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

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

Contents

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://www.microsoftpressstore.com/title/9780735666740

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:

http://www.microsoftpressstore.com/title/9780735666740

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.

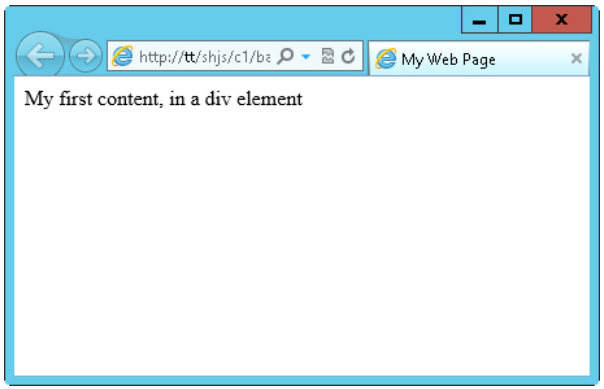

FIGURE 1-2 A basic webpage.

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.

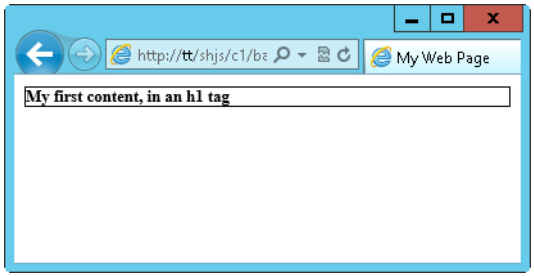

FIGURE 1-3 Applying a bit of style with CSS.

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).

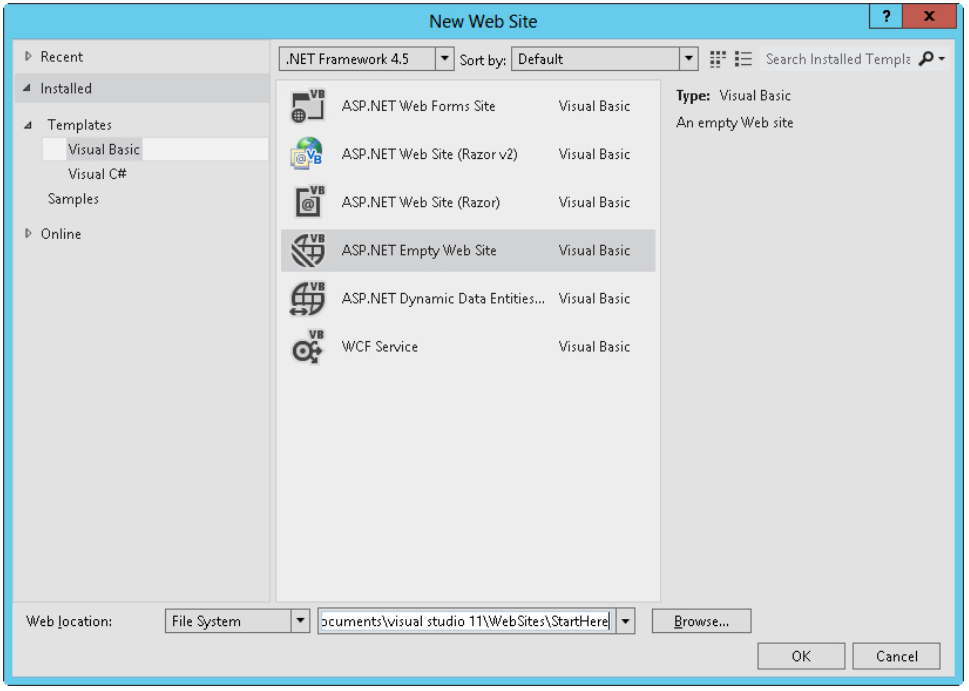

FIGURE 1-4 Creating an ASP.NET Empty Web Application project in Visual Studio 11.

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

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

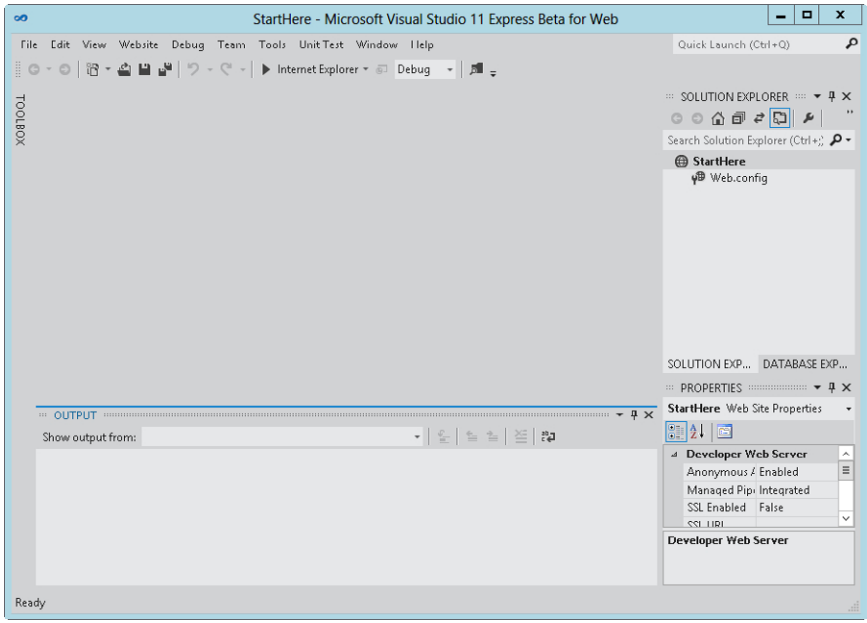

FIGURE 1-5 An empty web project in Visual Studio 11.

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.

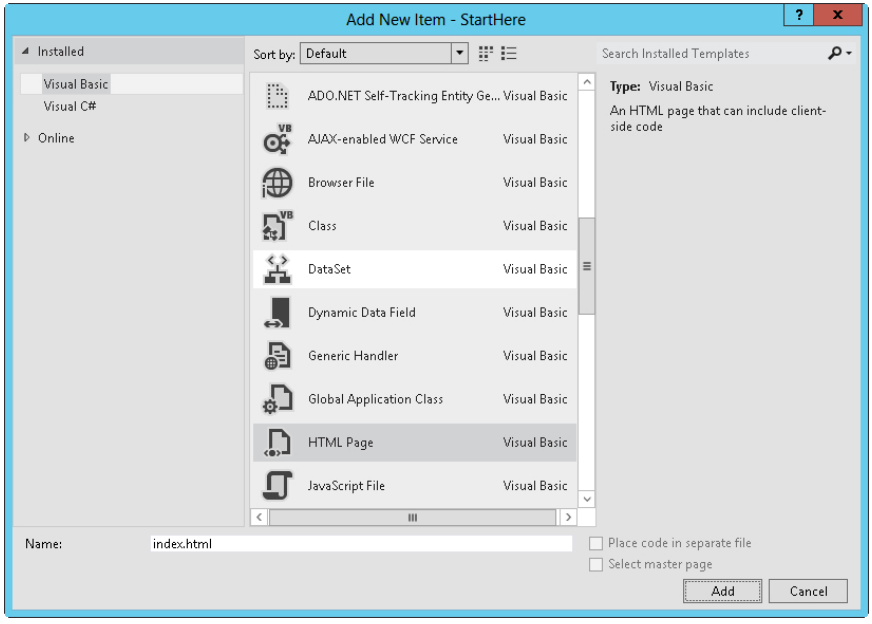

FIGURE 1-6 The New File dialog within Visual Studio 11 is where you add new files to your project.

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.

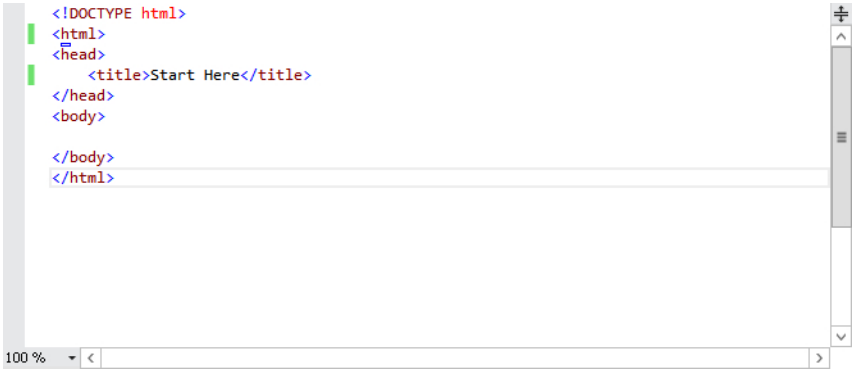

FIGURE 1-8 Making basic edits on your first webpage through Visual Studio.

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.

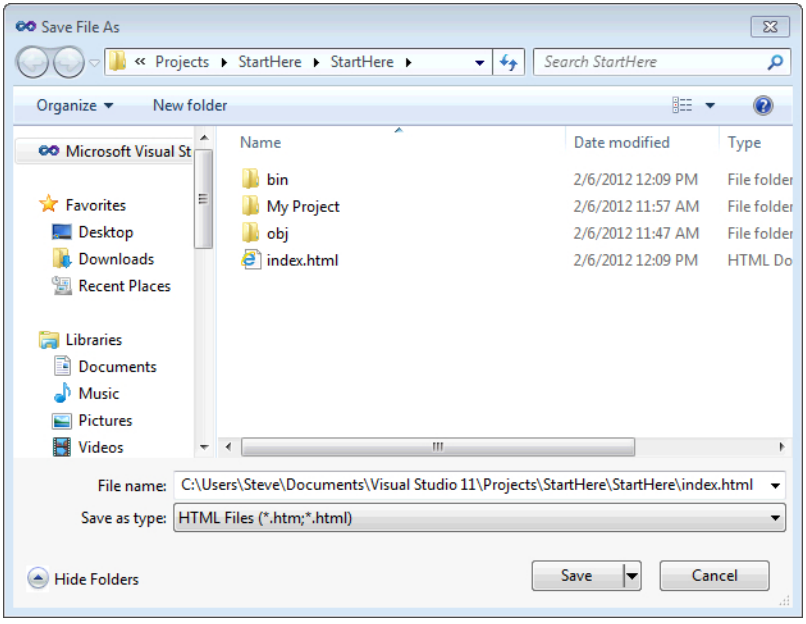

FIGURE 1-9 Saving the webpage as *index.html* within the project.

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.

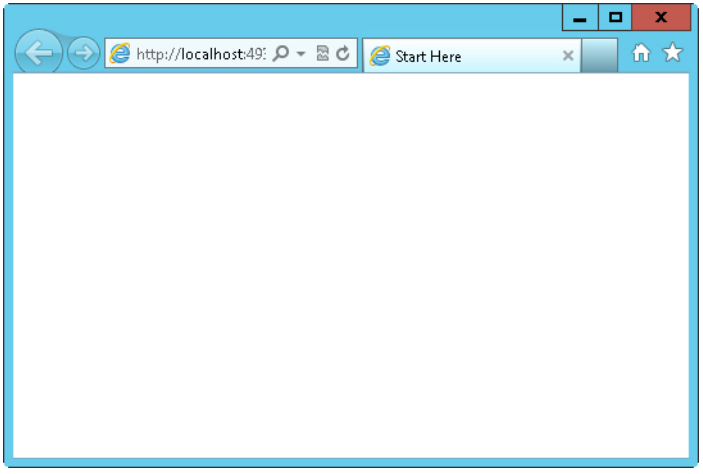

FIGURE 1-10 A successful run of your first webpage.

   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.

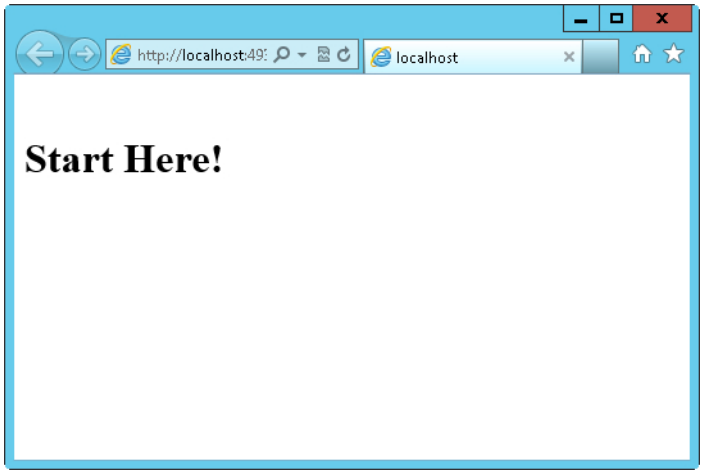

FIGURE 1-11 Running your first JavaScript program.

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.

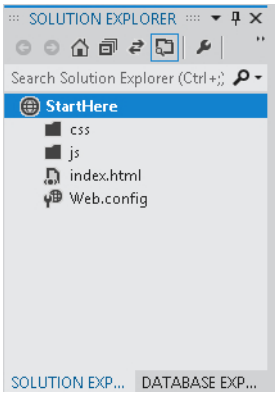

FIGURE 1-12 The final Solution Explorer with two folders added.

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.

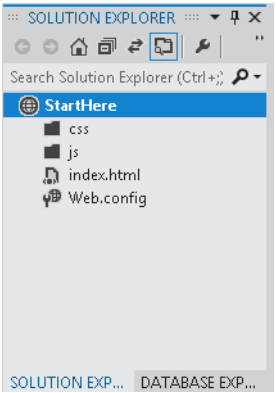

FIGURE 2-1 The folder structure for development for the book should include css and js folders within the project.

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.

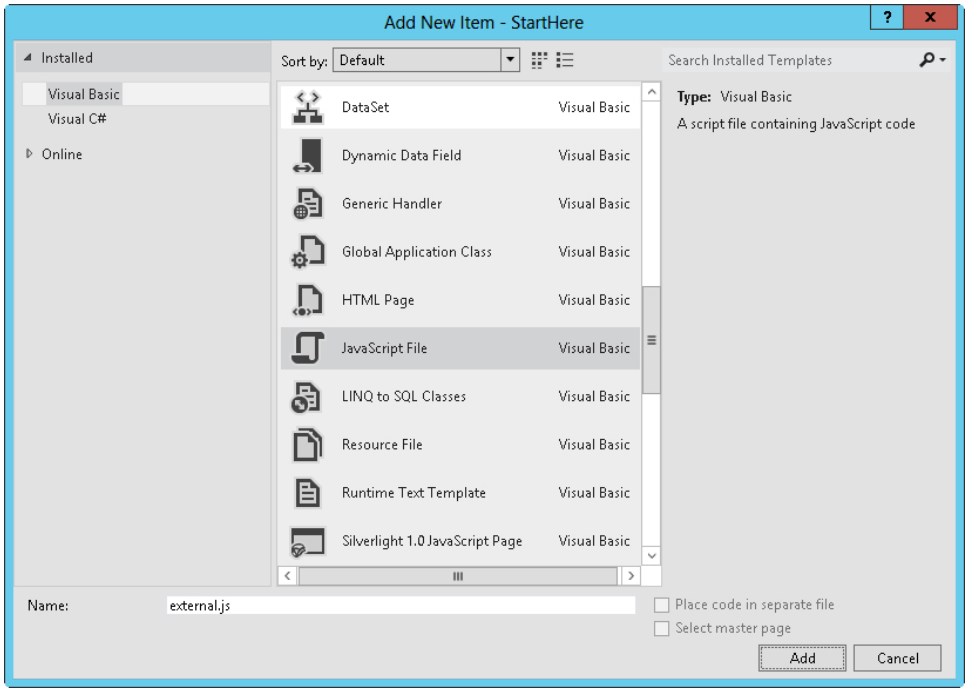

FIGURE 2-2 Creating a new JavaScript file.

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.)

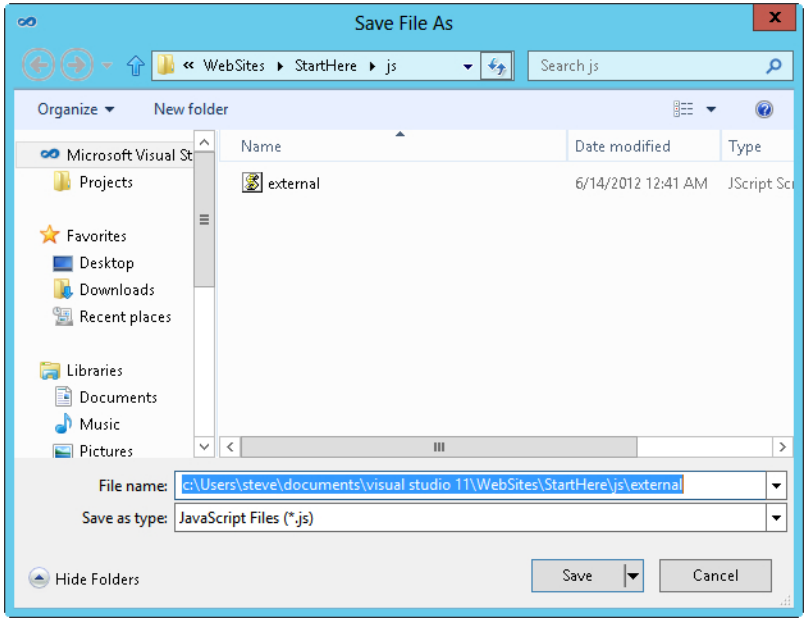

FIGURE 2-3 Saving the *external.js* file into the js folder.

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.

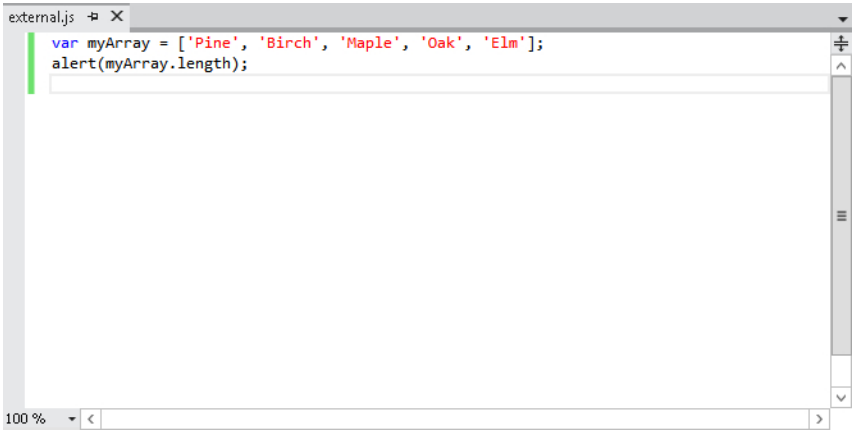

FIGURE 2-4 The *external.js* file should contain the code shown in this example.

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.

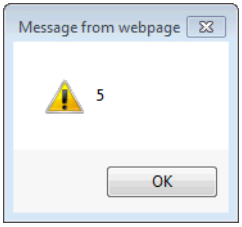

FIGURE 2-5 The length of the *myArray* array, thanks to the *length* property.

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.

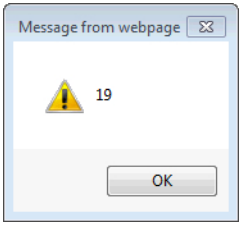

FIGURE 2-6 The alert shows that the length of *myArray* is 19.

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.

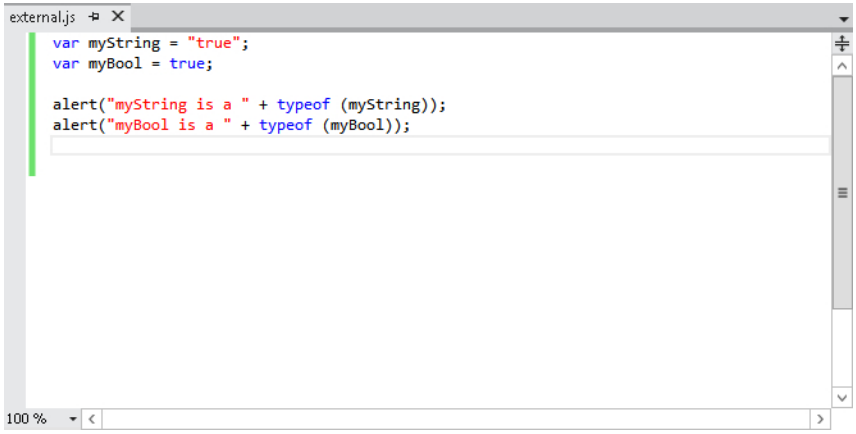

FIGURE 2-7 The *external.js* file should contain this JavaScript.

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.

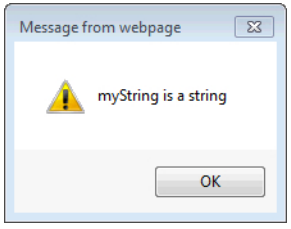

FIGURE 2-8 The variable *myString* is a string value, even though the string is set to the word *true*.

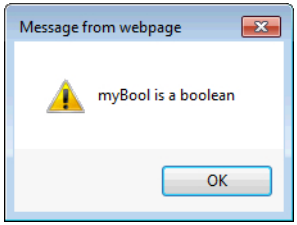

FIGURE 2-9 The variable *myBool* is a true Boolean value.

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.

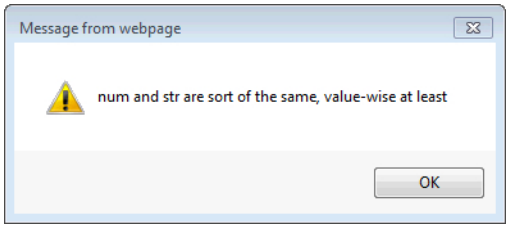

FIGURE 2-10 Testing equality operators with a string and number.

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.

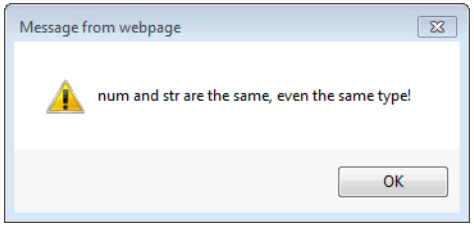

FIGURE 2-11 Evaluating two numbers in JavaScript, using the strict equality test.

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.

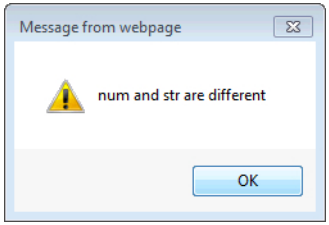

FIGURE 2-12 Testing equality with different values.

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.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z