Microsoft

# Start Here!™

# Learn JavaScript

Steve Suehring

# Ready to learn JavaScript programming?


Learn the fundamentals of programming with JavaScript—and begin building your first apps for the web and Windows® 8. If you have absolutely no previous experience, no problem—simply start here! This book introduces must-know concepts and getting-started techniques through easy-to-follow explanations, examples, and exercises. Then you'll put it all together by creating your first programs.

## Here's where you start learning JavaScript

- Understand how JavaScript works—and why it works the way it does
- Use JavaScript with HTML and CSS to add interactivity and manage styles
- Validate forms and react to events such as clicks
- Explore the tools and techniques for effective debugging
- Retrieve data from a server with JavaScript using AJAX
- Use jQuery and jQuery UI to enable apps to work across browsers

**GET CODE SAMPLES ONLINE**
Ready to download at
**http://go.microsoft.com/FWLink/?Linkid=258536**

*For system requirements, see the* ***Introduction.***

ISBN: 978-0-7356-6674-0


**U.S.A. $24.99**
Canada $26.99
[*Recommended*]

*Programming/JavaScript*

Start Here! Learn JavaScript

**Skill Level:** Beginner
**Prerequisites:** None

## RESOURCE ROADMAP

**Start Here!**
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects


**Step by Step**
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook


**Developer Reference**
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples


**Focused Topics**
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples


## About the Author

**Steve Suehring** is a technology architect who's written about programming, security, network and system administration, operating systems, and other topics for several industry publications, and he speaks at conferences and user groups. He is also author of *JavaScript Step by Step*, which is designed for experienced programmers new to JavaScript.

**microsoft.com/mspress**

Microsoft®

Microsoft

Start Here!™

# Learn JavaScript

Steve Suehring

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472



ISBN: 978-0-7356-6674-0

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.



**Acquisitions and Developmental Editor:** Russell Jones

**Production Editor:** Rachel Steely

**Editorial Production:** Dianne Russell, Octal Publishing, Inc.

**Technical Reviewer:** John Paul Mueller

**Copyeditor:** Roger LeBlanc

**Indexer:** Stephen Ingle

**Cover Design:** Jake Rae

**Cover Composition:** Zyg Group, LLC

**Illustrator:** Robert Romano and Rebecca Demarest

*I dedicate this book to Rebecca, Jakob, and Owen*

—Steve Suehring

# Contents at a Glance


| | | |
|---|---|---|
| | *Introduction* | *xiii* |
| CHAPTER 1 | **What Is JavaScript?** | **1** |
| CHAPTER 2 | **JavaScript Programming Basics** | **23** |
| CHAPTER 3 | **Building JavaScript Programs** | **45** |
| CHAPTER 4 | **JavaScript in a Web Browser** | **73** |
| CHAPTER 5 | **Handling Events with JavaScript** | **105** |
| CHAPTER 6 | **Getting Data into JavaScript** | **133** |
| CHAPTER 7 | **Styling with JavaScript** | **157** |
| CHAPTER 8 | **Using JavaScript with Microsoft Windows 8** | **187** |
| | *Index* | *207* |

# Contents

*Introduction* . . . . . *xiii*

**Chapter 1 What Is JavaScript? 1**

A First JavaScript Program . . . . . 2

Where JavaScript Fits . . . . . 3

HTML, CSS, and JavaScript . . . . . 4

JavaScript in Windows 8 . . . . . 9

Placing JavaScript in a Webpage . . . . . 9

Writing Your First JavaScript Program . . . . . 11

Writing JavaScript in Visual Studio 11 . . . . . 12

JavaScript's Limitations . . . . . 20

Summary . . . . . 22

**Chapter 2 JavaScript Programming Basics 23**

JavaScript Placement: Revisited . . . . . 23

Basic JavaScript Syntax . . . . . 26

JavaScript Statements and Expressions . . . . . 26

Names and Reserved Words . . . . . 27

Spacing and Line Breaks . . . . . 28

Comments . . . . . 29

Case Sensitivity . . . . . 30

Operators . . . . . 31

JavaScript Variables and Data Types . . . . . 32

Variables . . . . . 32

Data Types . . . . . 35

Looping and Conditionals in JavaScript . . . . . 39
Loops in JavaScript . . . . . 40
Conditionals in JavaScript . . . . . 41
Summary . . . . . 44

**Chapter 3 Building JavaScript Programs 45**

Functions . . . . . 45
Function Overview . . . . . 46
Function Arguments . . . . . 46
Calling Functions . . . . . 48
Return Values . . . . . 49
Function Examples . . . . . 50
Scoping Revisited . . . . . 54
Objects in JavaScript . . . . . 56
What Does an Object Look Like? . . . . . 56
Properties . . . . . 56
Methods . . . . . 58
Object Enumeration . . . . . 61
Classes . . . . . 63
Debugging JavaScript . . . . . 67
Debugging as a Process . . . . . 67
Debugging in Internet Explorer . . . . . 68
Summary . . . . . 71

**Chapter 4 JavaScript in a Web Browser 73**

JavaScript Libraries . . . . . 74
Getting jQuery . . . . . 74
Using a Local Copy of jQuery . . . . . 75
Using a CDN-Hosted jQuery Library . . . . . 78
Testing jQuery . . . . . 79
Getting jQuery UI . . . . . 81
Adding jQuery UI to a Project . . . . . 82
Testing jQuery UI . . . . . 86

The Browser Object Model . . . . . 89
Events and the *window* Object . . . . . 90
The *screen* Object . . . . . 90
The *navigator* Object . . . . . 92
The *location* Object . . . . . 93
The DOM . . . . . 95
DOM Versions . . . . . 95
The DOM Tree . . . . . 96
Retrieving Elements with JavaScript and jQuery . . . . . 98
Using jQuery, Briefly . . . . . 99
Retrieving Elements by ID . . . . . 100
Retrieving Elements by Class . . . . . 102
Retrieving Elements by HTML Tag Name . . . . . 102
Summary . . . . . 104

**Chapter 5 Handling Events with JavaScript 105**

Common Events with JavaScript . . . . . 105
Handling Mouse Events . . . . . 106
Preventing the Default Action . . . . . 110
Attaching to an Element with *On* . . . . . *112*
Validating Web Forms with jQuery . . . . . 113
Validating on Submit . . . . . 113
Regular Expressions . . . . . 118
Finding the Selected Radio Button or Check Box . . . . . 121
Determining the Selected Drop-Down Element . . . . . 122
The *click* Event Revisited . . . . . 125
Keyboard Events and Forms . . . . . 129
Summary . . . . . 131

**Chapter 6 Getting Data into JavaScript 133**

AJAX in Brief . . . . . 133
On Servers, *GET*s, and *POST*s . . . . . 134
Building a Server Program . . . . . 135

AJAX and JavaScript . . . . . 139
Retrieving Data with jQuery . . . . . 139
Using *get()* and *post()* . . . . . *140*
Building an Interactive Page . . . . . 141
Error Handling . . . . . 144
Using JSON for Efficient Data Exchange . . . . . 146
Using *getJSON()* . . . . . *147*
Sending Data to the Server . . . . . 148
Sending Data with *getJSON* . . . . . *148*
Sending Post Data . . . . . 153
Summary . . . . . 156

**Chapter 7 Styling with JavaScript 157**

Changing Styles with JavaScript . . . . . 157
CSS Revisited . . . . . 157
Changing CSS Properties . . . . . 159
Working with CSS Classes . . . . . 163
Determining Classes with *hasClass()* . . . . . *163*
Adding and Removing Classes . . . . . 164
Advanced Effects with jQuery UI . . . . . 167
Using Effects to Enhance a Web Application . . . . . 167
Using jQuery UI Widgets . . . . . 172
Other Helpful jQuery UI Widgets . . . . . 176
Putting It All Together: A Space Travel Demo . . . . . 176
Summary . . . . . 186

**Chapter 8 Using JavaScript with Microsoft Windows 8 187**

JavaScript Is Prominent in Windows 8. . . . . 187

A Stroll Through a Windows 8 Application . . . . . 190

Building a Windows 8 App . . . . . 193

Building the Application . . . . . 193

Code Analysis . . . . . 199

Defining a Splash Screen, Logos, and a Tile . . . . . 202

Summary. . . . . 204

*Index* *207*

# Introduction

JavaScript is a popular web programming language. Oops. I wrote that sentence five years ago. JavaScript is now much more than just a popular web programming language. In addition to web, JavaScript is now a central language for programming Windows 8 Apps. Using JavaScript, you can now not only write powerful applications for the web, but you can also write native Windows applications.

Now more than ever, people are looking to learn JavaScript—and not just developers—people who haven't programmed before, or who may have created a web page or two along the way, are recognizing the importance of JavaScript. It's a great time to learn JavaScript, and this book can help.

This book covers not only JavaScript programming for the web but also covers beginning Windows 8 programming with JavaScript. Even though programming or running JavaScript code doesn't require Microsoft tools, this book is noticeably Microsoft-centric. The one exception to not requiring Microsoft tools surrounds programming of Windows 8 Apps. If you're looking for a more generalized JavaScript programming book, please see my *JavaScript Step by Step* book, which, although more advanced, looks at JavaScript programming through a wider lens.

## Who Should Read This Book

This book is intended for readers who want to learn JavaScript but who don't have a formal background in programming. This characterization includes people who have perhaps created a web page, or simply been interested enough to view the source of a web page. It also includes people who are familiar with another programming language, but want to learn JavaScript.

Regardless of your background, if you're reading this, you're likely at the point where you want to learn JavaScript with some structure behind it. You'd like to write JavaScript code for practical applications, and also learn *why* it works.

In this book, you'll create the code for the examples, and test that code in one or more web browsers. You can write JavaScript in any text editor, but the book will use a free version of Microsoft Visual Studio as the JavaScript editor.

### Assumptions

This book assumes that you're familiar with basic computing tasks such as typing and saving files, as well as working with programs on the computer. The meaning of terms such as "web browser" should be clear to you as meaning programs such as Internet Explorer, Firefox, Chrome, Safari, Opera, and the like. A term like "text editor" shouldn't scare you away; hopefully you've fired up something like Notepad in Microsoft Windows before.

## Who Should Not Read This Book

This book is not intended for readers who already have extensive JavaScript programming experience. Additionally, if you're completely new to computers and aren't comfortable with the Internet and using computer software, this book might go somewhat fast for you.

Finally, if you're looking for a book to solve a specific problem with JavaScript or a book that shows JavaScript programs in a recipe-like manner, then this book isn't for you. Similarly, if you're not really interested in programming, and just want to learn how to add a counter or some other JavaScript widget to your page, there are plenty of free tutorials on the web that can help. Remember: this book shows not only *how* things work but also explains *why* things work as they do. Making something work once is easy, but explaining it and helping you understand why it works will help you for years to come.

## Organization of This Book

The book is organized into eight chapters that build upon each other. Early in the book you will see working code. While you can cut and paste, or use examples from the sample companion code, you'll have the most success if you enter the examples by hand, typing the code yourself. See the section "Code Samples" later in this Introduction for more information on working with the code samples.

## Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- The book includes several exercises that help you learn JavaScript.
- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close), means that you should select the first menu or menu item, then the next, and so on.

## System Requirements

Writing JavaScript doesn't technically require any specialized software beyond a web browser and a text editor of some kind. You will need the following hardware and software to complete the practice exercises in this book:

- While any modern operating system will work, you'll find it easier if you're on a later version of Windows, such as Windows 7 or Windows 8. Additionally, you'll need Windows 8 in order to follow some of the examples in the book that build Windows 8 Apps.
- Any text editor will suffice, but you'll find it easier to work through examples if you use Visual Studio 11, any edition (multiple downloads may be required if using Express Edition products)
- A computer that has a 1.6GHz or faster processor (2GHz recommended).
- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions).
- 3.5GB of available hard disk space.
- 5400 RPM hard disk drive.
- DirectX 9 capable video card running at 1024 x 768 or higher-resolution display.
- DVD-ROM drive (if installing Visual Studio from DVD).
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require Local Administrator rights to install or configure Visual Studio 11.

## Code Samples

There are numerous code samples throughout the book. As previously stated, you'll learn the most by typing these in manually. However, I realize that process can become mundane (and I'll even admit that I don't type in many examples when I read development books).

To help take the pain out of typing in code examples, this book reuses as much code as possible, so if you type it in once, in most cases you'll be able to reuse at least some of that code in later examples. This is both a blessing and a curse, because if you type it in incorrectly the first time—and don't get it working—then that problem will continue in later examples.

For simplicity, you'll concentrate most of your work on a single HTML and single JavaScript file within the book. This means that you won't need to create new files repeatedly; instead, you will reuse the files you already have by deleting or replacing code to create the new examples.

To help minimize errors you might make when creating the example code by hand, much of the code shown in the book (and all the formal examples) are included with the companion content for this book. These code examples, and indeed all of the code in the book, have been tested in Internet Explorer 10 and Firefox 10, along with a selection of other browsers such as Chrome and Safari in certain areas.

*http://go.microsoft.com/FWLink/?Linkid=258536*

Follow the instructions to download the *9780735666740_files.zip* file.

## Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the *9780735666740_files.zip* file that you downloaded from the book's website (name a specific directory along with directions to create it, if necessary).
2. If prompted, review the displayed end user license agreement. If you accept the terms, select the accept option, and then click Next.

**Note** If the license agreement doesn't appear, you can access it from the same web page from which you downloaded the *9780735666740_files.zip* file.

## Using the Code Samples

The code is organized into several subfolders corresponding to each chapter. Code samples are referenced by name in the book. You can load a code file and other files into a project in Visual Studio. or open the file and copy and paste the contents into the files that you'll build as part of the book.

# Acknowledgments

I've written a few books now and I'm thinking I should start an advertising program for the acknowledgements section. (Your name here for $25.) Thanks to Russell Jones and Neil Salkind for making this book possible. Since I wrote my last acknowledgements section, Owen Suehring was born and joins his brother Jakob in trying to distract me from the business of writing books. Speaking of distractions, follow me on Twitter: @stevesuehring.

Of course, it wouldn't be an acknowledgments section if I didn't thank Rob and Tim from Partners, and Jim Oliva and John Eckendorf. Thanks to Chris Tuescher. Pat Dunn and Kent Laabs: this is what I've been doing instead of updating your websites; I hope you enjoy the book more than updates to your sites.

## Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at *oreilly.com*:

*http://go.microsoft.com/FWLink/?Linkid=258535*

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

## We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

*http://www.microsoft.com/learning/booksurvey*

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

## Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Chapter 1

# What Is JavaScript?

**After completing this chapter, you will be able to**

- Understand JavaScript's role in a webpage
- Create a simple webpage
- Create a JavaScript program

**WELCOME TO THE WORLD** of JavaScript programming. This book provides an introduction to JavaScript programming both for the web and for Microsoft Windows 8. Like other books on JavaScript programming, this book shows the basics of how to create a program in JavaScript. However, unlike other introductory books on JavaScript, this book shows not only how something works but also *why* it works. If you're looking merely to copy and paste JavaScript code into a webpage there are plenty of tutorials on the web to help solve those specific problems.

Beyond the basics of how and why things work as they do with JavaScript, the book also shows best practices for JavaScript programming and some of the real-world scenarios you'll encounter as a JavaScript programmer.

Programming for the web is different than programming in other languages or for other platforms. The JavaScript programs you write will run on the visitor's computer. This means that when programming for the web, you have absolutely no control over the environment within which your program will run.

While JavaScript has evolved over the years, not everyone's computer has evolved along with it. The practical implication is that you need to account for the different computers and different situations on which your program might run. Your JavaScript program might find itself running on a computer from 1996 with Internet Explorer 5.5 through a dial-up modem just as easily as a shiny new computer running Internet Explorer 10 or the latest version of Firefox. Ultimately, this comes down to you, the JavaScript programmer, testing your programs in a bunch of different web browsers.

With that short introduction, it's time to begin looking at JavaScript. The chapter begins with code. I'm doing this not to scare you away but to blatantly pander to the side of your brain that learns by seeing an example. After this short interlude, the chapter examines where JavaScript fits within the landscape of programming for the web and beyond. Then you'll write your first JavaScript program.

## A First JavaScript Program

Later in this chapter, you'll see how to create your own program in JavaScript, but in the interest of getting you thinking about code right away, here's a small webpage with an embedded JavaScript program:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
</head>
<body>
<script type="text/javascript">
document.write("<h1>Start Here!</h1>");
</script>
</body>
</html>
```

You'll see later how to create a page like the one shown here. When viewed in a browser, the page looks like Figure 1-1. I'll show you how to create such a page later in the chapter.


**FIGURE 1-1** A basic JavaScript program to display content.

The bulk of the code shown in the preceding listing is standard HTML (HyperText Markup Language) and will be explained later. For now, you can safely ignore the code on the page except for the three lines beginning with <script type="text/javascript">.

The opening and closing script tags tell the web browser that the upcoming text is in the form of a script—in this case, it is of the type *text/javascript*. The browser sees that opening script tag and hands off processing to its internal JavaScript interpreter, which then executes the JavaScript. In this case, the entire code is merely contained on a single line: a call to the *write* method of the document object, which then places some HTML into the document.

The actual JavaScript comprises a single line:

```
document.write("<h1>Start Here!</h1>");
```

That line is enclosed within the opening and closing <script> tags. Each line of JavaScript is typically terminated by a semi-colon (;), and JavaScript gets executed from the top down. Each line is read in, then parsed and run, by the browser. The practical implication of the top-down execution is that if you have an error at the top, nothing below it will be executed by the browser. This can lead to some confusion when you're expecting something to happen, like having words appear on the screen when in fact an error occurred near the top of the program that prevented the code from ever being run. Luckily, there are tools and techniques for troubleshooting JavaScript, which are discussed later on. The one exception to this top-down execution is in the area of functions, and you'll see that later on, as well.

If you're already feeling a bit lost, don't worry. I'm going to back up and start from the beginning on JavaScript and describe its place on the web and among other programming languages.

**Note** The *document.write* usage shown here isn't always the preferred method for getting content onto a page. It works, but there are other ways to do it, though they can be more difficult to explain. For now, simpler is better.

## Where JavaScript Fits

JavaScript plays a key role in modern application development, but JavaScript comes from humble beginnings. A common misconception that new JavaScript programmers (and many other people) have is that JavaScript is somehow related to the Java programming language. Here's the first learning opportunity of this book: *JavaScript is not related to Java*. JavaScript was first built to enhance the web-browsing experience, but it's grown well beyond the browser to become an important programming language in Microsoft Windows 8.

JavaScript programs are made up of text, just like the text that makes up a page of this book. The text for JavaScript programs is created in a certain order and placed within a certain area so that a web browser will do something with it. In much the same way, the text in this book is sequenced: right now, you're reading this text, parsing its words, and producing results such as learning, validating your knowledge, or falling asleep.

In the world of computing, there is a model called *client-server*, where the client (think: *web browser*) requests resources (think: *webpage*) from a server, which is a different computer discoverable through a URL (Uniform Resource Locator, such as *http://www.microsoft.com*). When you request something from *http://www.microsoft.com*, your web browser is the client. It contacts the server for the webpage. The server sends the webpage (consisting of HTML, CSS, and JavaScript) back to the browser. The browser then shows, or *renders*, the page for your enjoyment.

The most common place that JavaScript runs is in the client web browser. This client-side execution should be differentiated from server-based languages such as C#, which run on the server. In the

webpage example already discussed, the Microsoft site is likely running a server-based language such as Microsoft Visual Basic or C# to create the webpages. JavaScript also runs in desktop widgets (notably in Windows 8), in PDF documents, and in other similar places. This section explores how JavaScript interacts with other client languages to provide a rich application experience.

**Note** JavaScript is also used as a server-side language with frameworks such as Node.js, but JavaScript's primary use remains as a client-side language.

## HTML, CSS, and JavaScript

The languages of front-end web development consist of HTML, CSS (Cascading Style Sheets), and JavaScript. HTML provides the descriptive elements that surround content to define the layout of the page. CSS provides the styling to make the marked-up content visually appealing. JavaScript provides the functional and behavioral aspects to both the content and the styling. These front-end languages are typically combined with back-end, server-side languages such as PHP, Visual Basic/C#, Java, or Python to create a full web application.

### HTML

HTML is the language of the web. It's the language developers use to create webpages. You can create working webpages and web applications with nothing more than just HTML; however, to today's sophisticated users, such pages would be boring and look horrible—but they would be webpages nonetheless. HTML is a *standard* or *specification* (two terms that are used relatively interchangeably, in this case) defined by the World Wide Web Consortium (W3C). There are several iterations of the HTML specification. The most current is version 5, known simply as *HTML5*.

HTML consists of *elements* enclosed in angle brackets (< and >). You already saw examples of HTML elements in the first tutorial in the chapter. HTML elements, also called *tags*, describe the content they enclose and how that content should be rendered by the browser. For example, an HTML tag for an image element is <img>. When the browser's rendering engine encounters that tag, it knows the contents of that tag should be a reference to an image file. Similarly, the  tag denotes a paragraph. Most tags, like , also have a closing or matching tag used to denote the end of that element. In the case of , the closing tag is . Other tags use a similar syntax, with a forward-slash (/) used to denote the end of the tag.

Tags can contain other content, called *attributes*, within their angle brackets. Attributes provide additional information about the content. Some attributes are generic and can be used with all tags, while others are specific to particular elements, such as the <img> tag. For example, the <img> tag uses the *src* attribute to specify the location of the image file that the browser will load and render. Here's an <img> tag that references an image called "SteveSuehring.jpg":

```
<img src="SteveSuehring.jpg">
```

Pages that adhere to the HTML standard (and every page you write should adhere to the standard) have certain elements that appear in a certain order. First among these is the Document Type Declaration, or *doctype*. (See the related sidebar for more information.) The HTML5 *doctype* is used throughout this book and looks like this:

```
<!DOCTYPE html>
```

An opening <html> tag follows the *doctype*. This <html> tag is then followed by the page's heading section. The heading section starts with a <head> tag and ends with the closing </head> tag. The <head> section is sort of a housekeeping area where information about the page itself is stored, such as the page's title. Additionally, the <head> section is where you find references to other files the page uses, such as files containing Cascading Style Sheets (CSS), and files containing JavaScript. You might also find CSS and JavaScript code placed directly into this heading section rather than in other referenced files.

**Tip** Don't confuse the <head> section with the heading or other visible elements on the page. The <head> section is used for housekeeping information about the page itself, not for display. The <title> is the only thing that displays from the <head> section, and that displays in the browser's title or tab bar only, not in the page itself.

## Document Types: DOCTYPE Declarations

Document Type Declarations, sometimes called DOCTYPE Declarations or DTDs, inform the parsing program (usually a web browser) what rules the webpage will follow for its syntax. If you fail to declare a DOCTYPE or use an incorrect DOCTYPE, the browser renders the page using its best guess, sometimes called *Quirks Mode*. In Quirks Mode, the browser chooses how to interpret elements and the resulting page might end up looking different than you intended.

The DTD used by some versions of Visual Studio is XHTML, though that varies widely among the versions and editions of Virtual Studio. The XHTML DTD looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

As you've seen, today's webpages should use HTML5 as a DOCTYPE so that they can take advantages of the advanced features offered by HTML5. The DTD for HTML5 is much simpler:

```
<!DOCTYPE html>
```

This book uses the HTML5 DTD exclusively for its projects.

The body of the webpage follows the closing </head> tag. A page's body begins with a <body> tag and ends with the corresponding </body> tag. Within the body of the webpage, you find the actual content, such as the text and images that make up what you see when you view the page in a

browser. The HTML to identify or *mark up* that content is also contained in the body. JavaScript code can also appear within the body of a webpage.

After the closing </body> tag, the page itself is closed by the </html> closing tag that matches the <html> tag that opened the webpage, way back up on the top. Here's a simple example that concisely shows you what I just spent five paragraphs explaining:

```
<!doctype html>
<html>
<head>
    <title>My Web Page</title>
</head>
<body>
    <div>My first content, in a div element</div>
</body>
</html>
```

Rendering this simple page in a browser results in a page similar to Figure 1-2.

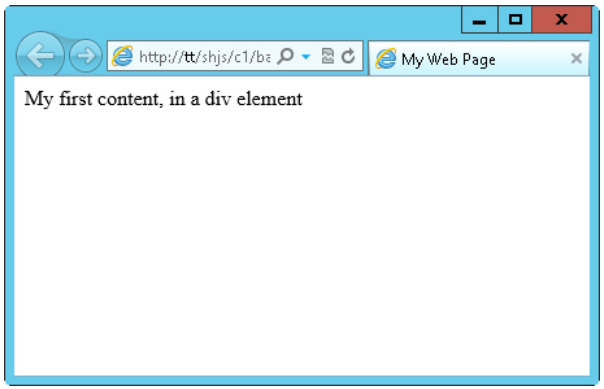


**FIGURE 1-2** A basic webpage.

**More Info** HTML is an expansive subject. There are countless books on the subject. This book won't cover the underlying HTML in much detail, but all of the examples will be explained as they pertain to JavaScript. For further reading on HTML, see the book *HTML5 Step by Step* by Faithe Wempen (Microsoft Press, 2011).

## CSS

HTML is frequently paired with CSS to help browsers deliver a visually appealing webpage. In essence, HTML describes the function of content, while CSS is responsible for providing the look and feel of that content. CSS enables page creators to define such things as colors, backgrounds (including background images), sizes of elements, and much more. For example, by adding some CSS to the

previous HTML example, I can change the size and add a border to the <h1> element. The CSS code is shown here in bold:

```
<!doctype html>
<html>
<head>
    <title>My Web Page</title>
</head>
<body>
    <h1 style="border: 1px solid black; font-size: 0.8em;">My first content,
        in an h1 tag</h1>
</body>
</html>
```

The result is shown in Figure 1-3. You can find this code as *basic.html* in the companion content.

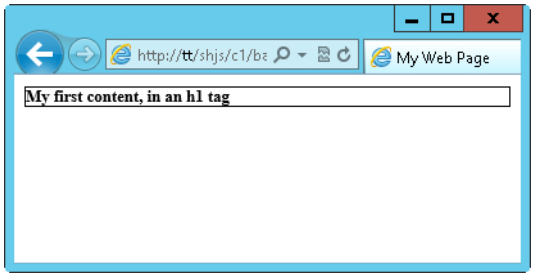


**FIGURE 1-3** Applying a bit of style with CSS.

This example adds a border to the element and changes the font size. Both of these changes were accomplished using CSS properties. A CSS property is a style attribute you want to change, such as the color or font of an element. CSS operates via rules, where you select items and then apply style rules to those selected items.

The properties available are dependent on both the browser and on the element being changed.

**Warning** Some browsers don't support all the features of CSS, so you need to make sure that the browsers that will view the page support the CSS properties you use in your pages.

The CSS in this example uses what's known as an *inline style* to change the element on which the style is applied. An inline style is CSS written directly into a tag, and it's perfectly OK to do that. However, it's more typical to use styles placed in an external CSS file and referenced by your HTML page. You'll see an example of external CSS later in this chapter. The main reason to use an external file is that CSS written for an element type can be applied across all elements of that type in a page. For example, if this page contained more than one <h1> tag, you could style all those in one sweeping line of CSS.

You can target CSS to only certain elements by using identifiers or *ids*. An id applies only to a single element on a page. CSS also uses the concept of *classes*. A class defines certain HTML elements that have common characteristics. For instance, you could create a class that includes only HTML elements in a certain section of a page. Collectively, tag names, ids, and class names are known as *selectors*. The essential syntax for CSS is as follows:

```
selector: { rule; }
```

Don't worry if all of this seems a bit hard to understand at the moment. Both the HTML and CSS should come into focus as you progress through the book.

**More Info** As with HTML, this book won't cover CSS in much detail beyond the scope of learning JavaScript. If you feel you need additional information or tutorials on CSS and how to use it, I invite you to take a look at *http://www.w3schools.com/css*.

### JavaScript

While HTML describes the function of content and CSS describes how to style that content, JavaScript is a programming language that's responsible for much of the behavioral or interactive elements seen on webpages and in web applications. This includes everything from drag-and-drop form elements to navigational menus to sliders, spinning graphics, and other enhancements (and sometimes annoyances) on websites. JavaScript is also frequently used to provide immediate, client-side form validation to check for errors and show feedback to the user.

Deploying a website or web application is a mix of designer, usability, and developer skills. Design skills provide the look and feel of elements, usability skills ensure that the look and feel works for the way end users will interact with the site or application, and finally, development skills make all of it work. Of course, one person can have more than one of these skills, or a team of people working on a web project might be needed in order to cover all these skills. Because you are reading a development book, it's safe for me to assume you want to learn or enhance that third web programming skill: development.

## JavaScript in Windows 8

Until recently, JavaScript was used primarily for client-side web applications, but that's changing. Windows 8 elevates JavaScript to a prominent role within the application development life cycle. For example, you can use JavaScript to create a fully functional application in Windows 8. These JavaScript programs have access to the file system and can interact with Windows itself through a library called Windows 8 Runtime (Windows RT). In other words, JavaScript is an equal partner in Windows 8 alongside more traditional application-level languages such as Visual Basic, C#, and so on. Much of JavaScript's power in Windows 8 comes through another library, called WinJS, which this book will cover in detail in Chapter 8, "Using JavaScript with Microsoft Windows 8."

## Placing JavaScript in a Webpage

The overall title for this section is "Where JavaScript Fits," but so far I've discussed only the conceptual environment in which JavaScript operates. It's time to fix that by discussing the literal location for JavaScript code on a webpage.

Just as you need to use an HTML <img> tag to inform the browser it should expect an image, you use the <script> tag to inform the browser it will be reading a script of some sort. There are a few different kinds or types of scripts web browsers can read; JavaScript is one of them.

You place JavaScript within the <head> or <body> section (or both) of an HTML document and you can place multiple scripts on a given page.

The browser executes JavaScript as it is encountered during the page-parsing process. This has practical implications for JavaScript developers. If your JavaScript program attempts to work with some elements of the HTML document before those elements have been loaded, the program will fail. For example, if you place JavaScript in the <head> section of a page and that code attempts to work with an HTML element that's all the way down at the bottom of the page, the program might fail because the browser doesn't yet have that element fully loaded. Unfortunately, the program will probably fail in subtle and difficult to troubleshoot ways; one time the program will work, and the next time it won't. That happens because one time the browser will have loaded that HTML element by the time it executes the JavaScript code, but the next time it won't. An even more fun (not really) failure scenario is when everything works in your local development environment on your computer but fails when deployed in real-world (and real slow network) conditions. One especially good method for solving this problem is with the jQuery *ready()* function, as you'll see later.

The basis of JavaScript's close coupling with a webpage is through the Document Object Model (DOM). Just as your perception of the elements on the page comes through a text editor or Visual Studio, the DOM represents the programmatic or browser view of the elements on a page. Much of what you do as a JavaScript programmer is work with the DOM. Unfortunately, the DOM works in slightly different ways depending on the web browser being used to render the page. Of course, if you've done any HTML and CSS work, you're already familiar with the different and nuanced ways in which browsers render pages. The same is true for JavaScript. You'll spend a nontrivial amount of time either writing for various browsers or troubleshooting why something isn't working in a given browser.

For anything but the most basic scripts, you should use external JavaScript files. This has the advantage of providing reusability, ease of programming, and separation of HTML from programming logic. When using an external script (which is how most of this book's examples will be constructed in later chapters), you use the *src* attribute of the <script> tag to point the browser at a particular JavaScript file, in much the same way you use the *src* attribute of an <img> tag to specify the location of an image in an HTML page. You'll see how to specify external scripts in more detail later in this chapter, but here's an example:

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

### Script Types

While we're discussing the <script> tag, it's a good time to discuss the *type* attribute, which you can see in the previous example. The *type* attribute specifies the Multipurpose Internet Mail Extensions (MIME) type of the script. There is a good deal of discussion as to whether the *type* attribute should even be used—and if it should, what it should contain for JavaScript. Some developers don't use the *type* attribute at all, while others use a *language* attribute instead. Some developers use both a *language* attribute and a *type* attribute.

There's also confusion as to whether to use *text/javascript*, *text/ecmascript*, or the newer *application/javascript* or *application/ecmascript* as the value for the *type* attribute. As a JavaScript programmer, be prepared to see variations of the *type* attribute, to see the *type* attribute missing, to see the *language* attribute, or to see some combination thereof.

The value I'll use throughout this book is *text/javascript* because, as of this writing, it enjoys the most compatibility and support across browsers. I've had the most success using the *type* attribute.

### No JavaScript? No Problem.

Sometimes JavaScript isn't available in your visitor's web browser. The user might be using assistive technologies or maybe she just disabled JavaScript manually. Whatever the case, the <noscript> tag helps if JavaScript isn't available in the browser. The <noscript> tag is used by most web browsers when scripting is unavailable. The content between the opening <noscript> and closing </noscript> tags displays for the user, typically informing the user (politely, of course) that JavaScript needs to be enabled in order for the site to function properly.

***See Also*** *See* http://www.w3schools.com/tags/tag_noscript.asp *for more information about using the <noscript> tag.*

## Writing Your First JavaScript Program

In this section, you create your first JavaScript program. As a side effect of doing so, you also create your first webpage with HTML.

JavaScript is tool-agnostic, meaning you can use any text editor or Integrated Development Environment (IDE) to write JavaScript. For example, Microsoft Visual Studio provides a powerful development experience for writing JavaScript. The same can be said for Eclipse, the open-source IDE. However, an IDE is certainly not a requirement for creating and maintaining JavaScript. You can also use a simple text editor such as Notepad or a more powerful text editor such as Vim to write JavaScript.

You're only a few pages into this book and you already have an important decision to make: What tool should you use to write JavaScript? The guidance I can offer for this choice is limited because I believe you should use whatever tool you're comfortable with for programming. If JavaScript required

a specialized IDE, the choice would be easy: you'd have to use that IDE. However, you can write JavaScript in anything from a simple text editor to a full programming IDE such as Eclipse or Visual Studio, so some might even argue that it's *easier* to just use a text editor.

For much of the independent web development and JavaScript writing that I do, I use Vim, because it's lightweight and gets out of the way. However, I also use Eclipse and Visual Studio for development, depending largely on the platform for which I'm writing code. The choice is yours as to how you prefer to develop JavaScript. Although this book shows examples in Visual Studio, you shouldn't feel that you must use that IDE to work with the code in this book. The one area where Visual Studio makes your life easier is when writing Windows 8 Apps. I'll stop short of saying that Visual Studio is required when writing JavaScript-based applications for Windows 8, but for the purposes of this book, Visual Studio is the platform you should expect to see for the Windows 8 chapter.

If you choose to not use Visual Studio, I'll assume you have a way (and the knowledge) to view the HTML pages you produce in a web browser. Many of the early examples won't require a web *server*, but the later chapters do require a web server. (Visual Studio includes a suitable web server.)

This book uses the Express Edition of Visual Studio 11 throughout. Visual Studio 11 Express Edition is available at no cost from Microsoft as explained in the following sidebar. No prior knowledge of Visual Studio is required for this book. Additionally, and just as importantly, writing JavaScript for the browser doesn't require the advanced features of Visual Studio. What you need to know will be shown along the way to complete a task, but you do need to have Visual Studio installed first.

### Getting Visual Studio 11 Express Edition

Visual Studio 11 Express Edition is available as a free download at *http://www.microsoft.com/visualstudio/11/downloads#express*, along with other tools related to development. For the purposes of the book, you want both the Express for Windows 8 and Express for Web. The first portion of this book uses the Express Edition for Web, while Chapter 8 requires the use of the templates specific to Windows 8 development. Installation of Visual Studio is typically a matter of executing the downloaded file from Microsoft, but you should refer to the documentation for the latest information at the time of installation.

## Writing JavaScript in Visual Studio 11

Writing JavaScript in Visual Studio 11 involves setting up a new project and writing the script or scripts to be used on the page or pages involved in your web application. If you're using a text editor or a different IDE, you can follow these examples and simply view the resulting file within a web browser locally. For this example, you create a simple webpage that displays text using JavaScript, in the same way that the first example showed earlier in the chapter.

The first step in programming JavaScript with Visual Studio is to create a new project. As a developer, you have a choice of several templates for beginning a project. For this example and most examples in this book, you'll use the ASP.NET Empty Web Application template, which is found in the Web category. The ASP.NET Empty Web Application template avoids much of the proprietary ASP.NET-related material that's not necessary for creating a JavaScript web application. Here are the steps:

1. Within Visual Studio, choose File, New Web Site.
2. In the New Web Site dialog that appears (shown in Figure 1-4), select the ASP.NET Empty Web Site. In the Web location text box, type **StartHere** as the name, (you might need to scroll to the right to get to the end of the File System and then click OK to create the website).

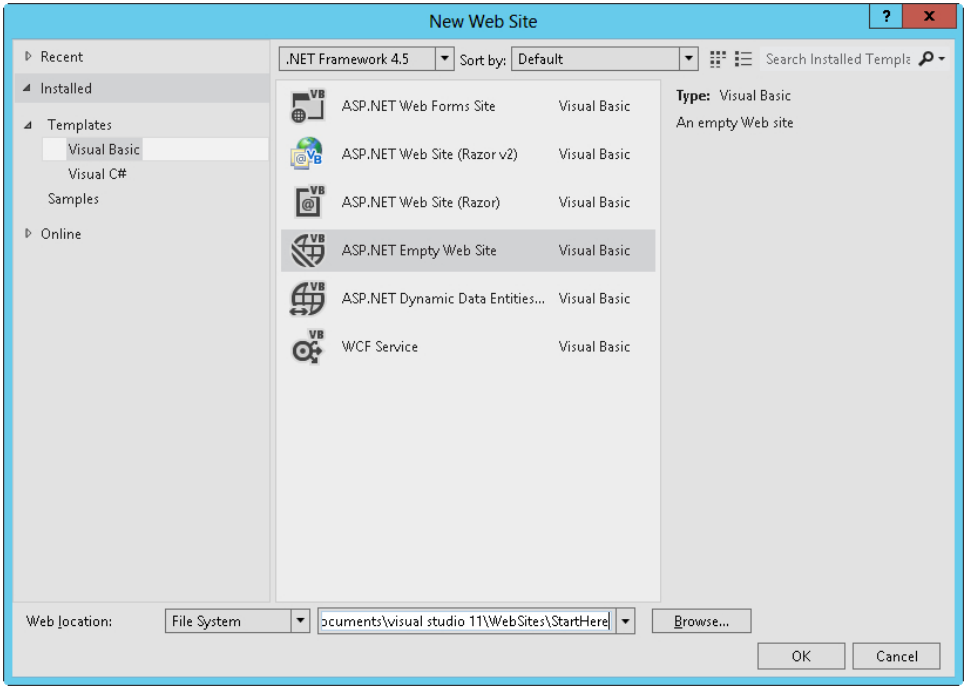


**FIGURE 1-4** Creating an ASP.NET Empty Web Application project in Visual Studio 11.

A blank, empty project opens, like the one shown in Figure 1-5.

Now it's time to add a file to the project.

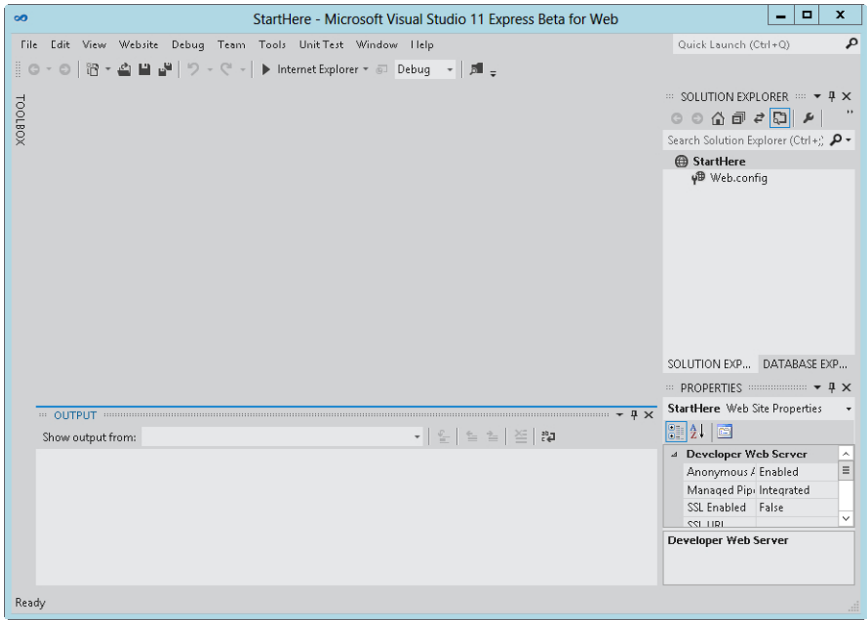


**FIGURE 1-5** An empty web project in Visual Studio 11.

**3.** For this first example, add an HTML file. On the File menu, select New File.

The New File dialog box appears, such as the one shown in Figure 1-6.

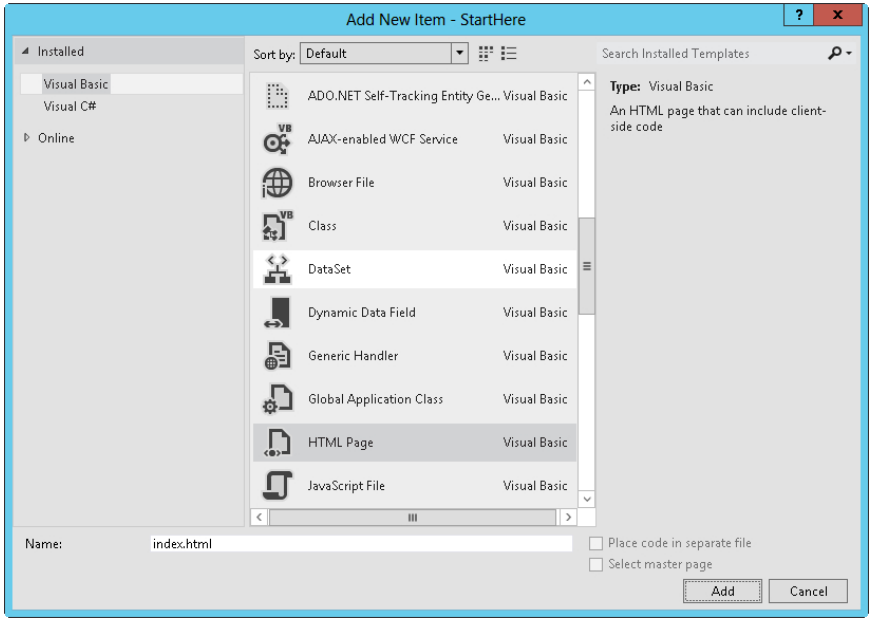


**FIGURE 1-6** The New File dialog within Visual Studio 11 is where you add new files to your project.

4. Click HTML Page, change the name to **index.html**, and then click Add.

   Depending on the version of Visual Studio you're using, you might see a basic HTML page that uses an XHTML document type similar to that in Figure 1-7. Some versions of Visual Studio use an HTML5 doctype as you've seen throughout the chapter, so your screen might look different than this.


**FIGURE 1-7** A beginning page with Visual Studio 11.

5. If your version of Visual Studio has the XHTML doctype as shown, the first thing you need to do is switch the DOCTYPE for the page. If your doctype is already the HTML5 doctype (<!DOCTYPE html>), there's nothing to change and you can skip this step. If you need to change it, highlight the entire DOCTYPE declaration in Visual Studio and delete it. Replace that long (and ugly) DTD with the following:

```
<!DOCTYPE html>
```

6. Within the <html> declaration, remove the *xmlns namespace* attribute. This applies only if it's present. Some versions of Visual Studio don't have this attribute present. If it's not there in yours, you don't need to change it.

   Regardless of the version of Visual Studio you have, you need to change the <title> tag so that it contains the words **Start Here**. With those three changes (which you can find in the companion content as *index.html*), the page should look like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
</head>
<body>
```

```
</body>
</html>
```

The file is still unsaved and will have the default name chosen by Visual Studio, as seen in Figure 1-8. It's time to save the file and view it in a web browser.

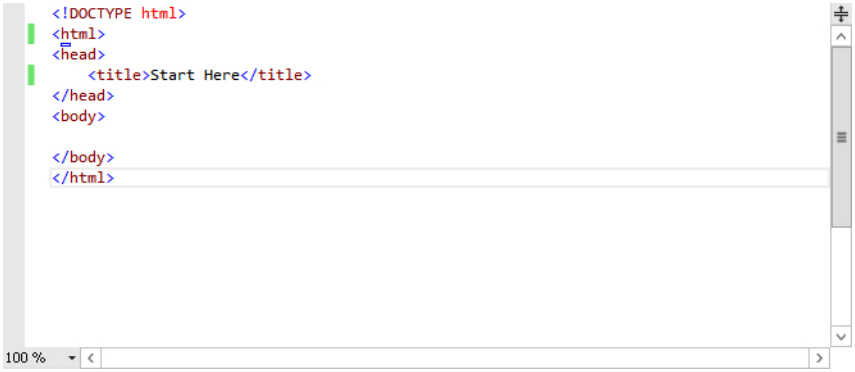


**FIGURE 1-8** Making basic edits on your first webpage through Visual Studio.

7. If you haven't already saved the file, save it now. On the File menu, click Save or use the Ctrl+S keyboard shortcut. In some versions of Visual Studio, you'll see a Save File As dialog, as shown in Figure 1-9. Save the file within your project, and name it **index.html**. Note that you might not see this dialog at all if you're using Visual Studio Express Edition for Web.

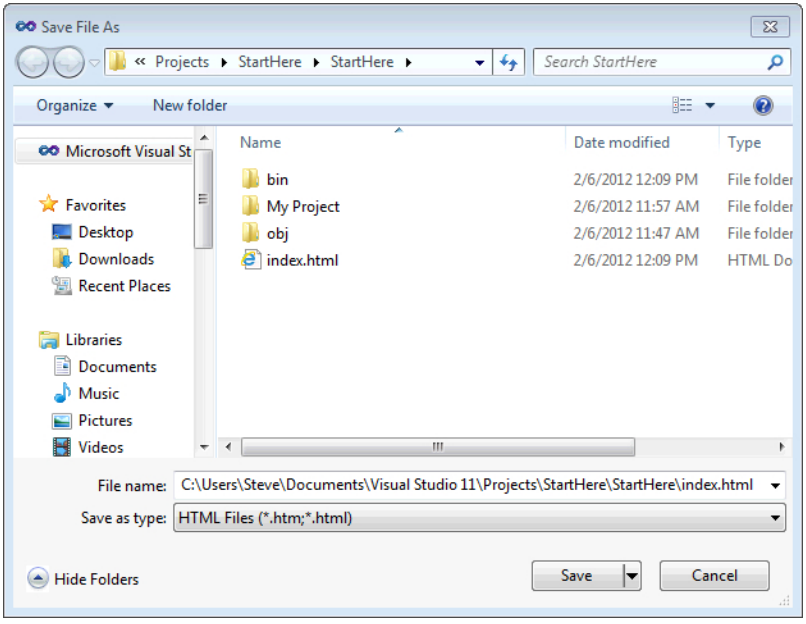


**FIGURE 1-9** Saving the webpage as *index.html* within the project.

**Tip** Make sure you save the file within the project itself, as shown in this dialog. Visual Studio might attempt to save the file outside of the project. (Note the two "StartHere" directories in Figure 1-9.)

8. With the file saved, the next step is to view it in a browser. The easiest way to do this in Visual Studio is to click the Run button on the toolbar or select Start Debugging from the Debug menu. When you do so, Visual Studio performs some background tasks, starts a web server (Internet Information Services, or IIS), and launches your default web browser. Note that you might see a dialog indicating that debugging isn't enabled. Accept the default, and click OK to modify the web.config to enable debugging.

   If all goes well, you eventually see a page like that shown in Figure 1-10.

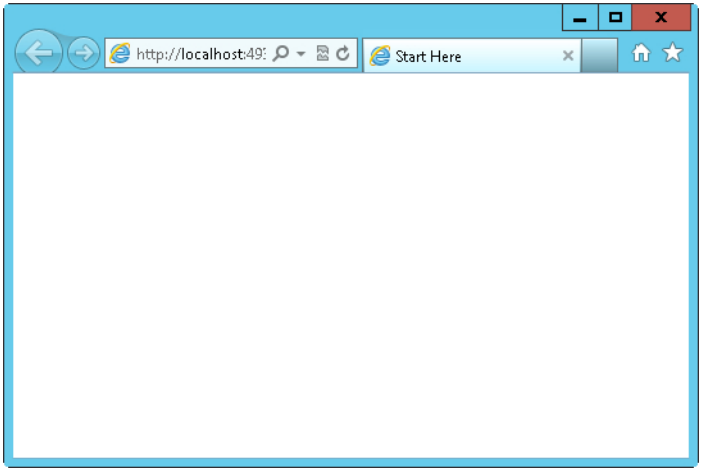


**FIGURE 1-10** A successful run of your first webpage.

   If you see a screen like that in Figure 1-10, your project ran successfully. Even though the page itself is blank, notice that the title bar reads "Start Here," thanks to the change you made to the <title> tag in the page.

9. Close your web browser.

   Closing your browser has the effect of stopping the project from running, and you'll be back at the source window for your *index.html* page.

10. Within *index.html*, add the following code between the opening <body> tag and the closing </body> tag (saved as *index-wjs.html* in the companion content):

```
    <script type="text/javascript">
        document.write("<h1>Start Here!</h1>");
    </script>
```

The code for the final page will look like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
</head>
<body>
    <script type="text/javascript">
        document.write("<h1>Start Here!</h1>");
    </script>
</body>
</html>
```

11. To run that code, on the Debug menu, click Start Debugging, or on the toolbar, click the green Run button.

    You'll now see a webpage like the one in Figure 1-11.

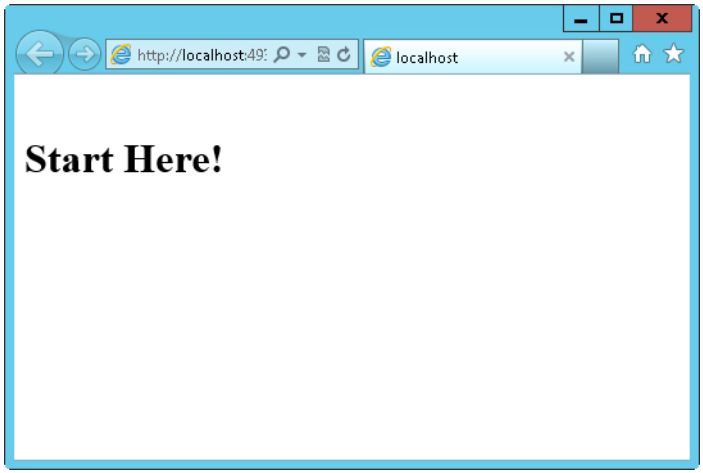


**FIGURE 1-11** Running your first JavaScript program.

Congratulations! You successfully created your first JavaScript program with Visual Studio. Prior to closing Visual Studio, you'll add two folders in preparation for future chapters.

12. In the Solution Explorer (normally on the right), right-click the name of the site, StartHere, click Add, and then click New Folder. Name the folder **js**.

13. Add another folder by using the same process: in Solution Explorer, right-click the StartHere project, click Add, and then click New Folder. Call this folder **css**. Your final Solution Explorer should look like Figure 1-12.

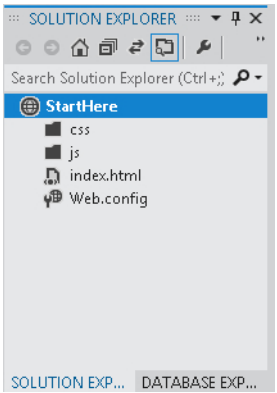


**FIGURE 1-12** The final Solution Explorer with two folders added.

### What If Your Code Didn't Work?

You might receive an error when attempting to run your program or view the webpage. Often, entering that error into your favorite search engine yields helpful results. However, here are some troubleshooting ideas that might help you along.

- **Check the syntax** JavaScript is case sensitive. Ensure that the case used in your code matches the example exactly. Also, ensure that your <script> tags are properly formatted and show up between the opening <body> and closing </body> tags.
- **Check the file location** If you save the webpage in the wrong location, the web server won't be able to find it, and you'll receive an error indicating that no default page was found. Move the file within the project, and run it again to solve this issue.
- **View Debug Output** Visual Studio displays output from its compile and readiness checks in the Output pane. You might find that your browser won't run or see other useful tidbits of information in the Output pane that will help solve the problem.

## JavaScript's Limitations

Prior to wrapping up the chapter you should see some of the limitations of JavaScript. Some of these are subtle and might not be obvious. JavaScript's largely client-side role means that there are some inherent issues you need to consider when developing with the language.

## Don't Rely on JavaScript for Data Security

JavaScript executes on the client. This means that anything—including data, or even the program itself—can be manipulated by users in any way they deem necessary. Users will try to send bad data back (for example, changing item prices in a shopping cart or anything they can get their virtual hands on), and they'll also try to inject scripts or their own programs into your code so that it gets executed by your server.

You should always assume that data is incorrect when it arrives back at your server, whether it's from a web form or through JavaScript. Only after proving, within your server-side programs, that the data is valid should you proceed to use it within an application. Under no circumstances should you use data without validating it on the server side, when it is under your control.

This warning specifically includes any JavaScript-based validation of form data. For example, JavaScript is frequently used to provide instant feedback to a user typing into a web form. However, just because there is JavaScript validation in place to ensure that data is properly formatted in the browser doesn't mean that it's actually formatted correctly. This data needs to be checked again on the server.

This is an area where I've seen developers attempt all sorts of trickery to make sure that the JavaScript's validation occurs rather than just check the data within the server program. None of the tricks work, so the time is better spent validating in your server-side program rather than attempting to add another layer of complexity.

One common misconception is that using a *POST* request rather than a *GET* request will keep the data secure. This is not true. Data sent by using *POST* is just as vulnerable as that sent through a *GET* (where the parameters show up in the address bar). So what can you do? Check the data in the server program.

## You Can't Force JavaScript on Clients

A visitor's web browser might not run JavaScript, or it might not run the version of JavaScript your program needs. As a developer, you must consider the case where JavaScript isn't available on the client, whether by their choice, due to accessibility, because of a device limitation, or for any reason. Your pages should work without JavaScript by providing an alternative means for accomplishing the task or interacting with your application and content.

However, today's advanced applications sometimes do require JavaScript. In such cases, you should detect which features are and aren't available and provide messaging to the user indicating that JavaScript is required for the application.

In essence, the only way you can fail is by assuming JavaScript will always be available and, hence, neglecting to create a way to handle its absence.

**More Info** For more information on feature detection and browser detection in general, see my more advanced book, *JavaScript Step by Step* (Microsoft Press, 2011).

A variation of this problem is that the versions of JavaScript and their implementations vary widely between web browsers and between versions of the browsers. JavaScript is defined in the ECMA-262 specification, but each browser vendor interprets that specification slightly differently, in much the same way that those same browser vendors interpret HTML and CSS standards differently. The good news is that there's always work for people who understand these browser differences; the bad news is that it can be very frustrating and time consuming to allow for such differences in your code so that it works in whatever browser your visitors are using.

This problem is slowly becoming less and less important. Inside organizations that standardize on specific browsers and versions, it's a minor problem. However, if your application will be used on the Internet, you need to accommodate differences between web browsers. The primary way to find out if your page works in other browsers is to test it in other browsers. This is not terribly difficult to do, but recently it has become even more cumbersome with the advent of mobile devices, which have their own browser implementations.

Microsoft provides free Application Compatibility images for Virtual PC. These are full operating systems with various versions of Internet Explorer installed on them. The current link for these is *http://www.microsoft.com/download/details.aspx?id=11575*, but if that link changes, an Internet search for "Internet Explorer Application Compatibility VPC" should result in the updated location for those images.

Other browsers such as Chrome, Firefox, and Safari are free downloads. I recommend that you acquire them and test your applications in those browsers, as well.

## Summary

This chapter introduced the basics of JavaScript, including what it can and can't do, and how it relates to other programming and markup languages. Within the chapter, you saw that JavaScript frequently works closely with the markup languages HTML and CSS to provide the interactivity that today's web users have come to expect. Windows 8 introduces a new level of importance for JavaScript by making it an equal partner in the application development life cycle.

You saw how to create a project in Visual Studio as well as how to create a simple JavaScript program. Part of that program involved learning where to place JavaScript code on a page. You also saw some key concepts about JavaScript, including that you can't always rely on JavaScript being available on the user's browser and that you should never rely on JavaScript for data security.

In the next chapter, you'll look at some specifics of the syntax of JavaScript—including mundane yet important details about spacing, comments, and case sensitivity—before getting into more fun details like looping and conditionals. That will set a brief, yet firm foundation upon which you can build complex programs in later chapters.

Chapter 2

# JavaScript Programming Basics

**After completing this chapter, you will be able to**

- Know where to place JavaScript in a webpage
- Understand basic JavaScript syntax
- Create JavaScript variables and understand common data types
- Use looping and conditional constructs in your JavaScript code

**NOW THAT YOU HAVE A TASTE** of what JavaScript can do, it's time to look at JavaScript's inner workings. Admittedly, the chapter won't get too far into the inner workings—just far enough to make you proficient but not overwhelmed. Between this chapter and the next, you'll have a firm foundation with which you'll be able to build programs and troubleshoot existing JavaScript programs.

In this chapter, you'll look at the syntax and rules of the language. This includes things as simple as how to place a comment in a program to how to create a variable and perform conditional logic. This material is largely condensed while still providing what you need to know. Like Chapter 1, "What Is JavaScript?" this chapter will also focus primarily on JavaScript and its use in web applications. However, the information applies to JavaScript for Microsoft Windows 8, as well.

## JavaScript Placement: Revisited

You learned in Chapter 1 that JavaScript programs are placed within the <head> or <body> sections of a webpage. For external scripts such as jQuery, it's common to place the reference in the <head> section, but there's nothing preventing you from placing the reference to the external script in either the head or body of the page. This section expands on the placement of JavaScript by creating a base page and an external JavaScript file. The files created here will be used as the basis for examples throughout the book.

The basic HTML page that will be used should look familiar if you've just come from Chapter 1. In fact, this chapter will use the project created in Chapter 1—specifically the structured layout as shown in Chapter 1—with css and js folders created in the project or document root. See Figure 2-1 for an example of this structure.

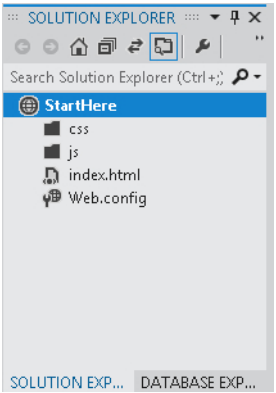


**FIGURE 2-1** The folder structure for development for the book should include css and js folders within the project.

Create a new file, or alter your existing *index.html* to match the code in the following sample. If you create a new file, save it as **index.html** (which you can find in the companion content as *index.html*).

```
<!DOCTYPE html>
<html>
<head>
<title>Listing 1-1</title>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
</body>
</html>
```

With *index.html* created, it's time to create an external JavaScript file. Within Microsoft Visual Studio, in Solution Explorer, right-click the js folder, click Add, and then click Add New Item. The Add New Item dialog box opens. Click JavaScript File and then in the Name text box, type **external.js**, as shown in Figure 2-2.

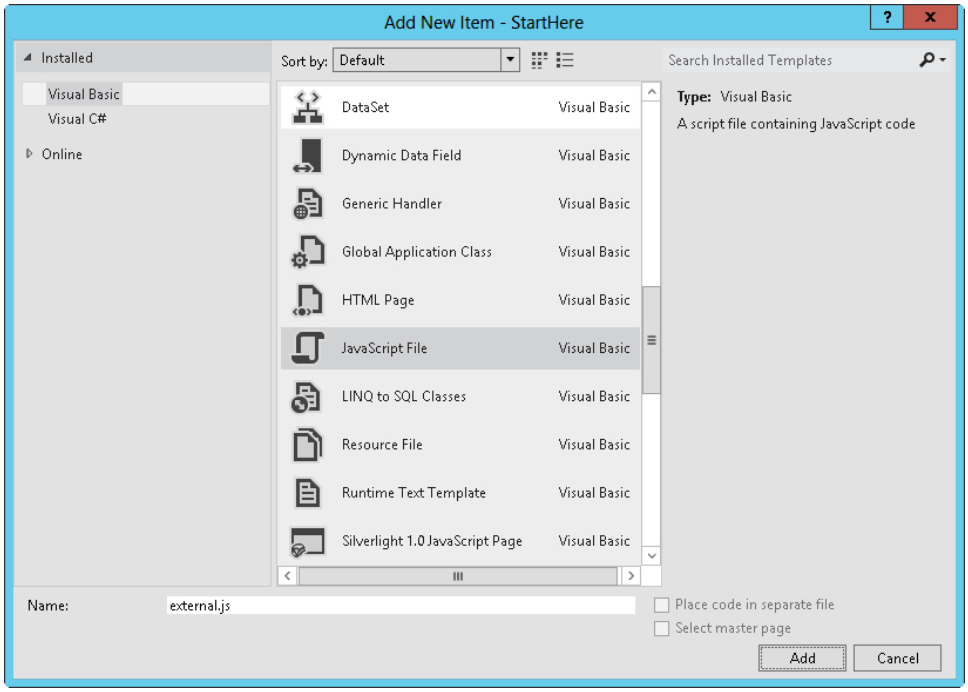


**FIGURE 2-2** Creating a new JavaScript file.

Click Add and a new JavaScript file appears.

Some versions of Visual Studio don't allow you to set the name of the file on creation, as in the previous example. If this is the case, the file will be named *JavaScript1.js*. However, the file referenced in the HTML file is *external.js* within the js folder. Therefore, the default *JavaScript1.js* file name will need to change.

Within Visual Studio, click File and then click Save As. The Save File As dialog box opens. Open the js folder and then enter **external.js** in the File Name text box, as shown in Figure 2-3. (Note that your screen shot might be slightly different than mine, and again, you only need to do this if your version of Visual Studio didn't allow you to set the name when you added the file.)

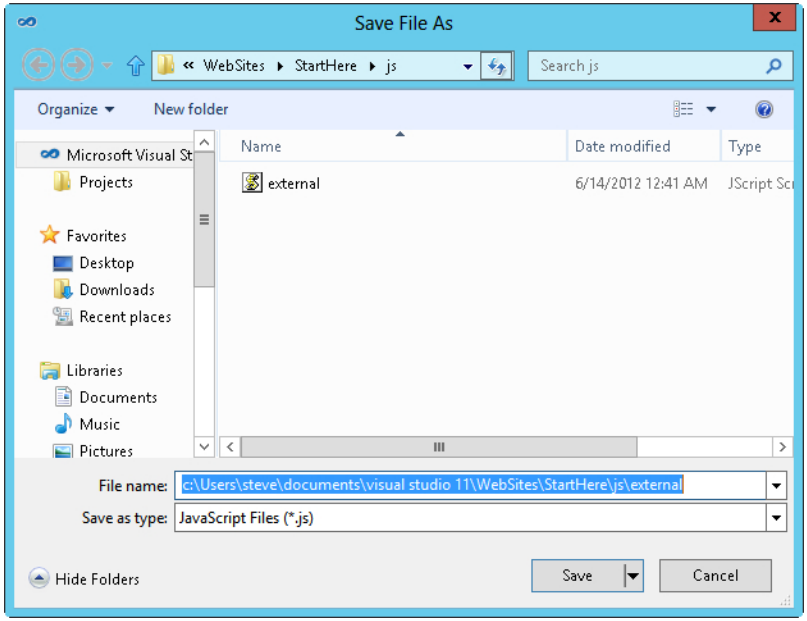


**FIGURE 2-3** Saving the *external.js* file into the js folder.

You now have a basic HTML page and an external JavaScript file. From here, the remainder of the chapter (and indeed the book) will use these files to show examples.

# Basic JavaScript Syntax

In much the same way that learning a foreign language requires studying the grammar and sentence structure of the language, programming in JavaScript (or any other computer language) requires learning the grammar and structure of a program. In this section, you'll learn some of the syntax of JavaScript.

## JavaScript Statements and Expressions

JavaScript is built around statements and expressions, where *statements* are simple lines of code and *expressions* produce or return values. Consider these two examples:

- Statement:

```
if (true) { }
```

- Expression:

```
var myVariable = 4;
```

In these examples, *myVariable* is the result, thus making it an expression, whereas nothing is returned from the *if (true)* conditional. While this admittedly is somewhat nuanced, what you need to know is that JavaScript has a certain structure that's made up of statements and expressions.

Lines of code are typically terminated with a semi-colon. The exceptions to this rule include conditionals, looping, and function definitions, all of which will be explained later.

One or more JavaScript statements and expressions make up a JavaScript program or script. (These two terms, program and script, are used interchangeably.) You saw examples of JavaScript programs in the previous chapter.

## Names and Reserved Words

JavaScript statements and expressions are made up of valid names (known as *identifiers* in the ECMA-262 specification) and words reserved for JavaScript itself. You saw several reserved words used already in the book. In JavaScript, the following are reserved words and therefore should be used only for their intended purpose; you can't use these as a variable or function name, for example.

**More Info** You can see the full ECMA-262 specification at *http://www.ecma-international.org/publications/standards/Ecma-262.htm*.

| | | | | |
|---|---|---|---|---|
| *break* | *delete* | *if* | *this* | *while* |
| *case* | *do* | *in* | *throw* | *with* |
| *catch* | *else* | *instanceof* | *try* | |
| *continue* | *finally* | *new* | *typeof* | |
| *debugger* | *for* | *return* | *var* | |
| *default* | *function* | *switch* | *void* | |

Several other words are reserved for future use; therefore, you shouldn't use these in your programs for your own purposes either.

| | | | |
|---|---|---|---|
| *class* | *extends* | *let* | *public* |
| *const* | *implements* | *package* | *static* |
| *enum* | *import* | *private* | *super* |
| *export* | *interface* | *protected* | *yield* |

When using JavaScript, you must use certain naming conventions. Valid names begin with a letter, a dollar sign ($), or an underscore (_) and cannot be a reserved word.

**Note** A backslash escape character (\) is also valid to begin names with, but its use is rather uncommon.

The following are valid names:

- *myVariable*
- *EMAILADDR*
- *SongName*
- *data49*

The following are invalid names:

- *var*
- *3address*
- *deci#*

The first example, *var*, is a reserved word and therefore cannot be used to name your own variables or functions. The second, *3address*, begins with a number, and the final example, *deci#*, contains a special character.

**Tip** Though not required, it's common to see variable and function names begin with a lowercase letter (such as *myVariable*). When variables begin with a lowercase letter and then use capital letters for other words, it's called *camelCase*. Other capitalization conventions exist. See *http://msdn.microsoft.com/library/ms229043.aspx* for more information.

## Spacing and Line Breaks

JavaScript largely ignores white space, or the space between elements and statements. Obviously, you need to separate words within a line by at least one space, but if you use two or more spaces, JavaScript typically won't care. That said, you'll spend less time chasing down difficult bugs if you just stick to standard single spacing. For example, this is valid:

```
var myVariable = 1209;
```

In this example, there's a single space between the keyword *var* and the name of the variable, *myVariable*. That space is required for the JavaScript interpreter to run the code.

Closely related to white space are line breaks or carriage returns, officially known in the JavaScript specification as *line terminators*. In general, line breaks are not required. In fact, you'll sometimes see JavaScript programs with no line breaks whatsoever. This is called *minification*; it reduces the size of the JavaScript downloaded by the visitor. However, I recommend that when you develop your programs, you use standard line breaks after each JavaScript statement and expression.

## Comments

Comments are lines within programs that aren't executed. Comments are frequently used to document code behavior within the code itself. Consider this code:

```
// myVariable is used to count characters
// Generate an alert when myVariable has more than 10 characters
// because this indicates we've exceeded some business rule.
if (myVariable > 10) {
```

In this example, there are three lines of comments prior to the *if* statement.

### Comment Style

The comment shown in the example indicates not only what *myVariable* does, but also why we're testing it. This is an important point to consider when using comments to document code. The time you spend writing the code is short relative to the time you spend maintaining it. It's quite obvious by looking at the code *if (myVariable > 10)* that it's testing to see if *myVariable* is greater than 10. However, what isn't clear from the code itself is *why* it's testing to see whether *myVariable* is greater than 10. In other words: What's the significance of 10? In this example, I commented that the "greater than 10 condition" means that the variable's content violates a business rule. Ideally, I'd also include which business rule was violated in the comment.

JavaScript comments come in two forms: single-line comments with a double slash (*//*), as you've seen, and the C-style multiline comment syntax (*/* */*), so named because of the C programming language. The double slash you saw in the first example is a single-line comment that indicates to the JavaScript interpreter that everything following the two slashes up to the next line break should be ignored.

The multiline comment structure indicates that everything beginning with the opening /* up to the closing */ should be ignored, as in this example:

```
/* myVariable is used to count characters
 Generate an alert when myVariable has more than 10 characters
 because this indicates we've exceeded some business rule.*/
if (myVariable > 10) {
```

One important point with multiline comment syntax is that multiline comments can't be nested. For example, this is invalid:

```
/* A comment begins here


  /*
   myVariable is used to count characters
   Generate an alert when myVariable has more than 10 characters
   because this indicates we've exceeded some business rule.
  */
  if (myVariable > 10) {

*/
```

In this example, the interpreter will happily begin the comment where you want it to, but once it encounters the first closing **/* sequence it will just as happily close the comment. Things will go haywire when the interpreter encounters the final closing **/* sequence, and an error will be the result.

**Tip** In practice, I find that I use the double slash convention most often. I do this for two reasons. First, it's easier to type two slashes. Second, the double slash comment syntax also allows me to comment out large sections of code using the multiline syntax and conveniently gets around the problem of multiple nested multiline comments shown in the previous example.

## Case Sensitivity

JavaScript is case sensitive. This fact alone trips up many programmers, experienced and new alike. When working with the language, the variable name *MYVARIABLE* is completely different than *myVariable*. The same goes for reserved words, and really everything else in the language. If you receive errors about variables not being defined, check the case.

Additionally, case sensitivity is essential for accessing elements from HTML pages with JavaScript. You'll frequently use the HTML *id* attribute to access the element with that *id* through JavaScript. The case of the *id* in your code needs to match the case of the *id* as written in HTML. Consider this HTML, which creates a link to an example website:

```
<a href="http://www.example.com" id="myExample">Example Site</a>
```

The HTML itself could be written in any case you want—all uppercase, all lowercase, or any combination you'd like. The web browser will show the page the same. However, you're now a JavaScript

programmer, and one thing you'll do frequently is access HTML from JavaScript. You might do this to change the HTML, create new parts of pages, change colors, change text, and so on. When you access HTML from within JavaScript, the case you use in the HTML suddenly becomes important. For example, you get access to that element in JavaScript with a special JavaScript function called *getElementById*, which, as the name suggests, retrieves an element using its *id* attribute, like so:

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

In this example code, the case of the *id* attribute's value (*myExample*) is essential. Trying to access the element using *MYEXAMPLE* or *myexample* or *MyExample* will not work. Just as important, the JavaScript function *getElementById* is itself case sensitive. Using *GETELEMENTBYID* or the more subtle *getElementByID* won't work. If you didn't care about case before, now's the time to start!

While we're on the subject of case, it's good practice to keep case sensitivity going throughout your code, whether it's JavaScript or something else. This is true both within code and for URLs and file names.

## Operators

JavaScript has operators to perform addition, subtraction, and other math operations, as well as operators to test for equality. The math-related operators are the same as those you learned in elementary school math class. You use the plus sign (+) for addition, a minus sign (–) for subtraction, an asterisk (*) for multiplication, and a forward slash (/) for division. Here are some examples:

```
// Addition
var x = 5 + 3.29;
// Subtraction
var number = 4901 - 943;
// Multiplication
var multiplied = 3.14 * 3;
//Division
var divide = 20 / 4;
```

Some important operators for programming are equality operators. These are used within conditionals to test for equality. Table 2-1 shows these equality operators.

**TABLE 2-1** Equality operators in JavaScript

| Operator | Meaning |
|---|---|
| == | Equal |
| != | Not equal |
| === | Equal, using a more strict version of equality |
| !== | Not equal, using a more strict version of inequality |

The difference between the normal equality operator (==) and the strict equality operator (===) is important. The strict equality test requires not only that the values match, but also that the types match. Consider this example:

```
var x= 42;
var y = "42";


x == y  // True
x === y // False
```

Later in the chapter, you'll create a sample program that tests these operators.

Relational operators test how a given expression relates to another. This can include simple things such as greater than (>) or less than (<), as well as the *in* operator and *instanceof* operator. You'll see examples and explanations of the *in* and *instanceof* operators as they're used.

There are also operators known as *unary* operators. These include the increment operator (++), the decrement operator (––), the delete operator, and the *typeof* operator. As with other operators used in this book, when it's not obvious, their use will be explained as you encounter them.

## JavaScript Variables and Data Types

Variables and data types define how a programming language works with user information to do something useful. This section will look at how JavaScript defines variables and the data types within the language.

### Variables

Variables contain data that might change during the course of a program's lifetime. Variables are declared with the *var* keyword. You've seen several examples of this throughout the book already, but here are a few more:

```
var x = 19248;
var sentence = "This is a sentence.";
var y, variable2, newVariable;
```

**Note** The third statement declares three variables at once but doesn't initialize them. Initializing a variable includes setting a value for that variable—the part with the equals sign (=).

Variable values are set or initialized by using the equals sign (=) or equals operator.

Two items are used in a similar fashion to variables: arrays and objects. Objects are discussed in the next chapter. In JavaScript, arrays are actually an object that acts like an array. For our purposes, we'll treat them as standard arrays, though, because the differences aren't important at this stage.

But wait, what's an array? Arrays are collections of items arranged by a numerical index. Put another way, think of the email messages in your inbox. If you're like me, that inbox is stacked with hundreds of emails. In programming terms, this is an array of emails. The emails in my inbox begin at 1 and continue through 163. In programming, however, it's typical for arrays to start at the number 0 and continue on—so rather than having emails 1 through 163, I'd have 0 through 162. When I want to access an item in the email inbox array, I would do so by using its numbered index.

In JavaScript, arrays are created with something called the array literal notation, *[]*, like this:

```
var myArray = [];
```

That syntax indicates that a variable called *myArray* will be an array. The line of code shown merely declares that this variable will be an array instead of a string or number, sort of like an empty email inbox.

Rather than declaring an empty array, you can also populate an array as you create it, such as in this example that creates an array containing the types of trees outside of my house:

```
var myArray = ['Pine','Birch','Maple','Oak','Elm'];
```

Arrays can contain values of any type, such as numbers and strings, mixed within the array, as in this example:

```
var anotherArray = [32.4,"A string value",4, -98, "Another string!"];
```

Arrays have a special property, *length*, that returns the length of the array. In JavaScript, the length represents the number of the final index that has been defined, which might or might not be the same as the number of elements defined. This calls for an example! This example uses the sample page created earlier in this chapter, along with the external JavaScript file created earlier.

Within the external JavaScript file, place the following code:

```
var myArray = ['Pine','Birch','Maple','Oak','Elm'];
alert(myArray.length);
```

The *external.js* JavaScript file should look like Figure 2-4.

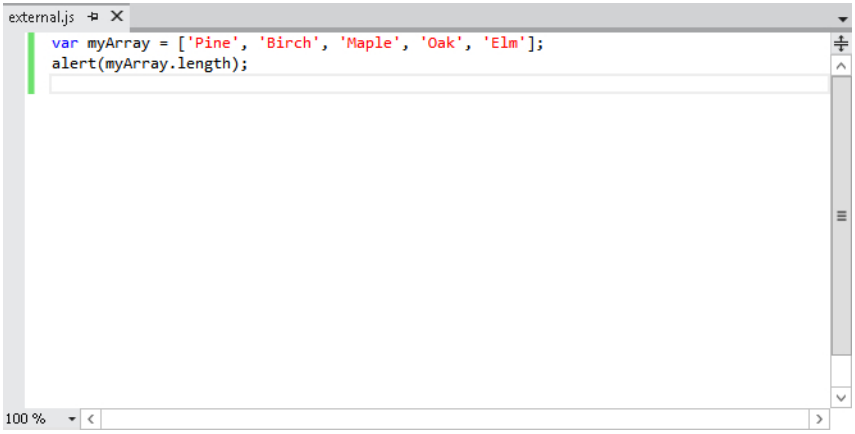


**FIGURE 2-4** The *external.js* file should contain the code shown in this example.

Now, view the page (*index.html*) in a browser. In Visual Studio, on the Debug menu, click Start Debugging, or press F5. Your browser opens and an alert appears, such as that shown in Figure 2-5, containing the length of the array.

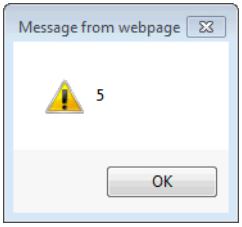


**FIGURE 2-5** The length of the *myArray* array, thanks to the *length* property.

Now, replace the code in *external.js* with this code:

```
var myArray = [];
myArray[18] = "Whatever";
alert(myArray.length);
```

Running this script yields an alert like the one shown in Figure 2-6.

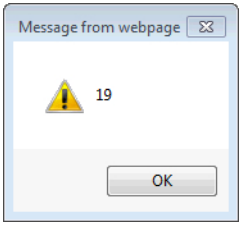


**FIGURE 2-6** The alert shows that the length of *myArray* is 19.

The alert shows a length of 19 even though only one element was defined. Therefore, be aware when using the length property that the number returned might not be the true length of all of the elements in the array.

Also, note that the example defined index number 18 of the array but the length returned was 19. Remember, array indexes begin with 0 not 1, so index 18 is really the 19th element in the array, thus the length of 19.

JavaScript defines several methods for working with arrays. Some highlights are shown in Table 2-2. They will be discussed further as they are used in the book.

**TABLE 2-2** Methods for working with arrays in JavaScript

| Method | Description |
|---|---|
| *concat* | Joins two arrays together or appends additional items to an array, creating a new array. |
| *join* | Creates a string from the values in an array. |
| *pop* | Removes and returns the last element of an array. |
| *push* | Adds or appends an element to an array. |
| *reverse* | Changes the order of the elements in the array so that the elements are backwards. If the elements were a, b, c, they will be c, b, a after the use of reverse. |
| *shift* | Removes the first element from the array and returns it. |
| *sort* | Sorts elements of an array. Note that this method assumes that elements are strings, so it won't sort numbers correctly. |
| *unshift* | Places an item or items at the beginning of an array. |

Additionally, later in this chapter you'll see how to cycle through each of the elements in an array using looping constructs in JavaScript.

## Data Types

The data types of a language are some of the basic elements or building blocks that can be used within the program. Depending on your definition of *data type*, JavaScript has either three or six data types. The main data types in JavaScript include numbers, strings, and Booleans, with three others, null, undefined, and objects, being special data types. We'll discuss most of these data types here and leave the discussion of objects for Chapter 3, "Building JavaScript Programs."

### Numbers

There is one number type in JavaScript, and it can be used to represent both floating-point and integer values (1.0 and 1, respectively). Additionally, numbers in JavaScript can hold negative values with the addition of the minus (–) operator, as in –1 or –54.23.

JavaScript has built-in functions or methods for working with numbers. Many of these are accessed through the *Math* object, which is discussed in Chapter 3. However, one handy function for working with numbers is the *isNaN()* function. The *isNaN()* function—an abbreviation for "Is Not a

Number"—is used to determine whether a value is a number. This function is helpful when validating user input to determine if a number was entered. Consider this example:

```
var userInput = 85713;
alert(isNan(userInput));
```

In this example, the output will be *false*, because 85713 is a number. It does take a bit of mental yoga to think in terms of the negative "not a number" when using this function. It would have been better for the function to be defined in the affirmative, as in "Is this a number?"

## Strings

Strings are sequences of characters (one or more) enclosed in quotation marks. The following are examples of strings:

```
"Hello world"


"B"


"Another 'quotable string'"
```

The last example bears some explanation. Strings can be quoted with either single or double quotation marks in JavaScript. When you need to include a quoted string within a string, as in the example, you should use the opposite type of quotation mark. In the example, double quotes were used to enclose the string, whereas single quotes were used to encapsulate the quoted string within.

You can also use an escape sequence or escape character to use quotation marks within the same type of quotation marks. The backslash (\) character is used for this purpose, as in this example:

```
'I\'m using single quotes within this example because they\'re common characters in
the text.'
```

Other escape sequences are shown in Table 2-3.

**TABLE 2-3** Escape sequences in JavaScript

| Escape Character | Sequence Value |
|---|---|
| \b | Backspace |
| \t | Tab |
| \n | Newline |
| \v | Vertical tab |
| \f | Form feed |
| \r | Carriage return |
| \\ | Literal backslash |

Strings have methods and properties to assist in their use. The *length* property provides the length of a string. Here's a simple example:

```
"Test".length;
```

This example returns *4*, the length of the word *Test*. The *length* property can also be used on variables, as in this example:

```
var myString = "Test String";
myString.length;  // returns 10
```

Strings include several other methods, including *toUpperCase* and *toLowerCase*, which convert a string to all uppercase or lowercase, respectively. Additionally, JavaScript allows concatenation or joining together strings and other types by using a plus sign (+). Joining a string can be as simple as this:

```
var newString = "First String, " + "Second String";
```

The preceding line of code would produce a variable containing the string *"First String, Second String"*. You'll see this type of concatenation throughout the book.

Other methods for changing strings include *charAt*, *indexOf*, *substring*, *substr*, and *split*. Some of these methods will be used throughout the remainder of this book and will be explained in depth as they are used.

## Booleans

Booleans have only two values, *true* and *false*, but you don't work with Booleans in the same way that you work with other variables. You can't create a Boolean variable, but you can set a variable to *true* or *false*. You'll typically use Boolean types within conditional statements (*if/else*) and as flags to test whether something is true or false.

An important point to remember with Booleans is that they are their own special type. When setting a variable to a Boolean, you leave the quotes off, like this:

```
var myValue = true;
```

This is wholly different than setting a variable to a string containing the word "true"—in which case, you include the quotes:

```
var myValue = "true";
```

Let's test that scenario with a code sample. The example uses the *index.html* page shown earlier, along with the external JavaScript file, *external.js*, created in this chapter. Remove any existing code from within the *external.js* file and replace it with this code:

```
var myString = "true";
var myBool = true;

alert("myString is a " + typeof(myString));
alert("myBool is a " + typeof(myBool));
```

Save *external.js*. It should look like the example in Figure 2-7.

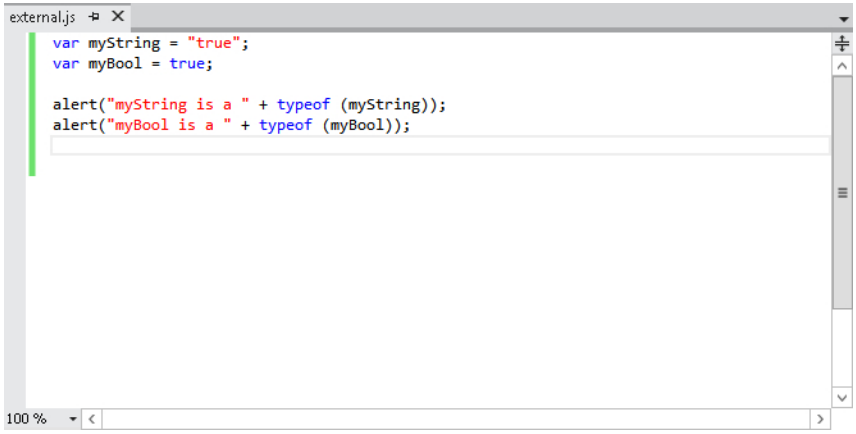


**FIGURE 2-7** The *external.js* file should contain this JavaScript.

No changes are necessary to the *index.html* file because *external.js* is already referenced in that file. Now view *index.html*. In Visual Studio, on the Debug menu, click Start Debugging, or press F5. Two alert dialogs boxes appear, like those shown in Figures 2-8 and 2-9. You can find the code for this example in *bool.html* and *bool.js* in the companion content.

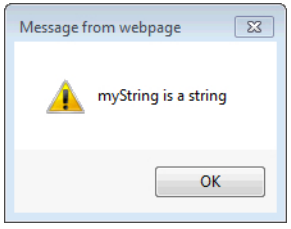


**FIGURE 2-8** The variable *myString* is a string value, even though the string is set to the word *true*.

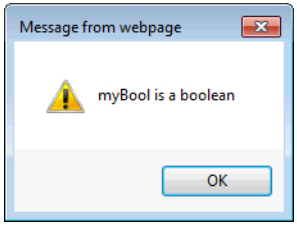


**FIGURE 2-9** The variable *myBool* is a true Boolean value.

This difference between the string *"true"* and the Boolean value *true* can be important when performing conditional tests. You'll learn more about conditional tests later in this chapter.

### Null

*Null* is a special data type in JavaScript. *Null* is nothing; it represents and evaluates to *false*, but *null* is distinctly different than empty, undefined, or Boolean. A variable can be undefined (you haven't initialized it yet, for example), or a variable can be empty (you set it to an empty string, for example). Both are different from *null*.

### Undefined

*Undefined* is a type that represents a variable or other element that doesn't contain a value or has not yet been initialized. This type can be important when trying to determine whether a variable exists or still remains to be set. You'll see examples of *undefined* throughout the book.

## Looping and Conditionals in JavaScript

In a programming language, loops enable developers to perform some action repeatedly, to execute code a certain number of times, or to iterate through a list. For example, you might have a list of links on a webpage that need to have a certain style applied to them. Rather than program each change individually, you could iterate (loop) through the links and apply the style to each link within the loop code. This section looks at the common ways for looping or performing iterations in JavaScript.

*Conditionals* are statements that look for the "truthiness" of a given expression. For example, is the number 4 greater than the number 2? If so, do something interesting. A conditional is another way of expressing that same concept programmatically. If a certain condition is met, your program will do something. There's also a condition for handling the situation when the specified condition *isn't* met, known as the *else* condition. Conditionals in this form and related conditionals are also examined in this section.

## Loops in JavaScript

*Loops* enable actions to be performed multiple times. In JavaScript (or any programming language), loops are commonly used to iterate through a collection such as an array and perform some action on each element therein.

**Note** You can find all of this code wrapped up into a single file called *loop.html* in the companion content.

The *for* loop is a primary construct for performing loop operations in JavaScript, as in this example:

```
var treeArray = ['Pine','Birch','Maple','Oak','Elm'];
for (var i = 0; i < treeArray.length; i++) {
    alert("Tree is: " + treeArray[i]);
}
```

In this example, each element in the array *treeArray* gets displayed through an *alert()* dialog. The code first instantiates a counter variable (*i*) and initializes it to 0, which (conveniently enough) is also the first index of a normal array. Next the counter variable *i* is compared to the length of the *treeArray*. Because *i* is 0 and the length of the *treeArray* is 5, the code within the braces will be executed. When the code within the braces completes, the counter variable, *i*, is incremented (thanks to the *i*++ within the *for* loop construct) and the whole process begins again—but this time, the counter variable is 1. Again, it's compared to the length of the *treeArray*, which is still 5, so the code continues. When the counter variable reaches 5, the loop will end.

A similar looping construct available in JavaScript is the *while* loop. With a *while* loop, the programmer has more flexibility because the test condition that's used can be changed by the programmer within the loop itself. Here's the previous example written using a *while* loop. Note the need to manually increment the counter within the loop:

```
var treeArray = ['Pine','Birch','Maple','Oak','Elm'];
var i = 0;
while (i < treeArray.length) {
    alert("Tree is: " + treeArray[i]);
    i++;
}
```

There is also a *foreach* statement available in JavaScript. However, as of this writing the *foreach* statement doesn't work in many older browsers. Therefore, this book will concentrate on the more common and widely supported *for* loop in most places.

Additionally, you can get a slight speed improvement by setting the array length to its own variable outside of the loop construct, in this manner:

```
var treeArray = ['Pine','Birch','Maple','Oak','Elm'];
var treeArrayLength = treeArray.length;
for (var i = 0; i < treeArrayLength; i++) {
    alert("Tree is: " + treeArray[i]);
}
```

This change means that the code doesn't need to examine the length of the *treeArray* for every iteration through the loop; instead, the length is examined once, before the loop starts.

## Conditionals in JavaScript

Conditionals are tests to determine what should happen when a given condition is met. Here's an example in plain English: "If it's snowing outside, I'll need to shovel the sidewalk." More precisely, "If the snow depth is greater than two inches, I'll need to shovel." Even more precisely, my wife will need to shovel (she likes to); however, I won't try to represent that final case programmatically. In JavaScript, this snow depth condition might be written as such:

```
if (snowDepth > 2) {
    goShovel();
}
```

The syntax for an *if* conditional calls for the test to be placed in parentheses and the code to be executed within braces, as in the preceding example. This construct is similar to the loop construct you saw in the previous section.

You can also use conditionals to define an "otherwise" scenario. Going back to the plain-English example: "If the snow depth is greater than two inches, go shovel; otherwise, watch the game." In code, you can represent this scenario with an *else* statement:

```
if (snowDepth > 2) {
    goShovel();
}
else {
    enjoyGame();
}
```

You aren't limited to evaluating single conditions; you can evaluate multiple condition scenarios, as well. For example, using the snow example one last time (I promise), I might like to go skiing if there is more than 10 inches of snow. This is represented in code using the *else if* statement, as shown here:

```
if (snowDepth > 10) {
    goSkiing();
}
else if (snowDepth > 2) {
```

```
    goShovel();
}
else {
    enjoyGame();
}
```

Note that the order of these conditionals is vital. For example, if I test whether the *snowDepth* is greater than 2 inches first, the code that checks whether the *snowDepth* is 10 inches would never be executed because snow that's 10 inches is also greater than 2 inches.

Conditions can also be combined into one set of tests, either logically together or as an either-or scenario. Here are some examples:

```
if (firstName == "Steve" || firstName == "Jakob") {
    alert("hi");
}
```

In this example, if the variable *firstName* is set to either Steve or Jakob, the code will execute. This code uses the logical OR syntax, represented by two pipe characters (||). Conditions can be joined with the logical AND syntax, represented by two ampersands (&&), as in this example:

```
if (firstName == "Steve" && lastName == "Suehring") {
    alert("hi");
}
```

In this example, if *firstName* is Steve and the *lastName* is set to Suehring, the code will execute.

## A Conditional Example

Earlier in the chapter, you learned about two types of equality operators: the double-equal sign (==) and the triple-equal sign (===). The triple-equal sign operator tests not only for value equality, but it also checks that each value is the same type. As promised, here's a more complete example. To try this example, use the sample page and external JavaScript file you created earlier in this chapter. This code can be found within the file *cond.html* in the companion content.

Within the *external.js* JavaScript file, place the following code, replacing any code already in the file:

```
var num = 42.0;
var str = "42";

if (num === str) {
    alert("num and str are the same, even the same type!");
}
else if (num == str) {
   alert("num and str are sort of the same, value-wise at least");
}
```

```
else {
    alert('num and str are different');
}
```

Save that file, and run the project in Visual Studio (press F5). An alert appears, such as the one shown in Figure 2-10. Note that you should be viewing the *index.html* page, because that's the location from which *external.js* is referenced.

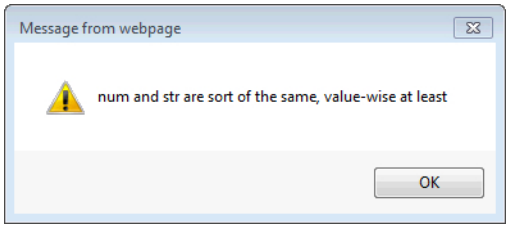


**FIGURE 2-10** Testing equality operators with a string and number.

Now remove the quotes from the *str* variable, so that it looks like this:

```
var str = 42;
```

View the page again, and you'll get an alert like the one shown in Figure 2-11.

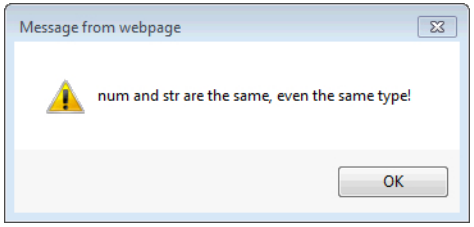


**FIGURE 2-11** Evaluating two numbers in JavaScript, using the strict equality test.

This second example illustrates a nuance of JavaScript that might not be apparent if you've programmed in another language. Notice that the *num* variable is actually a floating-point number, 42.0, whereas the *str* variable now holds an integer. As previously stated, JavaScript doesn't have separate types for integers and floating-point numbers; therefore, this test shows that 42.0 and 42 are the same.

As a final test, change the *num* variable to this:

```
var num = 42.1;
```

View the page one final time. An alert similar to the one shown in Figure 2-12 appears.

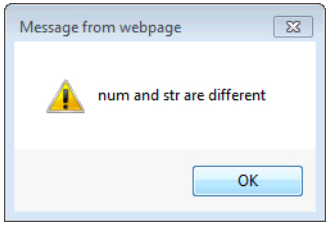


**FIGURE 2-12** Testing equality with different values.

This final example showed not only how to combine conditional tests, but also how the equality operators work in JavaScript.

## Summary

With this second chapter complete, you should now have a good grasp of the basic rudimentary syntax of JavaScript. In this chapter, you learned about comments, white space, names and reserved words, JavaScript statements and expressions, and case sensitivity. You also learned that you use variables to store data in programs and that JavaScript has several data types that include numbers, strings, Booleans, null, and undefined. These will be used throughout the book so that you get a better feel for their use in practice.

The chapter wrapped up with a look at looping, primarily through the use of *for* loops, and conditionals, mostly using *if/else* statements. The next chapter explores some of the more powerful areas of the language—namely, functions and objects. Both functions and objects are central to most modern programming languages, including JavaScript.

Chapter 3

# Building JavaScript Programs

**After completing this chapter, you will be able to**

- Understand and create functions in JavaScript
- Understand and create objects in JavaScript
- Debug JavaScript

**THE PREVIOUS TWO CHAPTERS CREATED A FOUNDATION** upon which you can build and enhance your knowledge of programming in JavaScript. It might not seem like it at first, but what you've seen so far gives you a great base from which to expand.

This chapter looks at functions and objects in JavaScript and ends with a discussion of debugging JavaScript. Debugging is key when programming in any language. Debugging includes not only the tools but also the techniques for successfully troubleshooting problems and figuring out why a program isn't working. This chapter discusses both.

## Functions

A *function* is a collection of one or more statements and expressions that can be executed from another part of the JavaScript program (or another program entirely). Earlier today, I went to an automated car wash. I drove my car into the car wash, and the car came out of the car wash clean. In this sense, I could say that the car wash is performing a function: it cleans my car.

Someone standing outside the car wash would see dirty cars going into the car wash and clean cars coming out. The dirty cars are the input to the car wash function, and clean cars are the output. What happens inside the car wash function isn't really important as long as at the end of the process the car is returned clean.

You now know all you need to know about functions, or at least you know all you need to know about my day so far. You already saw examples of functions in previous chapters. This section looks at functions in JavaScript, including how to create them and use them.

## Function Overview

A function groups one or more statements together to do something, like add two numbers or make a change to part of a webpage, or really anything else you can dream up. Functions can optionally accept input values, which are known as *arguments*, and functions can return a value, as well.

Functions begin with the keyword *function*, followed by parentheses for optional arguments and then opening and closing braces, like this function called *listMusic*:

```
function listMusic() {
    // Function code goes here
}
```

**Note** The part with the keyword *function* followed by the function name and the parentheses is called the *function declaration*.

Functions work great for reducing repeated code. Rather than creating top-down code to validate a form, you could create functions to perform basic validation based on the type of input. For example, if you know that all text fields on a form should be no more than 10 characters long, you could write a function to check the length:

```
function checkLength(textFieldValue) {
    if (textFieldValue.length > 10) {
        alert("Length too long: " + textFieldValue);
    }
}
```

Then whenever you needed to check the length of a text field, you call or run this function. Code inside of a function is not executed until the function is called, or invoked, by your JavaScript program.

## Function Arguments

Functions can accept arguments, which are essentially inputs into the function. For example, here's a function that adds two numbers:

```
function addNumbers(num1, num2) {
    var sum = num1 + num2;
}
```

This function accepts two arguments, arbitrarily called *num1* and *num2*, separated by commas. You could specify any number of arguments for a custom function, including none at all.

JavaScript doesn't complain if the function declaration indicates more arguments than you actually send. In the example shown, the *addNumbers* function expects two arguments, *num1* and *num2*. Calling that function with only one argument will not cause an error, but you'll likely get unexpected results because the function might rely on those arguments being set correctly. A similar scenario occurs if you call the function with more arguments than expected. JavaScript will silently ignore extra arguments as if they never existed. How rude!

With that said, in JavaScript the function declaration isn't the only place where you can define what arguments are expected. In other words, you don't have to declare the inputs in the function declaration as you saw in the example where two arguments (*num1* and *num2*) were included in the function declaration itself. JavaScript creates an array called *arguments* and populates it with the arguments sent into a function. For example, the previous function to add two numbers could be written as follows:

```
function addNumbers() {
    var num1 = arguments[0];
    var num2 = arguments[1];
    var sum = num1 + num2;
}
```

However, you should normally specify the arguments you're expecting to receive. By including the arguments in the function declaration, you make code maintenance much easier. Therefore, another way to accomplish this same thing is to accept an array as an argument for the function. This approach enables you to add an arbitrary amount of numbers inside the function. You can find an example of this within the *function.html* file in the companion content.

With that said, there are sometimes very good reasons for not including arguments in the function declaration. Consider the number addition example you've been working with in this section. It currently is capable of adding two, and only two, numbers. However, you can use the *arguments* array to refactor the function to add an arbitrary amount of numbers. Later in the chapter, you'll see how to do this.

### Refactoring

An important phase in a program's life cycle is *refactoring*. Refactoring means going back through the code later to improve it. This improvement can be anything from streamlining the code, adding comments, changing the code to take advantage of new features, or simply changing the spacing to make the code more readable. As programs evolve, refactoring makes the code more efficient.

Now you've seen more about functions and how to declare them, and you've also seen about function arguments—that they are the inputs to the function. Next up, you'll see how to actually use or *call* functions.

## Calling Functions

Functions are called, or invoked, by using them as you would a regular JavaScript statement and adding parentheses. For example, to call the *addNumbers* function from earlier, you use this syntax:

```
addNumbers(19, 39);
```

If the function doesn't accept arguments, the call looks like this:

```
myFunction();
```

You must include the parentheses when you call a function. You'll get unexpected results if you attempt to call a function without the parentheses. It's likely that the entire text of the function itself will be returned. That's usually not what you want.

A call to a function can use literal values (as you see with the literal numbers 19 and 39), variables, or in some cases, other functions. Here's an example in which two variables are set and then passed into the function:

```
var firstNumber = 93;
var secondNumber = 29;
addNumbers(firstNumber,secondNumber);
```

The order and type of argument sent into a function matters. It's therefore up to you, the programmer, to ensure that the arguments you send in are the correct type (numbers, in this example) and that they're sent in the correct order, as expected by the function. For example, if the function is expecting to receive two numbers so that it can add them, you can't send in two strings and expect the function to work correctly.

You might also have noticed that the variable names sent into the function are different than the variable names in the function declaration. The names don't need to match; in fact, it's probably better that they don't so that you (or whoever has to maintain your code later) won't encounter a name collision or other such difficult-to-troubleshoot code problems.

### Name Collisions

If I'm in a crowded room and someone yells, "Steve," chances are more than one person will turn around. This is a simple example of a *name collision*. A name collision in programming is the same. If you have two variables or functions with the same name, it can lead to errors in the program itself and also confusion when trying to troubleshoot or work with the code later.

In practice, there's usually a large time gap between when code is written and when a bug shows up. The more that you do to make things easier while writing the code, the less time you'll spend scratching your head later when trying to fix it.

## Return Values

The examples you've seen so far create functions, and the functions themselves do a bit of work to add numbers. But wait. What happens to those numbers inside of the function? Right now, nothing happens because you're not returning the result or doing something with the result within the function itself. JavaScript functions can return values with the help of the *return* keyword. The *return* keyword sends the expression that follows it back to the caller or invoking function, as in this example:

```
function addNumbers() {
    var num1 = arguments[0];
    var num2 = arguments[1];
    var sum = num1 + num2;
    return sum;
}
```

Note the addition of the *return sum* statement at the bottom of the function. In this example, the contents of the variable *sum* are returned to whomever called the function. That's all there is to returning a value from a function. However, with that said, there are some not-so-obvious characteristics of returning values from a function of which you should be aware. When *return* is called, execution in the function stops immediately and execution resumes outside of the function. Consider this example:

```
function addNumbers() {
    var num1 = arguments[0];
    var num2 = arguments[1];
    var sum = num1 + num2;
    return sum;
    alert("Sum is " + sum);
}
```

In the example, the *alert()* will never be executed because it occurs after the *return*.

Also, you can return only one value or expression by using *return*. The following are valid:

- *return sum;*
- *return sum * 5;*
- *return;*

The final example doesn't return a value at all; it just returns execution to the calling function. The following examples are not valid:

- *return sum, anotherValue;*
- *return*
  *myValue;*

The final example demonstrates that sometimes white space does matter. You cannot have a carriage return between the *return* and the value to be returned.

**Note** You can use an array if you need to return multiple values.

## Function Examples

In this section, you'll build example functions by using the HTML and JavaScript files shown in Chapter 1, "What Is JavaScript?," and Chapter 2, "JavaScript Programming Basics," as a foundation. If you don't have the project open, open your StartHere project in Microsoft Visual Studio. Alternatively, open the *example1.html* file in the companion content for Chapter 3. That file links to *example1.js* in the js folder.

### A Simple Example

This first example shows the *addNumbers* function you already saw in this chapter. This time you implement it yourself and do something with the result. Prepare for this example by removing any existing code from within the *external.js* file.

1. Place the following code into *external.js*. This code can be found in *example1.js* in the companion content.

```
function addNumbers(num1, num2) {
    var sum = num1 + num2;
    return sum;
}
var finalResult = addNumbers(28,51);
alert(finalResult);
```

With this code in *external.js*, the file should look like the code shown in Figure 3-1.

```
function addNumbers(num1, num2) {
    var sum = num1 + num2;
    return sum;
}
var finalResult = addNumbers(28,51);
alert(finalResult);
```

**FIGURE 3-1** The code for the first function example, deployed in the external JavaScript file in Visual Studio.

2. Additionally, for reference, the *index.html* file should look like the following code. If it doesn't, make changes as appropriate to match your *index.html* file to this one. You can find this HTML as *example1.html* in the companion content.

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
</body>
</html>
```

3. Save and run the project, or view *index.html* in a browser. An alert appears, similar to the one shown in Figure 3-2.

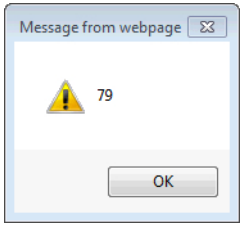


**FIGURE 3-2** The result from the first function example.

This example uses a function like the one shown early in the chapter, simply adding two numbers. Within that function, the *sum* variable is returned. Outside of the function, the code calls the function with the following line, which also places the returned value into a variable named *finalResult*:

```
var finalResult = addNumbers(28,51);
```

Next, the *finalResult* variable is sent to an alert and then to your browser. Note that this could also be written in one line, with the alert and the function call combined:

```
alert(addNumbers(28,51));
```

I chose to do it like the example to be more explicit, but either way is fine when you code your own programs!

## Refactoring *addNumbers()*

The next exercise in this section shows you how to refactor the *addNumbers()* function to accept any number of arguments and sum them up:

1. Begin with the example shown earlier that didn't include any arguments in the function declaration.

```
function addNumbers() {
    var num1 = arguments[0];
    var num2 = arguments[1];
    var sum = num1 + num2;
    return sum;
}
```

2. Place the following code into *external.js*, and remove any other code. Your beginning *external.js* should look like the one shown in Figure 3-3.

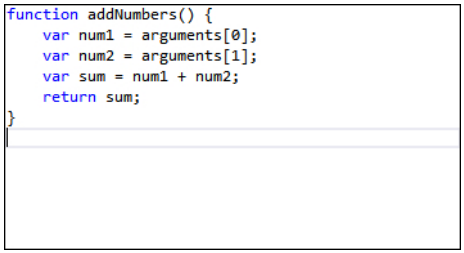


**FIGURE 3-3** The *external.js* file with the beginning code for this example.

3. Now alter the function so that it ends up like this:

```
function addNumbers() {
    var argumentsLength = arguments.length;
    var sum = 0;
    for (var i = 0; i < argumentsLength; i++) {
        sum = sum + arguments[i];
    }
    return sum;
}
```

This revised function examines the *arguments* array and sets the variable *argumentsLength* to the length of that array. The *sum* variable is declared and set to 0. This is important because setting that variable to 0 (zero) automatically casts it, or creates it, as a number, meaning that you can perform math with it, which happens inside of the *for* loop.

A *for* loop is then created, and each of the elements in the *arguments* array is added to the *sum* variable. Note that this assignment can also be written in a shortcut manner, like so:

```
sum += arguments[i];
```

However, as I did earlier, I chose to be more explicit rather than introduce new syntax (and possibly new confusion).

This function is invoked like so:

```
var finalResult = addNumbers(3, 5, 1);
```

As before, an alert is created with *finalResult*, too. The final code looks like this:

```
function addNumbers() {
    var argumentsLength = arguments.length;
    var sum = 0;
    for (var i = 0; i < argumentsLength; i++) {
        sum = sum + arguments[i];
    }
    return sum;
}

var finalResult = addNumbers(3, 5, 1);
alert(finalResult);
```

With that code in *external.js*, it should look like Figure 3-4.

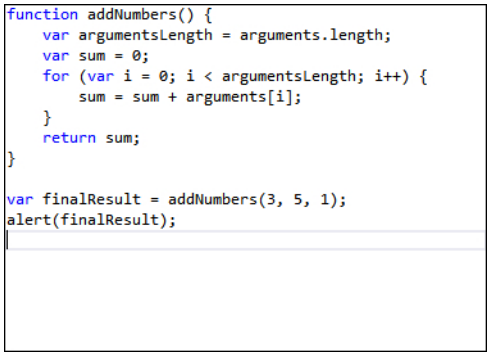


**FIGURE 3-4** The full code for the final function example.

4. Execute this code, or view it in a browser. You should see an alert like the one in Figure 3-5.

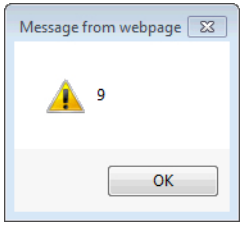


**FIGURE 3-5** The alert produced by the final function example in this section.

For more experimentation, you can alter the function call to send in any number of arguments, including no arguments at all. By instantiating the *sum* variable to 0, even if there are no arguments, you won't create an error condition; there just won't be anything to add to 0 because the *arguments* array will be empty.

With that example done, it's time to move on to objects in JavaScript. You'll see functions throughout the remainder of the book and also learn about their kin in the object-oriented programming world, where they're called *methods* instead of functions. Prior to looking at objects, it'll be helpful to look again at how variables are scoped in JavaScript.

## Scoping Revisited

You'll recall from Chapter 2 that JavaScript variables are globally scoped within JavaScript. This means that a variable declared at the top of your program is available to everything else within that program; all the other parts of your program can see and change the variable's contents.

This variable declaration rule, however, can be confusing to a new programmer because the rule doesn't apply equally—it depends on context. Variables are visible within the current function's scope and to any functions declared within the current function. This might not make sense to you quite yet because we haven't discussed functions. (That's the topic of the next chapter.) Here's an example to help explain. I include the <script> declaration here to show that this is the beginning of the script, although I won't normally include this declaration for other examples. You can find this entire example in *scope.html* in the Chapter 3 companion content.

```
<script type="text/javascript">
var globalNumber = 4;
alert("Global Number is: " + globalNumber);

function meToo() {
    var meTooNumber = 18;
    alert("Inside of function, global number is: " + globalNumber);
    alert("Inside of function, meTooNumber is: " + meTooNumber);
}
```

```
meToo();
alert("Outside of function, global number is: " + globalNumber);
alert("Outside of function, meTooNumber is: " + meTooNumber);
</script>
```

When this example is executed, three alert boxes are displayed, as shown in Figures 3-6 through 3-8. Note that your alert dialog boxes might look slightly different depending on the browser type and version you are using.


**FIGURE 3-6** A globally scoped variable, outside of a function.

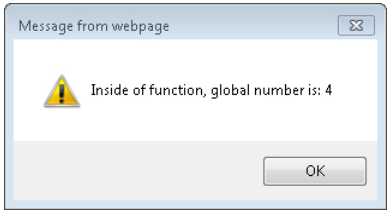


**FIGURE 3-7** A globally scoped variable, inside of an inner function.

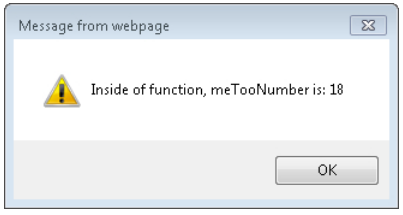


**FIGURE 3-8** A locally scoped variable inside of a function.

Looking closely at the code example, you'll notice that there are four calls to the *alert()* function but only three alerts actually appear. This is because the final call to the *alert()* function attempts to display a variable that doesn't exist in the global or main program scope. The variable, *meTooNumber*, was declared inside of the function and therefore cannot be accessed outside of the function in which it was declared.

# Objects in JavaScript

Objects are part of the programming paradigm known as *object-oriented programming*, sometimes shortened to OOP. JavaScript is not a full object-oriented language, but it behaves in a largely object-like manner. This enables the programmer to take advantage of some (but not all) of the good things that come with object-orientation.

As discussed in Chapter 2, in JavaScript everything except numbers, strings, Booleans, undefined, and null are objects. Arrays are objects in JavaScript and, well, objects are objects in JavaScript, too. Before diving head-first into objects, I'll back up and explain them with a real-world example.

Objects essentially are things. Looking outside of computer programming for an example, a guitar is an object. As I'm writing this chapter, I have a guitar next to me. The guitar has certain characteristics, like its color, the number of strings, whether it's an electric or acoustic guitar, and so on. The guitar next to me today is a red, six-string, electric guitar. The guitar can, along with my assistance, do things like strum a chord or play an individual string. It can therefore be said that this guitar object has certain properties and can perform certain actions.

As you'll see, objects in JavaScript can perform actions, known as *methods*, and also have characteristics, known as *properties*.

## What Does an Object Look Like?

In JavaScript, an object can be created in a number of ways. A primary way to create an object is by using two curly braces, like so:

```
var myObject = {};
```

This is called an *object literal*. Objects can also be created with the *new* keyword, like so:

```
var myObject = new Object;
```

In practice, I've seen more of the object literal notation in JavaScript programs, and that's the most common form you'll see in this book, as well.

## Properties

In the object-oriented world of computer programming, the color, number of strings, and type of guitar are called its *properties*. With JavaScript, properties can be created in two similar ways: by using dot notation or by adding properties when the object is created. Here's an example:

```
var guitar = {};   //  Create an object
guitar.color= "red";
guitar.strings = 6;
guitar.type = "electric";
```

You can also create properties in the declaration of the object itself, like so:

```
var guitar = {
    "color": "red",
    "strings": 6,
    "type": "electric"
};
```

### Dot Notation

Dot notation is the name used in programming when items are separated by dots. In the examples shown so far, the name of the object (guitar) is separated from the properties by a period or dot. So basically, dot notation is a fancy name for separating things by periods.

Just as when creating variables, you should always try to use valid, nonreserved words for properties. However, when creating properties, sometimes it just makes sense to use a reserved word in the context of that object. To do so, you create the property name in quotes, which is how you've seen objects created throughout this chapter.

Finally, you can also nest objects, as shown in this example (which you can find as *objectproperty.js* in the companion content):

```
var guitar = {
    "color": "red",
    "strings": {
        "number": 6,
        "smallGauge": 9,
        "largeGauge": 42
    },
    "type": "electric"
};
```

In this example, a nested object is created and contains properties about the strings of the guitar, including the number of strings and their gauge.

Properties are accessed by using dot notation or by including the property name in brackets. Here's an example:

```
guitar.color;   //red
guitar.strings.smallGauge;  // 9
guitar["color"];  //red
guitar["strings"]["smallGauge"]  //9
```

It's usually preferable from a readability standpoint to use dot notation when possible, but that's really a matter of developer preference and your coding standard, and I'll show you a combination of both ways of accessing properties throughout the book to make sure you're familiar with seeing either syntax.

## Methods

In addition to having properties, many objects do things. With my assistance, the guitar plays individual notes on a given string as well as chords with multiple strings. In JavaScript, you can create a function, store it as a property, and then call or invoke that function. These functions are essentially just like the functions you learned about earlier in the chapter, except they're called *methods* when used with objects. Methods are declared, they can accept arguments just like functions, and they can return values. For the purposes of this discussion, methods are just like functions that are added or attached to an object.

Building on the guitar object, here's a method to play a string:

```
var guitar = {
    "color": "red",
    "strings": 6,
    "type": "electric",
    "playString": function (stringName) {
        var playIt = "I played the " + stringName + " string";
        return playIt;
    } //end function playString
};
```

In this example, which you can find in *objectmethod.js* in the companion content, a method called *playString* is declared and accepts one argument. That variable, called *stringName*, is used to indicate the string to play on the guitar and is then used to create a message indicating that the guitar string was played.

This same method can be created later and added to the guitar object by using the dot notation discussed previously. Assuming that the *guitar* object has been created already, adding the method looks like this:

```
guitar.playString = function(stringName) {
        var playIt = "I played the " + stringName + " string";
        return playIt;
    } //end function playString
```

## Using the *this* Keyword

One of the more powerful aspects of object-oriented programming is the use of the *this* keyword. The *this* keyword provides self-referential access to an object's properties and methods. You can use *this* to perform advanced operations within an object's methods. For example, using *this*, you can get and set an object's properties in a consistent manner (sometimes known as *getters* and *setters*). To demonstrate, you'll work through an example. For this example, begin with the *StartHere* project in Visual Studio. You can also find this example as *this.html* in the companion content. That file links to *this.js* in the js folder.

1. Open *external.js* and remove any code in that file. Place the following code within *external.js*:

```
var guitar = {
    "color": "red",
    "strings": 6,
    "type": "electric",
    "playString": function (stringName) {
        var playIt = "I played the " + stringName + " string";
        return playIt;
    },
    "getColor": function() {
        return this.color;
    },
    "setColor": function(colorName) {
        this.color = colorName;
    }
};

alert(guitar.getColor());
guitar.setColor("blue");
alert(guitar.getColor());
```

2. Save *external.js*. It should look like Figure 3-9.

```
var guitar = {
    "color": "red",
    "strings": 6,
    "type": "electric",
    "playString": function (stringName) {
        var playIt = "I played the " + stringName + " str
        return playIt;
    },
    "getColor": function () {
        return this.color;
    },
    "setColor": function (colorName) {
        this.color = colorName;
    }
};

alert(guitar.getColor());
guitar.setColor("blue");
alert(guitar.getColor());
```

**FIGURE 3-9** *external.js* with object-related code.

3. Run the project to view the *index.html* file in a web browser. You'll first see an alert with the initial *color* property of the *guitar* object, as shown in Figure 3-10.

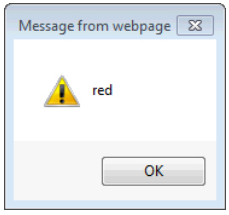


**FIGURE 3-10** The *color* property of the guitar object, accessed through a getter method.

Clicking OK to dismiss the alert reveals another alert, which indicates that the color has now been changed to blue, thanks to the setter method *setColor*. This is illustrated in Figure 3-11.

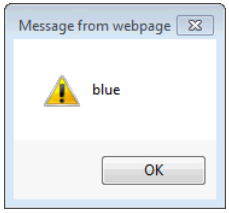


**FIGURE 3-11** The *color* property has been changed thanks to a setter method.

The two additions to the *guitar* object are the getter and setter methods, *getColor()* and *setColor()*, respectively. The *getColor()* method looks like this:

```
"getColor": function() {
        return this.color;
 }
```

This method merely returns the *color* property as it is currently set. The *setColor()* method looks like this:

```
"setColor": function(colorName) {
    this.color = colorName;
 }
```

The *setColor()* method sets the color as it is passed into the method call.

## Object Enumeration

You can traverse or enumerate the properties in an object by using a *for..in* loop, similar to the *for* loop syntax that you learned in Chapter 2. However, unlike the array examples you saw in that chapter, when enumerating an object with *for..in*, all properties and methods will be enumerated. Therefore, you need to employ the *typeof* function to make sure that the item currently being enumerated is actually a property. Here are two exercises to help illustrate the point. This exercise's HTML and code can be found in *typeof.html* in the companion content for Chapter 3 and *typeof.js* in the Chapter 3 js folder.

1. Clear any existing code out of *external.js* and place the following JavaScript in the file:

```
var telephone = {
    "numLines": 4,
    "usedLines": 0,
    "isLineAvail": function() {
        if (this.usedLines < this.numLines) {
            return true;
        } else {
            return false;
        }
    },
    "getLine": function() {
        if (this.isLineAvail) {
            this.usedLines++;
            return true;
        } else {
            return false;
        }
    },
```

```
    "startCall": function (line,dialNum) {
        if (this.getLine) {
            return "Called " + dialNum + " on line " + line;
        } else {
            return "No lines available at this time.";
        }
    },
    "endCall": function (line) {
        this.usedLines--;
    }
};



for (var propt in telephone) {
    document.writeln(propt);
}
```

Viewing the *index.html* file that includes *external.js* in a browser yields a page like the one shown in Figure 3-12. Note that depending on your browser, your output might appear on a single line.

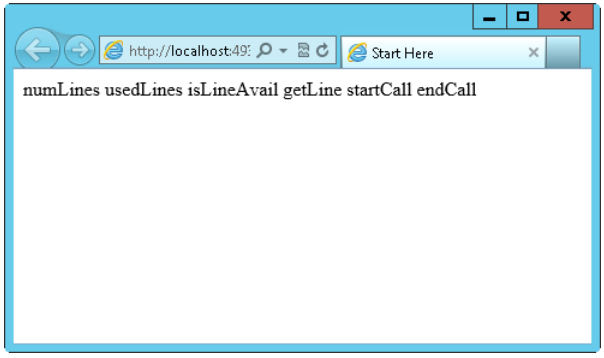


**FIGURE 3-12** Enumerating the properties (and methods) of an object in JavaScript.

2. Use of the *typeof* function is necessary to determine if a given value is truly a property or is a function. Therefore, change the *for..in* loop to include a call to the *typeof* function, like so:

```
for (var propt in telephone) {
    if (typeof(telephone[propt]) !== "function") {
        document.write(propt + "<br />");
    }
}
```

In this example, when *typeof* returns "*function*", you know that what's being enumerated is not a method and the example then writes it to the output, as shown in Figure 3-13.

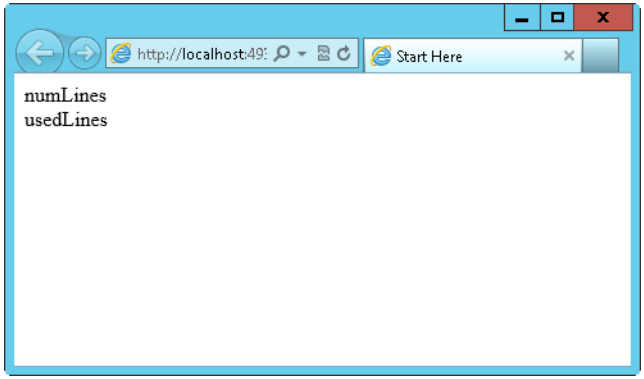


**FIGURE 3-13** Using *typeof* to eliminate methods from object enumeration.

## Classes

A powerful feature of object-oriented programming is a *class*, or the ability to create a type of object and then extend that object type to other types. For example, because guitars share many of the same characteristics, rather than create a guitar object for each guitar that I own, I can create a class by which I can have a generic object and then change the characteristics or properties for each of the objects. For example, all of my guitars are, well, guitars. This means that generically they're musical instruments. I can then create a generic musical instrument object.

Unfortunately, JavaScript doesn't have the concept of classes. However, it does provide for the creation of *pseudo-classes*, or objects that act like classes in certain key ways. This is accomplished by using a programming pattern. The simplest to explain and use is called a *constructor pattern*, so that's what I'll use here.

Creation of a pseudo-class involves making a function that, when called, returns an instance of an object. Behind the scenes, this object has a link to the function's prototype. This sounds confusing but really isn't so bad when you see it in action. So, here's an exercise for creating a pseudo-class for a *Person* object. This code can be found in the *person.html* and *js/person.js* files in the Chapter 3 companion content.

1. Clear any code out of *external.js* and create a *Person* constructor function, like so (the final code for this exercise can be found in *person.js* in the companion content for Chapter 2):

```
var Person = function(username,email,twitter) {
    this.username = username;
    this.email = email;
    this.twitter = twitter;
    this.listDetails = function() {
        document.write("Username: " + this.username + "<br />");
        document.write("E-mail: " + this.email + "<br />");
        document.write("Twitter ID: " + this.twitter + "<br />");
    }
}
```

2. Within *external.js*, instantiate a *Person* object by using the *new* keyword, in this manner:

```
var myPerson = new Person("steve","suehring@braingia.com","@stevesuehring");
```

3. Finally, call the method on the newly created *myPerson* object by placing the following code in *external.js*:

```
myPerson.listDetails();
```

The final code in *external.js* should look like this:

```
var Person = function(username,email,twitter) {
    this.username = username;
    this.email = email;
    this.twitter = twitter;
    this.listDetails = function() {
        document.write("Username: " + this.username + "<br />");
        document.write("E-mail: " + this.email + "<br />");
        document.write("Twitter ID: " + this.twitter + "<br />");
    }
}


var myPerson = new Person("steve","suehring@braingia.com","@stevesuehring");
myPerson.listDetails();
```

4. Run the project in Visual Studio, or view *index.html*. You should see results like those in Figure 3-14.

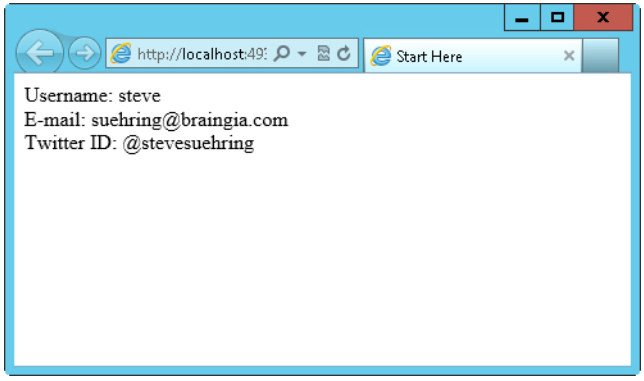


**FIGURE 3-14** Creating a *Person* object in JavaScript, and viewing the page in Internet Explorer.

Here's a slightly more detailed example to show the power of pseudo-classes. Though there's technically nothing here that hasn't already been introduced, the example does get a bit complex. You can find this example as *classes.js* in the companion content for Chapter 3.

This example begins with a slight variation of the *Person* object just introduced. The change to the *Person* object removes the Twitter id. However, rather than instantiate a *myPerson* object, an array of objects is used to set up several *Person* objects in a loop:

```
var Person = function (username, email) {
    this.username = username;
    this.email = email;
    this.listDetails = function () {
        document.write("Username: " + this.username + "<br />");
        document.write("E-mail: " + this.email + "<br />");
    }
}

// Create arrays of data
var usernames = ['steve', 'rebecca', 'jakob', 'owen'];
var emails = ['suehring@braingia.com', 'rebecca@braingia.com',
               'jake@braingia.com', 'owen@braingia.com'];

// Get length of usernames array
var usernamesLength = usernames.length;

// Create an array to hold Person objects
var myPeople = new Array();

// Iterate through all of the usernames and create Person objects
for (var i = 0; i < usernamesLength; i++) {
    myPeople[i] = new Person(usernames[i], emails[i]);
}

// Get length of the myPeople array
var numPeople = myPeople.length;

// Iterate through all of the Person objects in myPeople and
//    show their details.
for (var j = 0; j < numPeople; j++) {
    myPeople[j].listDetails();
}
```

This example first creates arrays of base data, called *usernames* and *emails*. Later in the book and while programming JavaScript in the real world (as opposed to the fake world you're living in while reading this book), you'll frequently have arrays of data such as form fields or returned data from a database, which will look similar to this array.

Next, the length of that array is gathered and set into a variable called *usernamesLength*. That variable is used within a *for* loop to walk through the user names and instantiate new *Person* objects, sending in the current user name and email address as part of that instantiation. Each of these newly created *Person* objects is sent into the *myPeople* array.

The length of the *myPeople* array is then placed into the *numPeople* variable, which is in turn used in a *for* loop. The final *for* loop in this example calls the *listDetails()* method of each *Person* object. The final result is that you have created and used several *Person* objects with a minimum amount of repeated code. You can run this code by placing it in *external.js* and running the project again. (Be sure to clear out any code from previous examples.) Figure 3-15 shows the results of this code.

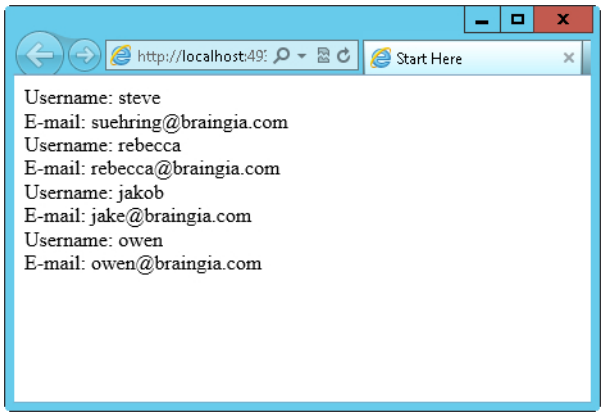


**FIGURE 3-15** Working with multiple objects using arrays and looping.

### Objects or Arrays?

You might notice some similarity between objects and arrays. Though this oversimplifies it a bit, I found it helpful when I first thought of it this way, so I'll pass it along to you: use objects when you need to include named parameters (properties); use arrays when a simple numeric index will suffice. There is more to it than that (arrays are actually objects), but it might help you frame the difference.

The preceding code might have stretched your understanding of JavaScript, but it provides a full example of a working, real-world program. You can use that program as a reference as you learn JavaScript. With that introduction to objects now complete, it's time to look at debugging in JavaScript.

# Debugging JavaScript

Troubleshooting programs is a core skill for any developer. This is known as *debugging*, and the debugging process begins when your code fails to work as you expected. Errors can present themselves for any number of reasons. Things like a missed semi-colon or closing bracket, a variable not created or named correctly, trying to use a function incorrectly, having the logic or flow incorrect, or hitting a bug or difference in how a browser interprets that JavaScript are just a few of the many bugs you'll encounter when programming in JavaScript.

The basic process for debugging JavaScript has changed over the past several years. Prior to just a few years ago, the best way to debug JavaScript was using the Firebug add-on for the Firefox web browser. In many ways, this is still true, and I find myself using Firebug and Firefox as a primary debugging tool. However, Microsoft has improved the debugging capabilities of both Visual Studio and Internet Explorer in recent versions, making it a viable method for a new JavaScript programmer. Finally, Google's Chrome browser has a powerful inspector similar to Firebug.

This section looks at debugging JavaScript with Internet Explorer. However, I strongly recommend seeking out the Firebug add-on as well as Chrome after you try out Internet Explorer for debugging. You should be performing cross-browser testing anyway, so using those browsers shouldn't present an undue burden.

## Debugging as a Process

The overall process of debugging is sometimes mysterious for new programmers (and even some experienced programmers). The process of debugging or troubleshooting is a matter of eliminating possibilities and testing solutions.

The most important phase of troubleshooting is the one in which you eliminate possibilities. During this time, the troubleshooter needs to remove as many factors as possible from the problem. For example, when debugging a web application, you encounter several "moving parts," including the server itself, the browser, the HTML, the CSS, and the JavaScript (and the network and any data from the server, and so on). So when you're trying to troubleshoot, you need to reduce and eliminate as many of those moving parts as possible.

One way to focus on the issue with JavaScript is to simply add an *alert()* at the top of the JavaScript, or even within the HTML, prior to other JavaScript in the <head> portion of the page. I find myself using *alert("I'm in this section");* (along with the name of the section) in various places just to see how far the JavaScript gets before it stops executing. This is an extremely simple yet effective way to troubleshoot picky JavaScript problems.

Once you narrow down the areas in which there's a problem, you can begin testing solutions. There's no magic potion for finding solutions when you find the trouble spot. However, using tools like Firebug and the tool to which I'm about to introduce you, you can work through the issues quicker.

## Debugging in Internet Explorer

Beginning with Internet Explorer 9, a tool called *F12 developer tools* is available. This set of tools is accessed by pressing F12 from within the browser. Figure 3-16 shows the F12 developer tools when viewed within the page from the previous section.

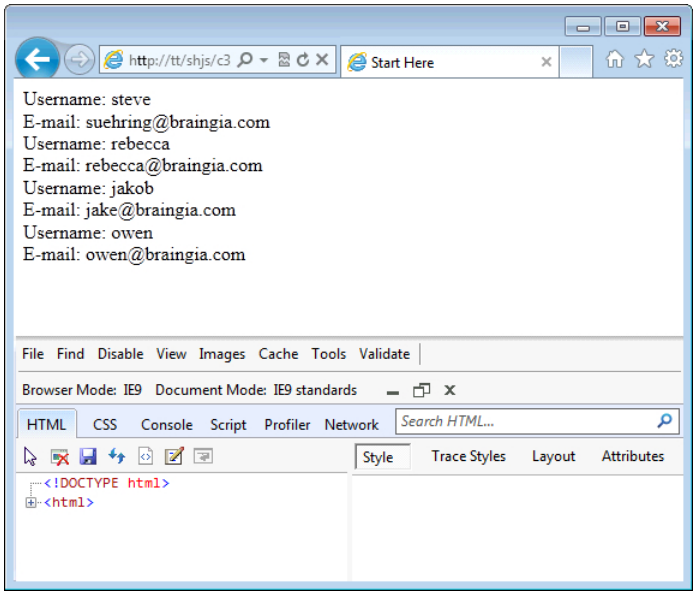


**FIGURE 3-16** The F12 developer tools in Internet Explorer.

Within the F12 developer tools console, you can examine the HTML, CSS, console, script, and other information via tabs within the interface. F12 developer tools are very powerful. This section gives you an overview so that you become familiar with them and their use for JavaScript. See *http://msdn.microsoft.com/library/IE/gg589507.aspx* for more information on F12 developer tools in Internet Explorer.

To demonstrate F12 developer tools, you'll use the *external.js* file you've used throughout the book so far to create a simple JavaScript program that contains an error. This code can be found in *debug.html* and *debug.js* in the companion content.

1. Clear any existing code out of *external.js* and place the following code in the file:

```
var myVar = 0;
if (myVar < 3) {
    alert("hello);
}
```

2. Save the file, and view the *index.html* file that calls *external.js* from Internet Explorer. Be sure to omit the closing double quote (") because you're trying to create an error.

3. If the F12 developer tools aren't open, open them by pressing F12. You might need to reload the page by pressing Ctrl+R.

   When looking at the code, you should receive an alert because *myVar* was defined and set to 0 and the condition looks for *myVar* being less than 3, which it is. Therefore, something went wrong. This is one of the more difficult aspects of working with JavaScript. When one error is encountered, in any script, no other JavaScript is executed. So anything below the error line won't be run.

   With this in mind, you need to have patience when troubleshooting complex JavaScript. Start with the first error, which in this case should be the only error. If you're viewing this page and have F12 developer tools open, your screen should look like the one shown in Figure 3-17.

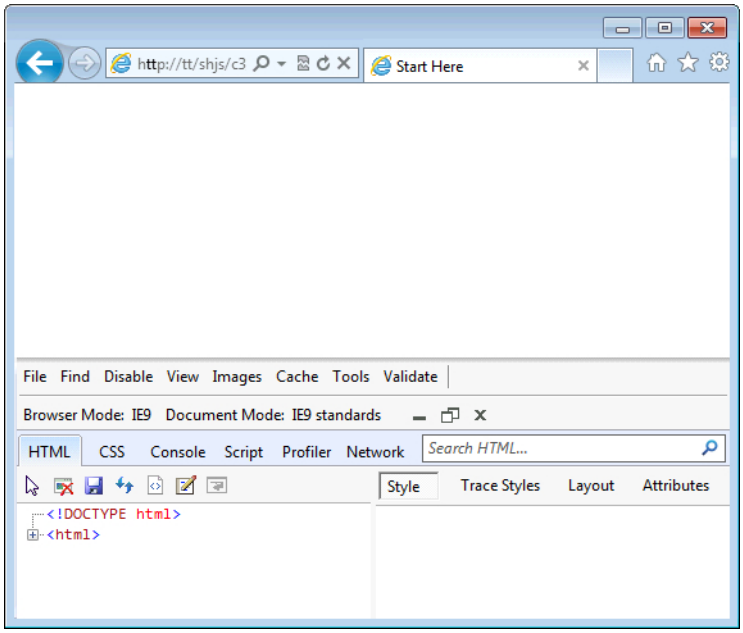


**FIGURE 3-17** Viewing a page with an error stops JavaScript processing.

4. Click the Console tab.

   Again, depending on your usage of F12 developer tools, you might need to reload the page by pressing Ctrl+R to activate this console. In the console, you'll see an error like the one shown in Figure 3-18.

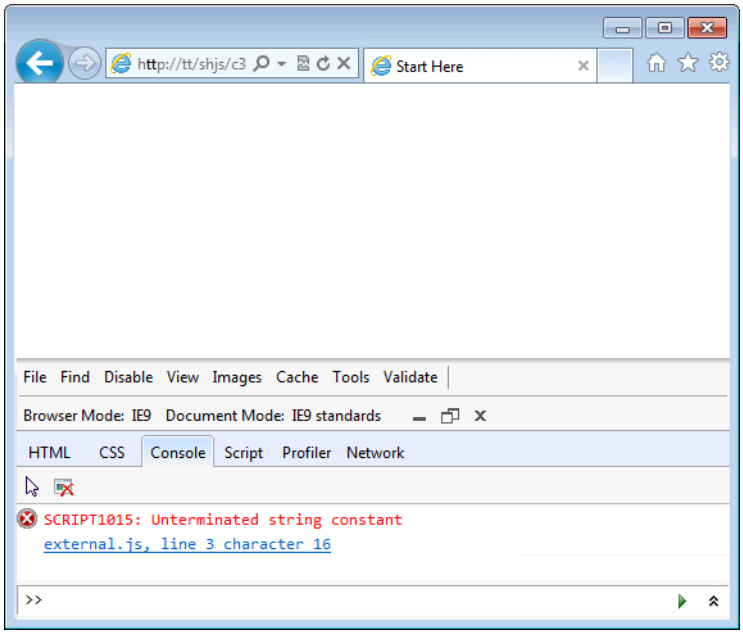


**FIGURE 3-18** An error in our JavaScript, viewed on the F12 developer tools Console tab.

As you can see from this message, "Unterminated string constant," there must be an error. Internet Explorer does its best to determine the line number and sometimes the position of the error; however, depending on how the HTML, CSS, and JavaScript interact, this line number and position might not always be accurate. Luckily, in our case it's correct. (Your line numbers might be slightly different than mine.) Looking at line 3 in my code, I see an *alert("hello);* and the problem is obvious: a missing double quote.

5. Add the missing double quote to the code. It should now look like this:

```
var myVar = 0;
if (myVar < 3) {
    alert("hello");
}
```

If you're using Visual Studio, you need to stop the project or stop debugging and then make the change.

6. Reload the page or rerun the project to show the alert, as you expected to see it (shown in Figure 3-19).


**FIGURE 3-19** The *alert()* after correcting the JavaScript error.

Congratulations, you've fixed your first JavaScript bug by using a debug console. From here, I invite you to browse various webpages across the Internet and go into F12 developer tools while browsing. You'll get a sense of the amount of complexity that goes into building many webpages. You might even be surprised by how many errors you encounter on webpages when using the console. Familiarize yourself with the various tabs, exploring the Network, Console, and other tabs according to your level of curiosity.

## Summary

This chapter looked in depth at some programming concepts in JavaScript. The chapter began with a look at functions, including how to create them and work with them to build more complex JavaScript programs. Functions are a central concept in JavaScript programming. Within the objects section, you learned how to create objects and add properties, values, methods, and pseudo-classes. You learned how to enumerate the properties in an object and how to create several objects by using an array of data. Finally, this chapter wrapped up with a look at how to debug JavaScript. Internet Explorer now includes F12 developer tools, which are akin to Firebug for Firefox. The chapter showed an example of using F12 developer tools, but I also recommend using Firebug, as well.

The next chapter will look at JavaScript and how it interacts with the web browser through the Document Object Model. What you've learned so far has prepared you with all the JavaScript you need to create JavaScript programs. However, you next need to learn how to put that JavaScript programming knowledge into practice in its natural habitat: the web browser.

Chapter 4

# JavaScript in a Web Browser

**After completing this chapter, you will be able to**

- Install and use jQuery
- Install and use jQuery UI
- Work with the Browser Object Model
- Work with the Document Object Model

**THROUGHOUT THE BOOK SO FAR,** you've used a web browser to view the results of your JavaScript programming efforts. You've also gained a sense of the central role that the web browser plays in JavaScript and for JavaScript programming. And with that, you've only scratched the surface of how JavaScript interacts with the browser.

The Document Object Model, or DOM, is the interface through which JavaScript works with webpages. Just as JavaScript is defined by the ECMA-262 specification, the DOM is defined by the World Wide Web Consortium (W3C). And just as different browsers implement JavaScript in slightly different ways, so too do browsers implement the DOM in different ways. This means that you need to program in one way for one browser and another way for another browser.

***See Also*** *The W3C site at* http://www.w3.org/DOM *has much more information on the DOM.*

Browsers are getting better in their interpretation of both JavaScript and the DOM, where "better" means that browsers are moving toward each other to standardize how they implement both JavaScript and the DOM. The result is less work for the JavaScript programmer and less one-off solutions to make a webpage work in a certain browser. The DOM can be used for much more than JavaScript programming, but because this is a JavaScript book, the focus will remain on its use and relation to JavaScript.

The DOM is actually one item of the overall Browser Object Model (BOM). The BOM enables you to do things like work with events, detect information about the visitor and the visitor's browser, create and resize windows, and more.

This chapter looks at the BOM and the DOM and how to use JavaScript with both. Working with the BOM and DOM can be difficult because of the sheer number of subtle differences between browsers. Luckily, there are collections of code functions, known as libraries, available to assist in this effort and remove much of the effort of programming around browser differences. This chapter begins with a look at one of the more popular JavaScript code libraries, jQuery, and its companion for advanced effects, jQuery UI.

## JavaScript Libraries

Programming libraries and frameworks assist developers by providing tools and helpers for common and time-consuming tasks. One of the greatest additions to the JavaScript programmer's toolbox over the past several years is the popularity and strength of libraries and frameworks for JavaScript. Prior to the various libraries becoming available, the JavaScript programmer needed to write extensive code to perform things like changing content, validating web forms, and other behavioral aspects of websites. Today's JavaScript programmers take advantage of libraries to make their job easier.

One of the most popular JavaScript libraries is called jQuery. jQuery, along with its complementary project, jQuery UI, are essential additions for JavaScript development. jQuery's popularity is confirmed by its inclusion in some versions of Microsoft Visual Studio.

This book features jQuery, and you'll see it used heavily throughout the book. Where appropriate, I'll also point out the traditional JavaScript method for accomplishing the same task. In the upcoming sections, you'll download jQuery and jQuery UI and add them to your StartHere project from previous chapters.

## Getting jQuery

jQuery and jQuery UI can be downloaded and used locally in your project, or they can be hosted elsewhere and accessed via a Content Delivery Network (CDN). For production websites, I strongly recommend downloading jQuery for use in your local project or site; see the "Hosted vs. Local for Libraries" sidebar for reasons why you should keep jQuery local for live websites. However, during development and for this book, it's acceptable to use the CDN-hosted version of jQuery. jQuery UI is a bit different insofar as there are also Cascading Style Sheets (CSS) themes that come with it. This makes it a bit more difficult to use in a CDN-based solution. With that in mind, I'll show how to use jQuery UI locally.

### Hosted vs. Local for Libraries

Time is of the essence when serving webpages, as is reliability. Popular libraries and other resources, such as fonts, are available both as downloads and through CDNs. Using a locally hosted copy of resources is the best way to ensure reliability while also increasing speed for your webpages. This is because the locally hosted copy of the resource, such as jQuery, is wholly under your control. Your web server is the one serving the file, and you can ensure that the local file is available at all times that your site is available.

When a file is hosted locally, you save a DNS query and sometimes a new TCP connection compared to when that file is hosted on a CDN. On the CDN, the visitor's web browser or device needs to make an additional DNS query and open up a new TCP connection to that CDN server to obtain the resource. This can be a time-consuming process, speaking in relative terms to the milliseconds that it should take for resources to load.

The counter argument is that the CDN is more reliable than your web server, and this is generally true. But if your web server is down, your web application is down and thus won't need the resource anyway. So it doesn't matter how reliable the CDN is! A CDN also provides the most current version of a library, thus alleviating you from having to update your local copy. Finally, a CDN takes a bit of load away from your server because the file is served from someone else's server.

## Using a Local Copy of jQuery

This section looks at how to obtain jQuery for the locally hosted option. If you use a CDN-hosted copy of the jQuery library to follow the examples in this book, you can safely skip this and jump to the "Using a CDN-Hosted jQuery Library" section in this chapter.

You can obtain jQuery from *http://jquery.com*. Once you are there, download the production version and save it to an appropriate location on your computer—typically, your Downloads folder. You'll copy it into the Visual Studio project later; for now, just save it somewhere that you can remember (or find) for the upcoming exercise.

**Note** You can find the code for this exercise, along with version 1.7.1 of jQuery in the companion content for Chapter 4. However, I recommend using the latest version of jQuery available when you work on this exercise. The HTML for this exercise is called *jquery.html* in the companion content for Chapter 4. Also, note that the Development version of jQuery (which you'll see on the jQuery website) is used if you want to do development on jQuery itself.

To add jQuery to your project and to the specific page in which it will be used, follow these steps:

1. Open your StartHere project in Visual Studio, if it isn't open already, by clicking File and then clicking Open Project.

2. Within the StartHere project, right-click within the Solution Explorer pane (usually on the right side of the screen). In the context menu that opens, click Add and then click Existing Item.

   You'll be presented with the Add Existing Item dialog.

3. Within the Add Existing Item dialog, navigate to your jQuery file (for example, mine's called *jquery-1.7.1.min.js*) and click Add. Note that you might need to select Script or All Files from the file type drop-down in order to see the jQuery file. This dialog is shown in Figure 4-1.

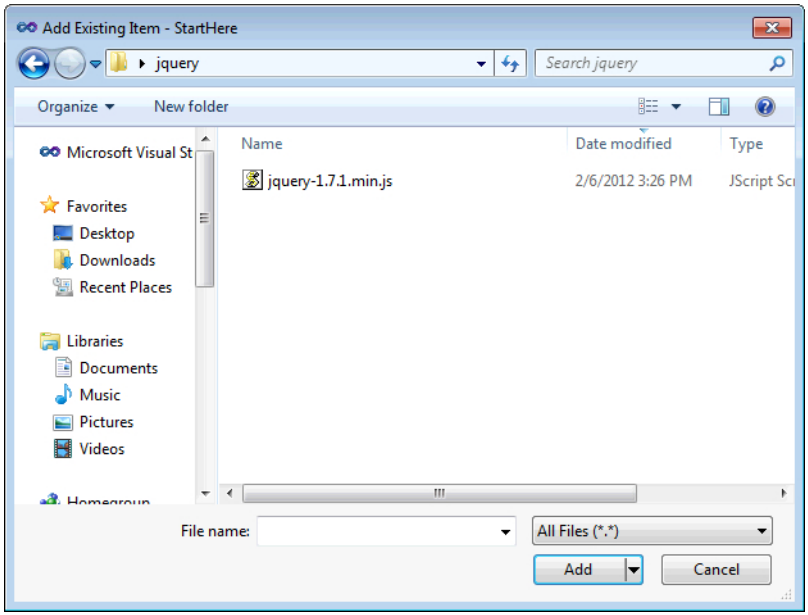


**FIGURE 4-1** Adding jQuery to a project by right-clicking within the Solution Explorer pane.

When you do so, Visual Studio copies the jQuery file into your project and Solution Explorer is updated, like the one shown in Figure 4-2.

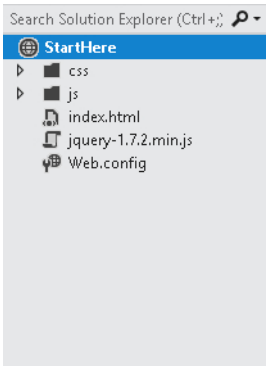


**FIGURE 4-2** jQuery now shows up within Solution Explorer after adding it to the project.

4. With jQuery added to your project, the next step is to add it to the page in which it will be used. You accomplish this using the <script> tag with the addition of the *src* attribute, like so:

```
<script type="text/javascript" src="jquery-1.7.1.min.js"></script>
```

Place this <script> tag within the <head> section of the page. For example, adding jQuery to the page created earlier in the chapter looks like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <script type="text/javascript" src="jquery-1.7.1.min.js"></script>
</head>
<body>
    <script type="text/javascript">
        document.write("<h1>Start Here!</h1>");
    </script>
</body>
</html>
```

In this code, you can see the addition of the <script> tag within the <head> of the page.

**Note** Depending on your settings, if you view this page some versions of Internet Explorer prompt you to allow blocked content. If you receive such a warning when viewing this page, allow the content.

You've now added jQuery to the page, but you haven't yet done anything with it. You'll see how to use jQuery in an upcoming section of this chapter and throughout the book.

## Using a CDN-Hosted jQuery Library

You can use the jQuery library hosted on a CDN. This section looks at how to add the CDN-hosted version of jQuery to your project. Note that you don't need to perform the actions in this section if you followed the previous section and already added jQuery to your project.

Adding jQuery, whether hosted locally or through a CDN, is essentially the same task, with the same syntax. The only change you make is to the location provided to the *src* attribute. With a locally hosted jQuery copy, the *src* attribute points to that copy on the server itself. With CDN-hosted jQuery, the *src* attribute uses the URL of the remote jQuery copy, like so:

```
<script type="text/javascript" src="http://ajax.aspnetcdn.com/ajax/jQuery/
jquery-1.7.1.min.js"></script>
```

Place the <script> tag in the <head> section of the document. Using the page created earlier as an example, the full markup with CDN-hosted jQuery looks like the following code (which you can find as *cdnjquery.html* in the Chapter 4 companion content):

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <script type="text/javascript" src="http://ajax.aspnetcdn.com/ajax/jQuery/
jquery-1.7.1.min.js"></script>
</head>
<body>
    <script type="text/javascript">
        document.write("<h1>Start Here!</h1>");
    </script>
</body>
</html>
```

**Note** Depending on your settings, some versions of Internet Explorer prompt you to allow blocked content when you try to view this page. If you receive such a warning, choose to allow the content. Also, the example code points to the Microsoft CDN for jQuery. There are other CDNs available, including one from Google. See *http://docs.jquery.com/Downloading_jQuery#CDN_Hosted_jQuery* for the most up-to-date list of CDNs and their corresponding URLs.

That's all there is to adding jQuery to your page. Next you'll see how to test jQuery to make sure it's working as expected on this sample page. jQuery will be used throughout the book.

## Testing jQuery

Now that jQuery is linked within your page, either locally or through a CDN, it's time to use it for the first time. To do so, you'll start with the simple page created earlier in the chapter:

1. Begin with the example page from earlier in the chapter. Remove the existing <script> within the body so that you have a base that looks like the following. Note that your location for jQuery might be different. (The one shown here is a CDN-hosted version from the previous section; you might be using the locally hosted version from earlier in the chapter.)

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <script type="text/javascript" src="http://ajax.aspnetcdn.com/ajax/jQuery/
jquery-1.7.1.min.js"></script>
</head>
<body>
</body>
</html>
```

2. Add an empty  element within the body of this markup. The markup will look like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <script type="text/javascript" src="http://ajax.aspnetcdn.com/ajax/jQuery/
jquery-1.7.1.min.js"></script>
</head>
<body>
    <div id="testDiv"></div>
</body>
</html>
```

   The  element was added with an identifier (id) of "*testDiv*". Now let's add some jQuery.

3. Just prior to the closing </body> tag, add the following JavaScript:

```
<script type="text/javascript">
$(document).ready(function () {
    $("#testDiv").text("Start Here, now with jQuery");
});
</script>
```

The final page should look like this:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <script type="text/javascript" src="http://ajax.aspnetcdn.com/ajax/jQuery/
jquery-1.7.1.min.js"></script>
</head>
<body>
    <div id="testDiv"></div>
    <script type="text/javascript">
        $(document).ready(function () {
            $("#testDiv").text("Start Here, now with jQuery");
        });
    </script>
</body>
</html>
```

4. With that code in place, run the project by pressing F5 or choosing Start Debugging from the Debug menu. You should see a page like Figure 4-3.

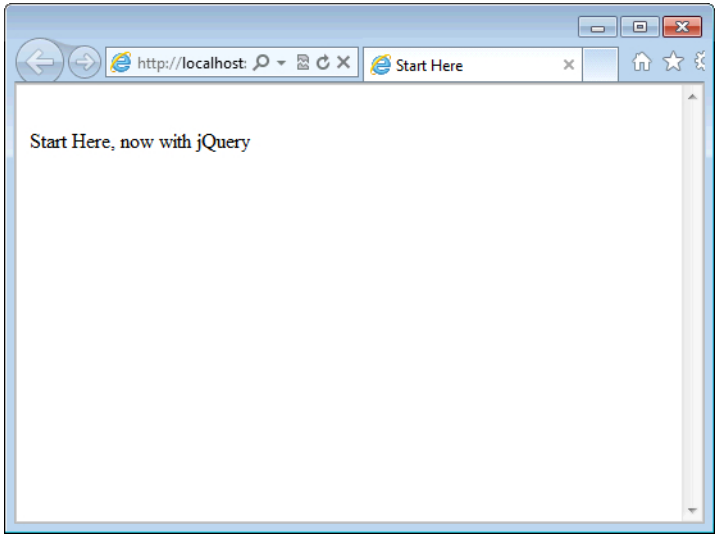


**FIGURE 4-3** Using jQuery to add content to a page.

If you don't see the page as expected, Visual Studio should show you an error. For example, I purposefully (as far as you know) had a syntax error in my version of this file. Visual Studio alerted me that I had a typo and even highlighted the line where the error appeared, as you can see in Figure 4-4.

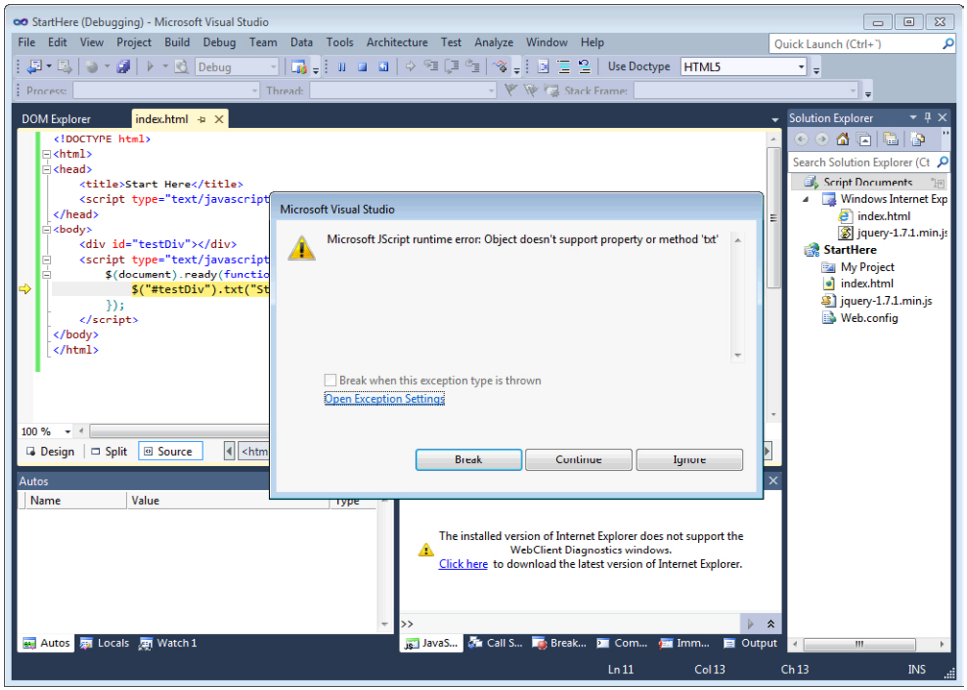


**FIGURE 4-4** Visual Studio provides some helpful debugging for certain errors.

The error I created for this example is one that Visual Studio can find. However, if you receive nothing more than an empty page when attempting to run the example, troubleshooting becomes a bit more difficult. However, Internet Explorer includes some additional debugging that is available through its Developer Tools add-in. In Internet Explorer 9, you can view the F12 developer tools by pressing F12. Take a look back at Chapter 3, "Building JavaScript Programs," for more information on debugging.

I will cover much more detail about jQuery throughout the book to help explain what the code example did. The short explanation is that you used the jQuery *ready()* function, which ensures that the DOM-related elements on the page are ready for manipulation by scripts. Within the *ready()* function, you accessed the *testDiv* using an id selector and changed its text.

You've now added jQuery to your page and used it to add content to a page. Next up is jQuery UI.

## Getting jQuery UI

You can obtain jQuery UI from *http://jqueryui.com*. jQuery UI is built around a modular architecture that enables developers to download only components that are used in their application. For the purposes of this book, you'll download all of it. On the jQuery UI site, click the Build Custom Download link. By default, all of the components are selected, so simply select Download and save the file to your computer.

**Note** The steps for downloading jQuery UI might change over time. The goal of the download is to get all of the jQuery UI components regardless of how that's accomplished by the time you read this book.

jQuery UI comes as a zip file with JavaScript contained in the js folder and related CSS stored in the CSS folder. Extract the contents of this zip file to your computer. For example, I created a folder called jqueryui on my Desktop and extracted the contents into that folder, as shown in Figure 4-5.

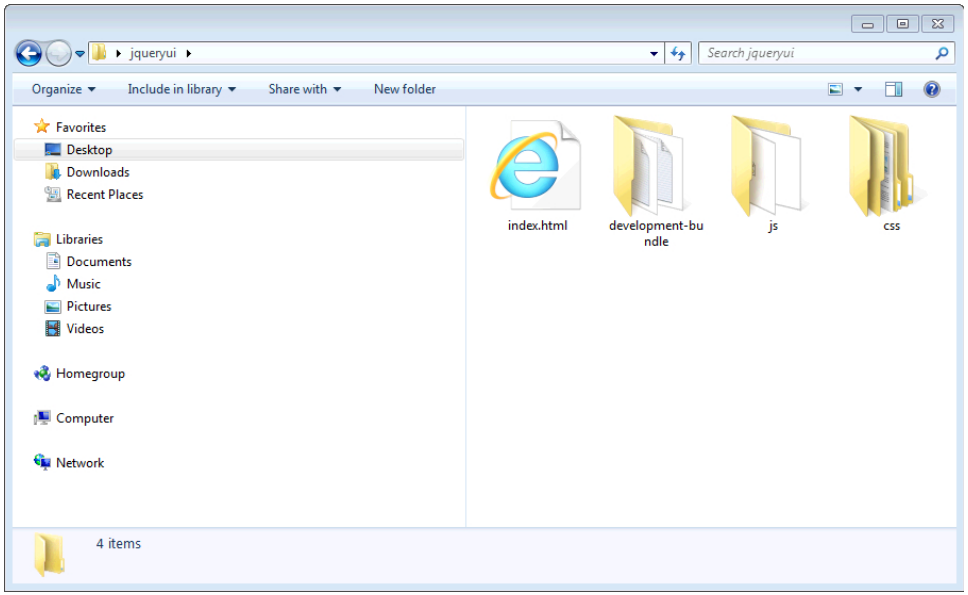


**FIGURE 4-5** Extracting jQuery UI to its own folder.

## Adding jQuery UI to a Project

Adding jQuery UI to a project isn't quite as simple as adding jQuery. The difficulty is because jQuery UI includes themes and CSS. But the power and ease jQuery UI provides makes the additional work to get it worthwhile. And the process really isn't that cumbersome once you've done it a couple times. Essentially, the process involves placing the CSS and JavaScript within your project. If you're not using Visual Studio, you need to place the css and js directories in a location that's available to your web browser (or web server, if you're using a server for development). Here's an exercise for creating the folders, which also includes moving the jQuery file into the js folder. If you already created css and js folders in your project from previous chapters, you only need to move the jQuery file as part of this exercise.

1. Open your StartHere project if it isn't already open within Visual Studio.
2. Within Solution Explorer (usually on the right side like in the figures you've seen so far), right-click the project name StartHere, navigate to Add and select New Folder. A new folder will be created within your project. The new folder should be named css.
3. Do the same action again: right-click, navigate to Add, select New Folder, and add a folder named js.
4. If you have an *external.js* file in the project, ensure that it appears in the js folder as shown, or drag it to the js folder. Finally, drag the jQuery file (if you have one locally) into the js folder. Your project should look like the one shown in Figure 4-6, with two folders (css and js) and the jQuery file within the js folder.

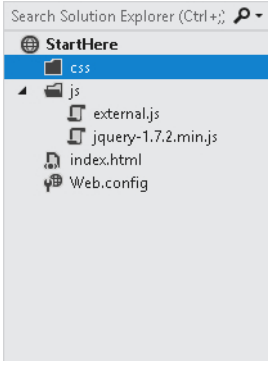


**FIGURE 4-6** Two folders created with Solution Explorer to hold CSS and JavaScript files.

5. With the folders created, you'll next copy the CSS extracted from the jQuery zip file and place it into the css folder you created in your project. Using Windows Explorer, navigate to the folder where you extracted the jQuery UI zip file. (See Figure 4-5.)

6. Within the extracted folder, open the CSS folder. You'll see another folder with the name of the theme you chose. Currently, the default theme is named ui-lightness. You can see an example of this folder in Figure 4-7.

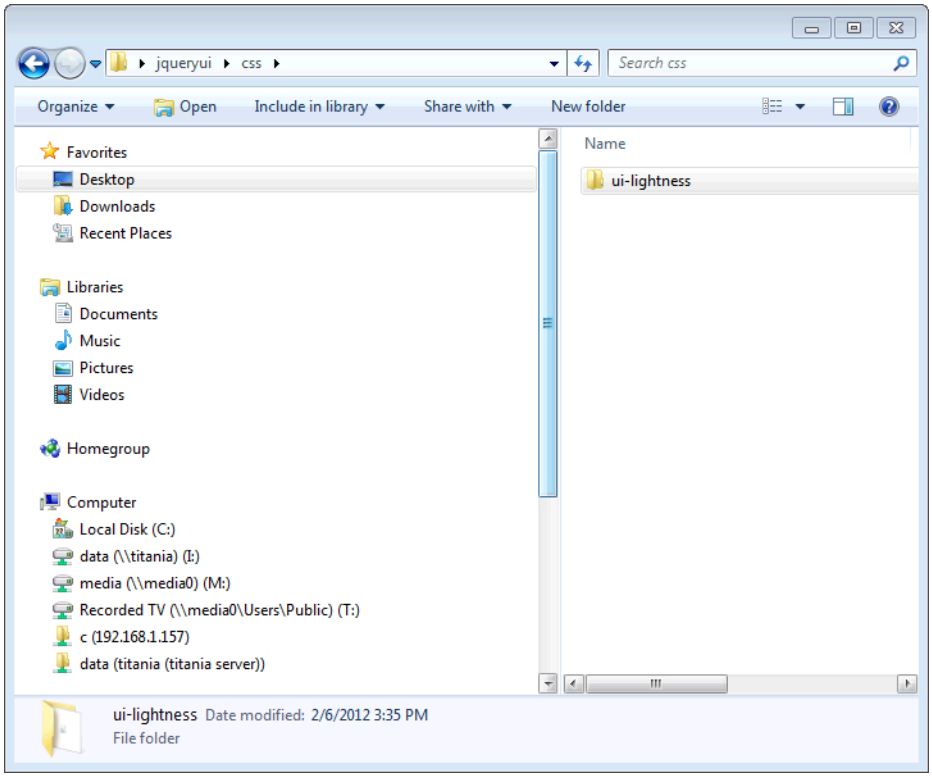


**FIGURE 4-7** The ui-lightness theme in the CSS folder in the jQuery UI distribution.

7. Right-click the theme folder (ui-lightness) and select Copy.

8. Now navigate back to Visual Studio and click the css folder inside of Solution Explorer. Right-click the css folder and select Paste. Your Solution Explorer should now look Figure 4-8, with the ui-lightness folder inside of the css folder.

**Note** If you didn't paste into the correct area, right-click and delete the ui-lightness folder and try again.

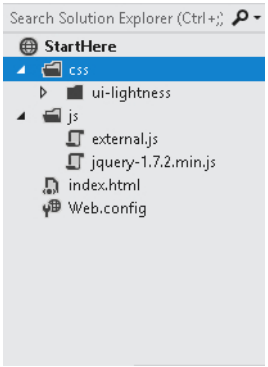


**FIGURE 4-8** The ui-lightness folder added underneath the css folder within the StartHere project.

And finally, you can add the jQuery UI JavaScript file within the js folder in your project. Right-click the js folder in Solution Explorer and select Add, and then click Existing Item. The Add Existing Item dialog opens so that you can navigate to the folder in which you extracted the jQuery UI zip file. Within the jQuery UI folder, open the js folder and select the jQuery UI js file. As before, you might need to change the file type so that you can see All Files or Script files in order to see the js file. An example is illustrated in Figure 4-9.

**Tip** Be sure you select the file named with jquery-ui and not the standard jquery file.

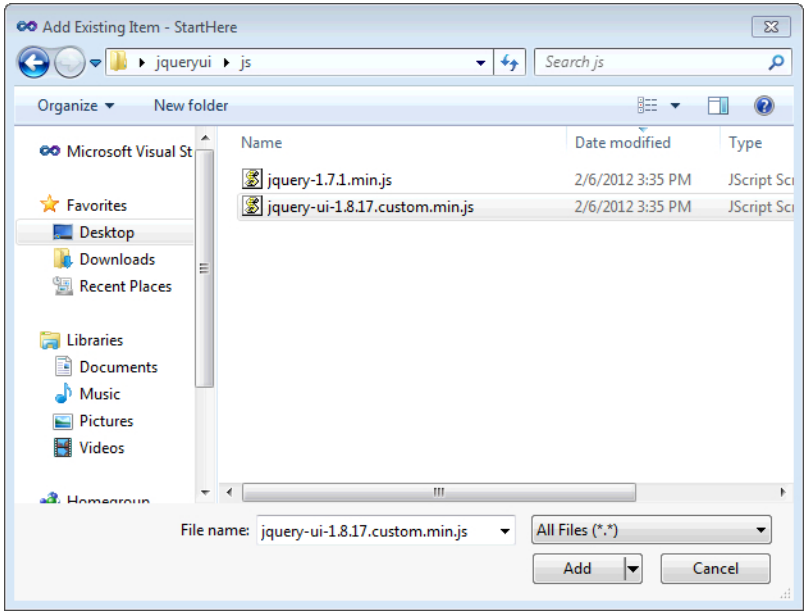


**FIGURE 4-9** Adding the jQuery UI JavaScript file to your project.

Solution Explorer for your project will now look similar to Figure 4-10.

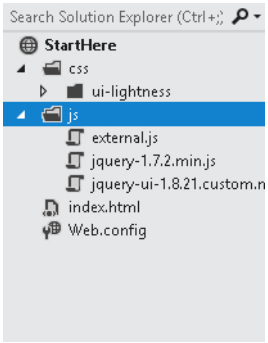


**FIGURE 4-10** The project's Solution Explorer now has JavaScript files for jQuery and jQuery UI, along with CSS for jQuery UI.

## Testing jQuery UI

Now that jQuery UI has been added to your project, it's time to test it. To do so, you'll add references to the CSS as well as a reference to the jQuery UI JavaScript file, both of which are now included in your project. These references are added in the <head> section, wrapping around the reference you added for jQuery earlier in the chapter. Note that the order is important for these. You should load the CSS file first, then jQuery itself, and finally jQuery UI.

You need to first add a reference to the CSS, which looks like this:

```
<link type="text/css" href="css/ui-lightness/jquery-ui-1.8.17.custom.css"
rel="Stylesheet" />
```

You'll then add a reference to the jQuery UI JavaScript file, which looks like this:

```
<script type="text/javascript" src="js/jquery-ui-1.8.17.custom.min.js"></script>
```

Note that the theme, ui-lightness, included as the *href* value might be different in your implementation. This name corresponds to the name you see within Solution Explorer within the css folder. Additionally, the version of jQuery UI will certainly be different in your version and will likely even change during the writing of this book. This caveat applies to both the jQuery UI css file and the jQuery UI js file found within your project. Be sure to change these versions to correspond to your version!

Using the code in *index.html* created earlier as a base, here's the code with the addition of links to the CSS and JavaScript for both jQuery UI and jQuery. Note that I've removed the  and <script> within the body. Otherwise, the main two changes are to add the <link> for the jQuery UI CSS and the <script> tag for the jQuery UI JavaScript file:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
rel="Stylesheet" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js"></script>
</head>
<body>
</body>
</html>
```

You'll now build on the code here to test one of the effects included in jQuery UI. For this exercise, use the HTML code just shown within your StartHere project in Visual Studio. This HTML, found as *jqueryui.html* in the companion content, should be placed in the *index.html* file and replace any existing HTML in that file.

1. Using the previous HTML code as a base, the following HTML markup should be added to the <body> section of the page:

```
<div id="startHere" style="border: 3px solid black;">
    <h1>Start Here</h1>
</div>
<div>
    <a href="#" id="hiderLink">Hide it!</a>
</div>
```

2. Below the markup (but still within the <body> section), add the following JavaScript:

```
<script type="text/javascript">
    $("#hiderLink").click(function () {
        $("#startHere").hide();
    });
</script>
```

The entire page should look like the following (noting, again, that your jQuery version numbers will be different):

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
      rel="stylesheet" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
```

```
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js">
      </script>
</head>
<body>
    <div id="startHere" style="border: 3px solid black;">
        <h1>Start Here</h1>
    </div>
    <div>
        <a href="#" id="hiderLink">Hide it!</a>
    </div>
    <script type="text/javascript">
        $("#hiderLink").click(function () {
            $("#startHere").hide();
        });
    </script>
</body>
</html>
```

3. View this page in a web browser. You should see a page similar to the one shown in Figure 4-11.

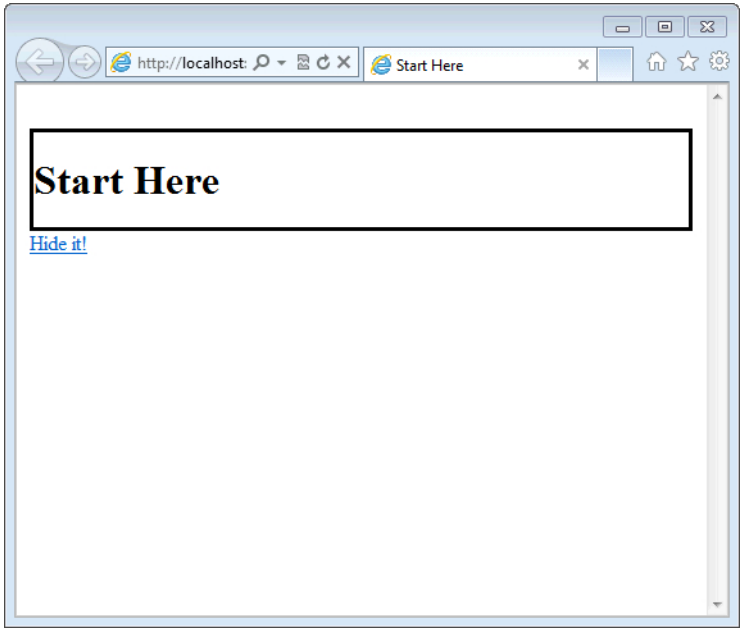


**FIGURE 4-11** Building a page to work with jQuery UI.

4. You might already have clicked it, but if you haven't, click the "Hide it!" link. The words "Start Here" will disappear. See Figure 4-12.

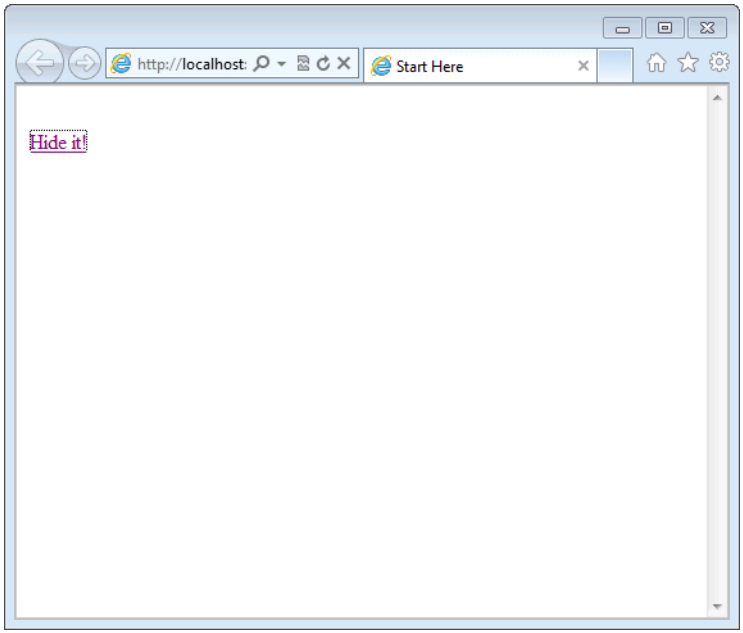


**FIGURE 4-12** Hiding an element using the jQuery UI *hide* function.

Admittedly, you might not be too impressed. In fact, jQuery itself includes a *hide()* function. But there's a couple points to realize: first, there's a corresponding *show()* function you can use to get your content back. Second, jQuery UI offers several effects that define exactly how the element should hide. Here's a taste of one of them.

Within the script section in your page, change the *hide()* function to look like this:

```
$("#startHere").hide("slide", {}, 2000);
```

Reload the page, and click the "Hide it!" link. The Start Here *div* now slides off the screen, taking about 2 seconds (2000 milliseconds) to do so.

jQuery UI will be covered in much greater depth throughout the book. For now, the take-away is that jQuery and jQuery UI can be used to make your life better, taking some of the hard work in creating complex behaviors and making it easy.

## The Browser Object Model

The Browser Object Model (BOM) defines a set of objects that are available to the JavaScript programmer for working with a web browser, and it defines the documents rendered within the browser window. That's quite a sentence. I'll try to explain it another way. Using the BOM, you can get information about the user's browser (through the *navigator* object), you can look through the browser

history (through the *history* object), and you can work with the webpage (through the *document* object. There are other objects or pieces of the BOM as well. This section will look at some of the objects included in the BOM.

## Events and the *window* Object

Developers can use the *window* object to work with and react to events in the browser, such as when the mouse cursor moves over an element or when a click is encountered. Additionally, the *window* object contains methods for working with timers. You'll see an example of using a timer later in this chapter.

Much of the *window* object's functionality revolves around the event models presented by web browsers. There are several event models and several levels of support for those event models within and across web browsers. This means that you need to program for each and every difference and nuance within each browser. In general, working with events is much more efficient and easier when using jQuery. jQuery accounts for these differences in both event-model support and model implementation within browsers. With this in mind, I'll pay particular attention to using jQuery for event handling rather than show you less efficient (and less preferred) methods for event handling. Event handling is discussed more in Chapter 7, "Styling with JavaScript."

**Note** An in-depth discussion of event models is beyond the scope of this book. If you'd like more information on event models and how to work with them, I recommend my more advanced-level book, *JavaScript Step by Step* (Microsoft Press, 2011).

## The *screen* Object

The *screen* object is used to determine things about the user's viewing environment. Some of the properties of the *screen* object are

- *availHeight*
- *availWidth*
- *colorDepth*
- *height*
- *width*

These properties can be used to help make decisions about how to display content. For example, if you know that the available height of a user's window is smaller than an element in your design, you

can choose to not display that content or display a substitute item that's more appropriate for the user's viewing ability.

A simple way to view the properties of the *screen* object is to write them to your screen using a *for.. in* loop. This pattern will be used in the next few sections to enumerate the properties of the various BOM objects. The following code can be found in *screen.html* in the companion content and *screen.js* in the js folder of the Chapter 4 companion content:

1. If your StartHere project isn't open, open it now. Clear any code inside of *external.js*, and place the following code in the file:

```
for (propt in screen) {
    document.write("Screen property " + propt + " is currently: ");
    document.write(screen[propt] + "<br />\n");
}
```

2. Save *external.js*. It should look similar to Figure 4-13.

```
for (propt in screen) {
    document.write("Screen property " + propt + " is currently: ");
    document.write(screen[propt] + "<br />\n");
}
```

**FIGURE 4-13** The *external.js* file containing code for looping through screen properties.

3. With *external.js* saved, open *index.html* and clear any HTML from that file, replacing it with the following:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
</body>
</html>
```

4. View the corresponding *index.html* in a web browser. You'll be presented with a page containing the properties of your current window, like the one shown in Figure 4-14.

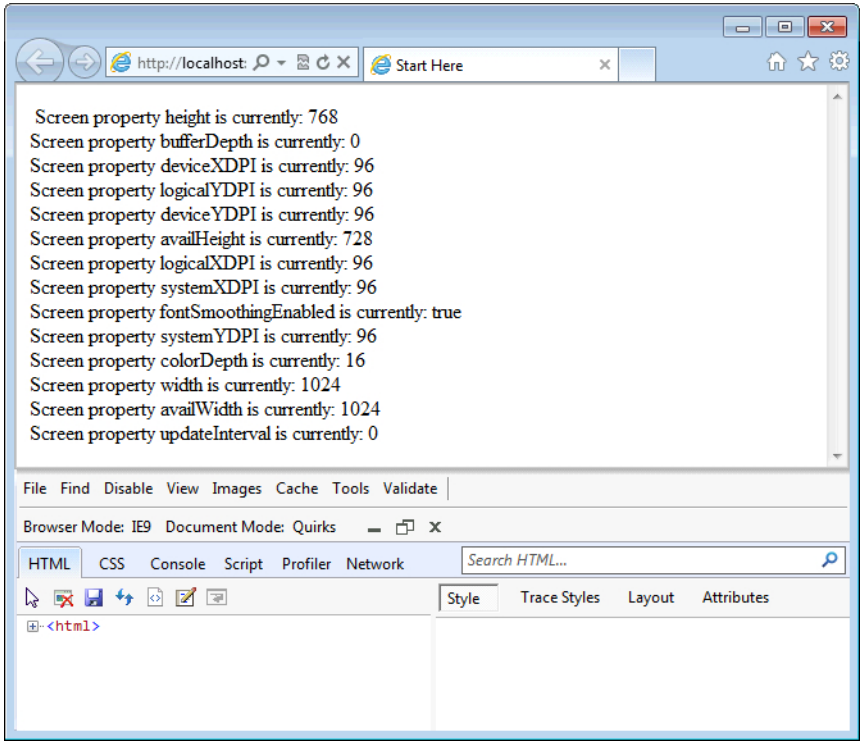


**FIGURE 4-14** Screen properties in Internet Explorer 9.

Note that your screen properties will likely look different depending on the version of the browser that you're using and your actual screen size.

## The *navigator* Object

Whereas the *screen* object contains information about the user's screen such as the screen resolution, the *navigator* object provides information about the actual browser itself. Things like the name of the browser, its version, and other items are obtained through the *navigator* object. To see the properties available through the *navigator* object, modify the code from the previous section to enumerate the *navigator* object instead of the *screen* object, like so:

```
for (propt in navigator) {
    document.write("navigator property " + propt + " is currently: ");
    document.write(navigator[propt] + "<br />\n");
}
```

If you want to see this code in action, simply replace the existing code in *external.js* with the code from this example and view *index.html* in your web browser. Alternatively, the HTML can be found in *navigator.html* in the Chapter 4 code and *navigator.js* in the Chapter 4 js folder. The result should look like Figure 4-15.

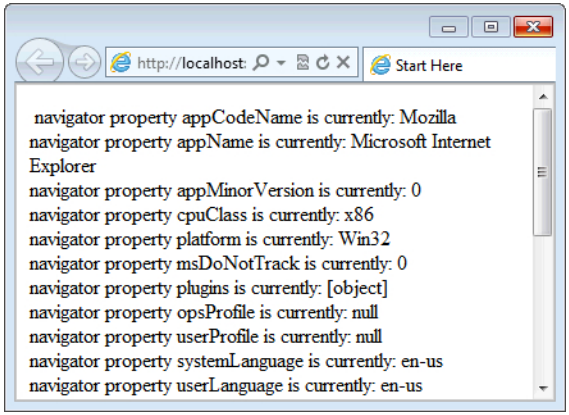


**FIGURE 4-15** Enumerating the *navigator* object.

The *navigator* object is frequently used to gather information about the visitor, such as the name of the browser that the visitor is using and a list of available plugins. This information is sometimes accurate, but saying that also implies that the information coming from the *navigator* object is also sometimes inaccurate.

For example, the *userAgent* property can be used to crudely determine the browser but, like everything else coming from the client, that *userAgent* can be trivially altered or faked by the visitor. Additionally, as discussed in Chapter 1, "What Is JavaScript?," JavaScript might not be available at all, and if so, the *navigator* object would be unavailable, as well.

This book won't show any examples of using the *userAgent* property because doing so is not a best practice. The best practice for designing across browsers is to enhance the sites features based on what the client has available. This concept is called *progressive enhancement*, and you'll see examples of it throughout the book.

## The *location* Object

Another interesting object within the BOM hierarchy is the *location* object. The *location* object contains information about the currently loaded URL (Uniform Resource Locator), the query string, and other information about the request as it arrived at the web server. Modifying the previous *for..in* loop to enumerate the *location* object results in code that looks like this:

```
for (propt in location) {
    document.write("location property " + propt + " is currently: ");
    document.write(location[propt] + "<br />\n");
}
```

**Note** This code can be found in *location.html* and *location.js* in the companion content.

The *location* object is frequently used to redirect a visitor to another page. Depending on the browser, a *replace()* method might be available that places the new page into the browser's history. If the *replace()* method isn't available, an *href* property is available that results in the browser being redirected. Here's an exercise to create an immediate redirect to another page. This code can be found as *redir.html* and *redir.js* in the companion content. Later in this chapter, you'll see an example that uses a timer to automatically redirect or provides a link for the visitor to click if JavaScript isn't available.

1. Begin by opening the StartHere project in Visual Studio and then opening *index.html*.
2. Within *index.html*, place the following markup:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
<a href="http://www.braingia.org?shjs">Click here if you're not redirected</a>
</body>
</html>
```

3. Save *index.html*.
4. Open *external.js*, and place the following code, replacing any other code that's already there from previous exercises:

```
if (typeof(location.replace) !== "undefined") {
    location.replace("http://www.braingia.org?shjs-replace");
} else {
    location.href = "http://www.braingia.org?shjs-href";
}
```

   This code first looks to see if the *location.replace()* method is available. It does this by making sure that the *typeof* the *location.replace()* method is not undefined. (Note the strict form of equality test, *!==*.). If *location.replace()* is available, it's used to send the visitor to a website. If *location.replace()* is not available, *location.href* is used instead. Within the *index.html* page itself, there's an anchor <a> element that contains a link to the same site. If JavaScript isn't available in the visitor's browser, the visitor can click this link.

5. Save *external.js*, and run the project. A web browser opens, and you're immediately redirected to the page specified in the code.

The code just shown can be called from a timer so that an interstitial page is shown prior to the redirect occurring. Here's the code to do so (also found as *timer.js* and *timer.html* in the companion content):

```
function redirect() {
    if (typeof(location.replace) !== "undefined") {
        location.replace("http://www.braingia.org?shjs-replace");
    } else {
        location.href = "http://www.braingia.org?shjs-href";
    }
}

setTimeout('redirect()',5000);
```

This code takes the *if* conditional from the previous exercise and wraps it in a function called *redirect()*. After the *redirect()* function is closed, the *setTimeout()* method is called. *setTimeout()* accepts two parameters: the function or code to execute when the timer expires, and the length of the timer itself, in milliseconds. In this example, the *redirect()* function is called after 5 seconds (5000 milliseconds).

# The DOM

The Document Object Model (DOM) provides a way to access elements of an HTML document. The DOM creates a tree-like structure to represent an HTML document, and through this hierarchical structure you can manipulate the DOM through JavaScript. This section discusses the DOM and then shows examples of using the DOM through JavaScript. The examples shown in JavaScript are rather brief in favor of showing more examples in jQuery, which is more likely how you'll work with the DOM in practice.

## DOM Versions

As alluded to earlier, the DOM is a specification put forth by the W3C to define a hierarchical representation of a document. In the case of this book, I'm going to concentrate on the DOM and how it relates to HTML. Then, even more specifically, I'll show you how to use JavaScript to work with the DOM.

The DOM has different versions, known as *levels*. DOM Level 0 is known as the legacy DOM and concentrates mainly on giving programmatic access to form elements. Support for DOM level 0 still exists in web browsers, but its use is frowned upon. You should use later levels of DOM support—namely, levels 1 and 2.

Web browsers offer various amounts of support for each level of the DOM, and most of them implement that support in slightly different ways. The biggest differences have historically been found in Internet Explorer, where an entirely different event model was used.

### Internet Explorer and Standards

Internet Explorer, especially prior to version 9, causes headaches for JavaScript developers due to its implementation and support for web standards. Many other browsers implemented the standards roughly the same, but hacks had to be employed to get the same functions and pages to display in Internet Explorer. Seeing more than 59,000,000 results in a search for "Internet explorer javascript hack" means that it's not just me!

Luckily, Internet Explorer has gotten much better at its standards adherence and implementation of key elements, so these hacks are becoming less important. Additionally, the rising popularity of jQuery has also abstracted these differences and made them less of an issue for common JavaScript development tasks.

## The DOM Tree

The DOM represents HTML documents in a tree-like structure, or rather an uprooted tree structure because the trunk of the tree is on top. For example, consider this simple HTML structure:

```
<html>
<head>
<title>Hello World</title>
</head>
<body>
<p>Here's some text.</p>
<p>Here's more text.</p>
<p>Link to the <a href="http://www.w3.org">W3</a></p>
</body>
</html>
```

Figure 4-16 shows this HTML structure represented as the DOM tree.

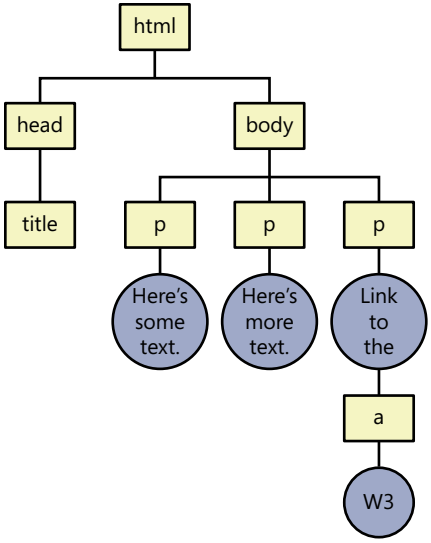


**FIGURE 4-16** The DOM representation of an HTML structure.

## HTML Attributes and the DOM

JavaScript and jQuery include methods for traversing, or walking through, the DOM's tree-like structure. This means that you can make a change to all the elements of a certain type on the page. However, in practical terms, you'll more likely want to work with elements individually, based on their type or based on their position within the DOM.

There are two HTML attributes that make it easier to work with individual elements and elements of a certain type. These are the id and class attributes. The id attribute provides a unique identifier for an element within an HTML page. The id should be unique—you shouldn't share the same id among multiple elements. If you need to share an id among multiple elements, you should use a class. A class is used to define an identifier that is shared among multiple elements. Here's an example of each:

```
<div id="myDiv">This is a div element with an id</div>
<div class="myClass">This is a div element with a class</div>
```

Classes and ids can be used together on the same element as well. A typical use of a class is to style elements in the same way. For example, if you want all of the  elements to be colored a certain way, you use a class and then apply a CSS style to that class. An id can be used to target a specific element as well.

In much the same way as you do for CSS, you can also access elements through JavaScript by using an id or a class, as you'll see next.

## Retrieving Elements with JavaScript and jQuery

This section looks at retrieving elements using JavaScript and jQuery. The section begins with a look at jQuery and its use and then jumps back into the DOM.

The following HTML structure will be used for this and subsequent sections unless otherwise noted. This code is found in *elements.html* in the companion content. Note that this code still refers to the old *external.js* from the previous exercise and therefore will redirect you. Later in the chapter, you'll update *external.js*, so don't try to view this HTML in a browser just yet!

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
<div id="myDiv">
<p><a id="silverwareLink" href="http://en.wikipedia.org/wiki/
Silverware">Silverware</a></p>
    <ul id="myUL">
        <li class="classList">Fork</li>
        <li class="classList">Spoon</li>
        <li class="classList">Knife</li>
    </ul>
</div>
</body>
</html>
```

### Order Is Important

While putting together the example HTML for this chapter, I mistakenly had the <script> tag to load the *external.js* file listed before the <script> tag to load jQuery. Unfortunately, the examples weren't working, and I spent a bit of time troubleshooting why seemingly simple code was silently failing but sometimes working.

The answer was that *external.js* was loaded before jQuery. Essentially, I unknowingly ignored my own advice. Flipping the order so that jQuery was loaded first fixed the problem, and the correct version is shown in the example earlier.

## Using jQuery, Briefly

Earlier in this chapter, you saw how to include jQuery in a webpage, and the example shown in this section contains a <script> tag for jQuery. jQuery really shines when it comes to working with the DOM. jQuery's syntax is slightly different than the DOM, as you saw earlier in this chapter.

jQuery defines its own function, called *jquery()*. This enables you to call any jQuery-related functions through the *jquery()* function itself. However, jQuery also defines a shortcut to the *jquery()* function, *$()*. Using *$()* to access jQuery-related functions is far more common than using *jquery()*, and this is the syntax you'll see throughout the book. When you see *$()*, you should know that it's a reference to *jquery()* itself, so whatever follows within the parentheses is jQuery-specific. This will become clearer (and seemingly unimportant) as you work with jQuery.

This section explains jQuery in more depth to prepare you for working with the DOM later in the chapter. There are two key things you need to know about right now: the *ready()* function and selectors.

### The *ready()* Function

When a webpage is loaded, the web browser reads and parses the JavaScript and HTML in order, from top to bottom. This means that JavaScript code might be loaded and executed prior to the page being loaded. Therefore, if the JavaScript code needs to work with a certain element of the page before that element is loaded, the JavaScript might fail.

JavaScript solves this by using a *load()* or *onload()* method that is executed when the page is loaded. However, doing so has some drawbacks and, when using jQuery, it is not recommended. The jQuery *ready()* function, part of the *document* object, provides a way to execute JavaScript after the DOM has loaded. The jQuery *ready()* function is different than the *onload()* method, however, because the *onload()* method will wait to execute after everything is loaded in the page, including images. The jQuery *ready()* function waits only for the DOM—the HTML itself, not the images.

The *ready()* function is used when you have JavaScript that needs to execute immediately on page load, such as when you need to change something on the page prior to the page being completely loaded by the browser.

Both the *onload()* method and the *ready()* function can be invoked only once per page, and using both is generally not recommended. This means you need to plan your programs accordingly so that they're invoked or called from within the *ready()* function. Note that this does not mean that you need to place all of your JavaScript inside of the *ready()* function; rather, you need to place any load-dependent functions inside of the *ready()* function—anything that needs to occur right away or react to something that the user does.

Later in this section and throughout the book, you'll see the *ready()* function in use, and the function will become clearer as you work through examples throughout the book.

### jQuery Selectors

The selector is central to jQuery. The selector defines the elements on which the jQuery code will execute. Selectors are incredibly powerful ways of grouping elements in the DOM and performing group operations on them. Not only can you select elements by their HTML id and class, but you can also select elements by their ancestry—for example, only <a> elements that are within an <li> element—and you can select elements in numerous other ways.

In the example found earlier in the chapter, you retrieved an element by using a jQuery id selector, and you'll see more examples of this later in the chapter as well. You can also retrieve elements by their class, and you'll see examples of this later in the chapter too. The more advanced selectors will be discussed as they're encountered in the book.

## Retrieving Elements by ID

The *getElementById()* method is a primary method for working with the HTML elements and the DOM through JavaScript. A typical usage of the *getElementById()* method is to instantiate a variable with an element and then work with one or more of the element's attributes. Using the HTML from earlier in this section as a guide, here's an example that retrieves the value of the *href* attribute from the <a> element on the page, using its id as the selector. You can find the code as *getelementbyid.js* in the companion content.

```
var aLink = document.getElementById("silverwareLink");
alert(aLink.href);
```

Once it is retrieved, you can set or change attributes. This behavior will be shown later in this chapter.

The typical use of jQuery doesn't call for element retrieval as in the example just shown. When using jQuery, it's typical to chain operations together. For example, rather than retrieve the element and set it to a variable as in the previous example, you typically just retrieve the attribute directly when using jQuery, as in this example:

```
alert($('#silverwareLink').attr('href'));
```

In this example code, the *href* attribute is retrieved from the HTML element with the id *silverwareLink*. This is accomplished with the help of the jQuery *attr()* function. The jQuery-related code is enclosed in the standard JavaScript *alert()* function.

Here's an exercise to work with this code and the HTML presented earlier in the section. To prepare for this exercise, open your StartHere project if it isn't already open. The entire code for this exercise can be found in *retrievebyid.html* and *retrievebyid.js* in the companion content.

1. Within the *index.html* file, change the HTML to match the example from earlier in this section. The changes from the existing *index.html* are highlighted in bold.

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
<div id="myDiv">
<p><a id="silverwareLink" href="http://en.wikipedia.org/wiki/
Silverware">Silverware</a></p>
    <ul id="myUL">
        <li class="classList">Fork</li>
        <li class="classList">Spoon</li>
        <li class="classList">Knife</li>
    </ul>
</div>
</body>
</html>
```

2. Save *index.html*.

3. Clear any code out of *external.js*, and place the following code inside the file. (You can find this code as *retrievebyid.js* in the companion content.)

```
$(document).ready(function(){
    alert($('#silverwareLink').attr('href'));
});
```

4. Save *external.js*.

5. Now run the code by pressing F5 or viewing the *index.html* in a web browser. You should receive an alert like the one shown in Figure 4-17.

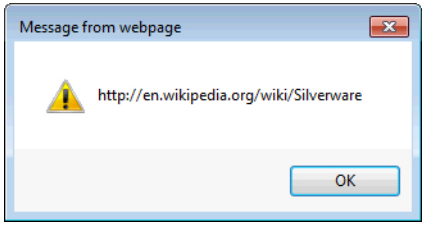


**FIGURE 4-17** Retrieving the value from the *href* attribute with jQuery.

The code in this exercise used *the jQuery ready()* function. Within the *ready()* function, the *alert()* was called with jQuery inside to retrieve the value of the *href* for the HTML id *silverwareLink*.

The next section discusses retrieving elements by their CSS class, and later you'll see how to change attributes such as *href* and others using jQuery.

## Retrieving Elements by Class

Like the *getElementById()* function, JavaScript also offers a *getElementsByClassName()* function. The *getElementsByClassName()* function retrieves all elements with the given CSS class. However, *getElementsByClassName()* is not supported in older versions of Internet Explorer (starting around version 8); therefore, its use isn't recommended quite yet. However, jQuery includes a perfectly valid class selector that does work in older browsers. Whereas the jQuery id selector uses a hash or pound sign (#), the class selector uses a single dot, like this:

```
$('.classList')
```

That code selects all elements on a page with the class of *classList*. Yes, it really is that easy. A primary difference between an id selector and the class selector is that the class selector typically returns multiple items. A typical scenario is to use the jQuery *each()* function to iterate through the elements retrieved. The *each()* function is akin to a *for* loop, iterating through a group of things. The HTML shown earlier includes several <li> elements that share the same class. Retrieving those elements and then iterating through them with jQuery's *each()* function looks like this:

```
$('.classList').each(function() {
    alert($(this).text());
});
```

This code, which you can find as *classlist.html* and *classlist.js* in the companion content, creates a function inside of the *each()* function. This inner function, which by the way is an anonymous function, merely uses the *alert()* function to show the text of each element. The *each()* function is roughly equivalent to the *for* loop you worked with earlier in the book. Two new items here are the jQuery version of *this*, which is written as *$(this)* in jQuery, and the *text()* function, which retrieves the plain text of the selected item.

If you want to run this code, simply add it within the *$(document).ready()* function from the exercise earlier in the chapter.

## Retrieving Elements by HTML Tag Name

You can also work with the DOM by retrieving HTML elements through their tag name, as in <li>, <a>, , and so on. Like the class example, retrieving elements by tag name is useful when you need to do something with the elements as a group. You might notice already, though, that retrieving by tag name is somewhat less specific than retrieving by class. For example, you can easily retrieve all of the

<a> elements on a page, but it's more likely that you want to retrieve only one of those elements or a more specific grouping of the <a> elements on a page. There are two ways to solve the specificity problem when retrieving by tag name: one uses standard JavaScript, and the other uses jQuery. I'll show both in this section.

JavaScript has a function called *getElementsByTagName()* you can use to retrieve HTML elements. An easy way to add some specificity when using this function is to retrieve its parent element. For example, in the HTML example being used in this chapter, you have this hierarchy:

```
<ul id="myUL">
    <li class="classList">Fork</li>
    <li class="classList">Spoon</li>
    <li class="classList">Knife</li>
</ul>
```

Using JavaScript, a typical scenario is to retrieve the parent element and then use a *for* loop to enumerate its children. Here's the code to do so:

```
window.onload = function() {
    var ulElm = document.getElementById("myUL");
    var liElms = ulElm.getElementsByTagName("li");
    for (var i = 0; i < liElms.length; i++) {
        alert(liElms[i].innerHTML);
    }
};
```

This code, which you can find in *getelementbytagname.html* and *getelementbytagname.js* in the companion content, sets the *onload* method of the *window* object to a function. Inside of that function, the <ul> element is selected with the help of the previously seen *getElementById()* method, and the result is placed in a variable called *ulElm*. Next, each of the <li> elements within the *ulElm* are retrieved with the *getElementsByTagName()* method. Those <li> elements are placed in a variable called *liElms*. Finally, a simple *for* loop is created and each of the *liElm* variables is sent to the screen using an *alert()*.

jQuery includes selectors for retrieval by tag name. They have essentially the same syntax you already saw for jQuery selectors. Recall that the id-based selector is a pound sign (#) and the class-based selector is a dot (.). Retrieving by tag name with jQuery leaves those off and retrieves by the tag name itself, much like the native JavaScript *getElementsByTagName()* function. Here's an example:

```
$(document).ready(function() {
    $('li').each(function() {
        alert($(this).text());
    });
});
```

This example uses the same *each()* function you already saw and iterates through each <li> element on the page displaying its text. You might notice the difference between this example and the previous native JavaScript example. The native JavaScript example first retrieved the specific <ul> by its id and then only iterated through the <li> elements within or beneath that <ul>. The jQuery example shown here actually retrieves all <li> elements on the page. However, the same positional or hierarchical behavior can be accomplished thanks to the advanced selector syntax of jQuery. Here's one way to accomplish the same hierarchical syntax, which you can find as *tagselector.html* and *tagselector.js* in the companion content:

```
$(document).ready(function() {
    $('#myUL li').each(function() {
        alert($(this).text());
    });
});
```

The syntax used for this example selects all <li> elements within the id *myUL*. There are several ways to select elements based on their position within the document. See *http://api.jquery.com/category/selectors* for more information on selectors.

## Summary

This chapter showed a lot of material on the power and strength of using JavaScript with the web browser. The Browser Object Model (BOM) and the Document Object Model (DOM) provide the structure within which JavaScript accesses elements of a web browser and webpage.

The chapter first showed how to install and use jQuery and jQuery UI. These libraries are important pieces in a JavaScript programmer's toolkit and enable you to use advanced behaviors and styling that works across multiple web browsers. This means that you can concentrate on the logic of your program rather than on how to get something to work in all browsers.

The chapter looked next at the BOM and some important objects within it. This included a look at event handling, the screen object, and the *navigator* and *location* objects.

The chapter wrapped up with an explanation of the DOM and its importance in JavaScript programming. Examples were shown using native JavaScript functions to retrieve elements from a webpage. Examples were also shown using jQuery.

You've now seen how to retrieve elements. In later chapters, you'll see some of the things you can do with those elements once they're retrieved.

Chapter 5

# Handling Events with JavaScript

**After completing this chapter, you will be able to**

- Recognize common JavaScript events
- Understand how to handle events with JavaScript and jQuery
- Validate data entered into a web form

**EVENTS ARE THINGS YOU REACT TO** (or sometimes ignore intentionally). Consider what happens when the temperature in your house or dwelling falls below a certain value—for example, 69 degrees Fahrenheit in my house. The event occurs when the temperature falls below 69; the handling of that event is done by a thermostat. The thermostat reacts or handles the event by starting the furnace. When the temperature falls from 71 to 70, the thermostat doesn't care; it ignores such events.

For webpages, events include such things as mouse clicks, moving the mouse over an element, clicking on a form element, submitting a form, or even entering an individual keystroke. When programming JavaScript web applications, you react to user clicks and other user-initiated actions through *events*. *Event handling* is simply the term used to describe how you programmatically react to those events.

This chapter looks at events in the context of the web browser, focusing specifically on areas where JavaScript is frequently used, such as validating web form data.

## Common Events with JavaScript

Interacting with a webpage in just about any way results in several events. Even simply loading the page results in the *load()* event, which was discussed in the previous chapter. Moving the mouse around the page, clicking on elements, filling out a form, submitting the form—all these actions also

result in one or more events. There is a fairly complex system in place to both handle events and make them available to the JavaScript programmer. The result of this complexity is that you can react to the events and do something with them, or you can ignore them.

**Note** Read more about event-handling history and complexity at *http://www.quirksmode.org/js/introevents.html*.

Table 5-1 lists some common events.

**TABLE 5-1** Select events available to JavaScript

| Event | Description |
|---|---|
| *mouseover* | Generated when the mouse cursor moves over an element. |
| *mouseout* | Generated when the mouse cursor leaves an element. |
| *mousedown* | Generated when the mouse button is clicked; the first half of a mouse click, when the button is depressed. |
| *mouseup* | Generated when the mouse button is released; the second half of a mouse click. |
| *click* | Generated when the mouse button is pressed and released. |
| *dblclick* | Generated when the mouse button is clicked twice. |
| *mousemove* | Generated as the mouse moves. This event is useful for tracking where the mouse is on the page. |
| *scroll* | Generated when the user scrolls within the document. |
| *focus* | This event fires when a given form field gains focus, whether through a mouse click or by tabbing into the field. |
| *blur* | This event is related to the focus event. When a given form field loses focus, the blur event fires. |
| *submit* | Generated when a form is submitted. |
| *keypress* | This event is generated when a key is pressed. |
| *keydown* | Generated when a key is pressed, just prior to the *keypress* event. |
| *keyup* | Generated when a key is released. |

The rest of the chapter will show examples of how you can use JavaScript to react to events.

## Handling Mouse Events

The browser generates mouse events in reaction to users working with a mouse or a pointing device such as a trackpad or a finger touch. For example, when a user moves the mouse cursor over an element, you can use JavaScript to change that element or to do something else entirely, such as add a new element.

**Note** As you'll see later in this book, you can now also handle some of these events with the help of the CSS *:hover* selector.

jQuery includes functions for handling mouse events such as *mouseover*, *mouseup*, *mousedown*, and so on. When using mouse events, you typically use them in tandem—for example, capturing a *mouseover* to change the look of an element, and capturing the *mouseout* event to restore it to its initial appearance. jQuery also includes functions that handle such tandem events, thus saving you even more work. Here's a simple exercise that creates a hover effect using jQuery. You can find this example in the Chapter 5 sample code in the files *moveit.html* and *moveit.js*. To create this example, follow these steps:

1. Begin by opening the StartHere project if it's not already open.

2. Open *index.html* and place the following HTML within the file:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="js/external.js"></script>
<style type="text/css">
    #mover, #movedest { border: 1px solid black; padding: 3px; }
</style>
</head>
<body>
    <div id="mover">
        Div 1: <p class="moveit">Some Text That Moves</p>
    </div>
    <div id="movedest">
        Div 2:
    </div>
</body>
</html>
```

   The preceding HTML creates two  elements, one with a <p> element in it. Both  elements have borders thanks to the Cascading Style Sheets (CSS) included within the file. This CSS helps you to see the effects that you'll create with jQuery in this exercise.

3. Save *index.html*.

4. Open *external.js*, and delete any existing code in that file, replacing it with the following JavaScript:

```
$(document).ready(function() {
    $('#mover').hover(function() {
         $('.moveit').appendTo("#movedest");
    });
});
```

This code again uses the *ready()* function and places a *hover* function on the element with the id of mover. Within the *hover()* function, any element with the *moveit* class is appended to the element with the id of *movedest*. This code uses the jQuery *appendTo* function, which adds the selected element to the end of the element specified in the parameter—in this case, the *movedest* .

5. Save *external.js*.

6. Load *index.html* into your browser—typically, you do this by pressing F5. You'll be presented with a page like that shown in Figure 5-1.

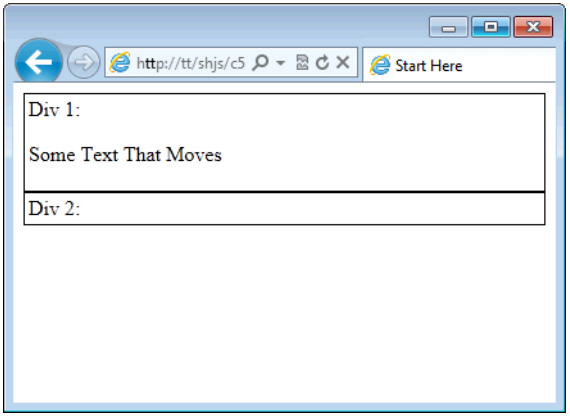


**FIGURE 5-1** The page built in this exercise has two *div* elements, each of which has a border.

7. Move your mouse cursor anywhere within the box for Div 1 and the text will shift into Div 2, as shown in Figure 5-2.

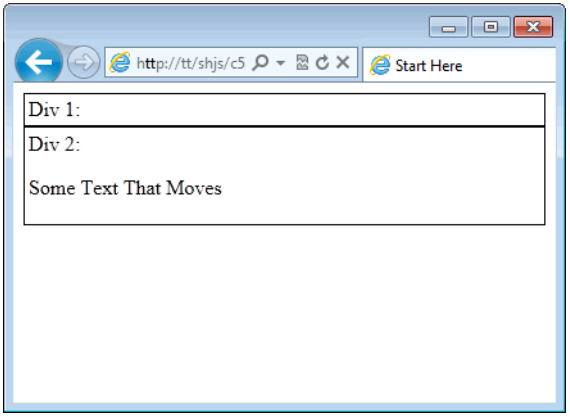


**FIGURE 5-2** By moving the mouse over the Div 1, you move the text to Div 2.

This exercise used the jQuery *hover* function, which provides much of the same functionality you typically use when building mouse-related behaviors. You can find other related functions in the jQuery manual at *http://api.jquery.com/category/events/mouse-events*. The manual includes a discussion of *mouseover*, *mouseout*, and other events.

You can also react to click events. A useful jQuery function for doing so is the *toggle()* function. Using *toggle()* on an element creates a switch-like experience, where the first click performs one action and a second click restores the original state—much like turning a switch on and then turning it back off again. The following exercise uses the *toggle()* function.

Begin with the HTML from the previous exercise. The required HTML should already be present in *index.html*; if it's not, refer back to the previous exercise. If necessary, save *index.html*. You can also find this code in the files *toggle.html* and *toggle.js* in the companion content.

1. Open *external.js*, and delete any existing code in the file.
2. Enter the following code into *external.js*:

```
$(document).ready(function() {
        $('#mover').toggle(function() {
                 $('.moveit').appendTo("#movedest");
        }, function() {
                $('.moveit').appendTo("#mover");
        });
});
```

3. Save *external.js*.
4. View *index.html* in a browser. You'll see a page similar to that from Figure 5-1.
5. Click inside Div 1. The text moves into Div 2, just as you saw in the previous exercise and as was shown in Figure 5-2.
6. Now, click inside Div 1 again. The text moves back into Div 1. If you keep clicking in Div 1, the text will flip back and forth between the two *div* elements.

The code for this exercise used the *toggle()* function. The *toggle()* function accepts two function arguments, which are essentially the thing you want *toggle()* to do. In this case, the first click caused the element with the class *moveit* to be appended to the *div* with the id *movedest*. The second function appended that same element back to the *div* with the id *mover*, causing the toggling action you can see in the browser as you click.

The *toggle()* function essentially captures mouse click events. jQuery also includes a function for capturing click events, aptly titled *click()*. The *click()* function is often used when you don't need to toggle the on/off type of action of the *toggle()* function, but rather you just want to do something when the mouse is clicked on an element.

To see *click()* in action, borrow the code from the previous exercise. Specifically, keep the same HTML and change the call to *toggle()* in *external.js* to *click()* instead. The *external.js* file should look like this:

```
$(document).ready(function() {
    $('#mover').click(function() {
         $('.moveit').appendTo("#movedest");
    });
});
```

You can also find this code in *clickmove.js* in the Chapter 5 companion content. Now load *index.html* again. This time you need to click inside Div 1 to make the text move into Div 2.

## Preventing the Default Action

When you click a link in a browser, the default action is to load whatever document or URL is contained in the link. However, there are certain times when you want to prevent that default action from occurring. For example, if a user is in the middle of a long process, you might not want him navigating away before he gets a chance to save his work. This exercise uses the *click()* function—and it also shows how to prevent the default action from occurring. You can find the code for this example in *click.html* and *click.js* in the companion content.

1. Open your StartHere project, if it's not already open.
2. Within the StartHere project, open *external.js* and delete any code within the file.
3. Place the following JavaScript in *external.js*, replacing any code in there already:

```
$(document).ready(function() {
        $('#myLink').click(function() {
                alert($('#myLink').text() + " is currently unreachable.");
        });
});
```

4. Save *external.js*.
5. Open *index.html* within the project, and delete any existing HTML from the file.
6. Place the following markup in *index.html*:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="external.js"></script>
</head>
```

```
<body>
<p><a href="http://www.braingia.org" id="myLink">braingia.org</a></p>
</body>
</html>
```

7. Save *index.html*.

8. Run the project, or view *index.html*. You'll see a page similar to Figure 5-3.

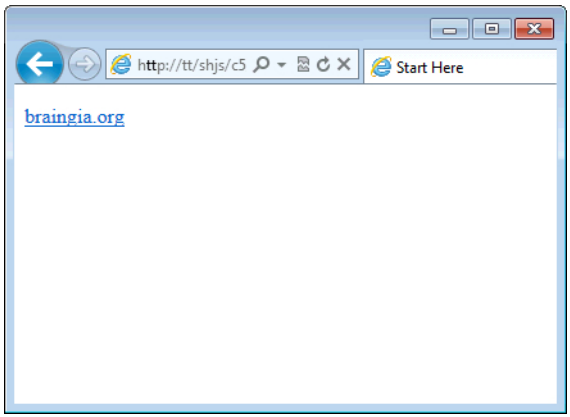


**FIGURE 5-3** Viewing the page from this exercise in a web browser.

9. Click the link. You get an alert like that shown in Figure 5-4.

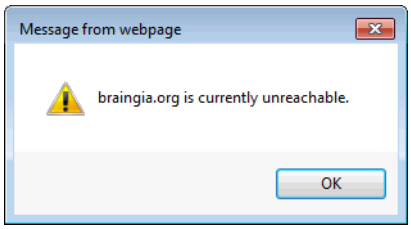


**FIGURE 5-4** The alert generated by this exercise.

10. Click OK to dismiss the alert dialog. You'll immediately notice that the browser is actually following the link from the <a> element. However, in reality, based on the message shown in the alert, I wanted to prevent the browser from following the link.

    Next you add code to prevent the browser from taking its default action and following the link. There are two ways to do that. The first way is to use the *preventDefault()* method, which is a JavaScript-standard way to prevent a default action from occurring. The second way is to return *false*. The *return false* statement prevents the default action from occurring because

you're inside a function that handles the event. Here's the code for each method. You can modify *external.js* to return *false* by adding a single line of code, shown in bold here:

```
 $(document).ready(function() {
    $('#myLink').click(function() {
        alert($('#myLink').text() + " is currently unreachable.");
        return false;
    });
});
```

11. Add *return false* in that location within *external.js*, save the file, and then view *index.html* in a browser. When you click the link, you'll notice that the alert dialog still shows up but clicking OK to clear that dialog doesn't result in the browser continuing on to the site.

    The second method, using *preventDefault()* requires adding the calling event within the function declaration. Notice the two bolded lines in this code (also found in the companion content as *preventdefault.js* in the Chapter 5 js folder):

```
$(document).ready(function() {
    $('#myLink').click(function(e) {
        alert($('#myLink').text() + " is currently unreachable.");s
        e.preventDefault();
    });
});
```

Which method to use? The answer is: it depends. The *preventDefault()* method prevents only the default action and doesn't prevent the actual event from continuing. In contrast, *return false* does both. For many actions, such as the one you saw in this example, *preventDefault()* is enough to do what you want. Additionally, if you're not inside a function, as in this example, *return false* might not work as you expect—meaning it might not work at all or might stop the entire program from continuing.

***See Also*** *For more information on event handling and jQuery, see* http://api.jquery.com/event.preventDefault *and* http://api.jquery.com/event.stopPropagation.

## Attaching to an Element with *On*

The click event handler and other event handlers in jQuery work quite well for most uses. There are times, however, when you might need to attach an event handler to something that doesn't yet exist on the page. For example, if part of your application adds or creates new elements on the page, you can't connect the *click* event or anything else to those elements (easily). Fortunately, jQuery has a function called *on()* that binds an event to elements on the page. In the next section, you'll see the *on()* function used to prevent a form submission.

# Validating Web Forms with jQuery

One of JavaScript's earliest uses was providing validation on forms that visitors filled out on the web. Because JavaScript runs in the user's browser, any invalid form elements or fields that were required to be filled in could be checked locally on the user's computer, without having to send the form's contents all the way back to the server. This was especially important in the days of dial-up modems, where transferring that data back and forth took a noticeable amount of time. This section examines some of the ways to validate user input in a web form.

What exactly makes a web form or its contents valid depends entirely on what you define as valid. For example, you could require that to be valid, the contents of an email address field must contain an @ symbol, or that a phone number field must contain a certain number of digits or even a specific format. You can also check to make sure that required fields contain a value.

jQuery has a number of selectors related to forms. Some of these are shown in Table 5-2.

**TABLE 5-2** Common jQuery form selectors

| Selector | Syntax Example | Description |
|---|---|---|
| *:button* | *$(':button')* | Selects all buttons |
| *:checkbox* | *$(':checkbox')* | Selects all check boxes |
| *:hidden* | *$(':hidden')* | Selects hidden input fields |
| *:image* | *$(':image')* | Selects images that are buttons |
| *:input* | *$(':input')* | Selects all form elements |
| *:password* | *$(':password')* | Selects password input types |
| *:radio* | *$(':radio')* | Selects all radio buttons |
| *:reset* | *$(':reset')* | Selects reset buttons |
| *:submit* | *$(':submit')* | Selects submit buttons |
| *:text* | *$(':text')* | Selects all text input fields |

In addition to these selectors, there are also *filters*, which find form fields that are checked or selected, as is the case with radio buttons or check boxes. These filters are *:checked* and *:selected*, and you'll see their use demonstrated later in this chapter.

## Validating on Submit

The simplest way to validate a web form is to check the form's contents when the user clicks on the submit button. The process essentially involves validating each field that requires validation, and stopping the default action (submit) from occurring if there are fields within the form that contain invalid or missing required data. Here's an exercise that will help illustrate simple form validation. You can find this code in the files *validsubmit.html* and *validsubmit.js* in the companion code.

1. Open your StartHere project if it's not already open.

2. Open *index.html*, delete any contents in the file, and then replace them with this code:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="external.js"></script>
</head>
<body>
<form id="myForm" name="myForm" action="#" method="POST">
<fieldset>
        <label for="customerName">Name:* </label>
        <input type="text" name="customerName" id="customerName">
        <br />
        <label for="emailAddress">E-mail:* </label>
        <input type="text" name="emailAddress" id="emailAddress">
        <br />
        <hr>
        <label for="musicFormat">Order Type: </label><br />
        Carryout: <input type="radio" name="orderType" value="carryout"><br />
        Delivery: <input type="radio" name="orderType" value="delivery">
        <br />
        <br />
        <label for="pizzaToppings">Pizza Toppings: </label><br />
        The Works! <input type="checkbox" name="toppings" id="works"
                    value="works"><br />
        Mushrooms: <input type="checkbox" name="toppings" id="mush"
                    value="mush"><br />
        Peppers: <input type="checkbox" name="toppings" id="peppers"
                  value="peppers"><br />
        Sausage: <input type="checkbox" name="toppings" id="sausage"
                  value="sausage"><br />
        <br />
        <br />
        <label for="crust">Crust Type: </label>
        <select name="crust">
        <option value="">Please choose...</option>

        <option value="thin">Thin</option>
```

```
        <option value="deepdish">Deep Dish</option>
        </select>
        <br />
        <hr>
        <input type="submit" name="submitForm">
        <p>* - Indicates required field</p>
</fieldset>
</form>
</body>
</html>
```

3. Save *index.html*.

4. Open *external.js*, delete any contents, and place the following JavaScript into the file:

```
$(document).ready(function() {
    $('#myForm').submit(function() {
        var formError = false;
        if ($('#customerName').val() == "") {
            formError = true;
        }
        else if ($('#emailAddress').val() == "") {
            formError = true;
        }

        if (formError) {
            alert("One or more required fields missing.");
            return false;
        } else {
            alert("Form valid, submitting to server.");
            return true;
        }
    });
});
```

5. Save *external.js*.

6. Now run the project or view *index.html* in a browser. You'll see a form like the one shown in Figure 5-5.

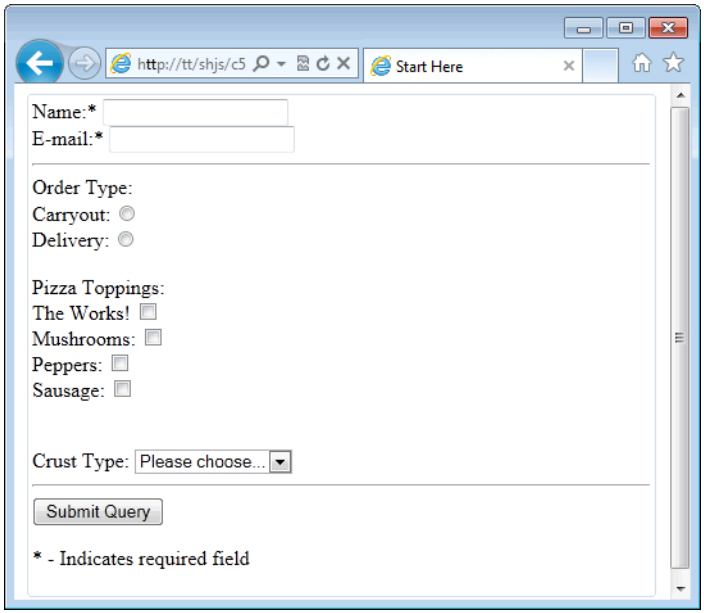


**FIGURE 5-5** An example HTML form.

7. Without filling in any form details, click the Submit Query button. You'll be presented with an alert indicating that the form is missing required fields, as depicted in Figure 5-6.

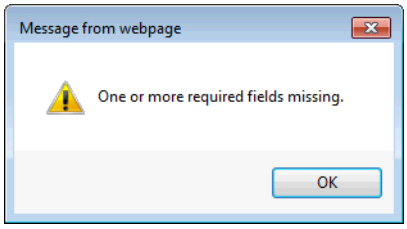


**FIGURE 5-6** Basic form validation.

8. Fill something into both of the text fields (Name and E-mail) and click submit. You'll be shown an alert indicating that the form has been successfully validated and will be submitted to the server. This is shown in Figure 5-7, but because there's no action on the form, the form's data won't go anywhere.

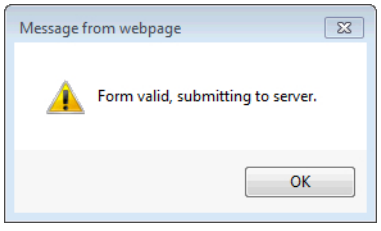


**FIGURE 5-7** The form has been validated.

9. Take some time and work with this validation exercise, clearing out the form and just using spaces instead of words, for example.

Although this code doesn't introduce much new material, you'll find it useful to work through the JavaScript so that you can gain experience with how this example is structured. The entire example is shown within the *ready()* function, which you already saw.

First, a function is attached to the *submit()* event. The *submit()* event is just like other events you've seen in this chapter except that it's attached to a form (because it doesn't make sense anywhere else):

```
$('#myForm').submit(function() {
```

Within the *submit()* function, a variable is created to store the state of the validation. This variable, called *formError*, is set to *false*:

```
    var formError = false;
```

Next, the *submit()* event's function performs a simple check to test whether the user entered values into the text box elements. This is accomplished through an *if* conditional that uses the jQuery *val()* function. The *val()* function retrieves the filled-in value for a form field. In this case, if either of the form elements is blank (empty), the code sets the *formError* variable to *true*.

```
        if ($('#customerName').val() == "") {
            formError = true;
        }
        else if ($('#emailAddress').val() == "") {
            formError = true;
        }
```

At the end of the *submit()* function, the *formError* variable is evaluated. If *formError* is found to be *true*, the page shows an alert indicating that one or more required fields haven't been filled in. Additionally, and more importantly, the function returns *false*, which—as you already saw—means that the default action for the form will be cancelled, and it won't be submitted until the user corrects the error and submits the form again for validation.

```
        if (formError) {
            alert("One or more required fields missing.");
            return false;
        } else {
            alert("Form valid, submitting to server.");
            return true;
        }
```

With this code explained, you're probably already counting the ways in which invalid data could still get back to the server. Luckily, the server program is always required to validate input data as well! Right now, a simple empty space will suffice to make the fields valid, which is probably not what you want for either a Customer Name or E-mail Address form field. There's also no validation on the E-mail Address field itself. Making sure there's at least an @ symbol in the email address would be a good start. Therefore, I'll show some examples to improve on the validation.

## Regular Expressions

Being new to programming, you might have never heard of the term *regular expression*. It's a term used to describe text processing and matching by using symbols to represent characters. Regular expressions make it possible to validate that an email address contains an @ symbol, that a phone number field contains only numbers (and maybe dashes or parentheses), that date fields are valid, and much, much more.

Entire books have been written on regular expressions, and a lot of information is available on the Internet about them as well. This section (and indeed this book) won't try to introduce regular expressions beyond the cursory glance you get here. What this section does, however, is show you that there are advanced validation techniques available. It's good to have these techniques in mind for further research and learning.

Regular expressions are particularly helpful for form validation, and it's that limited focus that you'll see here. In JavaScript, regular expressions are objects you can define in two ways:

```
var myRegex = new RegExp(pattern,modifiers);
```

or

```
var myRegex = /pattern/modifiers;
```

Regular expression objects have several methods for processing text, one of which is of particular interest when validating forms: the *test()* method. Rewriting the previous validation code to use regular expressions calls for two primary changes. First, the regular expressions themselves need to be defined and then the *test()* method needs to be implemented within the conditionals.

You can find this code in the file *regex.js* in the js folder of the Chapter 5 companion content. This code will define two regular expressions, one for each text field. The first regular expression is used for the Customer Name text box on the form:

```
var customerRegex = /[\w\s]+/;
```

This regular expression creates something called a character class, which is a special character set to represent matching characters. Character classes are defined with brackets, [ and ]. The *\w* in the regular expression represents any valid letter or number (alphanumeric) and the underscore character. The *\s* represents a space character.

By placing these special characters into the character class, this regular expression will match any alphanumeric characters (and the underscore) as well as any number of spaces. The minimum number of letters or numbers is one, and there also must be one space somewhere in the string in order for this regular expression to match. So this regular expression would match even if the user entered *"b "* into the text field. However, even with this flaw, this regular expression validation is still a large improvement over the previous example, which simply checked to make sure that something was filled in.

Likewise, the regular expression used here to check an email address is also a bit crude:

```
var emailRegex = /@/;
```

The email regular expression merely looks for the presence of an @ symbol. It's tempting to try to get fancier with the email address regular expression and begin requiring one or more letters followed by an @ symbol, followed by one or more letters and then a dot, followed by more letters, as in a real email address like *suehring@braingia.com*. However, this quickly fails when considering email addresses with multiple dots in them or domain name structures that aren't common.

With the regular expressions defined, the next task is to change the conditionals that already exist in the program to use the *test()* method instead of the standard equality test used previously. The two conditionals now look like this:

```
        if (!customerRegex.test($('#customerName').val())) {
        }


        else if (!emailRegex.test($('#emailAddress').val())) {
        }
```

Essentially, the value of each field is given as the argument to the *test()* method of each regular expression. When the regular expressions match, it means someone filled in that field correctly, therefore the conditionals are negated with the exclamation point (!). In plain language, the conditionals now read, "If the customer name field doesn't contain at least one letter or number and one space, do this" and "Otherwise, if the email address field doesn't at least have an @ symbol, do this."

The final code looks like this:

```
$(document).ready(function() {
    var customerRegex = /[\w\s]+/;
    var emailRegex = /@/;
    $('#myForm').submit(function() {
           var formError = false;
        if (!customerRegex.test($('#customerName').val())) {
            formError = true;
        }
```

```
        else if (!emailRegex.test($('#emailAddress').val())) {
            formError = true;
        }

        if (formError) {
            alert("One or more required fields missing.");
            return false;
        } else {
            alert("Form valid, submitting to server.");
            return true;
        }
    });
});
```

## Other Common Matches

Table 5-3 shows some recipes for other common matches you might find useful.

**TABLE 5-3** Other common matches

| Regex | Type | Description |
|---|---|---|
| *var zipRegex = /[0-9]/;* | Digits for ZIP code | Match only digits, such as when validating a ZIP code. |
| *var zipRegex = /^[0-9]$/;* | Positional digits for ZIP code | Use special positional characters to make sure that the match works only if the ZIP contains only digits—that is, there's nothing before or after the digits. |
| *var zip5Regex = /^[0-9]{5}$/;* | Positional counts of digits for ZIP Code | Make sure that there are at least 5 digits, as in a 5 digit zip code. |
| *var zipRegex = /^[0-9]{5}\-[0-9]{4}$/;* | Positional counts of digits for ZIP+4 digits | Match a ZIP code plus 4 digits, which is a 5-digit ZIP followed by a dash and then 4 more digits. |
| *var dateRegex = /^[0-9]{4}\/[0-9]{2}\/[0-9]{2}$/;* | Positional counts of digits for the date | Match a date formatted like YYYY/MM/DD. |
| *var dateRegex = /^[0-9]{2}\/[0-9]{2}\/[0-9]{4}$/;* | Positional counts of digits for the date | Match a date formatted like MM/DD/YYYY. |
| *var urlRegex = /https?:\/\//;* | Web URL | Match a URL, either http or, optionally, https. |

**Note** These date examples don't validate that the month is less than 12 or the number of days in a given month. This regular expression happily matches a correctly formatted invalid date such as 99/99/9281 just as it would a valid date such as 01/01/2014. Also, as you can see, the forward-slash (/) character needs to be escaped, using the backslash (\) character; otherwise, it will be interpreted as the end of the regular expression.

You can plug the regular expressions from Table 5-3 into *external.js* in place of the *customerRegex* and experiment with them by filling out the Customer Name text box in the form (after reloading the page, of course).

Now you've seen the basics of how to validate required fields in a form. Next you'll see how to work with other common form elements.

## Finding the Selected Radio Button or Check Box

Radio buttons, such as those shown on the form for choosing Delivery or Carryout, are useful when you want the visitor to be able to select one, and only one, choice from a set of options. In this example, visitors can have their pizza delivered, or they can come pick it up—but they can't do both. In contrast, you use check boxes when you want users to be able to select multiple values from a list, such as with pizza toppings.

As mentioned earlier, jQuery includes a *:checked* filter for determining whether an element is checked or unchecked. You use the *:checked* filter in combination with a selector, and typically along with an additional filter, called *name*.

In the examples already shown, each text field had its own id assigned to it. However, elements such as radio buttons and check boxes all share the same name, and it can be cumbersome to assign a unique id value to each one. But by using a combination of selectors and filters correctly, you can drill down to any form element without having to give each element an id or class. Recall that this is the radio button structure in the HTML you already saw:

```
Carryout: <input type="radio" name="orderType" value="carryout"><br />
Delivery: <input type="radio" name="orderType" value="delivery"><br />
```

The code for retrieving the checked value from the *orderType* radio buttons looks like this:

```
$(':input[name="orderType"]:checked').val()
```

That single line of jQuery code retrieves the value of whichever radio button element is selected—in this case, either Carryout or Delivery. The code uses the *:input* selector, which retrieves all the form elements and then applies another selector to narrow down the results to the specific element named *orderType*. Finally, it applies the filter to retrieve only the selected element. If no element is selected, jQuery returns *undefined*.

With check boxes, multiple values can be selected. Getting the selected values means looping through the selected elements and testing each one to see if it's selected. In jQuery, you can do that in one simple call; in fact, the basic syntax for selecting selected ("checked") elements looks the same as that for determining which radio button is selected:

```
$(':input[name="toppings"]:checked')
```

However, because several elements might be selected, instead of merely using *val()* to retrieve the value as you can with the radio buttons, you use the jQuery *each()* function to loop through each selected element. The *each()* function is similar to a *for* loop. Within the *each()* function, you can then process the value of the current element by using the *this* keyword. Here's some sample code that creates an alert dialog for each selected element:

```
$(':input[name="toppings"]:checked').each(function() {
    alert($(this).val());
});
```

### Validating Radio Buttons and Check Boxes

Performing extended validation of values with regular expressions is done the same way for the values returned from radio buttons and check boxes as it is for text fields. However, it's possible to argue that extended validation of values isn't really required within JavaScript because the user doesn't need to enter those values. With that said, it's still a requirement that you validate those elements on the server because users still have the ability to change those values (and sometimes do so, just to see what will happen).

If no elements of a check box or radio button group are selected, their value is set to *undefined*. This makes it easy to test whether a user has selected any check boxes or radio buttons from a group. Here's an example for the radio buttons shown in this section that incorporates the *formError* logic from the earlier exercise:

```
if ($(':input[name="orderType"]:checked').val() == undefined) {
        formError = true;
    }
```

The process for validating check boxes is exactly the same. Note that you don't need to do this inside of an *each()* loop; in fact, you want to perform this validation before you bother looping through the values with *each()*.

```
if ($(':input[name="toppings"]:checked').val() == undefined) {
        formError = true;
    }
```

With radio buttons and check boxes done, next you'll see how to work with select or drop-down boxes.

## Determining the Selected Drop-Down Element

Another common form element is a drop-down box, known as a <select> in HTML terms. The example form uses a drop-down box so that users can select the crust type for their pizza. The crust can be one of multiple options, but it can't be more than one. Generally, radio buttons are good when there

are only a few options that can be shown on the page easily, while drop-down lists are better when there are many options, such as when choosing a U.S. state (such as Wisconsin, California, or Hawaii) from among all the states.

The example HTML shown used a <select> element that presented two options for the type of pizza crust:

```
<select name="crust">
<option value="">Please choose...</option>
<option value="thin">Thin</option>
<option value="deepdish">Deep Dish</option>
</select>
```

You can determine which drop-down element the user selected with the *:selected* filter. The syntax is the same as that for determining the status of check boxes and radio buttons:

```
$(':input[name="crust"] :selected').val()
```

The obvious difference is that the preceding code uses the *:selected* filter rather than the *:checked* filter, but a not-so-obvious change is that you must include a space between the selector *:input[name="crust"]* and the filter *:selected* because the filter needs to apply to the elements below the selector (the <option> elements) rather than the <select> element itself.

**Note** This type of select box can have only one value selected at a time, but a multiselect list can return multiple values, as in the check-box example shown earlier in this section. Therefore, you need to use *each()* to get all the possible values.

## Validating <select> Elements

Validating <select> elements is different than validating the other element types you've seen. Although the concepts are the same, the implementation is different. Unlike radio buttons and check boxes, there is no undefined state for <select> elements. In the example shown, the first element, labeled Please Choose, has an empty *value* attribute. Therefore, the validation process can simply check to see whether the value is not empty. This is similar to the first example of validation you saw in this chapter, and the code looks like this:

```
if ($(':input[name="crust"] :selected').val() == "") {
    formError = true;
}
```

## Wrapping Up Form Validation

You've now seen how to obtain values from various types of form elements and how to validate them as well. Here is the code from *external.js* so far, which provides full validation for the form from the exercise earlier in this chapter. If you want to experiment with this code, remove any code in your copy of *external.js* and replace it with the code shown here. You can also find this code in the files *allbasicvalidation.html* and *allbasicvalidation.js* in the companion content.

```
$(document).ready(function() {
    var customerRegex = /[\w\s]+/;
    var emailRegex = /@/;
    $('#myForm').submit(function() {
        var formError = false;
        if ($(':input[name="orderType"]:checked').val() == undefined) {
            formError = true;
        }
        else if (!customerRegex.test($('#customerName').val())) {
            formError = true;
        }
        else if (!emailRegex.test($('#emailAddress').val())) {
            formError = true;
        }
        else if ($(':input[name="crust"] :selected').val() == "") {
            formError = true;
        }
        else if ($(':input[name="toppings"]:checked').val() == undefined) {
            formError = true;
        }
        else {
            // Here's the first place that you know the form is valid.
            // So you can do fun things like enumerate through checkboxes.
            // $(':input[name="toppings"]:checked').each(function() {
            //  alert($(this).val());
            // });
        }

        if (formError) {
            alert("One or more required fields missing.");
            return false;
        } else {
            alert("Form valid, submitting to server.");
            return true;
        }
    });
});
```

The next section revisits the *click()* event to show another common form enhancement.

## The *click* Event Revisited

One feature found on many forms is the ability to react to the *click* event. In the example form used in this chapter, when the visitor selects The Works! as a pizza topping, all the related check boxes for toppings should become selected. Conversely, if a user clears any individual toppings, The Works! option should be cleared. You can perform this task in multiple ways. The method I'll show uses the jQuery *prop()* function.

The *prop()* function sets a property of an element. In this case, you want to set the *checked* property of several check boxes. Alternatively, if one of the other elements is left clear, you want to remove the *checked* property from the The Works! check box. You can accomplish this using the *removeProp()* function.

The other piece of this puzzle is to add a handler to the *click* event for each of the check boxes. You accomplish this with the *on()* function that you've already seen. Here's an exercise that implements this functionality. You can find this code in the companion content in the files *click-on.html* and *click-on.js*.

1. Open your StartHere project if it's not already open.
2. Open *external.js*, and clear any existing code from that file. Alternatively, you can add the code shown in bold to your existing *external.js*.
3. Within *external.js*, place the following code:

```
$(document).ready(function() {

    function theWorks(elm) {
        if (elm.val() == "works") {
            $('#mush').prop("checked","checked");
            $('#peppers').prop("checked","checked");
            $('#sausage').prop("checked","checked");
        } else {
            $('#works').removeProp("checked");
        }
    } //end function theWorks

  $(':input[name="toppings"]').on("click", function() {
        theWorks($(this));
    });

    // Validation Code
    var customerRegex = /[\w\s]+/;
    var emailRegex = /@/;
    $('#myForm').submit(function() {
        var formError = false;
        if ($(':input[name="orderType"]:checked').val() == undefined) {
            formError = true;
```

```
        }
        else if (!customerRegex.test($('#customerName').val())) {
            formError = true;
        }
        else if (!emailRegex.test($('#emailAddress').val())) {
            formError = true;
        }
        else if ($(':input[name="crust"] :selected').val() == "") {
            formError = true;
        }
        else if ($(':input[name="toppings"]:checked').val() == undefined) {
            formError = true;
        }
        else {
            // Here's the first place that you know the form is valid.
            // So you can do fun things like enumerate through checkboxes.
            // $(':input[name="toppings"]:checked').each(function() {
            //  alert($(this).val());
            // });
        }
        if (formError) {
            alert("One or more required fields missing.");
            return false;
        } else {
            alert("Form valid, submitting to server.");
            return true;
        }
    });
});
```

4. Save *external.js*
5. View *index.html* in a browser. You're presented with the same form you saw back in Figure 5-5.
6. Within the Pizza Toppings section, select The Works! All the check boxes should now automatically be selected, as illustrated in Figure 5-8.

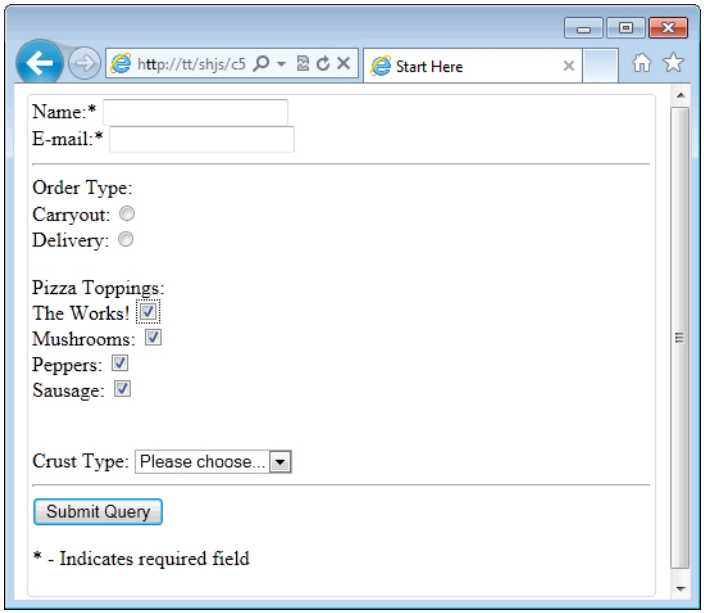


**FIGURE 5-8** Selecting The Works! automatically selects the check boxes for other toppings.

**7.** Clear the check boxes for Mushrooms (or Peppers or Sausage). Notice that the check box for The Works! also is cleared automatically, as shown in Figure 5-9.

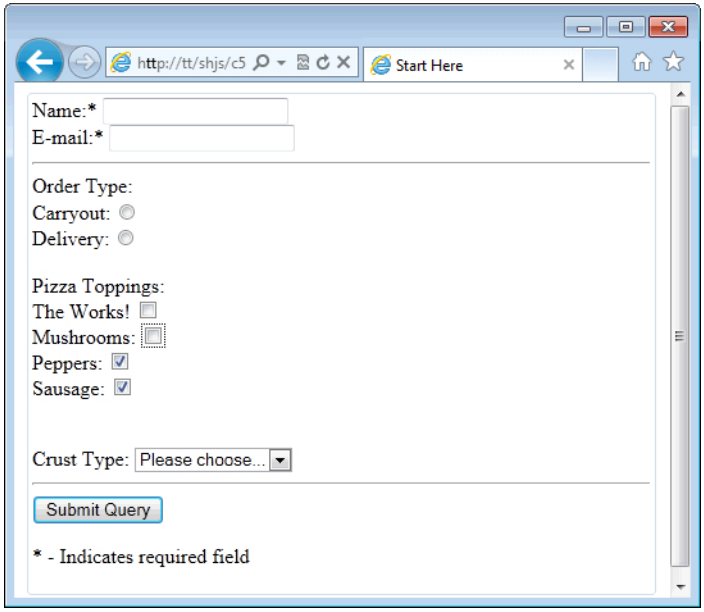


**FIGURE 5-9** Clearing any of the other toppings also clears The Works! because this pizza no longer has everything on it.

This exercise added to the previously explained validation code by adding a new function and a *click* event handler that uses the jQuery *on()* function that was explained previously in this chapter. Here's an explanation of this code:

```
function theWorks(elm) {
    if (elm.val() == "works") {
        $('#mush').prop("checked","checked");
        $('#peppers').prop("checked","checked");
        $('#sausage').prop("checked","checked");
    } else {
        $('#works').removeProp("checked");
    }
} //end function theWorks
```

The code creates a function called *theWorks*, which accepts one argument, *elm*—the element that was selected. Inside this function, if the value (*val()*) of the element is *works* (from the form's value), the *checked* property is enabled on each of the other toppings using the jQuery *prop()* function.

On the other hand, if the element's value is not *works*—in other words, if one of the other toppings was selected—The Works should be cleared. The code accomplishes with the *removeProp()* function, which removes the *checked* property from the element with the id *works*.

**Note** There are multiple ways to accomplish this same task, the one shown here is straightforward, easy to explain, and hopefully easy to understand.

Next the code adds a *click* event handler using the jQuery *on()* function. This is applied to all input elements with the name *toppings*. The *click* event handler does only one thing: it calls the function called *theWorks* and sends along the selected element to that function.

```
$(':input[name="toppings"]').on("click", function() {
      theWorks($(this));
  });
```

You've now seen how to validate several types of form elements, and you've also seen an extended example of the *click()* event. The final section in this chapter briefly illustrates the *keypress* event.

## Keyboard Events and Forms

You can access and handle several keyboard-related events with JavaScript. The *keypress* event is triggered when a key on the keyboard is pressed and a character is inserted. You can use related events, called *keydown* events, to detect when a key has been depressed, and you can use *keyup* events to detect when the key has been released. Keyboard events are used in various places in forms, such as for a type-ahead search box or to count characters in a text field.

This section shows you how to create a new form with a *text area* in it. Text areas are frequently used on message boards, forums, and other places where users can enter text and a character or length limit is imposed. On these forms, you sometimes see a Characters Remaining counter that counts down as you type. Twitter provides an example of this behavior. When entering a tweet, the characters remaining are counted down. Here's an exercise for building the similar behavior. This code can be found in the companion content as *char.html* and *char.js*.

1. Open your StartHere project if it's not already open.
2. Open *index.html*, and remove any existing HTML within the file.
3. Place the following HTML in *index.html*:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="external.js"></script>
</head>
<body>
<form id="myForm" name="myForm" action="#" method="POST">
<fieldset>
        <label for="messageText">Message: </label><br />
        <textarea rows="5" cols="20" name="messageText" id="messageText">
        </textarea>
        <br />
        <p>Characters Remaining: <span id="charRemain">100</span></p>
        <br />
        <hr>
        <input type="submit" name="submitForm">
</fieldset>
</form>
</body>
</html>
```

4. Save *index.html*.
5. Open *external.js*, and delete any code in the file.
6. Place the following JavaScript in *external.js*:

```
$(document).ready(function() {
        var charTotal = 100;
        $(':input[name="messageText"]').on("keyup",function() {
                var curLength = $(':input[name="messageText"]').val().length;
                var charRemaining = charTotal - curLength;
                $('#charRemain').text(charRemaining);
        });
});
```

7. Save *external.js*.
8. View *index.html* in a browser. You'll see a page like the one in Figure 5-10.

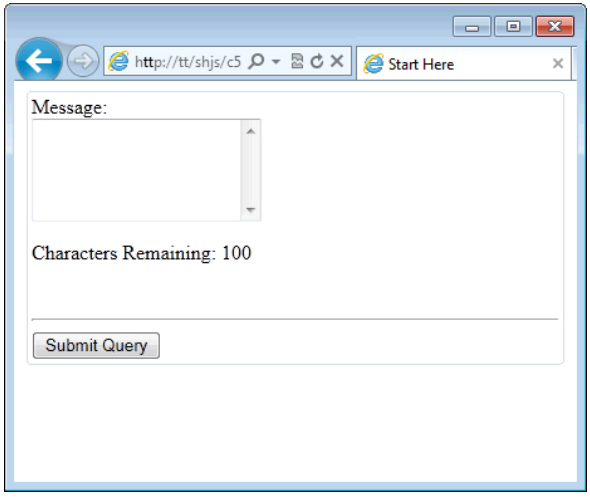


**FIGURE 5-10** Viewing the page in a web browser.

9. Begin typing into the Message text area. As you release each key, the counter will decrement, showing the number of characters remaining that you can type.

The code in this example, located inside the jQuery *ready()* function, uses a combination of jQuery and JavaScript to accomplish the task:

```
var charTotal = 100;
$(':input[name="messageText"]').on("keyup",function() {
        var curLength = $(':input[name="messageText"]').val().length;
        var charRemaining = charTotal - curLength;
        $('#charRemain').text(charRemaining);
});
```

First, the total number of allowed characters is assigned to a variable named *charTotal*. Next, the *keyup* event is bound to the text area with the name *messageText*. The *messageText* element is retrieved using the *:input* and *name* selectors you saw earlier in the chapter, and the *keyup* event is tied to that element with the jQuery *on()* function you already saw as well.

For the *keyup* event handler, a function is defined that first obtains the current length of that same text area's value (the text entered by the user). The value is retrieved with the *val()* function you also already saw. Then the code obtains the length of that value using the *length* property. This length is placed in a variable named *curLength*.

The *curLength* variable is then subtracted from the *charTotal* variable and placed in the variable called *charRemaining*. Finally, the *charRemain* element from the HTML form is retrieved using its id (*charRemain*), and the value in the *charRemaining* variable is placed on the HTML form by using the jQuery *text()* function.

Like the end of previous chapters, this final example uses a more complex example you can use in the real world. As you continue learning JavaScript, this example and its explanation can be used as a reference.

## Summary

This chapter looked at form validation and events, which are things that happen and that you can react to in order to perform actions (or sometimes intentionally ignore). Mouse movements, clicks, and keypresses are common events, as are form submissions.

You can react to events in any number of ways, and in the chapter you saw how to change HTML elements when the mouse moved over them and how to react to mouse clicks. After that, the chapter looked at form validation, adding basic validation when the form was submitted. This basic validation was expanded upon with the help of simple regular expressions. Next the chapter showed how to obtain values from form fields using the *val()* function and how to validate other types of form elements, such as radio buttons, drop-down lists, and check boxes.

An extended example was shown of the *click* event, which detailed how to set the *checked* property on check boxes as a reaction to a different check box being selected. This was accomplished with the *prop()* and *removeProp()* functions. Finally, keyboard events were discussed with an exercise that showed how to count down the number of characters remaining as characters were typed into a text area that had a limit on the number of characters it could accept.

Chapter 6

# Getting Data into JavaScript

**After completing this chapter, you will be able to**

- Understand the different ways to talk to servers
- See how to retrieve information from a server
- Use a JavaScript program to parse information in different formats
- Use JavaScript and jQuery to send data to a server

**YOU CAN LOOK JUST ABOUT ANYWHERE ON THE WEB** to find examples of sites that marry data with JavaScript. Sites like Google Maps, Twitter, Bing, and others all use JavaScript to fetch results, help you search, and help you click and drag a map. The data itself isn't stored in JavaScript; rather, it's retrieved using a combination of JavaScript and one or more programs on a server. In this way, web applications obtain data from the server without ever having to refresh the page. From the visitor's perspective, the web application operates much like a traditional desktop application, providing real-time feedback as the visitor interacts with the webpage.

All of this is accomplished with the help of a concept called AJAX. AJAX is an acronym for Asynchronous JavaScript and XML (Extensible Markup Language), and when it came about around 2005, it revolutionized the web experience.

This chapter looks at data retrieval with JavaScript using AJAX.

## AJAX in Brief

AJAX enables retrieval of data and content using only JavaScript. Prior to the advent of AJAX, when data needed to be retrieved from a server, the user typically needed to click a button to initiate the transfer. AJAX did away with that interaction, instead using a special function to send the request to

the server in the background. Unlike HTML, CSS, and JavaScript, AJAX isn't a specification or standard; rather, it's a concept or technique for data retrieval using JavaScript.

Though the X in AJAX indicates XML, there is no requirement that data needs to be formatted in XML, and this is a good thing. XML is very heavy, meaning that it requires a great deal of verbose and unnecessary syntax just to transfer simple data. Luckily, there are other ways to transfer data that are lightweight and more appropriate for transfer between the user and the server and back again. Later in the chapter, you'll see another way: JavaScript Object Notation (JSON).

In a typical AJAX scenario, the client's web browser uses JavaScript to send a request to a server using a JavaScript object called *XMLHttpRequest*. The server then returns the appropriate data, and the JavaScript program parses the response to do whatever action is required, such as move a map in response to a user zooming in or out, loading mail, or whatever is appropriate for that web application.

## On Servers, *GET*s, and *POST*s

AJAX retrieves data from a server-side program. The program on the server responds when your JavaScript program calls it. The server-side program can do whatever it is that server-side programs do—retrieve data from a database, return content, perform math, or do whatever is required. For AJAX to work, the server program needs to exist in the same domain as the JavaScript that calls it. This means that, from the JavaScript program written for this chapter, you can't call a server program on my website using an AJAX method.

The web operates using a protocol named Hypertext Transfer Protocol (HTTP), which is defined primarily by Request for Comments (RFC) number 2616 (*http://www.rfc-editor.org/rfc/rfc2616.txt*). The HTTP standard defines several methods for interacting with a server. These methods are used to send and retrieve data between the client (typically, the web browser) and the server (typically, Apache but also Microsoft Internet Information Services and others).

Two methods you see (and use) every day are the *GET* method and the *POST* method. When you request a webpage in your browser, the browser sends a *GET* request to the server in your browser's address bar. For most webpages, many more additional *GET* requests are sent to the server for each additional resource like images, JavaScript, and CSS. The server responds to each and every one of these *GET* requests with a status code and the result of the request, such as the JavaScript or image file itself.

The *POST* method is frequently used when working with web forms. The *POST* method sends the data from the web form back inside of the request itself rather than as part of the URL, as would be the case with a form that uses the *GET* method.

All of this information ties directly into programming JavaScript and AJAX. When programming AJAX, you need to create an HTTP connection using JavaScript and then send a request using either *GET* or *POST* to interact with a server program. It might sound complex, but the difficulty has largely been solved with jQuery. The next section shows how to create an AJAX request with JavaScript. Later in the chapter, you'll see more complex examples using jQuery.

As part of this chapter, I'll show a simple server-side program in both Microsoft Visual Basic .NET for those running Microsoft Visual Studio and in PHP as well. You'll be able to use that program for the examples in this chapter. Setting up a web server is beyond the scope of this book. However, Visual Studio includes a development server that will be used for the examples in this chapter and throughout the book. If you don't have Visual Studio, I recommend obtaining an Apache installation, which can run on Windows, Linux, or Mac. Another option is to use a hosting provider that offers PHP. These providers usually offer free setup and cost less than $5 a month.

## Building a Server Program

This section shows how to build a simple server program. A server program is necessary when working with AJAX. This program will be used for the initial example in this chapter, and you'll change it as you follow other exercises in this chapter. This program is an ASP.NET page that simply echoes a few values back and can be found as *WebForm1.aspx* in the companion code. I'll also show an example of the same code in PHP.

1. To begin this exercise, open your StartHere project in Visual Studio.
2. Within the StartHere project, right-click the StartHere project within Solution Explorer, click Add, and then click New Item. The Add New Item dialog will be shown, as shown in Figure 6-1.

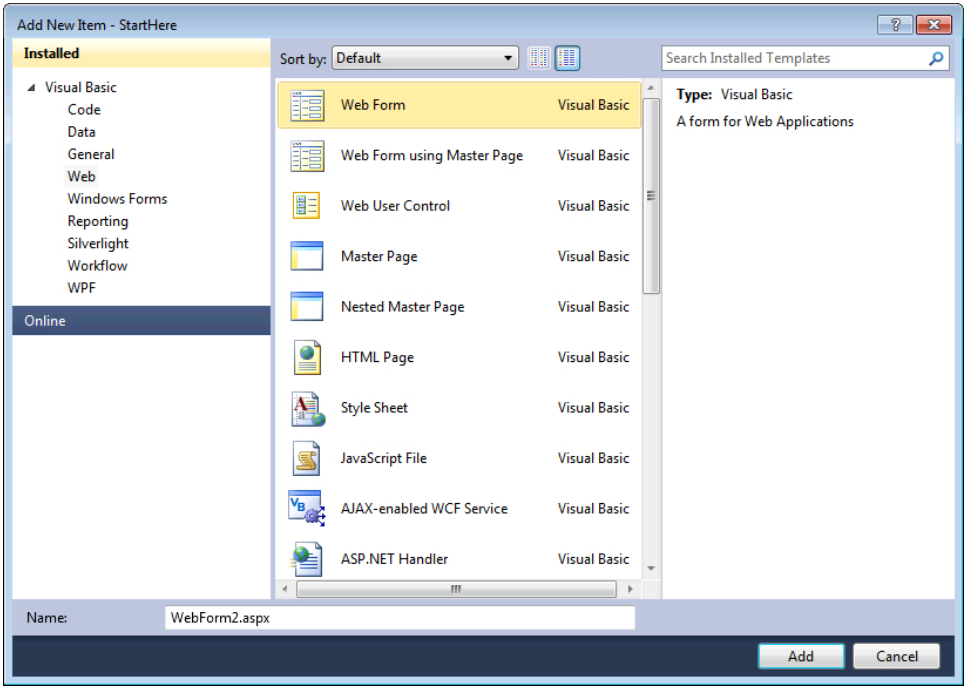


**FIGURE 6-1** Adding a new item to the StartHere project.

3. In the Add New Item dialog, select Web Form from the Web templates and click Add. A new web form will be created, similar to the one in Figure 6-2.

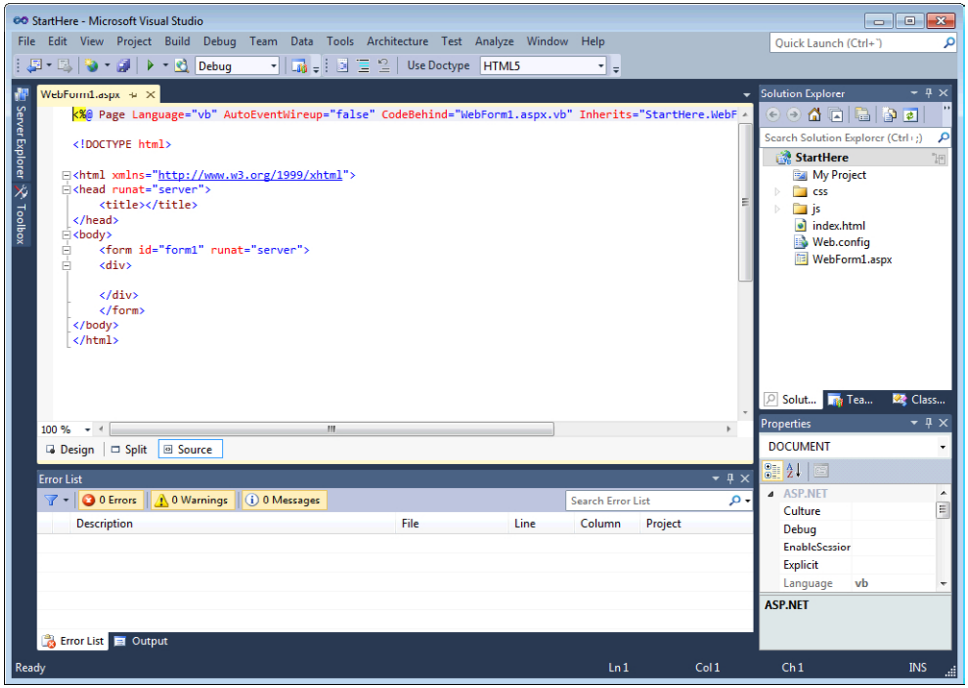


**FIGURE 6-2** A new web form in Visual Studio.

4. Inside of the new web form, delete everything, and replace it with the following code:

```
<%
Response.Write("Maple|Pine|Oak|Ash")
%>
```

The result should look like that shown in Figure 6-3.

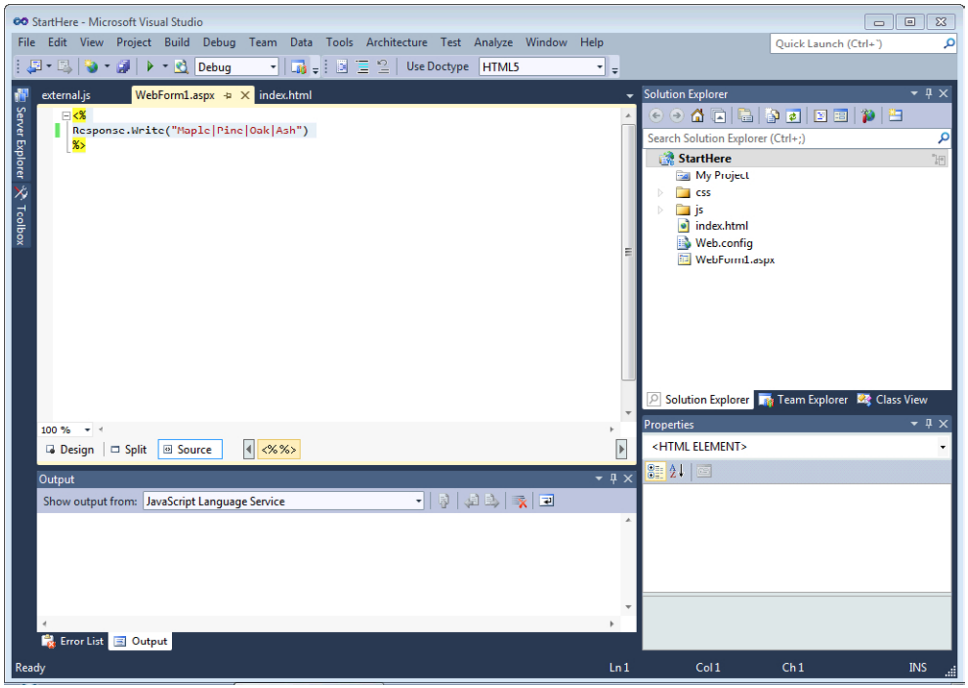


**FIGURE 6-3** The basic server-side program used for this chapter's examples.

5. Within Solution Explorer, right-click *WebForm1.aspx* and select View In Browser from the context menu. You'll be presented with a page like the one in Figure 6-4.

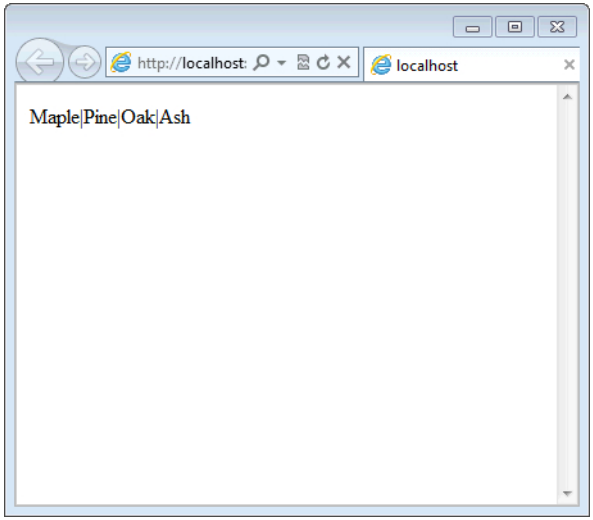


**FIGURE 6-4** *WebForm1.aspx* file viewed through the web browser.

With this code built, you now can create an AJAX implementation that runs on your local development computer.

## PHP Example

If you are using PHP, here's the source file to create a similar server program using PHP. This code can be found as *webform1.php* in the companion code:

```
<?php

echo "Maple|Pine|Oak|Ash";

?>
```

### Determine Your Development Server Port

Visual Studio creates a local development server that you use seamlessly when developing and running your applications locally. Visual Studio chooses a port for the server to run on, aside from the standard HTTP port 80. For the examples in the book, you need to find out what port your local server is using. To do so, open your StartHere project and select the StartHere project within Solution Explorer. The Project Properties pane is displayed in the lower right corner. Within that Project Properties pane, shown in Figure 6-5, there is a property called URL.

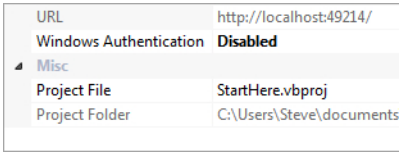


**FIGURE 6-5** The port chosen by Visual Studio on my computer is 49214, but yours will likely be different.

Copy that URL, especially noting the port. In the example, my local server is using port 49214 and therefore that's the URL I'll use throughout the examples in this chapter. However, there's only a 1 in 65,536 chance that you'll share that same port. Therefore, you need to change the URL in any examples to match your value.

Also, if you're running a regular web server, such as Apache, you'll likely be using the standard port 80 for communication. Either way, you need to update the URLs to match yours, unless you happen to be using port 49214.

## AJAX and JavaScript

Developing AJAX with JavaScript uses a special object called the *XMLHttpRequest* object. This object, used with its own methods and syntax, is responsible for sending requests to web servers to retrieve data.

Some slight differences between web browsers can make the *XMLHttpRequest* object challenging to use in the real world. This is largely solved by using jQuery, and in the next section you'll see how to use jQuery to make AJAX requests. The *XMLHttpRequest* object is instantiated like most other JavaScript objects, with the *new* keyword:

```
var myXHR = new XMLHttpRequest;
```

With the object created, you next create the connection, which involves calling the *open()* method, specifying the HTTP method (such as *GET* or *POST*), and specifying the URL. For example:

```
myXHR.open("GET","http://localhost:49214/WebForm1.aspx");
```

For *GET* methods, the next thing to do is send the data to the server, which is done with the aptly titled *send()* method:

```
myXHR.send();
```

The *send()* method can accept additional data, as might be the case when you use a *POST* method. With the request sent, the next step is to parse the response. As is the case with the example server program you created earlier in this chapter, you receive a text-based response.

As previously stated, this can get somewhat cumbersome, which is why it's almost always preferable to use jQuery for working with AJAX. However, if you'd like more information on using JavaScript for AJAX, I recommend my intermediate-level book, *JavaScript Step by Step* (Microsoft Press, 2011).

## Retrieving Data with jQuery

If you haven't figured it out already, I'll state it again: jQuery makes AJAX easier. Well, jQuery makes a lot of JavaScript programming easier, but it really shines in the area of AJAX. jQuery's methods for working with server-side data have evolved over the years. This section examines some of the primary methods for working with data from a server using jQuery. See *http://api.jquery.com/category/ajax* for a full list of methods related to AJAX.

## Using *get()* and *post()*

jQuery defines specific methods for exchanging data with a server. This means you don't need to set up and configure an *XMLHttpRequest* object or worry about the method used and so on. The two methods examined in this section are *get()* and *post()*. Both *get()* and *post()* are shortcut functions to the overall *ajax()* function in jQuery.

### Using the *get()* and *post()* Functions

The *get()* and *post()* functions accept several arguments, including the URL to use, any data to be passed to the server program, what to do with the returned data, and the type of data to be returned. Here's a simple example using *get()*. This code can be found as *basicindex.html* and *basicjs.js* in the Chapter 6 companion content.

```
$.get('/WebForm1.aspx', function(resp) {
    alert("Response was " + resp);
});
```

This *get()* function calls a file named *WebForm1.aspx*, retrieves the data, and then sends that response to the screen through an alert, shown in Figure 6-6.

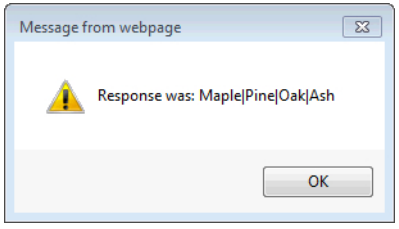


**FIGURE 6-6** Retrieving an AJAX-based alert from a server-side program using jQuery's *get()* method.

Here's a simple example using *post()*:

```
$.post('/WebForm1.aspx', function(resp) {
    alert("Response was " + resp);
});
```

The magic happens when you start sending data into the server programs and then do complex JavaScript operations with that returned data.

### Sending Data in the Request

Both *GET* and *POST* methods can send data to the server. The server then processes this data. For example, you might send in a postal code, and the server program looks up and returns the city to which the postal code belongs. You see examples of *GET* requests with additional data. This is called the *query string*. Here's an example:

```
http://www.contoso.com?zipCode=54481
```

**Note** contoso.com is a sample, nonfunctional domain. If you try this example, it won't actually do anything!

When data is sent in a *POST*, it doesn't get sent as part of the URL but as part of the HTTP request. There are two primary advantages of using *POST*. First, there is no limit on the length of the data that can be sent in a *POST* request. With *GET* requests, there are limitations on the length of the URL, and those limits differ by browser and browser version. The second advantage to *POST* requests is that it provides a tiny amount of additional security. With a *GET* request, a user can easily see the variable names and values being sent to the server and might be inclined to see what happens if those variables are changed. With a *POST* request, the values don't appear in the user's address bar, though the user still controls the values and can therefore change them.

**Note** Keep in mind that this security by obscurity does not provide any actual security. Both *GET* and *POST* requests can be changed by the user, so you still need to assume that the user sent in bad data and validate that data on the server.

## Building an Interactive Page

This exercise shows how to build elements of a web form using AJAX. It uses a server-based program to retrieve data and display it back on the page. This exercise assumes that you've completed the previous exercise for a server program to display four names of trees found in my yard. This code can be found as *inter.html* and *inter.js* in the companion content.

1. To begin the exercise, open the StartHere project.
2. Open the *index.html* file, delete any existing code in the file, and replace it with the following:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>

<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
    <form name="myForm" id="myForm" method="get">
        <label id="treeNameLabel" for="treeName">
        </label>
    </form>
</body>
</html>
```

3. Save *index.html*.

4. Open *external.js*, deleting any code in the file. Replace it with the following code, noting the name and URL used for the *WebForm1.aspx*. This might need to change based on the name you're using for the server program in your environment.

```
$(document).ready(function () {
    $.get('/WebForm1.aspx', function (resp) {
        var splitResp = resp.split("|");
        var splitRespLength = splitResp.length;
        for (var i = 0; i < splitRespLength; i++) {
            $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="' + splitResp[i] +
                    '">' + splitResp[i] + '<br />');
        }

    });
});
```

5. Save *external.js*.

6. Run this project, or view *index.html* in a web browser. You'll get a page like the one shown in Figure 6-7.

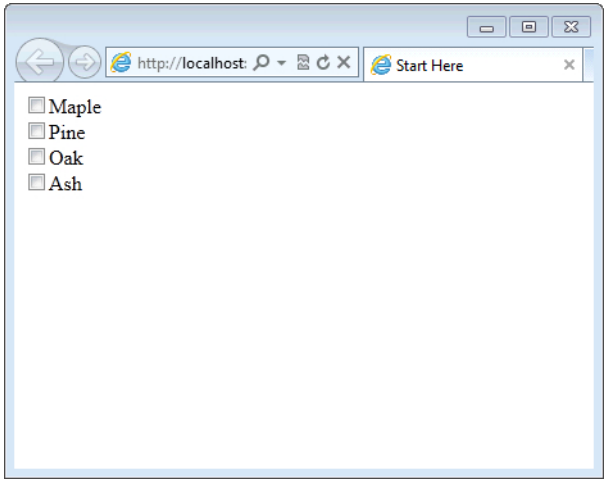


**FIGURE 6-7** Creating form elements with AJAX.

This exercise retrieved the names of trees from a server program and used that data to build check boxes on a form. I'll examine the JavaScript and jQuery used.

The first thing in *external.js* is a call to the *ready()* function, as shown in the following code block, so that the code will be executed after the DOM is available. If this isn't wrapped inside of the *ready()*

function, there's a good chance that the JavaScript will execute prior to all of the necessary form elements being available, which results in a mess indeed.

```
$(document).ready(function () {
```

The *get()* function, shown next, is called to create a *GET* request to the server with the URL /WebForm1.aspx. The response is then fed into an anonymous function that I defined.

```
    $.get('/WebForm1.aspx', function (resp) {
```

First the function splits the response, where it finds the pipe (|) character. Delimiting my source data with the pipe character makes it easy to split it like this. The results of that split are placed in a variable called *splitResp*:

```
        var splitResp = resp.split("|");
```

Because the *splitResp* variable will be used in a loop, the length of the *splitResp* array is set next into a variable called *splitRespLength*. Note that this isn't strictly necessary, but it does provide an ever-so-slight performance improvement:

```
        var splitRespLength = splitResp.length;
```

Next, a *for* loop is created. This is the heart of the code and where each form element is created:

```
        for (var i = 0; i < splitRespLength; i++) {
```

This line, split up over three lines, appends an <input> element to the label with id *treeNameLabel*. The *for* loop structure seen here is the same as previously seen throughout the book. The addition here is the jQuery *append()* function, which builds and places HTML inside of the selected element:

```
            $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="' + splitResp[i] +
                    '">' + splitResp[i] + '<br />');
        }
```

Finally, the functions are closed:

```
    });
});
```

Later in this chapter, you'll see how to send data into a server program and retrieve the results using AJAX.

## Error Handling

So far, you've seen what happens when an AJAX call works. Unfortunately, calls to server programs don't always work. This is especially true when you send data into the program, but also it could be a problem created by something as simple as the URL being incorrect or a bad Internet connection. This section looks at the simple and easy error handling available as part of the *get()* and *post()* functions in jQuery.

As previously stated, the *get()* and *post()* functions are shortcuts to the *ajax()* function. The *ajax()* function defines several additional parameters that aren't defined in the shortcut *get()* and *post()* functions. Of note is an error-handler option. However, that option is also accessible by both *get()* and *post()* as well, through the use of the *error()* function. The *error()* function is chained as part of the call to *get()* or *post()* and defines what to do when an error is encountered. Example time!

### Chaining an Error Handler

In this section, you'll add an error handler and then create an error condition to check your work. This code can be found as *errorhandler.html* and *errorhandler.js* in the companion content.

1. Open your StartHere project, and begin with the code from the previous exercise.

2. Within *external.js*, add a call to the *error()* function. This *error()* function will be called whenever there's a problem retrieving data from the AJAX server program. For this instance, you'll put a placeholder tree type (*Birch*) in the form that will be shown whenever there's an error. It's just as valid to place an error message indicating that the tree types couldn't be found instead of using this approach. Which one you choose depends on your needs. The error handling code to add looks like this:

```
.error(function () {
        $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="Birch"' +
                    '>Birch<br />');
    });
```

3. That code should be appended, or chained, to the *get()* function. The final version of *external.js* should look like this:

```
$(document).ready(function () {
    $.get('/WebForm1.aspx', function (resp) {
        var splitResp = resp.split("|");
        var splitRespLength = splitResp.length;
        for (var i = 0; i < splitRespLength; i++) {
            $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="' + splitResp[i] +
                    '">' + splitResp[i] + '<br />');
        }
```

```
    }).error(function () {
        $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="Birch"' +
                    '>Birch<br />');
    });
});
```

4. With that code in place and *external.js* saved, run the project or view *index.html*. You shouldn't notice anything different, the *get()* function should still work as it did before, and the browser should show a page like the one shown in Figure 6-7. If this doesn't work, be sure you added the *error()* function in the right place, noting specifically that there's a single dot (.) used to chain the *get()* and *error()* functions together. Next you'll intentionally create an error condition in order to test the error hander.

5. Close the browser or stop the project.

6. In *external.js*, change the URL within the *get()* function to a file that you know doesn't exist. For example, in the code shown, the URL is */WebForm1.aspx*. By changing that to *WebForm67.aspx*, an error condition will be created because *WebForm67.aspx* doesn't exist. That new line of code looks like this:

```
$.get('/WebForm67.aspx', function (resp) {
```

7. Save *external.js*.

8. Now run the project or view *index.html* again. This time you'll be presented with a browser window with only one check box, for Birch, as depicted in Figure 6-8.

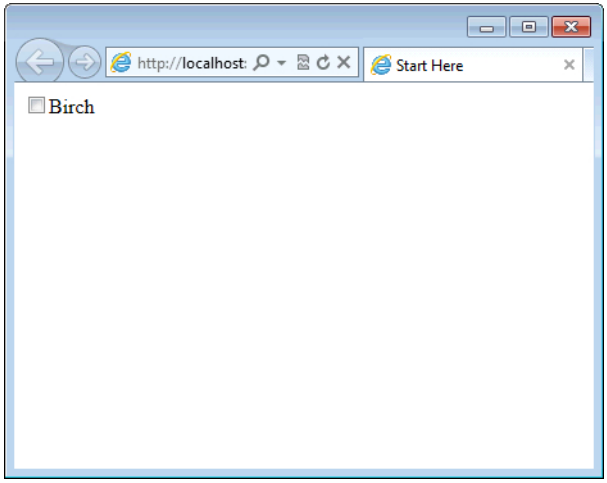


**FIGURE 6-8** The error handler is used to help when there are problems with an AJAX request.

9. Close your browser or stop the project.

10. Change *WebForm67.aspx* back to *WebForm1.aspx*, and rerun the project or view *index.html*. You'll be presented with a page like the one shown in Figure 6-7, with all of the options.

In this exercise, you added a backup plan, something to show the user when an error occurs with the AJAX call.

## Using JSON for Efficient Data Exchange

The examples you've seen so far use a simple data structure, one that can easily be represented and separated by a pipe character. However, you'll frequently need to represent more complex data structures and then consume them through AJAX. One way to pass this data back and forth is by using XML, the *X* in AJAX. However, XML requires a lot of extraneous data just to open and close data elements. For example, consider this XML to represent a person:

```
<person>
    <firstname>Steve</firstname>
    <lastname>Suehring</lastname>
    <emailAddresses>
        <primaryEmail>suehring@braingia.com</primaryEmail>
        <secondaryEmail>steve@braingia.com</secondaryEmail>
    </emailAddresses>
    <username>suehring</username>
    <lastSignon>10/19/2007</lastSignon>
</person>
```

All of the opening and closing tags for each element need to be passed from the server program to the client and then parsed. This can add up quickly, especially when considering data exchanged over slower mobile networks, not to mention the overhead to parse it on the client. Luckily, there's a better data exchange format available, known as JavaScript Object Notation (JSON). That same person structure shown with XML looks like this:

```
{
    "person": {
        "firstname": "Steve",
        "lastname": "Suehring",
        "emailAddresses": {
            "primaryEmail": "suehring@braingia.com",
            "secondaryEmail": "steve@braingia.com"
        },
        "username": "suehring",
        "lastSignon": "10/19/2007"
    }
}
```

The JSON-formatted data is much less verbose, but you don't lose any readability—you can easily still see which fields are which. Note that JSON and XML are not equivalent, though. See *http://www.json.org/fatfree.html* for more information on the two formats.

## Using *getJSON()*

jQuery includes functions to retrieve JSON from a server as well as functions to parse and work with JSON, thus making it the preferred data-exchange format for nearly all AJAX interactions.

The next exercise converts the previous exercise's code into JSON format, both on the server side and the JavaScript client side. This code can be found as *getjson.html* and *getjson.js* in the companion content, and the code ties to the server program *getjson.aspx* in the companion content.

1. Open your StartHere Project if it's not already open.
2. You'll first change the server program to return JSON-formatted data. Open *WebForm1.aspx*, removing any code that's there and replacing it with the following (also found as *getjson.aspx* in the companion content):

```
<%
    Dim str = "{ ""Trees"": { " & _
        " ""0"": ""Maple""," & _
        " ""1"": ""Pine""," & _
        " ""2"": ""Oak""," & _
        " ""3"": ""Ash""" & _
        "} }"
    Response.Write(str)
%>
```

3. Save *WebForm1.aspx*.
4. Open *external.js*, and delete any code within that file.
5. Place the following code in *external.js*:

```
$(document).ready(function () {
    $.getJSON('/WebForm1.aspx', function (resp) {
        $.each(resp["Trees"], function(element,tree) {
            $("#treeNameLabel").append('<input type="checkbox"' +
            'name="treeName" value="' + tree +
            '">' + tree + '<br />');
        })
    }).error(function () {
        $("#treeNameLabel").append('<input type="checkbox"' +
                    'name="treeName" value="Birch"' +
                    '>Birch<br />');
    });
});
```

6. Save *external.js*.

7. Run the project, or view *index.html* in your browser. You shouldn't notice any changes from the earlier exercise, and the page should look like the one shown in Figure 6-7.

Aside from the server-side program changes to send JSON-formatted data, the only substantive changes are within *external.js*. In that file, the *get()* function is changed to *getJSON()*, and the next change removes the code to split the response and loop using a *for* loop. Instead, the jQuery *each()* function is used to iterate through each of the results:

```
$.each(resp["Trees"], function(element,tree) {
```

Within the *each()* function, the same HTML code is built, with the change being that instead of the *for* loop variable, the variable is now named *tree:):*

```
    $("#treeNameLabel").append('<input type="checkbox"' +
    'name="treeName" value="' + tree +
    '">' + tree + '<br />');
})
```

Other than those changes, *external.js* remains the same, as does *index.html*.

So far, you've seen examples of both *get()* and *post()* requests, and now you've seen *getJSON()*. However, you're still missing a piece of the puzzle: how to send data to the server with AJAX.

# Sending Data to the Server

Many forms and user interactivity elements are built with the help of AJAX. Doing so frequently requires sending data to the server for processing. The server program then sends some response data back. All of the methods shown in this chapter so far (*get*, *post*, and *getJSON*) can send data to the server. This section examines how to send data to the server using *getJSON()* and *post()*. You might be asking where the example is with the *get()* method. It's exactly the same as the *getJSON()* function, so seeing *getJSON()* provides the functionality you need for both.

## Sending Data with *getJSON*

While *getJSON()* is fresh in your mind (and mine too hopefully), I'll show an example using that function. jQuery enables you to send data as an extra parameter, which is how you'll see it here. However, you can also append the data to the URL itself. Data is passed as the second parameter to the *getJSON()* function, and the *getJSON()* function automatically appends it to the URL as the first argument. For example, if you assume that a server program named *convertTemp.aspx* expects input data indicating the current temperature, the *getJSON()* call looks like this:

```
$.getJSON('convertTemp.aspx','curTemp=73',function(resp) {
```

Data can be sent as a string, as you see here, or as a map, like so:

```
$.getJSON('convertTemp.aspx', { curTemp: 73 }, function(resp) {
```

Both this and the next example with *post()* use a new server program. You'll use this program to build a web application to convert temperature between Fahrenheit and Celsius. Note you need to make a slight change to this program to use it with *post()*, and that will be shown later. This particular code is found as *tempconvert.html* and *tempconvert.js* in the companion content, and the server program is called *tempconvert.aspx*.

1. To begin this exercise, open your StartHere project.
2. Open *WebForm1.aspx*, and remove any existing code in the file.
3. Place the following code into *WebForm1.aspx*:

```
<%
    Dim inputTemp As String
    Dim inputTempType As String
    Dim outputTemp As Integer
    Dim outputTempType As String
    If IsNothing(Request.QueryString("temp")) Then
        inputTemp = ""
    Else
        inputTemp = CInt(Request.QueryString("temp"))
    End If
    If IsNothing(Request.QueryString("tempType")) Then
        inputTempType = ""
    Else
        inputTempType = Request.QueryString("tempType")
    End If
    If inputTempType = "f2c" Then

      outputTemp = (inputTemp - 32) * (5 / 9)
        outputTempType = "C"
    End If
    If inputTempType = "c2f" Then
        outputTemp = inputTemp * (9 / 5) + 32
        outputTempType = "F"
    End If
    Dim str = "{ ""Temp"": """ & outputTemp & """ " & _
        " ""TempType"": """ & outputTempType & """ " & _
        "}"
    Response.Write(str)
%>
```

This server-side program looks for input variables as part of a *GET* request. The variables are called *temp* and *tempType*, and you'll create a form with those variables next.

4. Save *WebForm1.aspx*.

5. Open *index.html*, and remove any existing HTML from the file.

6. Place the following HTML into *index.html*:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
    <h1>Temperature Conversions Galore!</h1>
    <form name="myForm" id="myForm" method="get">
        <label id="tempLabel" for="temp">
            Temperature to Convert:
            <input type="text" size="5" name="temp" id="temp" />
        </label>
        <br />
        <br />
        <label id="tempTypeLabel" for="tempType">
            Type of Conversion:
            <select name="tempType" id="tempType">
                <option value="f2c">Fahrenheit to Celsius</option>
                <option value="c2f">Celsius to Fahrenheit</option>
            </select>
        </label>
        <br />
        <br />
        <input type="submit" value="Convert!" name="submit" id="submit" />
        <br />
        <hr />
        <br />
        <span id="resultText">Result: <span id="result"></span></span>
    </form>
</body>
</html>
```

7. Save *index.html*.

8. Open *external.js*, and remove any existing code from that file.

9. Place the following code into *external.js*:

```
$(document).ready(function () {
    $('#myForm').on('submit', function () {
        var inputTemp = $('#temp').val();
        var inputTempType = $('#tempType').val();
        $.getJSON('/WebForm1.aspx',
        { "temp": inputTemp, "tempType": inputTempType },
            function (resp) {
            $("#result").text(resp["Temp"] + " " + resp["TempType"]);
        }).error(function () {
            $("#result").text('An unknown error has occurred.');
        });
        return false;
    });
});
```

10. *Save external.js.*

11. Now view *index.html* in a web browser. You'll see a page like the one in Figure 6-9.

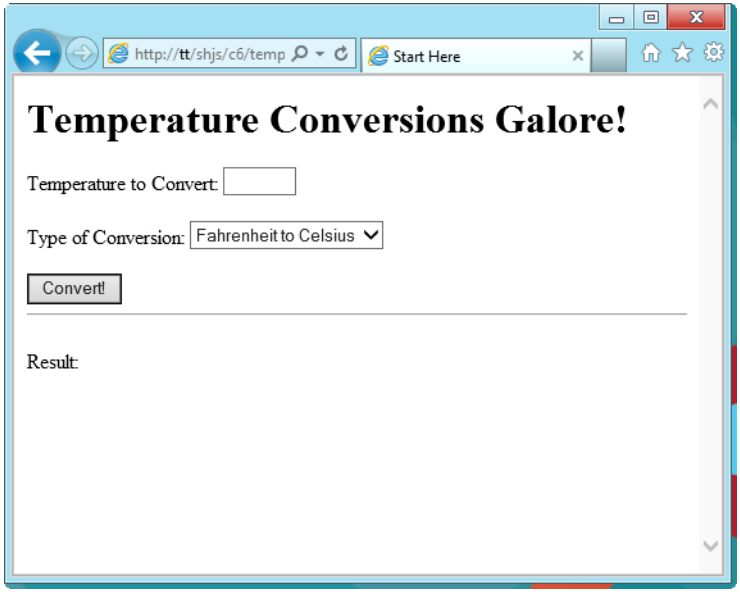


**FIGURE 6-9** The temperature conversion webpage.

12. Enter a number such as 50 into the Temperature To Convert text field, and click Convert! button.

13. You'll see the result in the Result area on the page, as shown in Figure 6-10.

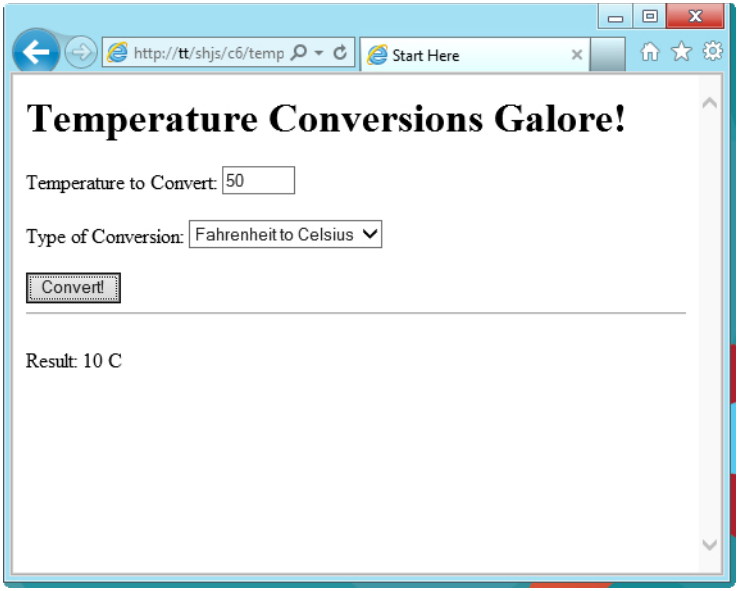


**FIGURE 6-10** A converted temperature.

14. Use the drop-down list, and enter in other temperatures to test your new temperature-conversion web application.

The JavaScript for this exercise looked like this:

```
$(document).ready(function () {
    $('#myForm').on('submit', function () {
        var inputTemp = $('#temp').val();
        var inputTempType = $('#tempType').val();
        $.getJSON('/WebForm1.aspx',
        { "temp": inputTemp, "tempType": inputTempType },
            function (resp) {
            $("#result").text(resp["Temp"] + " " + resp["TempType"]);
        }).error(function () {
            $("#result").text('An unknown error has occurred.');
        });
        return false;
    });
});
```

The first portion of this JavaScript attached a function to the *submit* event of the HTML form. This was explained in Chapter 5, "Handling Events with JavaScript," as were the next two lines, which gather the values of the form elements *temp* and *tempType*.

Next, the *getJSON()* function was called.

```
$.getJSON('/WebForm1.aspx',
```

The main difference here is that two data parameters were sent in as part of the *getJSON()* function call:

```
{ "temp": inputTemp, "tempType": inputTempType },
```

Next, the response was parsed and an error handler also was used:

```
function (resp) {
    $("#result").text(resp["Temp"] + " " + resp["TempType"]);
}).error(function () {
    $("#result").text('An unknown error has occurred.');
});
```

And finally, the default form submission behavior was prevented by returning *false* within the *submit* event:

```
return false;
```

## Sending Post Data

Using *post()* is sometimes preferred to send larger amounts of data than can be handled using the query string *GET* method. This section shows an example of using *post()* to send data to a server program and, just as importantly, how to inform the *post()* method what type of data will be returned. In this case, JSON data will be returned and *post()* needs to be told that explicitly, as you'll see. You can find the code as *tempconvert-post.html* and *tempconvert-post.js* in the companion content, and the server program is *tempconvert-post.aspx*.

1. Open your StartHere project if it's not already open.
2. If you followed the previous example, open *WebForm1.aspx* and change each of the *Request.QueryString* parameters to simply *Request*, like so:

   ```
   Request("temp")
   ```

3. Here's the full code for the *post()* version of the temperature converter, noting that the only changes are to how the data is retrieved by the server program:

   ```
   <%
       Dim inputTemp As String
       Dim inputTempType As String
       Dim outputTemp As Integer
       Dim outputTempType As String
   ```

```
    If IsNothing(Request("temp")) Then
        inputTemp = ""
    Else
        inputTemp = CInt(Request("temp"))
    End If
    If IsNothing(Request("tempType")) Then
        inputTempType = ""
    Else
        inputTempType = Request("tempType")
    End If
    If inputTempType = "f2c" Then
        outputTemp = (inputTemp - 32) * (5 / 9)
        outputTempType = "C"
    End If
    If inputTempType = "c2f" Then
        outputTemp = inputTemp * (9 / 5) + 32
        outputTempType = "F"
    End If
    Dim str = "{ ""Temp"": """ & outputTemp & """, " & _
        " ""TempType"": """ & outputTempType & """ " & _
        "}"
    Response.Write(str)
%>
```

4. Save *WebForm1.aspx*.

5. Open *index.html*, and change the form's method to *POST*, as shown here:

```
<form name="myForm" id="myForm" method="POST">
```

6. Save *index.html*.

7. Open *external.js*, and remove any code in that file.

8. Inside of *external.js*, place the following code:

```
$(document).ready(function () {
    $('#myForm').live('submit', function () {
        var inputTemp = $('#temp').val();
        var inputTempType = $('#tempType').val();
        $.post('/WebForm1.aspx',
            { "temp": inputTemp, "tempType": inputTempType },
            function (resp) {
                $("#result").text(resp["Temp"] + " " + resp["TempType"]);
            },
            "json"
        ).error(function () {
```

```
            $("#result").text('An unknown error has occurred.');
        });
        return false;
    });
});
```

9. Save *external.js*.

10. View *index.html* in a web browser. You should be presented with the same page as in the previous exercise. (See Figure 6-9.) And if you enter the same temperature, 50, you should receive the same result.

This exercise showed how to use the *post()* function to send data to a server and retrieve JSON-formatted results. The first of two substantive changes to the JavaScript used here was to change from *getJSON* to *post*:

```
$.post('/WebForm1.aspx',
```

The second change, and the one that's sometimes more difficult to place correctly, is the addition of the *dataType* parameter. That was accomplished with the string *json*, but that string needed to be placed after the *success* function, as you saw in the code. Getting all of the parentheses and braces placed correctly takes some practice and trial and error as well.

## Additional AJAX Options

All of the AJAX functions shown in this chapter—*get()*, *post()*, and *getJSON()*—are just shortcuts to the larger and more complex *ajax()* function in jQuery. The *ajax()* function can be used in place of any of these functions by setting the parameters accordingly. For example, instead of using *post()* you could use the following:

```
$.ajax({
type: "POST",
<other options here>
```

You might want to do this for coding-standards reasons or because you want to set additional AJAX options not available through the shortcut functions. See *http://api.jquery.com/jQuery.ajax* for more information on the *ajax()* function and its options.

Additionally, there is also an important function called *ajaxSetup()* you might want to use for extended features and options when using an AJAX function with jQuery. For example, I once had a problem with AJAX calls being cached by the browser. This meant that the same data was being used over and over again when, in fact, the program should've been obtaining new data from the server. To get around this, there's a *cache* option available with the AJAX functions. Setting that option to *false* forces the requests to not be cached and solves the problem.

See *http://api.jquery.com/jQuery.ajaxSetup* for more information on how to use *ajaxSetup()*.

## Summary

This chapter looked at interacting with servers using JavaScript. In the chapter, you saw the difference between the HTTP *GET* and *POST* methods and saw how to build an AJAX request for interacting with a program on a server.

You built several server-based programs in this chapter, including one to convert temperatures. The server-side programs were used by JavaScript programs that you wrote. Those programs used a combination of JavaScript and jQuery to retrieve and work with the data from the server.

The temperature conversion program built in this chapter isn't very nice-looking, but it gets the job done. In the next chapter, you'll look at how to style webpages with Cascading Style Sheets (CSS) and JavaScript to make pages look better and provide a better overall user experience.

Chapter 7

# Styling with JavaScript

**After completing this chapter, you will be able to**

- View CSS properties with JavaScript
- Set CSS properties with JavaScript
- View and set current CSS classes with JavaScript
- Use advanced effects and widgets from jQuery UI

**CASCADING STYLE SHEETS (CSS) ARE USED** to provide styling information for webpages. You see CSS all over but don't even think about it. That fancy background color on a webpage? CSS. Adding style to make web forms look good? CSS. Changing fonts and font sizes? CSS. With CSS, it's possible to create complex layouts while still making user interaction easy and intuitive.

Both JavaScript and jQuery have functions and ways in which to manipulate CSS and page styling. The chapter will primarily look at how to manipulate CSS with jQuery's functions because they are far superior to those offered through JavaScript.

## Changing Styles with JavaScript

JavaScript is frequently used to manipulate and change page styling. This is accomplished by either setting styles on HTML elements or by adding and removing CSS classes. For example, you saw how JavaScript is used to validate data entered into forms. By using CSS, you can also highlight the incorrectly filled-in form field. This section looks at styling with CSS using JavaScript and jQuery.

### CSS Revisited

In Chapter 1, "What Is JavaScript?," there was some initial coverage of CSS and its use to style webpages. In that chapter, you saw that CSS is placed onto HTML elements through selectors such as id

selectors, class selectors, and overall element selectors. Through the selectors, one or more properties—such as *font-color*, *background*, *font-size*—are then set to values to provide styling for the selected elements. Here's an example:

```
p {
font-size: 12px;
}
```

In this example, all <p> elements are selected, thus *p* is the selector. The property is *font-size*, and the value for that property is *12px*.

Applying that style to an individual paragraph through its id selector looks like this, assuming an id of *myParagraph*:

```
<p id="myParagraph">Here's some text</p>
#myParagraph { font-size: 12px; }
```

And applying a class to that same <p> tag and then applying the CSS to the class looks like this:

```
<p class="myClass" id="myParagraph">Here's some text</p>
.myClass { font-size: 12px; }
```

You use CSS classes when you need to apply the same style to a number of elements on the page, such as headings or paragraphs that you want styled all in the same way. This is more efficient than assigning an id to each element and applying those styles to each element through each id.

CSS styles also cascade, thus the name *Cascading Style Sheets*. This means that properties and their values are inherited within a page. For example, all paragraphs can be styled with a 12-px font size, certain elements can use a class to obtain a blue font, and then certain paragraphs can have borders assigned through their ids. The most specific properties, assigned through their ids, would still have a 12-px size font with blue color, inherited from their parent element's CSS.

It's typical to apply CSS through external style sheets, in much the same way you use external JavaScript files. External style sheets are connected to an HTML file with the <link> element, using this syntax:

```
<link rel="stylesheet" type="text/css" href="mycss.css" />
```

This <link> tag is placed inside of the <head> section of the page. The external style sheet contains the CSS itself—again, in much the same way that external JavaScript contains only the relevant JavaScript.

CSS can also be placed inside of the page itself and on elements themselves, but doing so is typically less favorable than using an external style sheet because external style sheets are easier to maintain and they promote reusability.

CSS is a broad and powerful language unto itself, and creating successful CSS-based layouts is a complex subject. If you'd like more information on CSS I recommend a book such as *CSS Mastery: Advanced Web Standards Solutions*, by Simon Collision, Andy Budd, and Cameron Moll (friendsofED, 2009) or the website *http://www.w3schools.com/css*.

## Changing CSS Properties

The jQuery *css()* function is used to both get and set individual CSS properties. With this function, you can select an individual element and see the value for any of its CSS properties. For example, if you have a  element styled with a red color, you could interrogate its CSS *color* property and find out that the value is set to *red*. Using this same *css()* function, you could also then change that value to something different, like green for example.

Here's an exercise that builds on the temperature-conversion program from Chapter 6, "Getting Data Into JavaScript," to add CSS and also add dynamic styling with the *css()* function.

Begin this exercise by opening your StartHere project if it's not already open. In that project, you should have files named *index.html*, *external.js* (located in the js folder), and *WebForm1.aspx*. These were created in Chapter 6. If you don't have those files, refer to the exercises in Chapter 6. You can also find these files as *exercise1.html*, *exercise1.js,* and *exercise1.aspx* in the Chapter 7 companion content. Additionally, the CSS for this exercise is found as *exercise1.css* in the Chapter 7 css folder.

1. Inside the StartHere project, right-click the css folder within Solution Explorer and select Add, New Item from the menu. The Add New Item dialog will appear.
2. Within the Add New Item dialog, shown in Figure 7-1, select Style Sheet from the Web menu.

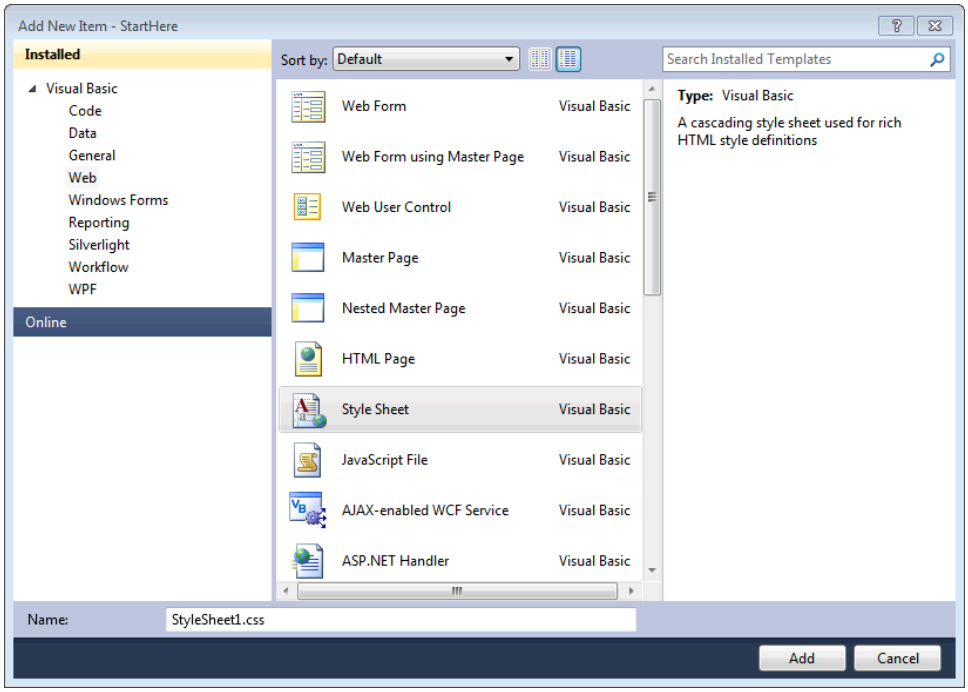


**FIGURE 7-1** Selecting a style sheet from the Add New Item dialog.

3. A style sheet will be created. By default, it is called *StyleSheet1.css*. Inside of that file, place the following CSS:

```
body {
    font-family: Arial, helvetica, sans-serif;
}

h1
{
    text-align: center;
    border: 1px solid black;
}
```

4. Save *StyleSheet1.css*.

5. Open *index.html*. Inside of *index.html*, and add a <link> element to include a style sheet:

```
<link rel="stylesheet" type="text/css" href="css/StyleSheet1.css" />
```

The <link> element should be placed within the <head> section, just below the <title> line. Here's the entire *index.html*. Note the only change made since Chapter 6 was adding that <link> element.

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
<link rel="stylesheet" type="text/css" href="css/StyleSheet1.css" />
<script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
<script type="text/javascript" src="js/external.js"></script>
</head>
<body>
    <h1>Temperature Conversions Galore!</h1>
    <form name="myForm" id="myForm" method="POST">
        <label id="tempLabel" for="temp">
            Temperature to Convert:
            <input type="text" size="5" name="temp" id="temp" />
        </label>
        <br />
        <br />
        <label id="tempTypeLabel" for="tempType">
            Type of Conversion:
            <select name="tempType" id="tempType">
                <option value="f2c">Fahrenheit to Celsius</option>
                <option value="c2f">Celsius to Fahrenheit</option>
            </select>
        </label>
        <br />
        <br />
        <input type="submit" value="Convert!" name="submit" id="submit" />
        <br />
        <hr />
        <br />
        <span id="resultText">Result: <span id="result"></span></span>
    </form>
</body>
</html>
```

6. Save *index.html*.

7. View *index.html* in a web browser by running the project. You'll see a page like that in Figure 7-2. Notice that the main <h1> element is now centered and that the fonts for the page are in the *arial*, *helvetica*, and *sans-serif* family.

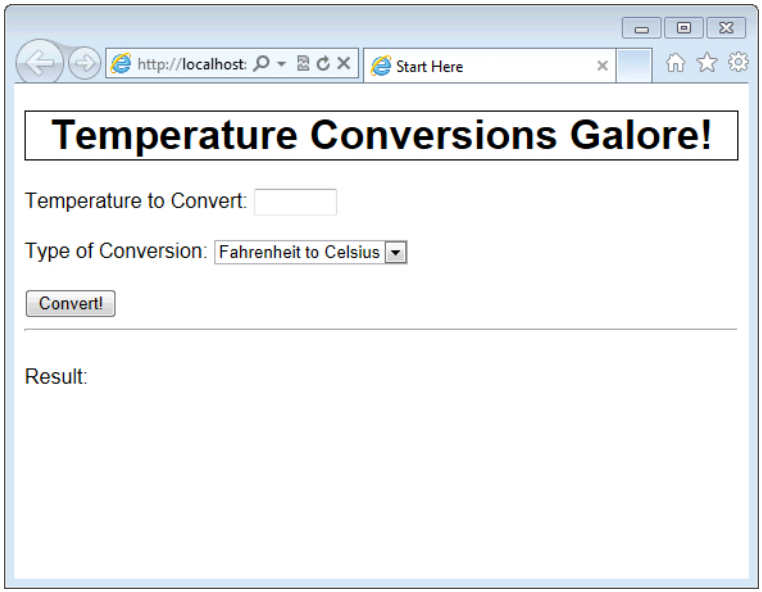


**FIGURE 7-2** The newly styled temperature-conversion form.

8. Close the browser.

9. Open *external.js*. Inside of *external.js*, you'll add some CSS styling as part of the form submission handler and also add a new click event handler. The code looks like this:

```
$(document).ready(function () {
    $('#myForm').on('submit', function () {
        var inputTemp = $('#temp').val();
        var inputTempType = $('#tempType').val();
        $.post('/WebForm1.aspx',
            { "temp": inputTemp, "tempType": inputTempType },
            function (resp) {
            $("#result").text(resp["Temp"] + " " + resp["TempType"]);
            $("#result").css("background-color", "#00FF00");
            },
            "json"
        ).error(function () {
            $("#result").text('An unknown error has occurred.');
            $("#result").css("background-color", "#FF0000");
        });
        return false;
    });
    $('#temp').on('click', function () {
        $("#result").text("");
        $("#result").css("background-color", "");
    });
});
```

10. Save *external.js*.

11. Now view *index.html* again by running it in Microsoft Visual Studio. This time type in a temperature and click the Convert! button. When a value is returned, the area with the result turns green. Additionally, when you click into the text field again, the result and its color are both cleared. This second part is accomplished with the new click event handler added to the temp text field.

In this exercise, you added styling to a web form with an external CSS file and you made changes to the fields with the help of the *css()* function. However, in doing so you now have two places where style information is set: in the external CSS file and in the JavaScript. This makes maintenance more difficult because you need to figure out where all the different styles come from and then make changes in multiple places.

There's a better way to change styles on HTML elements with JavaScript: by either adding or removing classes. This is the preferred method for styling elements. The next section shows how to use JavaScript to style with CSS classes.

# Working with CSS Classes

Classes are a primary method for applying CSS-based styles to elements on a page, and using classes is preferable when applying styles with JavaScript. jQuery has several functions for applying styles through classes using JavaScript, and this section looks specifically at the following functions:

- *hasClass()*
- *addClass()*
- *removeClass()*

## Determining Classes with *hasClass()*

Just as the *css()* function can be used to get the current status of a given CSS property, the *hasClass()* function can be used to determine if an HTML element currently has a certain CSS class applied to it. For example, you can create a class that adds a border and then apply that class' style to all the elements on a page that need a border. The CSS class definition looks like this:

```
.myBorder { border: solid 1px black; }
```

That class is then applied to an element with the class attribute, like so:

```
<h1 id="mainTitle" class="myBorder">Temperature Conversions Galore</h1>
```

Now you can use the *hasClass()* function to determine if the <h1> element has the border class. This is accomplished like so:

```
$('mainTitle').hasClass('myBorder");
```

The *hasClass()* function is helpful when you want to determine if an element is styled a certain way or has already been styled with a class.

## Adding and Removing Classes

jQuery includes functions to add classes and remove classes, aptly named *addClass()* and *removeClass()*, respectively. The class to be added needs to exist already in the CSS within the page or be linked through an external style sheet.

In the next exercise, you'll add some basic form validation to the temperature conversion application.

### Adding Error Styling to a Form

A frequent use of JavaScript is to provide real-time or near real-time feedback for form validation. You worked through form validation exercises already in the book. This next exercise highlights the field that hasn't been filled in so that the user can visually see the missing field. The code for this exercise is found in *formerror.html*, *formerror.js*, and *formerror.css* in the companion content. The code uses the same server-side program from the previous exercise, which can be found as *exercise1.aspx* in the companion content.

1. Open your StartHere project if it's not already open.
2. Within the StartHere project, open your style sheet. If you followed the previous exercise, the style sheet is located in the css folder and is called *StyleSheet1.css*.
3. Add the following class to the CSS file:

```
.formErrorClass
{
    background-color: #FF9999;
}
```

The CSS file should look like this:

```
body {
    font-family: Arial, helvetica, sans-serif;
}
```

```
h1


{
    text-align: center;
    border: 1px solid black;
}

.formErrorClass
{
    background-color: #FF9999;
}
```

4. Save *StyleSheet1.css*.

5. Open *external.js*.

6. In *external.js*, prior to the AJAX *post()* function, add the following validation logic:

```
        if (inputTemp == "") {
            $("#temp").addClass("formErrorClass");
            return false;
        } else {
            $("#temp").removeClass("formErrorClass");
        }
```

7. Within the click event handler for the *#temp* field, add a call to the *removeClass()* function:

```
$("#temp").removeClass("formErrorClass");
```

The final *external.js* should look like this:

```
$(document).ready(function () {
    $('#myForm').on('submit', function () {
        var inputTemp = $('#temp').val();
        var inputTempType = $('#tempType').val();
        if (inputTemp == "") {
            $("#temp").addClass("formErrorClass");
            return false;
        } else {
            $("#temp").removeClass("formErrorClass");
        }
        $.post('/WebForm1.aspx',
            { "temp": inputTemp, "tempType": inputTempType },
```

```
            function (resp) {
            $("#result").text(resp["Temp"] + " " + resp["TempType"]);
            $("#result").css("background-color", "#00FF00");
            },
            "json"
        ).error(function () {
            $("#result").text('An unknown error has occurred.');
            $("#result").css("background-color", "#FF0000");
        });
        return false;
    });
    $('#temp').on('click', function () {
        $("#result").text("");
        $("#result").css("background-color", "");
        $("#temp").removeClass("formErrorClass");
    });

});
```

8. Save *external.js*.

9. Now view *index.html* in your web browser.

10. Without filling in any temperature, click the Convert! button. The Temperature To Convert text box should turn a pale red color. This is shown in Figure 7-3, though the color might not show quite clearly in the book.

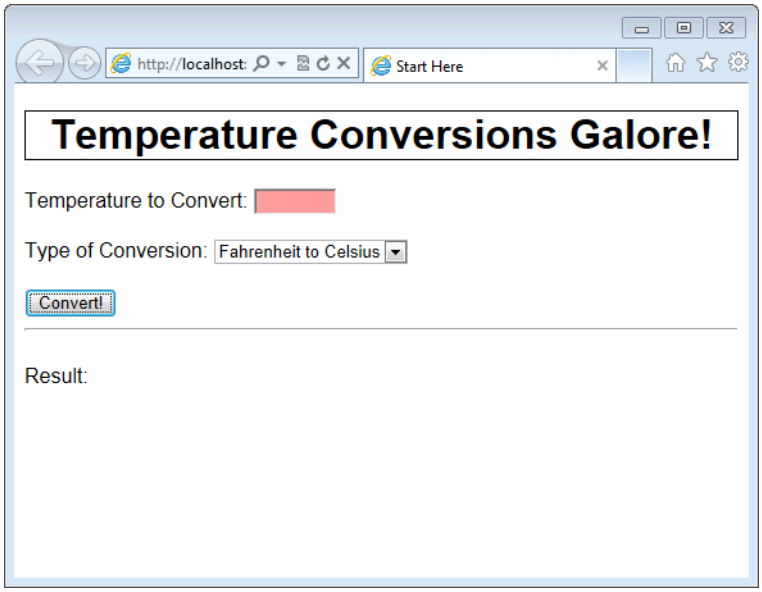


**FIGURE 7-3** Styling an error-handler class.

11. Now click into the Temperature To Convert box. The error class will be removed.

This exercise used *addClass()* and *removeClass()* to add a CSS class to a text field when the field wasn't filled in.

**Tip** One additional relevant function for working with CSS classes is the *toggleClass()* function. The *toggleClass()* function is functionally equivalent to using *addClass()* and *removeClass()*.

# Advanced Effects with jQuery UI

jQuery UI is a collection of advanced JavaScript and CSS combined to create effects and other user-interface elements. jQuery UI makes it easy to add things like pop-up calendars, drag-and-drop effects, and form autocompletion fields, and also to create advanced animated effects. In jQuery UI terms, there are *interactions* (such as drag and drop), *widgets* (such as a calendar/datepicker and tab), and *effects* (such as animation). This section looks at some of the effects available with jQuery UI and how to integrate those effects into your JavaScript. Note that the exercises in the rest of this chapter assume you have jQuery UI installed and configured. If this isn't the case, refer to the section "Getting jQuery UI" in Chapter 4, "JavaScript in a Web Browser," for an exercise used to install jQuery UI.

## Using Effects to Enhance a Web Application

jQuery UI includes numerous effects that can be applied to elements. These effects do things like slide the element in or out of the page, highlight an element, create a transfer effect, and more. This section builds a demonstration page so that you can see these effects in action. This code can be found as *effect.html*, *effect.js*, and *effect.css* in the companion content for Chapter 7.

1. Begin by opening your StartHere project.
2. Within the StartHere project, open *index.html*.
3. Clear any HTML out of *index.html*, and place the following markup there:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
        rel="stylesheet" />
    <link type="text/css" rel="stylesheet" href="css/StyleSheet1.css" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js">
        </script>
```

```
    <script type="text/javascript" src="js/external.js"></script>
</head>
<body>
    <div id="workBox">
    <div id="startHereBox">
        <div id="startHereHeader">
            <h1>Start Here</h1>
        </div>
        This is text, it goes inside the box, below the header.
    </div>
    <form action="#" method="POST">
        <select name="effect">
        <option value="bounce">Bounce</option>
        <option value="drop">Drop</option>
        <option value="explode">Explode</option>
        <option value="fade">Fade</option>
        <option value="fold">Fold</option>
        <option value="highlight">Highlight</option>
        <option value="puff">Puff</option>
        <option value="pulsate">Pulsate</option>
        <option value="shake">Shake</option>
        <option value="slide">Slide</option>
        <option value="transfer">Transfer</option>
        </select>
    <input type="submit" name="submit" value="Run Effect">
    <input type="button" name="reset" value="Reset Page">
    <span id="trash"></span>
    </form>
    </div> <!-- end workBox -->
</body>
</html>
```

4. Save *index.html*.

5. Open your CSS style sheet (typically called *StyleSheet1.css*), which should be in the css folder.

6. Delete any existing CSS from the style sheet, and replace it with this:

```
body
{
        font-family: arial, helvetica, sans-serif;
}

#workBox
{
        border: 3px solid black;
        padding: 10px;
}
```

```
#startHereHeader
{
        width: 300px;
        background: #999999;
        text-align: center;
}

#startHereBox
{
        width: 300px;
        height: 200px;
        background: #CCCCCC;
        margin-bottom: 25px;
}

.transfer
{
    border: 2px solid black;
}
```

7. Save *StyleSheet1.css*.

8. Open *external.js*, and delete any existing code from that file.

9. Place the following code inside of *external.js*:

```
$(document).ready(function () {
    $("form").on('submit', function () {
        var effect = $(":input[name=effect]").val();
        var options = {};
        var effectTime = 1000;
        if (effect === "transfer") {
            options = { to: "#trash", className: "transfer" };
            effectTime = 500;
        }
        $("#startHereBox").effect(effect, options, effectTime);
        return false;
    });
    $(":input[name=reset]").on('click', function () {
        $("#startHereBox").removeAttr("style");
    });

});
```

10. Save *external.js*.

11. View *index.html* in a web browser. You should receive a page like that in Figure 7-4.

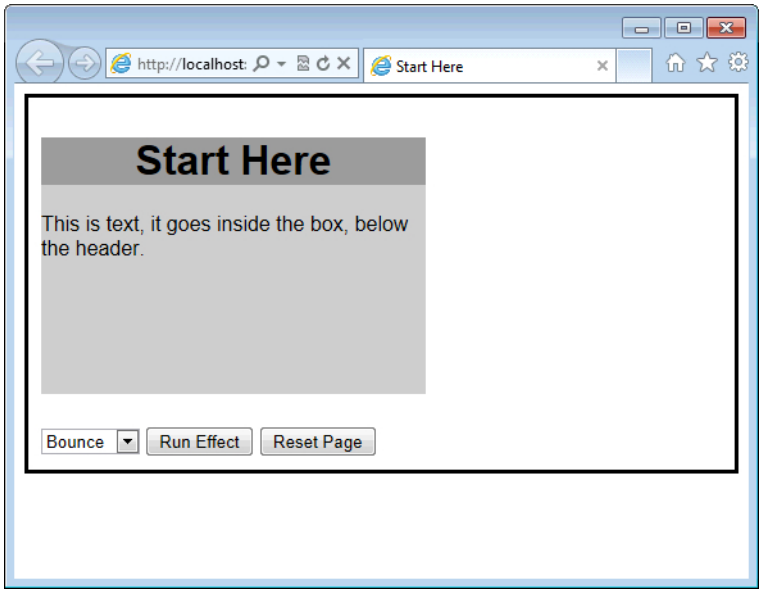


**FIGURE 7-4** Building a test page for jQuery UI effects.

12. Select any of the effects from the drop-down list (or leave the default Bounce effect selected), and click Run Effect. That particular effect will run.

    Some effects cause the Start Here box to disappear. If that happens, click Reset Page to bring the box back.

13. Work with the effects in the drop-down list, and think about ways in which those effects might be incorporated into a web application.

**Note** If you receive error messages about CSS or other items not being found, check the versions of both jQuery and jQuery UI that you include inside of *index.html*. These versions change rapidly and will be different by the time you read this sentence.

This exercise built a test page for jQuery effects. These effects are added with the *effect()* function.

The form's *submit* event was captured, just as you saw in previous exercises. Within the *submit* event handler, the value of the input element with the name of *effect* was gathered. Note the use of the *:input* selector rather than selecting by the element's id (because it didn't have an id):

```
var effect = $(":input[name=effect]").val();
```

Next, some default values are created. The *effect()* function accepts certain arguments, such as the name of the effect. Additionally, some effects—such as the transfer effect—require additional options

to be set. The final argument is the time to be used to run the effect, in milliseconds. This time is set to 1000 milliseconds (1 second) by default in this script. You can change this value to see how it changes the effect.

```
var options = {};
var effectTime = 1000;
```

Next, a conditional is used to determine if the incoming effect requested is the transfer effect. If it's the transfer effect, additional options need to be set. The transfer effect makes it look like the element was moved to another element and can be used to simulate throwing an element into a trashcan. The options set for the transfer effect tell it where to transfer to and how to style the transfer. Additionally, I set the *effectTime* to 500 milliseconds because that looks better with this particular effect (in my opinion, at least).

```
if (effect === "transfer") {
    options = { to: "#trash", className: "transfer" };
    effectTime = 500;
}
```

Finally, well, almost finally, the *effect()* function is called with the name of the effect, any options, and the *effectTime*:

```
$("#startHereBox").effect(effect, options, effectTime);
```

The *submit* event was used, so it should be set to *false* so that the form doesn't actually submit, and that's done next:

```
    return false;
});
```

And finally, a click event is added to the reset button to remove the effect styles and return the page to its normal state:

```
$(":input[name=reset]").on('click', function () {
    $("#startHereBox").removeAttr("style");
});
```

The power of these effects comes in when you incorporate them into pages to enhance the user experience. The concepts you learn and use here will assist you when building JavaScript applications in Microsoft Windows 8.

## Using jQuery UI Widgets

Now you've seen some effects that can be added to a web application with jQuery UI. This section looks at jQuery UI widgets, which are prebuilt combinations of effect, styling, and JavaScript that save time on common tasks.

### Building a Slider

Slider effects are frequently found on websites for hotel or airline reservations or places where there is a need to portray a range of values, such as acceptable takeoff and landing times for a flight. This section looks at using jQuery UI to build a slider effect.

Like other things related to jQuery and jQuery UI, the slider effect is simple to use but comes with powerful options. The most basic slider is accessed simply by calling the *slider()* function and attaching it to an element on the page, typically an empty .

For example, here's a basic page and JavaScript for the slider effect, found as *basicslider.html* in the companion content:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
        rel="stylesheet" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js"></script>
</head>
<body>
        <div id="slider"></div>
<script type="text/javascript">
$(document).ready(function () {
    $("#slider").slider();
});
</script>
</body>
</html>
```

When viewed in a browser, that HTML and JavaScript yields a very basic (and difficult to see) slider effect on the page, as shown in Figure 7-5.

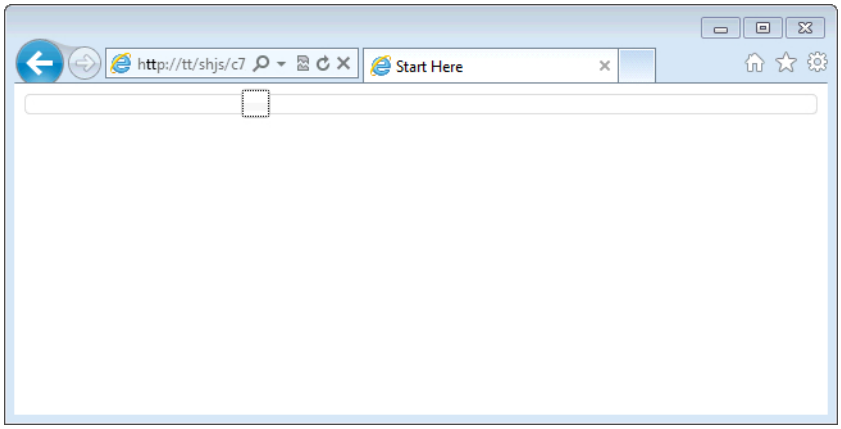


**FIGURE 7-5** A basic slider created with jQuery UI.

The *slider()* function can use numerous arguments that make the slider more useful and help tailor it to your needs. For example, the *value* and *values* options enable you to set values for points on the slider and the *step* option enables you to create stop points within the slider itself. In addition to the options for determining the behavior of the slider, sliders also have events that enable you to react as the user interacts with the slider. One such event is the *slide* event, which is fired when the user interacts with the slider to move it.

For example, here's a page that creates a slider and changes the amount by 50 for each point or step on the slider, found as *slider.html* in the companion content:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
        rel="stylesheet" />

    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js"></script>
</head>
<body>
        <div id="amount">$500</div>
        <br />
        <div id="slider"></div>
<script type="text/javascript">
```

```
$(document).ready(function () {
    $("#slider").slider({
                    step: 50,
                    min: 500,
                    max: 2000,
                    slide: function (event, uiElement) {
                        $("#amount").text("$" + uiElement.value);
                    }
                });
});
</script>
</body>
</html>
```

This code uses the *slider()* function and sends in several arguments, including the minimum and maximum amounts for the slider, how much to change the slider, and a function to handle the *slide* event. Viewing the page in a web browser results in a page like that shown in Figure 7-6.

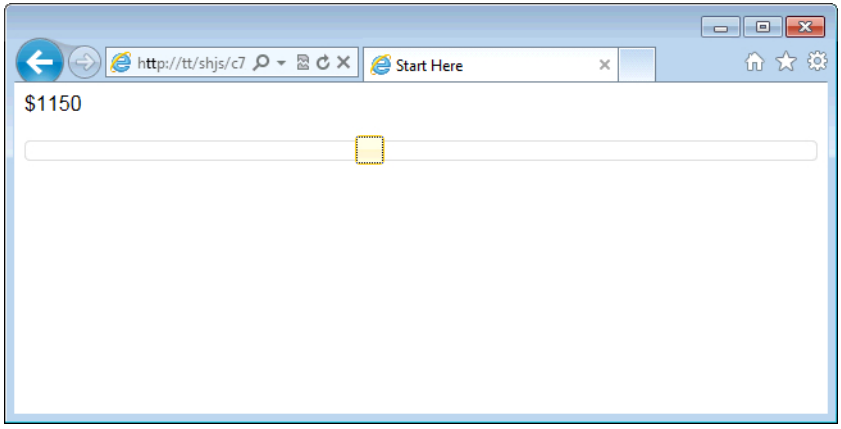


**FIGURE 7-6** Using a slider with additional options.

***See Also*** *See* http://jqueryui.com/demos/slider *for more information on sliders and their options.*

## Creating a Calendar

jQuery UI includes a calendar function called a *datepicker* that enables a themed pop-over calendar for use in forms and elsewhere. Like other functions, the datepicker is added with the *datepicker()* function. A basic sample page with a datepicker looks like the following code (which you can find as *datepicker.html* in the companion content):

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Here</title>
```

```
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.18.custom.css"
        rel="stylesheet" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.18.custom.min.js"></script>
</head>
<body>
        Date: <input type="text" name="date" id="date">
<script type="text/javascript">
$(document).ready(function () {
    $("#date").datepicker();
});
</script>
</body>
</html>
```

When viewed in a browser, it results in a page with an empty text field, like Figure 7-7.

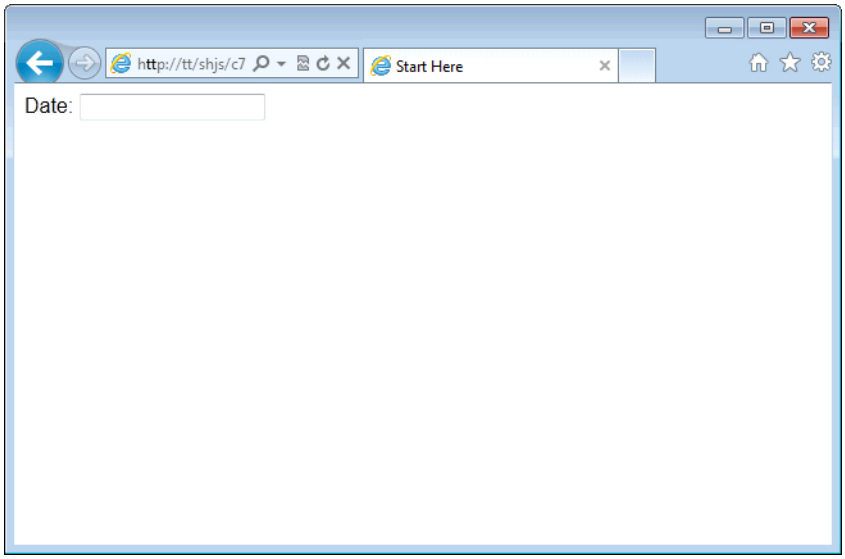


**FIGURE 7-7** A basic datepicker text field.

Clicking into the text field reveals a date picker, like the one shown in Figure 7-8.

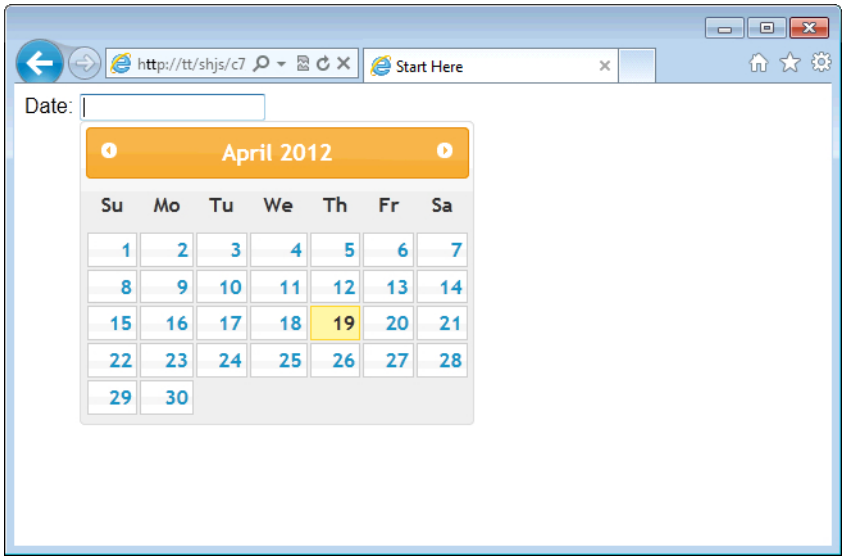


**FIGURE 7-8** A date picker.

Once a date is selected, it's automatically placed into the Date text box.

You can change the styling of the datepicker with jQuery UI, and you can set several options related to the datepicker, such as a range of dates, the formatting, and other such options. See *http://jqueryui.com/demos/datepicker* for more information on the datepicker and its options.

## Other Helpful jQuery UI Widgets

You've now seen just two of the many widgets that jQuery UI offers. There are many additional widgets available and many other aspects of jQuery UI that won't be discussed in this book. However, I invite you to peruse the jQuery UI website at *http://jqueryui.com*, paying special attention to the Demos And Documentation area, which contains excellent working demonstrations of jQuery UI features.

# Putting It All Together: A Space Travel Demo

You've come a long way in the book, seeing all sorts of HTML, CSS, and JavaScript used to create webpages, create web forms, react to events, style webpages, and create advanced effects. It's time to show a web application that uses many of the elements you've seen.

This section creates a web application to accept reservations for interplanetary travel. Your client, Adventure Works, wants people to book spaceflights on its site. The company also wants to provide an interface that lets people pick their price range, after which time the Adventure Works reservation system will tell the visitor which planets are available and the length of the trip.

At this time, your task is to create some basic mocked-up forms, which will then be used by a designer to add pretty pictures and other graphics and design elements. In other words, you don't need to make it look pretty, but it should be functional and responsive so that the Adventure Works executives can click through the application and see how it flows.

This application requires two server-side programs: one called *details.aspx* and one called *flights.aspx*. Both of these files can be found in the companion content (along with PHP equivalents). The other code is found as *space.html*, *space.js*, and *space.css* in the companion content for Chapter 7.

1. To begin creating the application, open your StartHere project.
2. Within the StartHere project, open *index.html*.
3. Delete any existing code from within *index.html*.
4. Place the following markup in *index.html*:

```
<!doctype html>
<html>
<head>
<title>Start Here</title>
    <link type="text/css" href="css/ui-lightness/jquery-ui-1.8.17.custom.css"
        rel="stylesheet" />
    <link rel="stylesheet" type="text/css" href="css/StyleSheet1.css" />
    <script type="text/javascript" src="js/jquery-1.7.1.min.js"></script>
    <script type="text/javascript" src="js/jquery-ui-1.8.17.custom.min.js">
        </script>
    <script type="text/javascript" src="js/external.js"></script>
</head>
<body>
    <h1><span class="underline">Adventure Works</span>
        <span id="subhead">Interplanetary Space Travel</span> </h1>
    <hr>
    <div id="container">
    <p>Price of Space Flight: $<span id="spanPrice">500,000</span></p>
        <div id="priceSlider"></div>
    </div>
    <div id="flights">
        <p>Available Flights at that Price</p>
        <ul id="flightList">

        </ul>
    </div> <!-- end flightList -->

  <div class="hidden" id="flightDetails">
        <p>Flight Details</p>
    </div>
</body>
</html>
```

5. Save *index.html*.

6. Open the style sheet, which is found in the css folder as *StyleSheet1.css*.

7. Clear any CSS from that file.

8. Place the following CSS in the style sheet:

```
body {
        font-family: arial, helvetica, sans-serif;
}
.underline { text-decoration: underline; }
.hiddenPrice { display: none; }
.hidden { display: none; }
#formDiv { height: 250px; }
#subhead { font-size: .6em; }

.detailsLi {
        list-style-type: none;
}

#priceSlider {
        height: 250px;
        margin-left: 100px;
}
#container {
        float: left;
        padding-left: 50px;
        width: 290px;
}

#flightList li {
        list-style-type: none;
        border: 1px dashed grey;
        text-align: center;
}

#flightList li:hover {
        background-color: grey;
}
#flights {
        float: left;
        height: 250px;
        padding-left: 50px;
}
```

```
#flightDetails {
        float: left;
        height: 250px;
        padding-left: 50px;
}

#priceSlider .ui-slider-handle {
    background: #000;
}

#priceSlider {
    background: #808080;
}
```

9. Save *StyleSheet1.css*.

10. Open *external.js*, and delete any JavaScript from that file.

11. Place the following JavaScript in *external.js*:

```
$(document).ready(function () {
    $.ajax({
        url: 'flights.aspx',
        dataType: "json",
        success: function(data) {
            var counter = 0;
            $.each(data, function(key, value) {
                $("#flightList").append('<li ' +
                       'id="flight' + counter + '"' +
                       ' class="flightLi">' +
                       value['trip'] + '<span class="hiddenPrice">' +
                       value['price'] + '</span></li>');
                counter++;
            });
        }
    });
    $("#priceSlider").slider({
        orientation: "vertical",
        min: 10000,
        max: 500000,
        step: 10000,
        value: 500000,
        slide: function (event, uiElement) {
            $("#flightDetails").html("<p>Flight Details</p>").
                addClass("hidden");
            var numRegex = /(\d+)(\d{3})/;
            var inputNum = uiElement.value;
```

```
            var strNum = inputNum.toString();
            strNum = strNum.replace(numRegex, '$1' + ',' + '$2');
            $("#spanPrice").text(strNum);
            $("#inputPrice").val(uiElement.value);
            $(".hiddenPrice").each(function() {
                if ($(this).text() > inputNum) {
                    $(this).parent().addClass("hidden");
                }
                else if ($(this).text() < inputNum) {
                    $(this).parent().removeClass("hidden");
                }
            });
        }
    });
    $(".flightLi").on('click', function () {
        $("#flightDetails").html("<p>Flight Details</p>").addClass("hidden");
        var myId = $(this).attr("id");
        $.ajax({
            url: "details.aspx",
            dataType: "json",
            data: { "flightID": myId },
            type: "POST",
            success: function (data) {
                $("#flightDetails").removeClass("hidden").append('<ul>' +
                        '<li class="detailsLi">Trip Duration: ' +
                          data['duration'] + '</li>' +
                        '</ul>');
            }
        });
    }); //end flightLi live click.
});
```

12. Save *external.js*.

13. Add a new item to the project. (See Chapter 6 for details on how to add a server-side program.) The new items should be a web form.

14. Clear any code from inside of the newly created web form, and place the following code in the file:

```
<%
    Dim str = "[ " & _
        "{ ""trip"": ""Around the block"", " & _
        " ""price"": 10000 " & _
        "}, " & _
```

```
        "{ ""trip"": ""Earth to Moon"", " & _
        " ""price"": 50000 " & _
        "}, " & _
        "{ ""trip"": ""Earth to Venus"", " & _
        " ""price"": 200000 " & _
        "}, " & _
        "{ ""trip"": ""Earth to Mars"", " & _
        " ""price"": 100000 " & _
        "}, " & _
         "{ ""trip"": ""Venus to Mars"", " & _
        " ""price"": 250000 " & _
        "}, " & _
        "{ ""trip"": ""Earth to Jupiter"", " & _
        " ""price"": 300000 " & _
        "}, " & _
        "{ ""trip"": ""Earth to Sun - One Way"", " & _
        " ""price"": 450000 " & _
        "}, " & _
        "{ ""trip"": ""Earth to Neptune"", " & _
        " ""price"": 475000 " & _
        "}, " & _
         "{ ""trip"": ""Earth to Uranus"", " & _
        " ""price"": 499000 " & _
        "} " & _
        "]"
    Response.Write(str)
%>
```

**15.** Choose Save As from the File menu.

**16.** Name this file **flights.aspx**.

**17.** Create another server-side program in the same way, by adding a new web form. Clear any code from this second web form.

**18.** Place the following code inside of this file:

```
<%
    Dim flightID As String
    Dim outputStr As String
    If IsNothing(Request("flightID")) Then
        flightID = "Unknown"
    Else
        flightID = Request("flightID")
    End If
    If flightID = "flight0" Then
        outputStr = "1 minute"
```

```
    ElseIf flightID = "flight1" Then
        outputStr = "5 Days"
    ElseIf flightID = "flight2" Then
        outputStr = "7 Days"
    ElseIf flightID = "flight3" Then
        outputStr = "8 Days"
    ElseIf flightID = "flight4" Then
        outputStr = "9 Days"
    ElseIf flightID = "flight5" Then
        outputStr = "12 Days"
    Else
        outputStr = "15 Days"
    End If

    Dim str = "{ ""duration"": """ & outputStr & """ }"
    Response.Write(str)
%>
```

19. Save this file as **details.aspx**.

20. Now view *index.html* in your web browser. You'll see a page like the one shown in Figure 7-9.

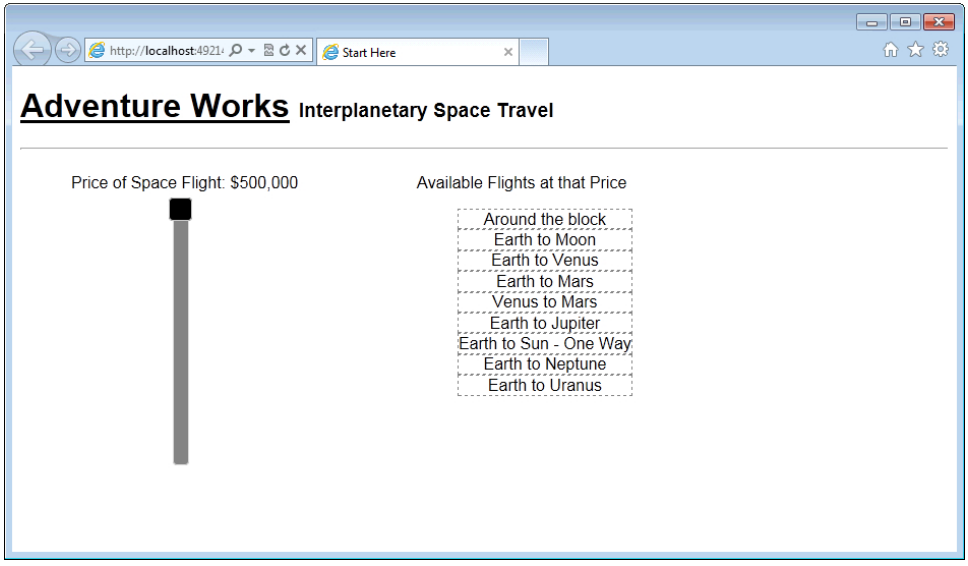


**FIGURE 7-9** The Adventure Works mock-up flight reservation form.

21. Move the slider down to lower the price. Notice how the list of available flights changes as the slider moves.

22. Set the slider at $250,000 and then click the Venus To Mars trip. You'll see the flight duration on the right side, as shown in Figure 7-10.

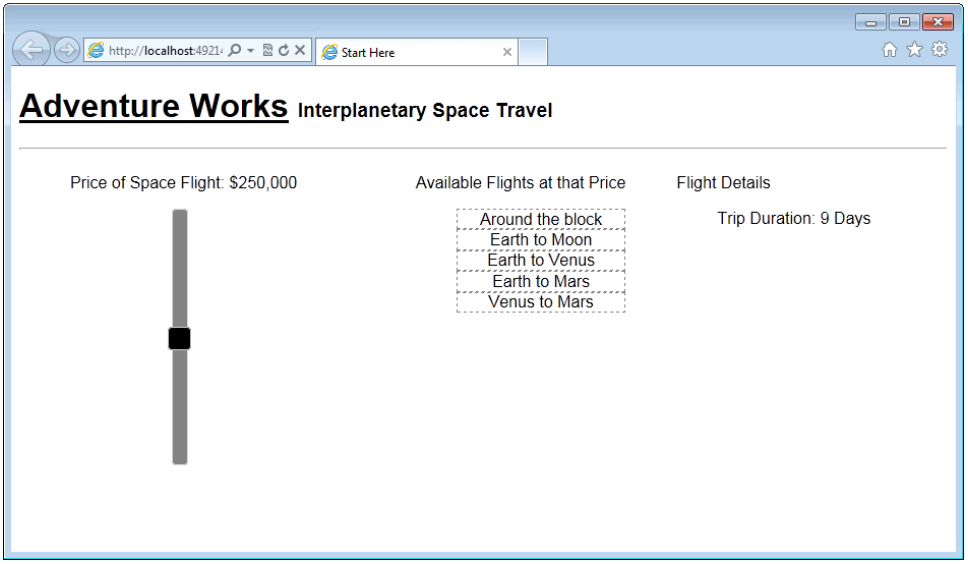


**FIGURE 7-10** The flight details for a trip from Venus to Mars.

## Code Analysis

I'll analyze the JavaScript for this project in a few different parts. The first portion retrieved a list of all available flights from the server (*flights.aspx*). Each of the returned results is appended to the *flightList ul* element on the page, and a hidden  element is created with the price for that flight:

```
$.ajax({
    url: 'flights.aspx',
    dataType: "json",
    success: function(data) {
        var counter = 0;
        $.each(data, function(key, value) {
            $("#flightList").append('<li ' +
                   'id="flight' + counter + '"' +
                   ' class="flightLi">' +
                   value['trip'] + '<span class="hiddenPrice">' +
                   value['price'] + '</span></li>');
            counter++;
        });
    }
});
```

Next in the code, the slider is built. The slider's orientation is set to vertical, and ranges are set:

```
$("#priceSlider").slider({
    orientation: "vertical",
    min: 10000,
    max: 500000,
    step: 10000,
    value: 500000,
```

The *slide()* function presents some interest. This function formats the value from the slider into a friendlier-looking display, such as $450,000 rather than just $450000. Within that function, each of the available flights is enumerated. If the slider's value (like 300000, 250000, etc) is greater than the current value from the *flightDetails*, then that flight is hidden. Conversely, if the flight details price is lower than the slider's value the flight is shown:

```
    slide: function (event, uiElement) {
        $("#flightDetails").html("<p>Flight Details</p>").addClass("hidden");
        var numRegex = /(\d+)(\d{3})/;
        var inputNum = uiElement.value;
        var strNum = inputNum.toString();
        strNum = strNum.replace(numRegex, '$1' + ',' + '$2');
        $("#spanPrice").text(strNum);
        $("#inputPrice").val(uiElement.value);
        $(".hiddenPrice").each(function() {
            if ($(this).text() > inputNum) {
                $(this).parent().addClass("hidden");
            }
            else if ($(this).text() < inputNum) {
                $(this).parent().removeClass("hidden");
            }
        });
    }
});
```

The last major piece of the JavaScript for this application sets up the click event handler for the flight. When a flight is clicked, its details are retrieved from a web service. These details could include seating availability, exact departure and arrival times, and so on. The example shows just a simple overall duration for that flight:

```
$(".flightLi").on('click', function () {
    $("#flightDetails").html("<p>Flight Details</p>").addClass("hidden");
    var myId = $(this).attr("id");
    $.ajax({
        url: "details.aspx",
        dataType: "json",
        data: { "flightID": myId },
```

```
            type: "POST",
            success: function (data) {
                $("#flightDetails").removeClass("hidden").append('<ul>' +
                        '<li class="detailsLi">Trip Duration: ' +
                          data['duration'] + '</li>' +
                        '</ul>');
            }
        });
    }); //end flightLi live click.
```

Here are the server-side programs written in PHP. The first file returns available flights (*flights.php*):

```
<?php

$travels = array();

$travels[] = array("trip" => 'Around the block',"price" => 10000);
$travels[] = array("trip" => 'Earth to Moon',"price" => 50000);
$travels[] = array("trip" => 'Earth to Venus',"price" => 200000);
$travels[] = array("trip" => 'Earth to Mars',"price" => 100000);
$travels[] = array("trip" => 'Venus to Mars',"price" => 250000);
$travels[] = array("trip" => 'Earth to Sun - One Way',"price" => 450000);
$travels[] = array("trip" => 'Earth to Jupiter',"price" => 300000);
$travels[] = array("trip" => 'Earth to Neptune',"price" => 475000);
$travels[] = array("trip" => 'Earth to Uranus',"price" => 500000);

print json_encode($travels);

?>
```

Here's *details.php*:

```
<?php

if (!isset($_POST['flightID'])) {
        die();
}
$data = array();
switch ($_POST['flightID']) {
        case "flight0":
                $data = array("duration" => "1 Minute");
                break;
        case "flight1":
                $data = array("duration" => "5 Days");
                break;
```

```
        case "flight2":
                $data = array("duration" => "7 Days");
                break;
        default:
                $data = array("duration" => "9 Days");
}

print json_encode($data);

?>
```

This application was created in just 59 lines of JavaScript with the help of jQuery and jQuery UI.

## Summary

This chapter covered the use of JavaScript with CSS. The chapter looked at CSS itself, and then it examined methods for determining and setting current CSS property values through the jQuery *css()* function. The ability to add and remove CSS classes was shown next by giving examples of the *hasClass()*, *addClass()*, and *removeClass()* functions.

Advanced effects with jQuery UI were shown in the chapter, and you built a test page to look at the *effect()* function. Some jQuery UI widgets were shown in the chapter as well. Finally, the chapter wrapped up with a web application that tied together concepts from throughout the book to build a flight reservation front page using elements from jQuery UI as well as AJAX calls.

Chapter 8

# Using JavaScript with Microsoft Windows 8

**After completing this chapter, you will be able to**

- Understand how Microsoft Windows 8 and JavaScript interact
- Understand a grid application
- Create a full Windows 8 App

**WINDOWS 8 CHANGES THE PARADIGM** for JavaScript. Granted, JavaScript has already grown into a primary development language for the web. But with Windows 8, Microsoft has elevated JavaScript even higher.

This chapter looks at JavaScript programming from the perspective of writing apps for Windows 8. Although Microsoft Visual Studio isn't required for development with JavaScript in Windows 8, the Software Development Kit (SDK) is. You should obtain the Windows 8 SDK from the Developer Downloads at MSDN, at *http://msdn.microsoft.com/windows/apps/br229516*. Additionally, to run the examples in this chapter, you need Windows 8.

**Note** If you're using a version of Visual Studio for Windows 8, you already have the SDK.

## JavaScript Is Prominent in Windows 8

With the release of Windows 8, Microsoft signaled a significant change to the landscape of client-side programming. Prior to Windows 8, JavaScript was (mainly) used as a programming language for the web—and everything you've seen so far in this book has been based on the premise that you're programming a web application that will be viewed through a browser of some sort. Microsoft Windows 8 changes all that.

Before Windows 8, if you wanted to build a program that ran natively on the operating system and wasn't reliant on a browser, you had to use a client-side language such as Microsoft Visual Basic, C#,

C++, or a similar language. But with Windows 8, you can now use HTML, CSS, and JavaScript to create those same native applications.

Luckily, what you've learned so far about JavaScript still applies when programming an application for Windows 8. Yet programming for Windows 8 is a different animal. When programming with JavaScript for Windows 8, you'll rely on certain application programming interfaces (APIs) and libraries to make your programs work.

### What's an API?

An API is a way for you to work with another developer's code. In this case, an API enables you to access certain functions and data in Windows 8. The API is defined by the person who creates the code library or program. In this case, the API is defined by Microsoft developers who created Windows 8.

Think of APIs as you would functions in jQuery: you need to include them in your programs, you call them like a function, and the function will sometimes accept arguments and sometimes return values.

Windows 8 introduces a new interface, the Windows 8 UI, which is based on the premise that the user will touch or tap the screen, rather than use mouse clicks. You can see an example of this interface in Figure 8-1.


**FIGURE 8-1** An example of the Windows 8 UI.

Within the Windows 8 UI, programs are represented through *tiles*. Tiles are dynamic on-screen areas that can be used not only to start a program, but also to convey updates and to otherwise entice the user to click, touch, and interact with them.

Unlike web development with HTML, CSS, and JavaScript, as of this writing you need a Microsoft-supplied Software Development Kit (SDK) to develop Windows 8 Apps. The SDK is akin to the jQuery library you've been using throughout the book. The SDK for Windows 8 Apps provides provides the interface to Windows 8 JavaScript libraries, and it enables you to connect to the inner workings of Windows 8.

The easiest way to develop Windows 8 Apps is to use Visual Studio 11. Doing so gives you access to the SDK in its native environment and also provide advantages like giving you access to IntelliSense to help make coding easier. The free Express Edition for Windows 8 includes the SDK, and you can download and install it from *http://www.microsoft.com/visualstudio/11/downloads*.

When you develop a Windows 8 App, you can choose from a variety of predefined layouts or design your own layout. For instance, some applications aggregate information from blog posts, while others provide navigational-based information on a series of screens. Choosing how your application needs to be laid out is an important step in the Windows 8 App development lifecycle.

In fact, good application design is such an important element that Microsoft has expended a lot of resources to help you design your applications. This is evidenced by the amount of CSS and JavaScript that's prewritten for you in the template layouts, but you also see it in the documentation available on Microsoft's website. See *http://msdn.microsoft.com/library/windows/apps/hh781237* for just one example of design recommendations. You also should check out *http://msdn.microsoft.com/en-us/windows/apps* for up-to-date information on Windows 8 App development. Although not everyone agrees with the guidelines, such guidelines have helped companies ensure a positive and consistent user experience, such as Apple has done for its iOS-based applications that are available through the Apple App Store.

Windows 8 Apps written in JavaScript are controlled through a combination of HTML, CSS, and JavaScript. Much of the JavaScript used in Windows 8 Apps is defined in the Windows JavaScript libraries and is connected to various HTML elements through the use of the HTML data attribute. Sometimes you'll see and use CSS classes to define the look and feel, and the behavior as well.

In this way, programming Windows 8 Apps is similar to what you've been doing through the book with jQuery and jQuery UI. The names are different, and there's a new set of functions to learn, but the concepts are the same.

Windows 8 Apps rely on data retrieved through JavaScript, using AJAX for example, or loaded from within the JavaScript file itself. Later in this chapter, you'll see an example of a grid-style Windows 8 App. That application's sample data is defined completely within JavaScript and then loaded through the Windows JavaScript libraries.

## A Stroll Through a Windows 8 Application

This section looks at a grid-style Windows 8 App.

To follow along in this section, create a new grid application in Visual Studio 11 by selecting New Project from the File menu in Visual Studio 11.

1. Select Grid Application from the Windows 8 JavaScript Templates, and type **MyGrid** for the project name.
2. A new grid application is created. From the Debug menu, select Start Debugging or press F5. The application starts, and you see a screen similar to the one shown in Figure 8-2.

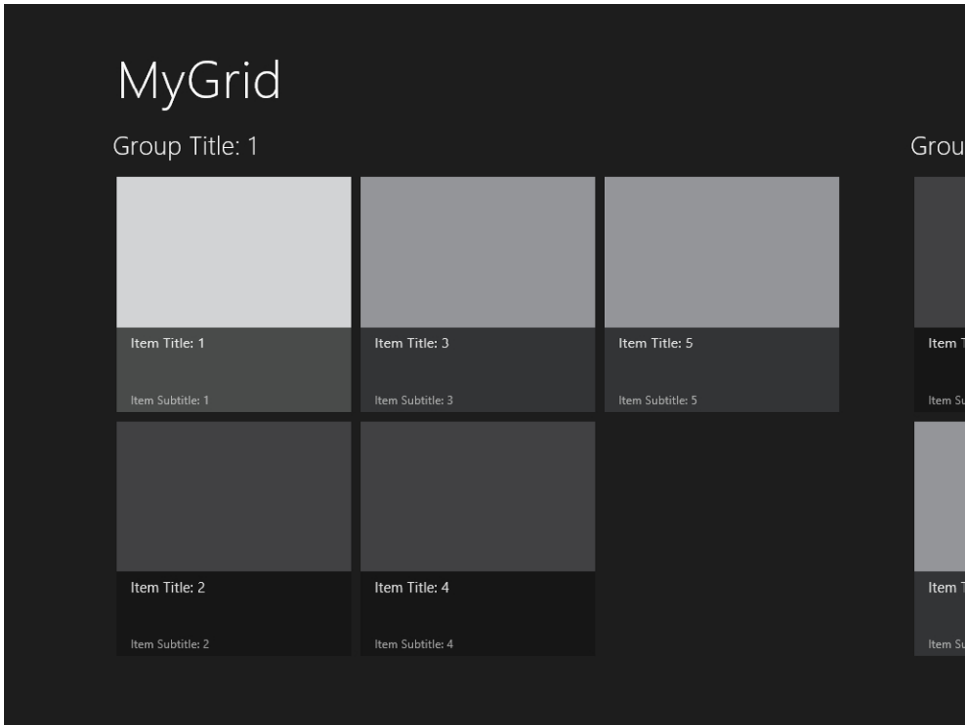


**FIGURE 8-2** The default grid application.

3. Switch back to Visual Studio by pressing Alt+Tab or by clicking and dragging the mouse cursor from the top of the application to the bottom.
4. Within Visual Studio, select Stop Debugging from the Debug menu.
5. With the project stopped, examine the files and resources within Solution Explorer. Inside Solution Explorer, you'll see a folder for css, html, images, and JavaScript (named js). Additionally, there are references and other files in the solution. There's also a file in the root folder named *default.html*.

6. Open *default.html*, and examine its contents.

The *default.html* file is structured similar to the HTML you've been working with throughout the book, with a DOCTYPE declaration, a head section, and a body section. Inside of the <head> section there are three references to the Windows JavaScript library, WinJS. Those references look like this:

```
<!-- WinJS references -->
<link href="//Microsoft.WinJS.0.6/css/ui-dark.css" rel="stylesheet">
<script src="//Microsoft.WinJS.0.6/js/base.js"></script>
<script src="//Microsoft.WinJS.0.6/js/ui.js"></script>
```

If you followed along through previous exercises, you'll recognize that the pattern used is strikingly similar to that of jQuery and jQuery UI. There is a CSS file, followed by a base JavaScript library, followed by a User Interface (UI) library. These libraries provide the necessary functions through which a Windows 8 App is built.

**Note** The order in which the WinJS references appear is important. Always make sure to include the *base.js* file prior to the *ui.js* file.

The next references found in *default.html* are all specific to this application. This means they'll be customized to whatever the application is trying to do, such as display a list of houses for sale, show news articles, and so on. The references include a CSS file and three JavaScript files for the application, as shown here:

```
<!-- MyGrid references -->
<link href="/css/default.css" rel="stylesheet">
<script src="/js/data.js"></script>
<script src="/js/navigator.js"></script>
<script src="/js/default.js"></script>
```

These files are responsible for loading the data and navigation elements for the application. In a real-world application, the data could be loaded from AJAX or stored in this file, as it is in the sample application.

Within the body section of the page is the next customization for the grid application. Specifically, within that section, the *div* element is important:

```
<div id="contenthost" data-win-control="MyGrid.PageControlNavigator" data-win-
options="{home: '/html/groupedItemsPage.html'}"></div>
```

This *div* uses two attributes specific to Windows 8 Apps: the *data-win-control* attribute and the *data-win-options* attribute. These attributes connect the HTML element to a specific WinJS JavaScript control. In this case, they connect the element to a *PageControlNavigator* control, which is a fancy name for some JavaScript functions that help users move around within your application.

Also specified in this *div* element is the *data-win-options* attribute, which relates to the *Page ControlNavigator* and provides the home page for the application. In this sample application, the home page display is defined in a page called *groupedItemsPage.html*, which you can find in the html folder.

The *groupedItemsPage.html* file contains HTML specific to loading this application. The page also contains its own specific JavaScript and CSS files, defined in the <*head*> section of the page. Much of the HTML found in this file won't be changed for a grid-style application. The *groupedItemsPage.html* file is responsible for loading the screen.

The layouts for group and item detail pages or displays within the application are defined in the other HTML files found within that html folder. However, unlike the link between *default.html* and *groupItemsPage.html*, the connection between these files is defined in JavaScript. As already discussed, the *groupItemsPage.html* includes its own JavaScript file. Opening *groupedItemsPage.js* shows the JavaScript specific for this page. In that file, you'll see that a detail page for a group is loaded from the *groupDetailPage.html* file.

When the application is running, clicking one of the Group Title labels loads that *groupDetailPage.html* content and its related JavaScript. In much the same way as the details of a group are loaded, the details of an item are also loaded with a specific HTML file.

## Quick Checkpoint: Windows 8 Apps

Now that you've seen some of the code behind a Windows 8 App, it's a good time to regroup and make sure we're on the same page. (I mean figuratively, not literally. If you aren't on the same page as I am, then how would we be reading this at the same time?)

- Windows 8 Apps can be built with HTML, CSS, and JavaScript.
- Much of the JavaScript used in Windows 8 Apps is stored in Windows JavaScript libraries, collectively called WinJS.
- The HTML used for Windows 8 Apps requires the use of specific data attributes within HTML, the use of specific CSS classes, and the execution of WinJS JavaScript functions for loading the layouts, navigation, and data related to the application.
- Data is typically loaded through JavaScript, whether it's gathered using AJAX, stored in files or accessed by other means.
- Programming Windows 8 Apps requires you to use many of the same skills and concepts as programming using jQuery and jQuery UI.
- Visual Studio 11 includes several templates to help jump-start development of common Windows 8 App layouts.

With that short list, it's time to look at building a Windows 8 App.

# Building a Windows 8 App

This section is a tutorial for building a Windows 8 App. The tutorial shows how an application is built from start to finish. The application makes an AJAX call using an API available from SiteReportr (*http://www.sitereportr.com*) to check the availability of websites. To perform this tutorial, you need an API key from SiteReportr. Keys are available for free by going to *http://www.sitereportr.com/customer/keygen.php*, and you can use them for a maximum of 90 days. To obtain a key, you need to provide an email address. The key will be mailed to the email address you supply, and the email address does not need to be the same as your Windows Live ID.

**Note** The terms and conditions of use of the API key might change between now and when you read this. The latest terms are available from SiteReportr.

### Promises, Promises

One of the objects available in the WinJS library is known as a *Promise*. A Promise is an object that provides data within an application in an asynchronous manner. You already saw how to retrieve data with AJAX earlier in the book. Sometimes you need to build the page prior to the data being available. In these cases, you use a Promise, which enables that element to ignore the fact that it's missing data and just continue on as normal.

Promises also enable advanced error handling and timeout handling for requests, as you'll see in the application example in this section. The Promise object requires three functions to be provided as arguments: one function defines what to do when things go well and you get data back, another function defines what to do if there's an error, such as a timeout, and the final function is a progress function.

## Building the Application

This application will use the Basic JavaScript template for Windows 8 Apps in Visual Studio 11 and relies heavily on the WinJS library. You can find this code as *default.html*, *default.js*, and *default.css* in the companion content for Chapter 8. Note that you need to change the application key found in the *default.js* file with the one you receive from SiteReportr or this code won't work.

1. Begin by opening Visual Studio 11.
2. Create a new project by selecting New Project from the File menu.
3. Select Blank Application from the Windows 8 templates, and type **mySiteReportr** as the name. Click OK to create the project. See Figure 8-3 for an example.

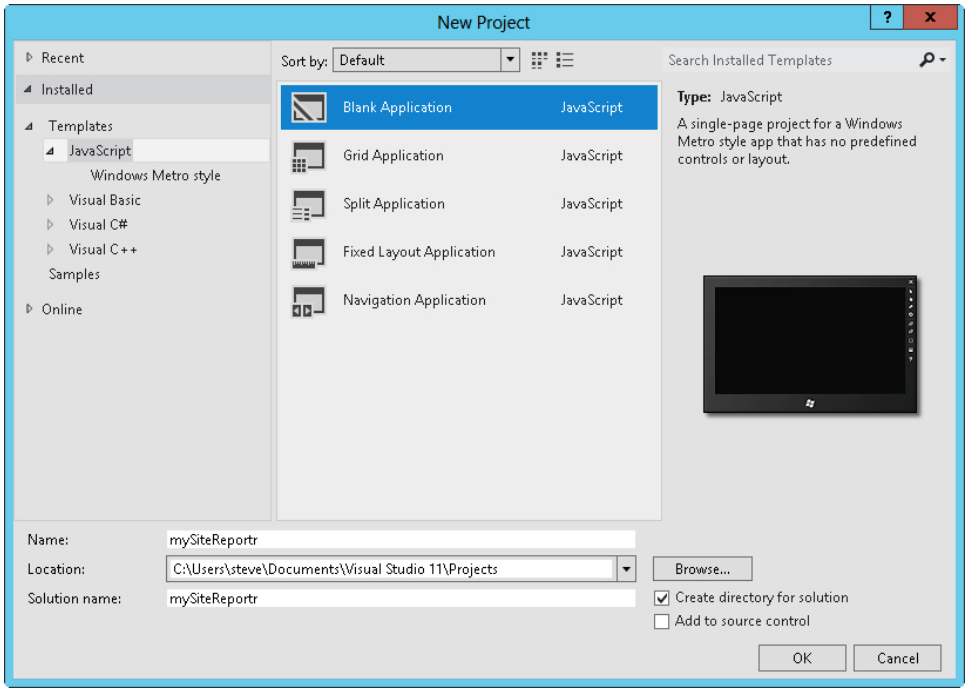


**FIGURE 8-3** Creating a new blank application in Visual Studio 11.

4. When you create a new blank application, the *default.js* JavaScript file is opened automatically. If it's not, open *default.js*, which is found in the js folder. There is a good amount of existing code in *default.js* already. Leave this code in place.

5. Near the top of *default.js*, find the variable declaration for *WinJS.application*. It looks like this:

```
var app = WinJS.Application;
```

6. Below that line, place the following two variable declarations:

```
var ui = WinJS.UI;
var utils = WinJS.Utilities;
```

7. Scroll to near the bottom of *default.js*, and place the following code on the line immediately below the *app.start();* line, opening a new blank line after *app.start()*. This code should go below *app.start()* but before the existing closing brace and parentheses.

```
function getSiteStatus() {
        var siteSubmit = document.getElementById("siteSubmit");
        siteSubmit.setAttribute("disabled", "true");
        var progress = document.getElementById("progress");
        progress.className = "win-ring";
```

```
    var siteURL = document.getElementById('siteURL').value;
    var resultMessage = document.getElementById('resultMessage');
    var resultStaticText = document.getElementById('resultStaticText');
    var appURL = "http://sr.sitereportr.com/siterep.php";

  var appID = "StartHere";
    // REPLACE WITH YOUR KEY VALUE:
    var appKey = "5150";
    var params = "guid=" + appID;
    params += "&site=" + siteURL;
    params += "&key=" + appKey;
    WinJS.Promise.timeout(10000, WinJS.xhr({
        url: appURL,
        type: "POST",
        headers: { "Content-type": "application/x-www-form-urlencoded" },
        data: params
    }).then(function complete(result) {
        var siteSubmit = document.getElementById("siteSubmit");
        siteSubmit.removeAttribute("disabled");
        var progress = document.getElementById("progress");
        progress.className = "hide";
        if (result.responseText != "Invalid URL") {
            var myResult = JSON.parse(result.responseText);
        } else {
            var myResult = "Unknown";
        }
        if (myResult.status == "Up") {
            resultMessage.innerHTML = myResult.status;
            resultMessage.className = "resultUp";
            resultStaticText.className = "";
            resultMessage.setAttribute("display", "inline");
        } else if (myResult.status == "Down") {
            resultMessage.innerHTML = myResult.status;
            resultMessage.className = "resultDown";
            resultStaticText.className = "";
            resultMessage.setAttribute("display", "inline");
        } else {
            if (myResult.status == "Message") {
                resultMessage.innerHTML = myResult.message;
            } else {
                resultMessage.innerHTML = "Unknown";
            }
            resultMessage.className = "resultUnknown";
            resultStaticText.className = "";
            resultMessage.setAttribute("display", "inline");
        }
    }, function error(error) {
```

```
                var siteSubmit = document.getElementById("siteSubmit");
                siteSubmit.removeAttribute("disabled");
                var progress = document.getElementById("progress");
                progress.className = "hide";
                resultMessage.innerHTML = "Error";
                resultMessage.className = "resultUnknown";
                resultStaticText.className = "";
                resultMessage.setAttribute("display", "inline");
            },
            function progress(result) {
                var progress = document.getElementById("progress");
                progress.className = "win-ring";
            }));
        } //end function getSiteStatus

        ui.Pages.define("default.html", {
            ready: function (element, options) {
                document.getElementById('siteSubmit').addEventListener("click",
getSiteStatus, false);
            }
        });
```

8. In that code, there's a variable declaration for *appKey*, which is currently set to *5150*. Change that value to the key value you obtained from SiteReportr. Note that the key you receive will likely be significantly longer than *5150*.

9. Save *default.js*.

10. Open *default.html* by selecting it from within Solution Explorer.

11. Within the <head> section, change the CSS reference to point to *ui-light* instead of *ui-dark*. It should look like this when you're done:

```
<link href="//Microsoft.WinJS.0.6/css/ui-light.css" rel="stylesheet">
```

12. In *default.html*, remove the <p> line within the <body> section:

```
<p>Content goes here</p>
```

13. With that line gone, place the following markup in between the opening <body> and closing </body>:

```
    <h1 class="titlearea win-type-ellipsis"><a href="http://sitereportr.
com">SiteReportr</a></h1>
    <div id="siteLine">
```

```
        <span id="urlSpan">URL: </span>
        <input type="text" size="50" maxlength="100" name="siteURL"
            id="siteURL" />
        <progress id="progress" class="win-ring hide"></progress>
        <input type="submit" name="submit" value="Go" class="go-button"
            id="siteSubmit" />
        <div id="statusLine"><span id="resultStaticText" class="hide">Status:
</span>
        <span id="resultMessage" class="hide"></span></div>
    </div>
```

14. Save *default.html*.

15. Open *default.css*, which is found in the css folder.

16. In *default.css*, below the closing bracket for the final *@media* screen definition, add the following CSS:

```
#siteLine {
   margin-left:  75px;
}

#resultStaticText {
    margin-left: 40px;
    text-align:right;
    color:#000;
    font-size:16px;
    line-height:38px;
    font-family:Arial;
    letter-spacing:2px;
    font-weight:bold;
}

#resultMessage {
    padding-top: 5px;
    padding-bottom: 5px;
    padding-right: 50px;
    padding-left: 50px;
    height:  41px;
    margin-left: 10px;
    border: 1px solid black;
    text-align:center;
    text-transform:uppercase;
    color:#000;
    text-shadow:1px 1px 1px rgba(0,0,0,0.85);
    font-size:24px;
    line-height:38px;
```

```
    font-family:Arial;
    letter-spacing:2px;
    font-weight:bold;
}

.go-button {
    width:182px;
    height:41px;
    margin-left: 20px;
    text-align:center;
    text-transform:uppercase;
    color:#fff;
    text-shadow:1px 1px 1px rgba(0,0,0,0.85);
    font-size:1.4em;
    line-height:38px;
    font-family:Arial;
    letter-spacing:2px;
    font-weight:normal;
}


.show {
    display: inline;
}

.hide {
    display:  none;
}

.hideVis {
    visibility: hidden;
}
.resultUp {
    background: #00FF00;
}

#statusLine {
    margin-top: 50px;
}

.resultDown {
    background:  #FF0000;
}

.resultUnknown {
    background:  #FF9900;
}
```

17. Save *default.css*.

18. Run this application by selecting Start Debugging from the Debug menu. The application will build, display a splash screen, and then finally display the application. It'll be like the one you see in Figure 8-4.


**FIGURE 8-4** Running your Windows 8 App.

19. Within the URL text box, type **braingia.org** and click Go. You should see an UP status indicator, like the one shown in Figure 8-5.


**FIGURE 8-5** Using your SiteReportr Windows 8 App.

20. Enter a site that doesn't exist or is otherwise down in the URL text box and click Go. The status should change to Down. Entering garbage text into the URL text box and clicking Go should result in an Unknown status.

21. When you're done experimenting, press Alt+Tab to switch back to Visual Studio.

22. In Visual Studio, select Stop Debugging from the Debug menu.

## Code Analysis

The HTML and CSS are standard, based on what you've already seen and therefore aren't analyzed here. The JavaScript, which is in *default.js*, is where you added the code to control the application. So that's where the analysis begins.

Inside of *default.js*, you added a function and a UI page. The UI page, defined at the bottom of your custom code, looks like this:

```
ui.Pages.define("default.html", {
    ready: function (element, options) {
        document.getElementById('siteSubmit').addEventListener("click",
            getSiteStatus, false);
```

```
    }
});
```

This code defines the JavaScript related to a page using the *Pages* object of the WinJS UI library. In that definition, a *ready* function is defined. This is akin to the jQuery *ready* functions you've seen throughout the book. Inside of that *ready* handler function, an event listener is added to the element with id *siteSubmit*. The event handler, which is a click event handler, will call the function *getSiteStatus* when the element is clicked.

The first part of the function *getSiteStatus*, seen here, sets up the initial environment by defining variables and setting values. Included among these variables is the URL that will be called at SiteReportr along with parameters that will be sent in the call.

```
function getSiteStatus() {
        var siteSubmit = document.getElementById("siteSubmit");
        siteSubmit.setAttribute("disabled", "true");
        var progress = document.getElementById("progress");
        progress.className = "win-ring";
        var siteURL = document.getElementById('siteURL').value;
        var resultMessage = document.getElementById('resultMessage');
        var resultStaticText = document.getElementById('resultStaticText');
        var appURL = "http://sr.sitereportr.com/siterep.php";
        var appID = "StartHere";
        // REPLACE WITH YOUR KEY VALUE:
        var appKey = "5150";
        var params = "guid=" + appID;
        params += "&site=" + siteURL;
        params += "&key=" + appKey;
```

Next, a Promise object is configured. This Promise object sets several parameters for the URL, the type of call (POST), and the additional data to be sent with the call. If you think this looks much like the *ajax()*, *get()*, and *post()* functions you saw earlier in the book, you're right.

Three functions are defined in the Promise code: *complete*, *error*, and *progress* functions. You can see all of them within the Promise object's definition:

```
        WinJS.Promise.timeout(10000, WinJS.xhr({
            url: appURL,
            type: "POST",
            headers: { "Content-type": "application/x-www-form-urlencoded" },
            data: params
        }).then(function complete(result) {
            var siteSubmit = document.getElementById("siteSubmit");
            siteSubmit.removeAttribute("disabled");
```

```
            var progress = document.getElementById("progress");
            progress.className = "hide";
            if (result.responseText != "Invalid URL") {
                var myResult = JSON.parse(result.responseText);
            } else {
                var myResult = "Unknown";
            }
            if (myResult.status == "Up") {
                resultMessage.innerHTML = myResult.status;
                resultMessage.className = "resultUp";
                resultStaticText.className = "";
                resultMessage.setAttribute("display", "inline");
            } else if (myResult.status == "Down") {
                resultMessage.innerHTML = myResult.status;
                resultMessage.className = "resultDown";
                resultStaticText.className = "";
                resultMessage.setAttribute("display", "inline");
            } else {
                if (myResult.status == "Message") {
                    resultMessage.innerHTML = myResult.message;
                } else {
                    resultMessage.innerHTML = "Unknown";
                }
                resultMessage.className = "resultUnknown";
                resultStaticText.className = "";
                resultMessage.setAttribute("display", "inline");
            }
        }, function error(error) {
            var siteSubmit = document.getElementById("siteSubmit");
            siteSubmit.removeAttribute("disabled");
            var progress = document.getElementById("progress");
            progress.className = "hide";
            resultMessage.innerHTML = "Error";
            resultMessage.className = "resultUnknown";
            resultStaticText.className = "";
            resultMessage.setAttribute("display", "inline");
        },
        function progress(result) {
            var progress = document.getElementById("progress");
            progress.className = "win-ring";
        }));
    } //end function getSiteStatus
```

Notable within this code is a call to the native JSON parsing function, aptly named *parse()*. The parse function—which you can see in the line *var myResult = JSON.parse(result.responseText);*—parses the JSON returned into an object, thus making its properties available with dot notation.

For example, the JSON returned from SiteReportr includes a property named *status* to represent whether the status is Up, Down, Unknown, or other. After being parsed with *JSON.parse* as seen in the previous paragraph, this status is available as *myResult.status*.

## Defining a Splash Screen, Logos, and a Tile

The splash screen is what displays between the time that the user clicks or taps the application tile and when the application is fully loaded. A custom splash screen adds personalization to the application and enhances the user experience.

To add a splash screen, you first need to create an image. Microsoft has several recommendations and requirements for splash screens:

- The image should be a transparent png format.
- The image should be 620 by 320 pixels (optional images at 868 by 420 and 1116 by 540).
- The image should clearly identify the app.

Other recommendations can be found at *http://msdn.microsoft.com/library/windows/apps/hh465338*.

Once you've created an image, you then add it to the application through its package manifest file. The package manifest file contains pointers to resources and overall definitions for your application. The package manifest is found in the file *package.appxmanifest*, which you can open by double-clicking it in Solution Explorer.

The package manifest file is organized into several tabs when viewed through Visual Studio. The Application UI tab contains information pertaining to the logo and application tile images as well as the splash screen. This is depicted in Figure 8-6.

The new splash screen is added by simply clicking Browse and locating the image. It will be added and available the next time you run the application. The same process is used to add tiles and logos to your application as well.

You've now seen a simple, yet fully functional, Windows 8 App. But there's more to Windows 8 Apps than what you've seen. For example, the user interface could be enhanced. Microsoft has several excellent resources and guidelines surrounding the user experience of Windows 8 applications (see *http://msdn.microsoft.com/library/windows/apps)*. More specifically, see "Designing UX for apps" at *http://msdn.microsoft.com/library/windows/apps/hh779072* for information on improving the design of your application.

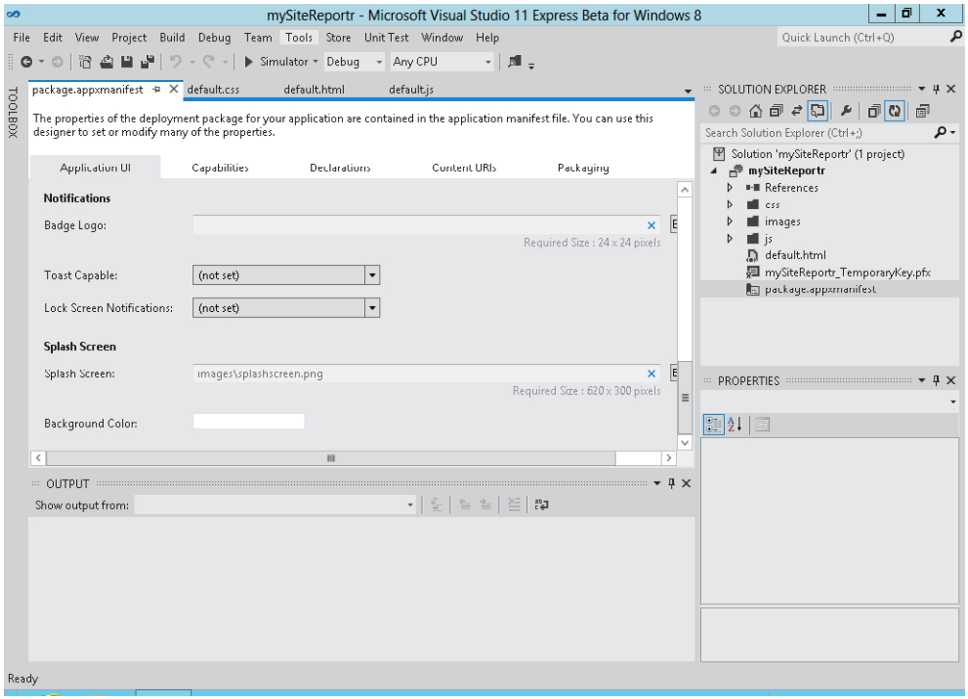


**FIGURE 8-6** The manifest file viewed through Visual Studio.

You can even monetize the app by selling it or by using an ad network such as Bing's AdCenter and placing the ads within the application. Unlike other platforms, Microsoft allows developers to use any ad network, as long as the ads meet the guidelines for the application. See *http://msdn.microsoft.com/library/windows/apps/hh694084* for information about monetizing the app and the Microsoft Advertising site at *http://advertising.microsoft.com/windowsadvertising/developer* for more information on these concepts, including an SDK for Microsoft Advertising.

## Beyond the Basics with Windows 8 Apps

Windows 8 Apps can do much more than you've seen here. Windows 8 Apps can read and write local files, including playing and recording audio and video. Windows 8 Apps can print, work with local devices, and read from and write to Microsoft SkyDrive. You could combine its accelerometer capabilities to create a great game too!

If you'd like to go beyond the basics, I invite you to peruse the Microsoft Windows 8 App site as well as purchase my intermediate-level book, *JavaScript Step by Step* (Microsoft Press, 2011), which has been updated for Windows 8 App development.

## Summary

This chapter was devoted to Windows 8 development with JavaScript. The chapter discussed the use of JavaScript and the WinJS JavaScript library. Within the chapter, you saw the Windows 8 interface and walked through a grid application.

The chapter showed a full example, creating a basic Windows 8 App that called a web service at SiteReportr to check the status of a website. You can use this code as the basis for any web service calls, such as those to other sites (such as Twitter).

Finally, the chapter discussed additional ways to enhance the application, including adding logos, tiles, and a splash screen.

# Index

## Symbols


$(), 99
&& (ampersands), 42
\ (backslash character), 120
<body>, 6
:button selector (jQuery), 113
:checkbox selector (jQuery), 113
:checked filter (jQuery), 113, 121
{} (curly braces), 56
 tag, 79, 107
. (dot), 102
! (exclamation point), 119
/ (forward-slash character), 120
<h1>, 7
# (hash or pound sign), 102
<head> section, 5, 9
<head> tag, 5
:hidden selector (jQuery), 113
<html>, 5
:image selector (jQuery), 113
<img>, 4, 9, 10
:input selector (jQuery), 113
<link> tag, 158
:password selector (jQuery), 113
| (pipe character), 143
|| (pipe characters), 42
<p> tag, 4
:radio selector (jQuery), 113
:reset selector (jQuery), 113
<script>, 9, 10
<script> declaration, 54
<script> tag, 3, 77, 78, 98, 99
<script type="text/javascript">, 2
<select>, 122
:selected filter (jQuery), 113, 123
<select> elements
  validating, 123
; (semi-colon), 3
 element (space travel demo), 183
\s (regular expressions), 118
:submit selector (jQuery), 113
@ symbol, 118
:text selector (jQuery), 113
<title>, 5
\w (regular expressions), 118


## A


AdCenter (Bing), 203
addClass() function, 164
adding
  CSS classes, 164–167
  error styling, to form, 164–167
addition, 46
addNumbers() function, 46, 50–52
  refactoring, 52–54
AJAX (Asynchronous JavaScript and XML), 133–138, 191, 193
  building an interactive page using, 141–143
  and building a server program, 135–138
  error handling with, 144–146
  JavaScript and, 139
  as server-side program, 134, 134–135
ajax() function, 200
ajaxSetup() function, 155
alert() function, 67
allbasicvalidation.html, 124
allbasicvalidation.js, 124
ampersands (&&), 42
angle brackets (< and >), 4
Apache, 134, 135, 138
API key (SiteReportr), 193
append() function, 143

appendTo function (jQuery), 108
Application Compatibility images for Virtual PC, 22
application programming interfaces (APIs), 188
app.start() function, 194
arguments
 function, 46–48
 types of, 48
arrays, 50
 as argument, 47
 objects vs., 66
 of base data, 65
ASP.NET Empty Web Application template, 12
Asynchronous JavaScript and XML. *See* AJAX
attributes, 4
availHeight property (screen object), 90
availWidth property (screen object), 90

## B

back-end languages, 4
backslash (\) character, 120
base.js, 191
Bing, 203
blur event, 106
Booleans, 37–39
borders
 in HTML, 7
Browser Object Model (BOM), 89–95
 Document Object Model and, 74
 events and window object in, 90
 location object in, 93–95
 navigator object in, 92–93
 screen object in, 90–92
browser(s). *See* web browser(s)
Budd, Andy, 159

## C

C#, 4, 187
C++, 188
calendar (jQuery UI widget), 174–176
calling
 functions, 48–49
calls, 3
cascading, 158
Cascading Style Sheets. *See* CSS (Cascading Style Sheets)
case sensitivity, 30–31
chaining (error handlers), 144–146
character classes, 118
Characters Remaining counter, 129
charRemain element, 131
charRemaining variable, 131
charTotal variable, 131
checkbox, finding selected, 121–122
Chrome, 22, 67
class attribute, 97
class(es), 63–66
 CSS. *See* CSS classes
 retrieving elements by, 102
click event, 106, 125–128
click() function, 109, 110
client computer
 unavailability of JavaScript on, 21–22
 variations in, 1
client-server model, 3
client-side execution, 4
closing tags, 4
Collision, Simon, 159
colorDepth property (screen object), 90
comments, 29–30
conditionals, 39, 41–44
 example, 42–44
 order of, 42
constructor patterns, 63
Content Delivery Network (CDN), 74–75
 jQuery library hosted on, 78
context of JavaScript, 3–10
CSS (Cascading Style Sheets), 4, 7–22, 157–159
 changing properties in, 159–163
 in jQuery, 82
 in jQuery UI, 86
 mouse events and, 107
 themes, 74
 working with classes in, 163–167
CSS classes, 163–167
 adding/removing, 164–167
 hasClass() function and, 163–164
css() function, 159
curLength variable, 131
curly braces ({}), 56
customerRegex (variable), 118

## D

databases, 65
data retrieval, 133–156
 with AJAX, 133–138

with jQuery, 139–146
JSON and, 146–148
and sending data to server, 148–155
data security as limitation of JavaScript, 20–22
data types, JavaScript, 35–39
Booleans, 37–39
null, 39
numbers, 35–36
strings, 36–37
undefined, 39
datepicker() function, 174–176
dateRegex (variable), 120
date, validation of, 120
dblclick event, 106
debugging, 67–71
in Internet Explorer, 68–71
as process, 67
in Visual Studio, 17
Debug Output viewing, 20
default.css, 193
default.html, 193
default.js, 193, 199
design skills, 8
desktop widgets, 4
developers, web, 8
Developer Tools add-in (Internet Explorer), 81
DOCTYPE declaration in Visual Studio, 16
Document Object Model (DOM), 9, 95–97
Browser Object Model and, 74
trees in, 96–97
versions of, 95–96
document object, write method of, 3
Document Type Declaration (doctype), 5
Document Type Declarations (DOCTYPE Declarations, DTDs), 5
document.write, 3
dot (.), 102
dot notation, 57
drop-down element, determining selected, 122–125
drop-down lists, radio buttons vs., 122

## E

each() function, 102, 122, 123
Eclipse, 11
ECMA-262 specification, 73
effect() function, 170–171
effects, enhancing a web application with, 167–171
elements. *See also* retrieving elements
HTML, 4
email addresses, validation of, 118
enumeration, object, 61–63
error() function, 145
errorhandler.html, 144
errorhandler.js, 144
errors, 67
AJAX calls, 144–146
in Visual Studio, 20, 80
error styling, adding, 164–167
events (event handling), 105–132
common events, 105–106
in Browser Object Model, 90
keyboard events and forms, 129–131
mouse events, 106–112
web forms, using jQuery to validate, 113–128
exclamation point (!), 119
execution, top-down, 3
expressions, 26–27
external.js, 59, 61, 63, 124
external style sheets, 158

## F

F12 developer tools, 68
file location checking, 20
filters, 113
finger touch, 106
Firebug, 67
Firefox, 22, 67
first program in JavaScript, 2–3, 11–22
focus event, 106
font size in HTML, 7
foreach statement, 40
for loop, 40
formerror.css, 164
formerror.html, 164
formerror.js, 164
formError variable, 117
form(s), adding error styling to, 164–167
forward-slash (/) character, 120
front-end languages, 4
function declaration, 46
functions, 45–55
arguments, function, 46–48
calling, 48–49
examples of, 50–54
overview, 46
return values for, 49–50
scoping and, 54–56
top-down execution and, 3

## G

getColor() method, 61
getElementById() method, 100, 103
getElementsByClassName() function, 102
getElementsByTagName() function, 103
getElementsByTagName() method, 103
get() function, 140, 143, 144, 145, 200
getJSON() function, 147–148, 155
  sending data with, 148–153
GET method, 134, 139, 140–141
GET requests, 21
getters, 59
grid-style application (Microsoft Windows 8), 190–193
groupDetailPage.html, 192
groupedItemsPage.html, 192

## H

hasClass() function, 163–164
hash sign (#), 102
heading, 5
height property (screen object), 90
hide() function (jQuery), 89
hosted libraries, using, 75
hover() function, 108
hover function (jQuery), 109
HTML5, 4, 5
HTML (HyperText Markup Language, 2, 4–22
  CSS vs., 7
HTML tag name, retrieving elements by, 102–104
Hypertext Transfer Protocol (HTTP), 134

## I

ID, retrieving elements by, 100–102
id attribute, 97
identifiers (ids), 8
if, 41
inline styles, 8
Integrated Development Environment (IDE), 11
interactions, 167
Internet Explorer
  allowing blocked content with, 77, 78
  debugging in, 67, 68–71
  Developer Tools add-in, 81
  DOM and, 95
  standards and, 96

## J

Java programming language, JavaScript vs., 3
JavaScript. *See also* specific topics
  about, 8
  context of, 3–10
  CSS and, 7–22
  first program in, 2–3, 11–22
  HTML and, 4–22
  limitations of, 20–22
  retrieving elements with, 98
  unavailability of, 10
JavaScript interpreter, 2
jQuery, 74–81
  form-related selectors in, 113
  get() and post() methods in, 140–141
  getting, 74–75
  retrieving data with, 139–146
  retrieving elements with, 98–104
  selectors in, 100
  testing, 79–81
  using a CDN-hosted jQuery library, 78
  using a local copy of, 75–77
  web form validation using, 113–128
  widgets in, 172–176
jquery() function, 99
jquery.html, 75
jQuery ready() function, 9
jQuery UI, 81–89
  adding, to a project, 82–86
  advanced styling effects using, 167–176
  getting, 81–82
  testing, 86–89
JSON (JavaScript Object Notation), 146–148, 202

## K

keyboard events and forms, 129–131
keydown events, 106, 129
keypress event, 106, 129
keyup events, 106, 129, 131
keyword function, 46

## L

legacy DOM, 95
libraries, 74, 188
  hosted vs. local, 75

limitations of JavaScript, 20–22
 data security, 20–22
 unavailability on client computer, 21–22
line breaks, 28
literal values, 48
load() method, 99
local libraries, using, 75
location object (Browser Object Model), 93–95
logical AND, 42
logical OR, 42
logo, in Windows 8 Apps, 202–203
loop.html, 40
loops (looping), 40–41

## M

matching tags, 4
messageText element, 131
methods, 58–61
Microsoft Advertising, 203
Microsoft Internet Information Services, 134
Microsoft SkyDrive, 204
Microsoft Windows 8, 187–206
 building a Windows 8 application in, 193–204
 grid-style application in, 190–193
 prominence of JavaScript in, 187–189
Moll, Cameron, 159
mousedown event, 106, 107
mouse events, 106–112
mousemove event, 106
mouseout event, 106, 107
mouseover event, 106, 107
mouseup event, 106, 107
MSDN, 187
Multipurpose Internet Mail Extensions (MIME), 10
multiselect lists, 123

## N

name collisions, 49
names, 27–28
navigator object (Browser Object Model), 92–93
new projects, creating, 12
Node.js, 4
Notepad, 11
null data type, 39
number, 35–36

## O

object enumeration, 61–63
object literals, 56
objects, 56–66
 appearance of, 56
 arrays vs., 66
 classes and, 63–66
 methods and, 58–61
 properties and, 56–58
 this keyword and, 59–61
 ways of creating, 56
on() function, 112, 125, 128
onload() method, 99
open() method, 139
operators, 31–32
order
 of conditionals, 42
 with HTML, 98

## P

PageControlNavigator control, 191
parentheses (in functions), 46, 48
parse() function, 202
parsing, 3, 9
PHP, 4
 creating a server program using, 138
pipe (|) character, 143
pipe characters (||), 42
pointing devices, 106
post() function, 144, 153, 155, 200
POST method, 134, 139, 140–141
POST requests, 21
pound sign (#), 102
preventDefault() method, 111, 112
programming, for web vs. other platforms, 1
programming in JavaScript, 23–44, 45–72
 case sensitivity and, 30–31
 comments, 29–30
 conditionals, 41–44
 data types, 35–39
 debugging and, 67–71
 expressions, 26–27
 functions, 45–55
 line breaks, 28
 looping, 40–41
 names, 27–28
 object enumeration and, 61–63

objects and, 56–66
operators, 31–32
reserved words, 27–28
scripts and, 23–44
spacing, 28
statements, 26–27
strings, 36–44
syntax, 26–32
variables, 32–35
Project Properties pane (StartHere project), 138
Promise object, 200
properties, 56–58
CSS, 7
prop() function, 125
pseudo-classes, 63, 65
Python, 4

## Q

Quirks Mode, 5
quotes, 57

## R

radio buttons
drop-down lists vs., 122
finding selected, 121–122
ready() function, 81, 99, 108, 117, 130, 142, 200
redirect() function, 95
refactoring, 47
addNumbers(), 52–54
regular expressions, 118–121
removeClass() function, 164
removing CSS classes, 164–167
rendering, 4
replace() method, 94
Request for Comments (RFC) number 2616, 134
reserved words, 27–28, 57
retrieving elements, 98–104
by class, 102
by HTML tag name, 102–104
by ID, 100–102
with jQuery, 99–100
return false statement, 111, 112
return keyword, 49
return values (for functions), 49–50

## S

Safari, 22
saving, in Visual Studio, 17
scoping, 54–56
screen object (Browser Object Model), 90–92
scripts, 2, 9
placing, 23–44
types of, 10
scroll event, 106
security, GET/POST requests and, 141
selectors, 8, 100, 157–158
semi-colon (;), 3
server-based languages, 4
server program. building a, 135–138
server(s), 3
retrieving data from, 133
sending data to, 148–155
sending data to the, 148–155
setColor() method, 61
setters, 59
setTimeout() method, 95
show() function (jQuery), 89
SiteReportr, 193
slider() function, 172–174
slider (jQuery UI widget), 172–174
Software Development Kit (SDK), 187, 189
Solution Explorer, 190
Solution Explorer area (Visual Studio), 19
space travel demo, 176–186
code analysis, 183–186
spacing, 28
special characters, in regular expressions, 119
splash screen, in Windows 8 Apps program, 202–203
splitResp variable, 143
src attribute, 4, 10, 77
statements, 26–27
strings, 36–37
styles and styling, 157–186
changing styles with JavaScript, 157–163
space travel demo, 176–186
using CSS classes to apply, 163–167
using jQuery UI for advanced effects, 167–176
submit button, 113
submit event, 106
submit() event, 117
submit() function, 117
submit, validating on, 113–118

syntax
 checking, 20
 JavaScript, 26–32

## T

tag name, retrieving elements by, 102–104
tags, HTML, 4
TCP connections, 75
temperature conversion program, 149–155
template layouts (Windows 8), 189
templates, 12
testing
 jQuery, 79–81
 jQuery UI, 86–89
test() method, 118, 119
text editors, 11
text, for JavaScript programs, 3
thermostats, 105
this keyword, 59–61
tile images, in Windows 8 Apps, 202–203
tiles, 189
toggle() function, 109
tool-agnosticism, 11
top-down execution, 3
trackpads, 106
transfer effect, 171
trees, in Document Object Model, 96–97
troubleshooting and debugging, 67–71
Twitter, 129
type attribute (<script> tag), 10

## U

ui.js, 191
UI page (Windows 8 Apps), 199
ulElm (variable), 103
unavailability of JavaScript, 10
 on client computer, 21–22
undefined data types, 39
URL property, 138
urlRegex (variable), 120
URL (Uniform Resource Locator), 3
userAgent property, 93
usernamesLength, 66

## V

val() function, 117, 131
validation
 of date, 120
 of email addresses, 118
 error styling for, 164–167
 of radio buttons and checkboxes, 122
 of web forms, using jQuery. *See* web forms, using jQuery to validate
 of <select> elements, 123
variable declaration rule, 54
variables, 48
 Booleans, 37–39
 JavaScript, 32–35
 names of, 48
viewing, Debug Output, 20
Vim, 11
Visual Basic, 4
Visual Studio, 5, 11, 67
 building a server program with, 135–138
 debugging in, 17
 development server in, 135
 DOCTYPE declaration in, 16
 error notification in, 80
 errors in, 20
 Express Edition of, 11, 12
 saving in, 17
 Solution Explorer area of, 19
 Windows 8 and, 11
Visual Studio 11, 189
 writing JavaScript in, 12–20

## W

W3C, 95
web applications, enhancing with effects, 167–171
web browser(s), 3, 73–104
 Browser Object Model and, 89–95
 CSS properties and, 7
 DOCTYPE and, 5
 Document Object Model and, 95–97
 first JavaScript program viewed in, 2
 JavaScript libraries and, 74
 JavaScript on other, 21
 jQuery and, 74–81, 98–104
 jQuery UI and, 81–89
web developers, 8

web forms, using jQuery to validate, 113–128
 click event and, 125–128
 drop-down element, determining selected, 122–125
 radio button or checkbox, finding selected, 121–122
 with regular expressions, 118–121
 on submit, 113–118
webpage, placing JavaScript in a, 9–22
webpages, 3
Wempen, Faithe, 6
while loop, 40
widgets, 167
 desktop, 4
 jQuery, 172–176
widgets, jQuery UI
 calendar, 174–176
 slider, 172–174
width property (screen object), 90
window object, in Browser Object Model, 90
Windows 8, 9–22
 Visual Studio and, 11
Windows 8 Apps, 193–204
 building, 193–199
 code analysis, 199–202
 grid-style, 190–193
 logo in, 202–203
 splash screen in, 202–203
 tile images in, 202–203
Windows 8 Runtime (Windows RT), 9
Windows 8 user interface, 188–189
WinJS, 9
WinJS library, 193
WinJS UI library, 200
words, reserved, 27–28
World Wide Web Consortium (W3C), 4, 73
write method, 3
www.w3schools.com, 8

## X

XHTML, 5
XML, AJAX and, 134, 146–147
XMLHttpRequest object, 134, 139, 140
xmlns namespace attribute (<html> tag), 16

## Z

zip5Regex (variable), 120
zipRegex (variable), 120