays, we already have the `Array.prototype.indexOf` function. But if you want to test for something other than simple equality, you can use `Array.prototype.find` and `Array.prototype.findIndex`. Both take an argument of a predicate function and an optional context in which to run. The predicate function, much like the predicate functions for `forEach`, `map`, and so on, takes arguments of element, index, and a reference back to the original code. Listing 11-7 shows an example.

Listing 11-7. Using `Array.prototype.find()`

```
var names = ['John', 'Jon', 'José', 'Joseph', 'Mike',
  'Andre', 'Melanie', 'Jaymi', 'Kathy', 'Jennifer'];

names.find( function ( element, index ) {
  if ( element.startsWith( 'J' ) ) {
    console.log( 'The name %s at position %d starts with "J"!', element, index );
  }
} );
```

Finally, there's `Array.prototype.fill`, which allows you to fill an array with values, optionally passing in start and end positions.

There is one more important improvement to arrays: the `spread` operator. For a long time, JavaScript programmers have wanted to be able to "unwind" an array as an argument to a function. Up through ECMAScript 5, there was no way to do this, though you could use `Function.prototype.apply`, which took

an array and spread it out as arguments to the function being called. This was unwieldy and unclear to say the least. Now, you can use the spread operator:

```
[1, 2, 3].push(...[4, 5, 6])
[1, 2, 3, 4, 5, 6]
```

Nice, right? In many ways, this makes push, pop, shift, and unshift more powerful, and makes splice and concat much more specialized.

Polyfills

The Traceur transpiler does not support these expansions of standard types. Traceur, in general, is focused on new syntactical changes, not expansions of existing types.

On the other hand, ES6-shim supports all of these features. And you can find polyfills for most of these functions, often broken out by function or by type, at Addy Osmani's ES6 Tools page.

New Collection Types

JavaScript has suffered from a dearth of collection implementations. Before ECMAScript Harmony, your only choices for native data strutures were the Array and the Object. This was hardly ideal. ECMAScript 6 introduces the Set, WeakSet, Map, and WeakMap as new data structures. At their most basic, these intend to be a new Collections API for JavaScript. In the future, you should be able to think of JavaScript collections as Map, WeakMap, Set, WeakSet, and Array, leaving Objects to do work as just objects, not double-duty as maps/associative arrays as well.

Let's talk about the non-Weak versions of these objects first. The Map is a set of keys and values. The keys can be any primitive or object value, as can the values. These are explicitly intended to replace Objects as a data structure. There are some critical differences:

- Keys for Objects are always Strings, whereas keys for Maps can be any data type.

- Maps have a property for size, Objects do not; put another way, you have to track the size of your Objects manually, where Maps do this automatically.

- Objects have prototypes. Strictly speaking, Maps have prototypes as well, but Map instances do not have prototypes the way that Object instances do. This is important in that Objects have default keys, where Maps do not.

Sets are Arrays that guarantee uniqueness. Set entries must be unique according to === (triple-equals). Data can be retrieved from a Set in insertion order. A Set can be created from an Array by passing the Array as an argument to the Set constructor. It is not easy to convert a Set to an Array in cross-browser ECMAScript, but you could always iterate over the elements of a Set with for...of or Set.prototype.forEach and push the individual element onto an array, should you need to.

The new collections are supported by recent editions of both Firefox and Chrome. Internet Explorer has "basic" support for Map, WeakMap, and Set (though not WeakSet!). This means that IE knows about the type, and has some of it implemented, but does not, for instance, allow you to use new Map(iterable) to create an instance of Map. But you can create empty Maps, Sets, and WeakMaps and then add elements to them with the appropriate API calls. Full support is expected in the next edition of IE, according to status.modern.ie.

That's Weak

So what's the deal with the WeakSet and the WeakMap? To understand these, you have to understand a little bit about JavaScript garbage collection. In general, an object is available for garbage collection when its reference count is zero. That means there are no extant references to the object in question: no variables, no keys, no values, no entries and so on. The entries in Maps and Sets count as references.

Imagine that you have created a reference to a complex object and stored it in a variable. Later, you also stashed that reference in a Set. Before your function or section of code returned, you deallocated the variable, setting it equal to null. Great, right? All cleaned up! Not so fast. The entry in the Set persists and your object is not elligible for garbage collection until you remove it from the Set (or deallocate the entire Set!).

Enter WeakSets and WeakMaps. In the case of a WeakSet, the object references are held weakly. This means that the instance held in the Set does not count toward overall reference count. The reference is weakly held. If all other references to the object have been deallocated, the object is available for garbage collection. This can be quite useful for memory management but somewhat tricky if you were counting on that reference sticking around.

WeakMaps are similar in that their keys are held weakly, the same way as the entries for a Set are held weakly. Additionally, keys in a WeakMap can *only* be Objects, not primitves. Unlike regular Maps, the keys for WeakMaps cannot be iterated, because their state is nondeterministic, due to potential garbage collection. You will have to maintain a list of keys yourself, perhaps as an array, if you want access to it.

Collections API

The various collection implementations share a common API. Not all functions are found on all collections, but most are shared by two or more classes of collection.

Function/Property	Set	WeakSet	Map	WeakMap	Description
size	Y	Y	Y	N	Number of entries in the collection.
add(e)	Y	Y	N	N	Adds an element to the Set.
clear()	Y	Y	Y	Y	Removes all elements from the collection.
delete(k)	Y	Y	Y	Y	For Maps, deletes the entry for this key; for Sets, deletes the entry for this value.
has(k)	Y	Y	Y	Y	Returns Boolean true or false based on whether the key (Maps) or value (Sets) is present in the collection.
get(k)	N	N	Y	Y	Gets the value associated with this key.
set(k, v)	N	N	Y	Y	Sets the key k to value v.
entries()	Y	N	Y	N	Returns an iterator which will return individual arrays for key/ value pairs (Maps) or two-element arrays of each value (Sets).
forEach(fn, [scope])	Y	N	Y	N	Iterates over the elements of the collection, running the function *fn* once for each element (values for Sets, key/value pairs as an array for Maps).
keys()	Y	N	Y	N	Returns the keys for the collection. In Sets, keys and values are the same thing.
values()	Y	N	Y	N	Returns the values for the collection.

Polyfills

As with the extensions to JavaScript types, Traceur does not implement ES5 transpilations of these features. That is the realm of the polyfill. And, as noted earlier, the ES6-Shim polyfill has ES5-compliant implementations of Map, Set, WeakMap, and WeakSet. There's also the harmony-collections polyfill (`https://github.com/Benvie/harmony-collections`), which implements only Map, Set, WeakMap, and WeakSet.

Keep in mind, though, that while a polyfill can ape the API, it cannot duplicate a critical feature of WeakMaps and WeakSets: holding references weakly so that they don't count toward overall reference count for garbage collection purposes. That can only be implemented through changes to the JavaScript engine. It can only emulate the behavior of a WeakMap, without having an actual relationship with the garbage collector whereby weakly held references can be released.

Summary

In this chapter, we have tried to give you an overview of some of the new features coming to JavaScript with ECMAScript 6. In doing so, we have necessarily overlooked a large amount of the specification. Some time in 2015, the specification will settle into its final form, and APress will offer several books covering the material.

Meanwhile, we have focused on the tools that ameliorate some of JavaScript's more annoying quirks (think arrow functions here). We also walked through how to use some features that many programmers are using today (promises, classes, modules). We are very excited about the future of JavaScript. With ECMAScript 6 well defined by the time this book is published, we are looking forward to ECMAScript 7, which TC39 is already starting to work on!

■ ■ ■

DOM Reference

This appendix serves as a reference for the functionality provided by the Document Object Model discussed in Chapter 5.

Resources

DOM functionality has come in a variety of flavors, starting with the original prespecification DOM Level 0 on up to DOM Level 3. One of the things to understand about the DOM is that it is considered a living standard. Each level is describes the features and behaviors that are added. The DOM itself is a representation of the document with nodes and properties the can have events associated with them.

If you wanted to understand some of the details of DOM the W3C's web sites serve as an excellent reference for learning how the DOM should work as well as the Web Hypertext Application Technology Working Group (WHATWG):

- *HTML DOM Level 3:* *http://www.w3.org/TR/DOM-Level-3-Core/*

- *WHATWG DOM:* *https://dom.spec.whatwg.org/*

Additionally, there exist a number of excellent references for learning how DOM functionality works, but none is better than the resources that exist at Quirksmode.org, a site run by Peter-Paul Koch. He has comprehensively looked at every available DOM method and compared its results in all modern browsers (plus some). It's an invaluable resource for figuring out what is, or is not, possible in the browsers that you're developing for. Another source is also caniuse.com created by Alexis Devera. Here you can search for a feature you would like to use and see a compatibioity table for witch browsers support that feature.

Terminology

In Chapter 5 on the Document Object Model and in this appendix, I use common XML and DOM terminology to describe the different aspects of a DOM representation of an XML document. The following words and phrases are terminology that relate to the Document Object Model and XML documents in general. All of the terminology examples will relate to the sample HTML document shown in Listing A-1.

Listing A-1. A Reference Point for Discussing DOM and XML Terminology

```
<!doctype html>
<html>
<head>
    <title>Introduction to the DOM</title>
</head>
```

```
<body>
    <h1>Introduction to the DOM</h1>
    <p class="test">There are a number of reasons why the DOM is awesome,
        here are some:</p>
    <ul>
        <li id="everywhere">It can be found everywhere.</li>
        <li class="test">It's easy to use.</li>
        <li class="test">It can help you to find what you want, really quickly.</li>
    </ul>
</body>
</html>
```

Ancestor

Very similar to the genealogical term, *ancestor* refers to the parent of the current element, and that parent's parent, and that parent's parent, and so on. In Listing A-1 the ancestor elements of the <ul> element are the <body> element and the <html> element.

Attribute

Attributes are properties of elements that contain additional information about them. In Listing A-1 the <p> element has the attribute *class* that contains the value *test*.

Child

Any element can contain any number of nodes (each of which are considered to be *children* of the parent element). In Listing A-1 the <ul> contains seven child nodes; three of the child nodes are the <li> elements, the other four are the endlines that exist in between each element (contained within text nodes).

Document

An XML *document* consists of one element (called the *root node* or *document element*) that contains all other aspects of the document. In Listing A-1 the <html> is the document element containing the rest of the document.

Descendant

An element's *descendants* include its child nodes, its children's children, and their children, and so on. In Listing A-1 the <body> element's descendants include <h1>, <p>, <ul>, all the <li> elements, and all the text nodes contained inside all of them.

Element

An *element* is a container that holds attributes and other nodes. The primary, and most noticeable, component of any HTML document is its elements. In Listing A-1 there are a ton of elements; the <html>, <head>, <title>, <body>, <h1>, <p>, <ul>, and <li> tags all represent elements.

Node

A *node* is the common unit within a DOM representation. Elements, attributes, comments, documents, and text nodes are all nodes and therefore have typical node properties (for example, nodeType, nodeName, and nodeValue exist in every node).

Parent

Parent is the term used to refer to the element that contains the current node. All nodes have a parent, except for the root node. In Listing A-1 the parent of the <p> element is the <body> element.

Sibling

A *sibling* node is a child of the same parent node. Generally this term is used in the context of previousSibling and nextSibling, two attributes found on all DOM nodes. In Listing A-1 the siblings of the <p> element are the <h1> and <ul> elements (along with a couple white space–filled text nodes).

Text Node

A *text node* is a special node that contains only text; this includes visible text and all forms of white space. So when you're seeing text inside of an element (for example, hello world!), there is actually a separate text node inside of the <b> element that contains the "hello world!" text. In Listing A-1, the text "It's easy to use" inside of the second <li> element is contained within a text node.

Global Variables

Global variables exist within the global scope of your code, but they exist to help you work with common DOM operations.

document

This variable contains the active HTML DOM document, which is viewed in the browser. However, just because this variable exists and has a value, doesn't mean that its contents have been fully loaded and parsed. See Chapter 5 for more information on waiting for the DOM to load. Listing A-2 shows some examples of using the document variable that holds a representation of the HTML DOM to access document elements.

Listing A-2. Using the Document Variable to Access Document Elements

```
// Locate the element with the ID of 'body'
document.getElementById("body")

// Locate all the elements with the tag name of <div>.
document.getElementsByTagName("div")
```

HTMLElement

This variable is the superclass object for all HTML DOM elements. Extending the prototype of this element extends all HTML DOM elements. This superclass is available by default in Mozilla-based browsers and Opera. It's possible to add it to Internet Explorer and Safari. Listing A-3 shows an example of binding new functions to a global HTML element superclass. Attaching a hasClass function provides the ability to see whether an element has a specific class.

Listing A-3. Binding New Functions to a Global HTML Element SuperClass

```
// Add a new method to all HTML DOM Elements
// that can be used to see if an Element has a specific class, or not.
HTMLElement.prototype.hasClass = function( class ) {
    return new RegExp("(^|\\s)" + class + "(\\s|$)").test( this.className );
};
```

DOM Navigation

The following properties are a part of all DOM elements and can be used to traverse DOM documents.

body

This property of the global HTML DOM document (the document variable) points directly to the HTML <body> element (of which there should only be the one). This particular property is one that has been carried over from the days of DOM 0 JavaScript. Listing A-4 shows some examples of accessing the <body> element from the HTML DOM document.

Listing A-4. Accessing the <body> Element Inside of an HTML DOM Document

```
// Change the margins of the <body>
document.body.style.margin = "0px";

// document.body is equivalent to:
document.getElementsByTagName("body")[0]
```

childNodes

This is a property of all DOM elements, containing an array of all child nodes (this includes elements, text nodes, comments, etc.). This property is read-only. Listing A-5 shows how you would use the childNodes property to add a style to all child elements of a parent element.

Listing A-5. Adding a Red Border Around Child Elements of the <body> Element Using the childNodes Property

```
// Add a border to all child elements of <body>
var c = document.body.childNodes;
for ( var i = 0; i < c.length; i++ ) {
    // Make sure that the Node is an Element
    if ( c[i].nodeType == 1 )
        c[i].style.border = "1px solid red";
}
```

documentElement

This is a property of all DOM nodes acting as a reference to the root element of the document (in the case of HTML documents, this will always point to the <html> element). Listing A-6 shows an example of using the documentElement to find a DOM element.

Listing A-6. Example of Locating the Root Document Element From Any DOM Node

```
// Find the documentElement, to find an Element by ID
someRandomNode.documentElement.getElementById("body")
```

firstChild

This is a property of all DOM elements, pointing to the first child node of that element. If the element has no child nodes, firstChild will be equal to null. Listing A-7 shows an example of using the firstChild property to remove all child nodes from an element.

Listing A-7. Removing All Child Nodes From an Element

```
// Remove all child nodes from an element
var e = document.getElementById("body");
while ( e.firstChild )
    e.removeChild( e.firstChild );
```

getElementById(elemID)

This is a powerful function that locates the one element in the document that has the specified ID. The function is only available on the document element. Additionally, the function may not work as intended in non-HTML DOM documents; generally with XML DOM documents you have to explicitly specify the ID attribute in a DTD (Document Type Definition) or schema.

This function takes a single argument: the name of the ID that you're searching for, as demonstrated in Listing A-8.

Listing A-8. Two Examples of Locating HTML Elements by Their ID Attributes

```
// Find the Element with an ID of body
document.getElementById("body")

// Hide the Element with an ID of notice
document.getElementById("notice").style.display = 'none';
```

getElementsByTagName(tagName)

This property finds all descendant elements—beginning at the current element—that have the specified tag name. This function works identically in XML DOM and HTML DOM documents.

In all modern browsers, you can specify * as the tag name and find all descendant elements, which is much faster than using a pure-JavaScript recursive function.

This function takes a single argument: the tag name of the elements that you're searching for. Listing A-9 shows examples of getElementsByTagName. The first block adds a highlight class to all <div> elements in the document. The second block finds all the elements inside of the element with an ID of *body*, and hides any that have a class of *highlight*.

Listing A-9. Two Code Blocks That Demonstrate How getElementsByTagName Is Used

```
// Find all <div> Elements in the current HTML document
// and set their class to 'highlight'
var d = document.getElementsByTagName("div");
for ( var i = 0; i < d.length; i++ ) {
    d[i].className = 'hilite';
}
```

```
// Go through all descendant elements of the element with
// an ID of body. Then find all elements that have one class
// equal to 'hilite'. Then hide all those elements that match.
var all = document.getElementById("body").getElementsByTagName("*");
for ( var i = 0; i < all.length; i++ ) {
    if ( all[i].className == 'hilite' )
        all[i].style.display = 'none';
}
```

lastChild

This is a reference available on all DOM elements, pointing to the last child node of that element. If no child nodes exist, lastChild will be null. Listing A-10 shows an example of using the lastChild property to insert an element into a document.

Listing A-10. Creating a New <div> Element and Inserting It Before the Last Element in the <body>

```
// Insert a new Element just before the last element in the <body>
var n = document.createElement("div");
n.innerHTML = "Thanks for visiting!";

document.body.insertBefore( n, document.body.lastChild );
```

nextSibling

This is a reference available on all DOM nodes, pointing to the next sibling node. If the node is the last sibling, nextSibling will be null. It's important to remember that nextSibling may point to a DOM element, a comment, or even a text node; it does not serve as an exclusive way to navigate DOM elements. Listing A-11 is an example of using the nextSibling property to create an interactive definition list.

Listing A-11. Making All <dt> Elements Expand Their Sibling <dd> Elements Once Clicked

```
// Find all <dt> (Defintion Term) elements
var dt = document.getElementsByTagName("dt");
for ( var i = 0; i < dt.length; i++ ) {
    // Watch for when the term is clicked
    dt[i].onclick = function() {
        // Since each Term has an adjacent <dd> (Definition) element
        // We can display it when it's clicked

        // NOTE: Only works when there's no whitespace between <dd> elements
        this.nextSibling.style.display = 'block';
    };
}
```

parentNode

This is a property of all DOM nodes. Every DOM node's parentNode points to the element that contains it, except for the document element, which points to null (since nothing contains the root element). Listing A-12 is an example of using the parentNode property to create a custom interaction. Clicking the Cancel button hides the parent element.

Listing A-12. Using the parentNode Property to Create a Custom Interaction

```
// Watch for when a link is clicked (e.g. a Cancel link)
// and hide the parent element
document.getElementById("cancel").onclick = function(){
    this.parentNode.style.display = 'none';
};
```

previousSibling

This is a reference available on all DOM nodes, pointing to the previous sibling node. If the node is the first sibling, the previousSibling will be null. It's important to remember that previousSibling may point to a DOM element, a comment, or even a text node; it does not serve as an exclusive way to navigate DOM elements. Listing A-13 shows an example of using the previousSibling property to hide elements.

Listing A-13. Hiding All Elements Before the Current Element

```
// Find all elements before this one and hide them
var cur = this.previousSibling;
while ( cur != null ) {
    cur.style.display = 'none';
    cur = this.previousSibling;
}
```

Node Information

These properties exist on most DOM elements in order to give you easy access to common element information.

innerText

This is a property of all DOM elements (which only exists in non-Mozilla-based browsers, as it's not part of a W3C standard). This property returns a string containing all the text inside of the current element. Since this property is not supported in Mozilla-based browsers, you can utilize a workaround (where you use a function to collect the values of descendant text nodes). Listing A-14 shows an example of using the innerText property and the text( ) function from Chapter 5.

Listing A-14. Using the innerText Property to Extract Text Information From an Element

```
// Let's assume that we have an <li> element like this, stored in the variable 'li':
// <li>Please visit <a href="http://mysite.com/">my web site</a>.</li>

// Using the innerText property
li.innerText

// or the text() function described in Chapter 5
text( li )

// The result of either the property or the function is:
"Please visit my web site."
```

nodeName

This is a property available on all DOM elements that contains an uppercase version of the element name. For example, if you have an <li> element and you access its nodeName property, it will return LI. Listing A-15 shows an example of using the nodeName property to modify the class names of parent elements.

Listing A-15. Locating All Parent <li> Elements and Setting Their Class to current

```
// Find all the parents of this node, that are an <li> element
var cur = this.parentNode;
while ( cur != null ) {
    // Once the element is found, and the name verified, add a class
    if ( cur.nodeName == 'LI' )
        cur.className += " current";
    cur = this.parentNode;
}
```

nodeType

This is a common property of all DOM nodes, containing a number corresponding to the type of node that it is. The three most popular node types used in HTML documents are the following:

- Element node (a value of 1 or document.ELEMENT_NODE)

- Text node (a value of 3 or document.TEXT_NODE)

- Document node (a value of 9 or document.DOCUMENT_NODE)

Using the nodeType property is a reliable way of making sure that the node that you're trying to access has all the properties that you think it does (e.g., a nodeName property is only useful on a DOM element; so you could use nodeType to make sure that it's equal to 1 before accessing it). Listing A-16 shows an example of using the nodeType property to add a class to a number of elements.

Listing A-16. Locating the First Element in the HTML <body> and Applying a header Class to It

```
// Find the first element in the <body>
var cur = document.body.firstChild;
while ( cur != null ) {
    // If an element was found, add the header class to it
    if ( cur.nodeType == 1 ) {
        cur.className += " header";
        cur = null;

    // Otherwise, continue navigating through the child nodes
    } else {
        cur = cur.nextSibling;
    }
}
```

nodeValue

This is a useful property of text nodes that can be used to access and manipulate the text that they contain. The best example of this in use is the text function presented in Chapter 5, which is used to retrieve all the text contents of an element. Listing A-17 shows an example of using the nodeValue property to build a simple text value function.

Listing A-17. A Function That Accepts an Element and Returns the Text Contents of It and All Its Descendant Elements

```
function text(e) {
    var t = " ";
    // If an element was passed, get its children,
    // otherwise assume it's an array
    e = e.childNodes || e;

    // Look through all child nodes
    for ( var j = 0; j < e.length; j++ ) {
        // If it's not an element, append its text value
        // Otherwise, recurse through all the element's children
        t += e[j].nodeType != 1 ?
            e[j].nodeValue : text(e[j].childNodes);
    }

    // Return the matched text
    return t;
}
```

Attributes

Most attributes are available as properties of their containing element. For example, the attribute ID can be accessed using the simple element.id. This feature is residual from the DOM 0 days, but it's very likely that it's not going anywhere, due to its simplicity and popularity.

className

This property allows you to add and remove classes from a DOM element. This property exists on all DOM elements. The reason I'm mentioning this specifically is that its name, className, is very different from the expected name of *class*. The strange naming is due to the fact that the word *class* is a reserved word in most object-oriented programming languages; so its use is avoided to limit difficulties in programming a web browser. Listing A-18 shows an example of using the className property to hide a number of elements.

Listing A-18. Finding All <div> Elements That Have a Class of special and Hiding Them

```
// Find all the <div> elements in the document
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    // Find all the <div> elements that have a single class of 'special'
    if ( div[i].className == "special" ) {
        // And hide them
        div[i].style.display = 'none';
    }
}
```

getAttribute(attrName)

This is a function that serves as the proper way of accessing an attribute value contained within a DOM element. Attributes are initialized with the values that the user has provided in the straight HTML document.

The function takes a single argument: the name of the attribute that you want to retrieve. Listing A-19 shows an example of using the getAttribute( ) function to find input elements of a specific type.

Listing A-19. Finding the <input> Element Named text and Copying Its Value Into an Element With an ID of preview

```
// Find all the form input elements
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // Find the element that has a name of "text"
    if ( input[i].getAttribute("name") == "text" ) {

        // Copy the value into another element
        document.getElementById("preview").innerHTML =
            input[i].getAttribute("value");
    }
}
```

removeAttribute(attrName)

This is a function that can be used to completely remove an attribute from an element. Typically, the result of using this function is comparable to doing a setAttribute with a valueof " " (an empty string) or null; in practice, however, you should be sure to always clean up extra attributes in order to avoid any unexpected consequences.

This function takes a single argument: the name of the attribute that you wish to remove. Listing A-20 shows an example of unchecking some check boxes in a form.

Listing A-20. Finding All Check Boxes in a Document and Unchecking Them

```
// Find all the form input elements
var input = document.getElementsByTagName("input");
for ( var i = 0; i < input.length; i++ ) {

    // Find all the checkboxes
    if ( input[i].getAttribute("type") == "checkbox" ) {

        // Uncheck the checkbox
        input[i].removeAttribute("checked");

    }

}
```

setAttribute(attrName, attrValue)

This is a function that serves as a way of setting the value of an attribute contained within a DOM element. Additionally, it's possible to add in custom attributes that can be accessed again later while leaving the appearance of the DOM elements unaffected. setAttribute tends to behave rather strangely in Internet Explorer, keeping you from setting particular attributes (such as class or maxlength). This is explained more in Chapter 5.

The function takes two arguments. The first is the name of the attribute. The second is the value to set the attribute to. Listing A-21 shows an example of setting the value of an attribute on a DOM element.

Listing A-21. Using the setAttribute Function to Create an <a> Link to Google

```
// Create a new <a> element
var a = document.createElement("a").

// Set the URL to visit to Google's web site
a.setAttribute("href","http://google.com/");

// Add the inner text, giving the user something to click
a.appendChild( document.createTextNode( "Visit Google!" ) );

// Add the link at the end of the document
document.body.appendChild( a );
```

DOM Modification

The following are all the properties and functions that are available to manipulate the DOM.

appendChild(nodeToAppend)

This is a function that can be used to add a child node to a containing element. If the node that's being appended already exists in the document, it is moved from its current location and appended to the current element. The appendChild function must be called on the element that you wish to append into.

The function takes one argument: a reference to a DOM node (this could be one that you just created or a reference to a node that exists elsewhere in the document). Listing A-22 shows an example of creating a new <ul> element and moving all <li> elements into it from their original location in the DOM, then appending the new <ul> to the document body.

Listing A-22. Appending a Series of <li> Elements to a Single

```
// Create a new <ul> element
var ul = document.createElement("ul");

// Find all the first <li> elements
var li = document.getElementsByTagName("li");
for ( var i = 0; i < li.length; i++ ) {

    // append each matched <li> into  our new <ul> element
    ul.appendChild( li[i] );
}

// Append our new <ul> element at the end of the body
document.body.appendChild( ul );
```

cloneNode(true|false)

This function is a way for developers to simplify their code by duplicating existing nodes, which can then be inserted into the DOM. Since doing a normal insertBefore or appendChild call will physically move a DOM node in the document, the cloneNode function can be used to duplicate it instead.

The function takes one true or false argument. If the argument is true, the node and everything inside of it is cloned; if false, only the node itself is cloned. Listing A-23 shows an example of using this function to clone an element and append it to itself.

Listing A-23. Finding the First <ul> Element in a Document, Making a Complete Copy of It, and Appending It to Itself

```
// Find the first <ul> element
var ul = document.getElementsByTagName("ul")[0];

// Clone the node and append it after the old one
ul.parentNode.appendChild( ul.cloneNode( true ) );
```

createElement(tagName)

This is the primary function used for creating new elements within a DOM structure. The function exists as a property of the document within which you wish to create the element.

■ **Note**   If you're using XHTML served with a content-type of application/xhtml+xml instead of regular HTML served with a content-type of text/html, you should use the createElementNS function instead of the createElement function.

This function takes one argument: the tag name of the element to create. Listing A-24 shows an example of using this function to create an element and wrap it around some other elements.

Listing A-24. Wrapping the Contents of a <p> Element in a <strong> Element

```
// Create a new <strong> element
var s = document.createElement("strong");

// Find the first paragraph
var p = document.getElementsByTagName("p")[0];

// Wrap the contents of the <p> in the <strong> element
while ( p.firstChild ) {
    s.appendChild( p.firstChild );
}

// Put the <strong> element (containing the old <p> contents)
// back into the <p> element
p.appendChild( s );
```

createElementNS(namespace, tagName)

This function is very similar to the createElement function, in that it creates a new element; however, it also provides the ability to specify a namespace for the element (for example, if you're adding an item to an XML or XHTML document).

This function takes two arguments: the namespace of the element that you're adding, and the tag name of the element. Listing A-25 shows an example of using this function to create a DOM element in a valid XHTML document.

Listing A-25. Creating a New XHTML <p> Element, Filling It With Some Text, and Appending It to the Document Body

```
// Create a new XHTML-compliant <p>
var p = document.createElementNS("http://www.w3.org/1999/xhtml", "p");

// Add some text into the <p> element
p.appendChild( document.createTextNode( "Welcome to my site." ) );

// Add the <p> element into the document
document.body.insertBefore( p, document.body.firstChild );
```

createTextNode(textString)

This is the proper way to create a new text string to be inserted into a DOM document. Since text nodes are just DOM-only wrappers for text, it is important to remember that they cannot be styled or appended to. The function only exists as a property of a DOM document.

The function takes one argument: the string that will become the contents of the text node. Listing A-26 shows an example of using this function to create a new text node and appending it to the body of an HTML page.

Listing A-26. Creating an <h1> Element and Appending a New Text Node

```
// Create a new <h1> element
var h = document.createElement("h1");

// Create the header text and add it to the <h1> element
h.appendChild( document.createTextNode("Main Page") );

// Add the header to the start of the <body>
document.body.insertBefore( h, document.body.firstChild );
```

innerHTML

This is an HTML DOM–specific property for accessing and manipulating a string version of the HTML contents of a DOM element. If you're only working with an HTML document (and not an XML one), this method can be incredibly useful, as the code it takes to generate a new DOM element can be cut down drastically (not to mention it is a faster alternative to traditional DOM methods). While this property is not part of any particular W3C standard, it still exists in every modern browser. Listing A-27 shows an example of using the innerHTML property to change the contents of an element whenever the contents of a <textarea> are changed.

Listing A-27. Watching a <textarea> for Changes and Updating a Live Preview With Its Value

```
// Get the textarea to watch for updates
var t = document.getElementsByTagName("textarea")[0];

// Grab the current value of a <textarea> and update a live preview,
// everytime that it's changed
t.onkeypress = function() {
    document.getElementById("preview").innerHTML = this.value;
};
```

insertBefore(nodeToInsert, nodeToInsertBefore)

This function is used to insert a DOM node anywhere into a document. The function must be called on the parent element of the node that you wish to insert it before. This is done so that you can specify null for nodeToInsertBefore and have your node inserted as the last child node.

The function takes two arguments. The first argument is the node that you wish to insert into the DOM; the second is the DOM node that you're inserting before. This should be a reference to a valid node. Listing A-28 shows an example of using this function to insert the *favicon* (the icon that you see next to a URL in the address bar of a browser) of a site next to a set of URLs on a page.

Listing A-28. Going Through All <a> Elements and Adding an Icon Consisting of the Site's Favicon

```
// Find all the <a> links within the document
var a = document.getElementsByTagName("a");
for ( var i = 0; i < a.length; i++ ) {

    // Create an image of the linked-to site's favicon
    var img = document.createElement("img");
    img.src = a[i].href.split('/').splice(0,3).join('/') + '/favicon.ico';

    // Insert the image before the link
    a[i].parentNode.insertBefore( img, a[i] );
}
```

removeChild(nodeToRemove)

This function is used to remove a node from a DOM document. The removeChild function must be called on the parent element of the node that you wish to remove.

The function takes one argument: a reference to the DOM node to remove from the document. Listing A-29 shows an example of running through all the <div> elements in the document, removing any that have a single class of *warning*.

Listing A-29. Removing All Elements That Have a Particular Class Name

```
// Find all <div> elements
var div = document.getElementsByTagName("div");
for ( var i = 0; i < div.length; i++ ) {
    // If  the <div> has one class of 'warning'
    if ( div[i].className == "warning" ) {

        // Remove the <div> from the document
        div[i].parentNode.removeChild( div[i] );
    }
}
```

replaceChild(nodeToInsert, nodeToReplace)

This function serves as an alternative to the process of removing a node and inserting another node in its place. This function must be called by the parent element of the node that you are replacing.

This function takes two arguments: the node that you wish to insert into the DOM, and the node that you are going to replace. Listing A-30 shows an example of replacing all <a> elements with a <strong> element containing the URL originally being linked to.

Listing A-30. Converting a Set of Links Into Plain URLs

```
// Convert all links to visible URLs (good for printing
// Find all <a> links in the document
var a = document.getElementsByTagName("a");
while ( a.length ) {

    // Create a <strong> element
    var s = document.createElement("strong");

    // Make the contents equal to the <a> link URL
    s.appendChild( document.createTextNode( a[i].href ) );

    // Replace the original <a> with the new <strong> element
    a[i].replaceChild( s, a[i] );
}
```

Index

177

■ F

■ G

■ H

■ I, J

■ K

■ L

■ Y, Z

Pro JavaScript Techniques

Second Edition

John Resig

Russ Ferguson

John Paxton

Pro JavaScript Techniques

I will forever be grateful to my dad. Even now I'm still learning from his example. Dad, Rodd, we miss you both. Watching Doctor Who and The Twilight Zone has never been the same.

—Russ Ferguson

For Andreina, who always believed.

—John Paxton

Contents

About the Authors

John Resig is a developer at Khan Academy and the creator of the jQuery JavaScript library. In addition to *Pro JavaScript Techniques*, he's also the author of *Secrets of the JavaScript Ninja* (Manning, 2012).

John is a Visiting Researcher at Ritsumeikan University in Kyoto working on the study of Ukiyo-e (Japanese woodblock printing). He has developed a comprehensive woodblock print database and image search engine, located at http://ukiyo-e.org.

Russ Ferguson is a developer and instructor working in and around New York City. He is currently a manager with SunGard Consulting Services, developing applications for clients like Morgan Stanley and Comcast. For many years Russ has been an instructor for Pratt Institute and Parsons School of Design.

He has developed applications for both start-ups and established organizations like Chase Bank, Publicis Groupe, DC Comics, and MTV/Viacom.

Some of his interests are encouraging young people to code and the ways technology changes media consumption and participation.

Other interests include practicing Japanese, writing, film, concerts, and finding a good wine or sake. Tweets can be found @asciibn.

John Paxton is a programmer, trainer, author, and presenter who lives in his native New Jersey. Studying history at Johns Hopkins University, he discovered that he spent more time in the computer lab than at the document archives. Since then, his career has oscillated between programming and training, working with many of the various languages used in web development over the last 15 years. He now concentrates on JavaScript and Java, with the occasional nostalgic visits to Perl and XML. He can be found on Twitter @paxtonjohn and at his website: speedingplanet.com.

About the Technical Reviewers

Ian Devlin is interested in all things web, and currently works as a senior web developer at a web agency based in Düsseldorf, Germany. He is an HTML5 Doctor and a founding contributor to Intel's HTML5 Hub and has written articles for a number of developer zones such as Mozilla, Opera, Intel, and Adobe, and for *net* magazine. He has also written a book on HTML5 multimedia and has been technical reviewer for a number of Apress books.

Mark Nenadov is a software developer with around 15 years of experience, predominantly with open source technologies. Mark lives in Essex, Ontario, Canada with his lovely wife and their two adorable daughters—with a son on the way. When he's not developing software or spending time with his family, he is usually hiking, observing wildlife, reading, researching history, reviewing/editing manuscripts, or writing. Mark is an avid poet and his poems have appeared in publications in the United States, Canada, Pakistan, India, Australia, England, and Ireland.

Acknowledgments