Microsoft

developer

// Step by step

JavaScript

Third Edition

Intermediate

Steve Suehring

// Step by step

Your hands-on guide to JavaScript fundamentals

Expand your expertise—and teach yourself the fundamentals of JavaScript. If you have previous programming experience but are new to JavaScript, this tutorial delivers the step-by-step guidance and coding exercises you need to master core topics and techniques.

Discover how to:

- Work with JavaScript syntax, variables, and data types
- Master techniques for building cross-browser programs
- Speed up and simplify app development with jQuery
- Quickly retrieve data from a server using AJAX requests
- Adapt your app for mobile devices with jQuery Mobile
- Build Windows 8 apps using HTML, CSS, and JavaScript

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. He is also the author of *Start Here! Learn JavaScript*, a book designed for people with no previous programming experience.

Practice Files + Code

Available at:
http://aka.ms/JavaScriptSbS/files

Companion eBook

See the instruction page at the back of the book

microsoft.com/mspress

ISBN: 978-0-7356-6593-4

9 780735 665934 90000

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Programming/JavaScript

Microsoft

Microsoft

JavaScript Step by Step, Third Edition

Steve Suehring

ISBN: 978-0-7356-6593-4

3 4 5 6 7 8 9 10 11 LSI 8 7 6 5 4 3

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: Melanie Yarbrough

Editorial Production: nSight, Inc.

Technical Reviewer: John Grieb

Copyeditor: nSight, Inc.

Indexer: nSight, Inc.

Cover Design: Twist Creative • Seattle

Cover Composition: Karen Montgomery

Illustrator: nSight, Inc.

[2013-09-09]

I would like to dedicate this book to Mom and Dad.

—Steve Suehring

Contents at a glance

Contents

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

Much has changed since the first edition of JavaScript Step by Step was written in 2007: the underlying JavaScript specification received a major update; Microsoft released new versions of Internet Explorer and Windows; the Chrome browser came of age, as did mobile web usage; and JavaScript development frameworks have matured and are now ubiquitous.

This third edition of *JavaScript Step by Step* builds on the foundation laid down by the first two editions. While the underlying architecture of the JavaScript language has remained largely the same, use of JavaScript has become pervasive, with huge year-over-year increases and an expanded importance to developers. With that in mind, the layout and coverage of the book have also remained largely the same, with some notable exceptions. The book now places extra emphasis on JavaScript event handling and the use of jQuery to speed development. The book also includes a final section on Windows 8 development using JavaScript. However, this book is most definitely not Microsoft-centric.

One of the first things I asked prior to accepting the offer to write *JavaScript Step by Step* was whether it had to focus on Microsoft products. The answer was a firm “no.” The book was and is intended to be a general tutorial on using JavaScript, including best practices for using JavaScript on the web.

The biggest influence Microsoft has had on this book was to make sure that I used the term “Internet Explorer” when referring to IE. You’ll see this absence of bias reflected throughout the book, which includes exercises built using plain text editors as well as full-featured development tools. While it’s true that most of the screen shots show Internet Explorer (I almost said IE), the code was also tested across several other browsers, including Chrome and Firefox. In fact, much of the book’s code was written in Vim, and tested in a cross-browser fashion.

Throughout the book, you’ll find highlights and additions for the new features in the latest version of JavaScript. Also, the examples used in the book received greater scrutiny in multiple browsers to reflect the reality of today’s web landscape. Reader feedback from the earlier editions is reflected in the content of this edition and was the impetus for adding jQuery and emphasizing event handling.

Housekeeping aside, this book provides an introductory look at JavaScript, including some of its core functions as well as features and paradigms such as Asynchronous JavaScript and XML (AJAX).

The first part of the book examines JavaScript and helps you get started developing JavaScript applications. You don't need any specific tools for JavaScript development, so you'll see how to create JavaScript files in Microsoft Visual Studio, and in Appendix B, in Eclipse and in Notepad (or any other text editor). The book examines JavaScript functions and the use of JavaScript in the browser, along with the aforementioned jQuery. Finally, the book provides coverage of Windows 8 app development using HTML, CSS, and JavaScript.

Who should read this book

This book is for beginning JavaScript programmers or people who are interested in learning the basics of modern JavaScript programming: the language syntax, how it works in browsers, what the common cross-browser problems are, and how to take advantage of AJAX and third-party libraries such as jQuery to add interactivity to your webpages.

Assumptions

This book expects that you have at least a minimal understanding of concepts surrounding web development. You should be at least somewhat familiar with HTML. CSS is also helpful to know, but neither HTML nor CSS are required prerequisite knowledge for completing this book. The examples shown provide all the HTML and CSS whenever necessary.

Who should not read this book

This book isn't meant for experienced JavaScript programmers.

Organization of this book

This book is divided into four sections, each of which focuses on a different aspect of JavaScript programming. Part I, "Javawhat? The where, why, and how of JavaScript," provides the foundation upon which JavaScript is programmed. Included in this part are chapters to get you up to speed creating JavaScript programs as well as chapters discussing the syntax of JavaScript. Part II, "Integrating JavaScript into design," looks closely at the interactions between JavaScript and its primary role of web programming.

Part III, “AJAX and server-side integration,” shows the use of JavaScript to retrieve and parse information from web services. Finally, Part IV, “JavaScript and Windows 8,” shows how to create a Windows 8 app with HTML, CSS, and JavaScript.

Conventions and features in this book

This book takes you step by step through the process of learning the JavaScript programming language. Starting at the beginning of the book and following each of the examples and exercises will provide the maximum benefit to help you gain knowledge about the JavaScript programming language.

If you already have some familiarity with JavaScript, you might be tempted to skip the first chapter of this book. However, Chapter 1, “JavaScript is more than you might think,” details some of the background history of JavaScript as well as some of the underlying premise for this book, both of which might be helpful in framing the discussion for the remainder of the book. Chapter 2, “Developing in JavaScript,” shows you how to get started with programming in JavaScript. If you’re already familiar with web development, you might already have a web development program, and therefore you might be tempted to skip Chapter 2 as well. Nevertheless, you should become familiar with the pattern used in Chapter 2 to create JavaScript programs.

The book contains a Table of Contents that will help you to locate a specific section quickly. Each chapter contains a detailed list of the material that it covers.

The coverage of Windows 8 app development is limited to the final section of the book, so if you’re not interested in making a Windows 8 app (it’s really easy) then you can safely skip that last section without missing any of the valuable information necessary to program in JavaScript for the web. If you’re looking for a more comprehensive book on Windows 8 development with HTML5 and JavaScript, a beginner’s book, *Start Here! Build Windows 8 Apps with HTML5 and JavaScript* will be available from Microsoft Press in May (pre-order here: *http://oreil.ly/build-w8-apps-HTML5-JS*).

In addition, you can download the source code for many of the examples shown throughout the book.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- An operating system capable of running a web server. For the section on Windows 8 development, you'll need Windows 8, but none of the other examples require Windows.
- A text editor such as Notepad, Vim, or an Integrated Development Environment (IDE) such as Visual Studio or Eclipse. For Windows 8 development, you'll specifically need Visual Studio 2012 for Windows.
- An Internet connection so you can download software and chapter examples.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. The code for those exercises and many other examples can be downloaded from:

http://aka.ms/JavaScriptSbS/files

Follow the instructions to download the 9780735665934_files 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 9780735665934_files.zip file that you downloaded from the book's website to a location that is accessible by your web server.
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 webpage from which you downloaded the 9780735665934_files.zip file.

Using the Code Samples

Unzipping the sample code creates several subfolders, one for each chapter of the book. These subfolders may contain additional subfolders, based on the layout necessary for a given chapter.

Acknowledgments

There are so many people that help in the book writing process. I'd like to first thank Russell Jones at Microsoft Press. He has provided excellent guidance and is always a pleasure to work with. Thanks to John Grieb for providing excellent technical feedback for this edition. I should also thank Jim Oliva and John Eckendorf, if for no other reason than I do so in every other book I write.

I'd also like to thank Terry Rapp for being understanding about my scheduling conflicts. Thanks to Chris Tuescher for years of friendship and support. Many people have helped me through the years, and it all led to me sitting here, writing this sentence. Duff Damos, Kent Laabs, Pat Dunn, and the entire Nightmare Productions Ltd and Capitol Entertainment crews are just as responsible for getting me here as anyone. Thanks to Dave, Sandy, Joel, and the gang at Ski's. Thanks also to Mrs. Mehlberg and Mrs. Jurgella for extra attention and just being great.

Finally, thank you, dear reader. This book has been highly successful (at least by my standards) and your feedback and contact over the years has been helpful. Please follow me on Twitter @stevesuehring or drop me a line by going to my website at *http://www.braingia.org*.

Errata & 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://aka.ms/JavaScriptSbS/errata

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*

PART I

Javawhat? The where, why, and how of JavaScript

The first section of the book, by far the largest, includes just about everything you need to know about basic JavaScript syntax. You'll start things off by learning some of the background and history of JavaScript. It may sound boring, but Chapter 1 will help to frame the discussion and tone of the remainder of the book.

Chapter 2 gets you started with JavaScript development by showing how to create a page with JavaScript using Visual Studio. See Appendix B for information about using Eclipse or a text editor such as Notepad.

The discussion of JavaScript syntax begins in Chapter 3 and continues for the remainder of the first part of the book. You'll learn about conditionals, operators, functions, loops, and even get an introduction to jQuery.

CHAPTER 1

JavaScript is more than you might think

After completing this chapter, you will be able to

- Understand the history of JavaScript.
- Recognize the parts of a JavaScript program.
- Use the *javascript* pseudo-protocol.
- Understand where JavaScript fits within a webpage.
- Understand what JavaScript can and cannot do.
- Understand how JavaScript is used in Windows 8.

A brief history of JavaScript

JavaScript isn't Java. There! With that clarification out of the way, you can move on to bigger, more important learning, like how to make cool sliders. In all seriousness, JavaScript is one implementation of a specification known as ECMAScript. You'll learn more about ECMAScript later in this chapter.

Where did JavaScript come from? You might not know the rich and storied history of JavaScript—and you might not really care much about it, either. If that's the case, you might be tempted to jump ahead to the next chapter and begin coding JavaScript. Doing so, of course, would be a mistake—you'd miss all the wonderful information that follows in this chapter. And understanding a bit about the history of JavaScript is important to understanding how the language is implemented in various environments today.

JavaScript was originally developed by Brendan Eich at Netscape sometime in 1995–1996. Back then, the language was called LiveScript. That was a great name for a new language—and the story could have ended there. However, in an unfortunate decision, the folks in marketing had their way, and the language was renamed to JavaScript. Confusion soon ensued. You see, Java was the exciting new language at the time, and someone decided to try to capitalize on Java's popularity by using its name. As a result, JavaScript found itself associated with the Java language. This was a disadvantage for JavaScript, because Java, although popular in the sense that it was frequently used, was also unpopular because it had earned a fairly bad reputation—developers used Java in websites to present data or to add useless enhancements (such as annoying scrolling text). The user experience suffered because Java required a plug-in to load into the web browser, slowing down the browsing process and causing grief for visitors and accessibility problems. Only in recent years has JavaScript begun to separate from this negative Java association, but, almost weekly, I still hear people confuse Java and JavaScript. You'll hopefully no longer do that!

JavaScript is not a compiled language, which makes it look and feel like a language that lacks power. But programmers new to JavaScript soon came to realize its strengths and usefulness for both simulating and creating interactivity on the World Wide Web. Up until that realization, programmers developed many websites using only simple Hypertext Markup Language (HTML) and graphics that often lacked both visual appeal and the ability to interact with the site's content. With Microsoft Windows 8, JavaScript now has an avenue for creating full-fledged applications that don't rely on the web browser.

Early JavaScript concentrated on client-side form validation and working with images on webpages to provide rudimentary, although helpful, interactivity and feedback to the visitor. When a visitor to a website filled in a form, JavaScript instantly validated the contents of the web form rather than making a round-trip to the server. Especially in the days before broadband was pervasive, preventing the round-trip to the server was a great way to help applications seem a little quicker and more responsive—and it still is.

Enter Internet Explorer 3.0

With the release of Microsoft Internet Explorer 3.0 in 1996, Microsoft included support for core JavaScript, known in Internet Explorer as JScript, and support for another scripting language called Microsoft Visual Basic, Scripting Edition, or VBScript. Although JavaScript and JScript were similar, their implementations weren't exactly the same. Therefore, methods were developed to detect which browser the website visitor was using and respond with appropriate scripting. This process is known as *browser detection*, and is discussed in Chapter 11, "An introduction to jQuery." Although it is considered undesirable for most applications, you'll still see browser detection used, especially with the advent of mobile devices that have their own special look and feel.

And then came ECMAScript

In mid-1997, Microsoft and Netscape worked with the European Computer Manufacturers Association (ECMA) to release the first version of a language specification known as ECMAScript, more formally known as ECMA-262. Since that time, all browsers from Microsoft have implemented versions of the ECMAScript standard. Other popular browsers, such as Firefox, Safari, and Opera, have also implemented the ECMAScript standard.

ECMA-262 edition 3 was released in 1999. The good news is that browsers such as Microsoft Internet Explorer 5.5 and Netscape 6 supported the edition 3 standard, and every major browser since then has supported the version of JavaScript formalized in the ECMA-262 edition 3 standard. The bad news is that each browser applies this standard in a slightly different way, so incompatibilities still plague developers who use JavaScript.

The latest version of ECMAScript, as formalized in the standard known as ECMA-262, was released in late 2009 and is known as ECMA-262 edition 5. Version 4 of the specification was skipped for a variety of reasons and to avoid confusion among competing proposals for the standard. ECMA-262 edition 5.1 is becoming more widely supported as of this writing and will likely (I'm hopeful) be in versions of popular browsers such as Internet Explorer, Chrome, Firefox, Opera, and Safari by the time you read this book.

It's important to note that as a developer who is incorporating JavaScript into web applications, you need to account for the differences among the versions of ECMA-262, and among the many implementations of JavaScript. Accounting for these differences might mean implementing a script in slightly different ways, and testing, testing, and testing again in various browsers and on various platforms. On today's Internet, users have little tolerance for poorly designed applications that work in only one browser.

Accounting for those differences has become much easier in the last few years, and there are two primary reasons. First, web browsers have consolidated around the specifications for HTML, CSS, and JavaScript, and the vendors have worked to bring their interpretation of the specifications closer to one another. The second reason that accounting for differences has become easier is that JavaScript libraries have become more popular. Throughout the book, I'll show the use of the jQuery library to make JavaScript easier.

Important It is imperative that you test your websites in multiple browsers—including web applications that you don't think will be used in a browser other than Internet Explorer. Even if you're sure that your application will be used only in Internet Explorer or if that's all you officially support, you still should test in other browsers. This is important both for security and because it shows that you're a thorough developer who understands today's Internet technologies.

So many standards...

If you think the standards of JavaScript programming are loosely defined, you're right. Each browser supports JavaScript slightly differently, making your job—and my job—that much more difficult. Trying to write about all these nuances is more challenging than writing about a language that is implemented by a single, specific entity, like a certain version of Microsoft Visual Basic or Perl. Your job (and mine) is to keep track of these differences and account for them as necessary, and to try to find common ground among them as much as possible.

The DOM

Another evolving standard relevant to the JavaScript programmer is the *Document Object Model (DOM)* standard developed by the World Wide Web Consortium (W3C). The W3C defines the DOM as "a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of documents." What this means for you is that you can work with a specification to which web browsers adhere to develop a webpage in a dynamic manner. The DOM creates a tree structure of objects for HTML and Extensible Markup Language (XML) documents and enables scripting of those objects. JavaScript interacts heavily with the DOM for many important functions.

Like JavaScript, the DOM is interpreted differently by every browser, making life for a JavaScript programmer more interesting. Internet Explorer 4.0 and earlier versions of Netscape included support for an early DOM, known as Level 0. If you use the Level 0 DOM, you can be pretty sure that you'll find support for the DOM in those browsers and in all the browsers that came after.

Microsoft Internet Explorer 5.0 and Internet Explorer 5.5 included some support for the Level 1 DOM, whereas Windows Internet Explorer 6.0 and later versions include some support for the Level 2 DOM. The latest versions of Internet Explorer, Chrome, Firefox, Safari, and Opera support the Level 3 DOM in some form. Safari provides a representation of the WebKit rendering engine. The WebKit rendering engine is also used as the basis for the browser on devices such as the iPhone and iPad and on Android-based devices.

If there's one lesson that you should take away while learning about JavaScript standards and the related DOM standards, it's that you need to pay particular attention to the code that you write (no surprise there) and the syntax used to implement that code. If you don't, JavaScript can fail miserably and prevent your page from rendering in a given browser. Chapter 12, "The Document Object Model," covers the DOM in much greater detail.

Tip The W3C has an application that can test the modules specified by the various DOM levels that your web browser claims to support. This application can be found at *http://www.w3.org/2003/02/06-dom-support.html*.

What's in a JavaScript program?

A JavaScript program consists of *statements* and *expressions* formed from *tokens* of various categories, including keywords, literals, separators, *operators*, and *identifiers* placed together in an order that is meaningful to a JavaScript interpreter, which is contained in most web browsers. That sentence is a mouthful, but these statements are really not all that complicated to anyone who has programmed in just about any other language. An expression might be:

```
var smallNumber = 4;
```

In that expression, a token, or reserved word—*var*—is followed by other tokens, such as an identifier (*smallNumber*), an operator (=), and a literal (*4*). (You learn more about these elements throughout the rest of the book.) The purpose of this expression is to set the variable named *smallNumber* equal to the integer *4*.

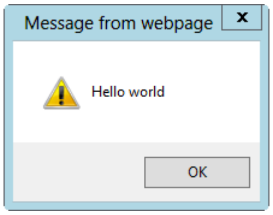
Like in any programming language, statements get put together in an order that makes a program perform one or more functions. JavaScript defines functions in its own way, which you read much more about in Chapter 7, "Working with functions." JavaScript defines several built-in functions that you can use in your programs.

Using the *javascript* pseudo-protocol and a function

1. Open a web browser.
2. In the address bar, type the following code and press Enter:

   ```
   javascript:alert("Hello World");
   ```

   After you press Enter, you see a dialog box similar to this one:



Hello world

Congratulations! You just programmed your first (albeit not very useful) bit of JavaScript code. However, in just this little bit of code, are two important items that you are likely to use in your JavaScript programming endeavors: the *javascript* pseudo-protocol identifier in a browser and, more importantly, the *alert* function. You'll examine these items in more detail in later chapters; for now, it suffices that you learned something that you'll use in the future!

Note Internet Explorer 10 in Windows 8 sometimes doesn't display or use the *javascript* pseudo-protocol correctly.

JavaScript is also *event-driven*, meaning that it can respond to certain events or "things that happen," such as a mouse click or text change within a form field. Connecting JavaScript to an event is central to many common uses of JavaScript. In Chapter 11, you see how to respond to events by using JavaScript.

JavaScript placement on your webpage

If you're new to HTML, all you need to know about it for now is that it delineates elements in a webpage using a pair of matching tags enclosed in brackets. The closing tag begins with a slash character (/). Elements can be nested within one another. JavaScript fits within *<SCRIPT>* tags inside the *<HEAD> </HEAD>* and/or *<BODY> </BODY>* tags of a webpage, as in the following example:

```
<!doctype html>
<html>
<head>
```

```
<title>A Web Page Title</title>
<script type="text/javascript">
// JavaScript Goes Here
</script>
</head>
<body>
<script type="text/javascript">
// JavaScript can go here too
</script>
</body>
</html>
```

JavaScript placed within the *<BODY>* tags executes as it is encountered by the browser, which is helpful when you need to write to the document by using a JavaScript function, as follows (the function calls are shown in boldface type):

```
<!doctype html>
<html>
<head>
<title>A Web Page Title</title>
<script type="text/javascript">
// JavaScript Goes Here
</script>
</head>
<body>
<script type="text/javascript">
document.write("hello");
document.write(" world");
</script>
</body>
</html>
```

Because of the way browsers load JavaScript, the current best practice for placing JavaScript in your HTML is to position the *<SCRIPT>* tags at the end of the *<BODY>* element rather than in the *<HEAD>* element. Doing so helps to ensure that the content of the page is rendered if the browser blocks input while the JavaScript files are being loaded.

When you're using JavaScript on an Extensible Hypertext Markup Language (XHTML) page, the less-than sign (<) and the ampersand character (&) are interpreted as XML, which can cause problems for JavaScript. To get around this, use the following syntax in an XHTML page:

```
<script type="text/javascript">
<![CDATA[
    // JavaScript Goes Here
]]>
</script>
```

Browsers that aren't XHTML-compliant don't interpret the CDATA section correctly. You can work around that problem by placing the CDATA section inside a JavaScript comment—a line or set of lines prefaced by two forward slashes (*//*), as shown here:

```
<script type="text/javascript">
```

```
//<![CDATA[
    // JavaScript Goes Here
//]]>
</script>
```

Yes, the code really is that ugly. However, there's an easy fix for this: use external JavaScript files. In Chapter 2, "Developing in JavaScript," you learn exactly how to accomplish this simple task.

Document types

If you've been programming for the web for any length of time, you're probably familiar with Document Type declarations, or DOCTYPE declarations, as they're sometimes called. One of the most important tasks you can do when designing your webpages is to include an accurate and syntactically correct DOCTYPE declaration section at the top of the page. The DOCTYPE declaration, frequently abbreviated as DTD, lets the browser (or other parsing program) know the rules that will be followed when parsing the elements of the document.

An example of a DOCTYPE declaration for HTML 4.01 looks like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">
```

If you're using a Microsoft Visual Studio version earlier than version 2012 to create a web project, each page is automatically given a DOCTYPE declaration for the XHTML 1.0 standard, like this:

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

HTML version 5 uses a much simpler DOCTYPE:

```
<!DOCTYPE html>
```

If you fail to declare a DOCTYPE, the browser interprets the page by using a mode known as *Quirks Mode*. Falling back to Quirks Mode means that the document might end up looking different from your intention, especially when viewed through several browsers.

If you do declare a DOCTYPE, making sure that the resulting HTML, cascading style sheet (also known as CSS), and JavaScript also adhere to web standards is important so that the document can be viewed as intended by the widest possible audience, no matter which interface or browser is used. The W3C makes available an online validator at *http://validator.w3.org/*, which you can use to validate any publicly available webpage.

Tip Use the Markup Validator regularly until you're comfortable with coding to standards, and always check for validity before releasing your web project to the public.

What JavaScript can do

JavaScript is largely a complementary language, meaning that it's uncommon for an entire application to be written solely in JavaScript without the aid of other languages like HTML and without presentation in a web browser. Some Adobe products support JavaScript, and Windows 8 begins to change this, but JavaScript's main use is in a browser.

JavaScript is also the J in the acronym AJAX (Asynchronous JavaScript and XML), the darling of the Web 2.0 phenomenon. However, beyond that, JavaScript is an everyday language providing the interactivity expected, maybe even demanded, by today's web visitors.

JavaScript can perform many tasks on the client side of the application. For example, it can add the needed interactivity to a website by creating drop-down menus, transforming the text on a page, adding dynamic elements to a page, and helping with form entry.

Before learning about what JavaScript can do—the focus of this book—you need to understand what JavaScript can't do, but note that neither discussion is comprehensive.

What JavaScript can't do

Many of the operations JavaScript can't perform are the result of JavaScript's usage being somewhat limited to a web browser environment. This section examines some of the tasks JavaScript can't perform and some that JavaScript shouldn't perform.

JavaScript can't be forced on a client

JavaScript relies on another interface or host program for its functionality. This host program is usually the client's web browser, also known as a *user agent*. Because JavaScript is a client-side language, it can do only what the client allows it to do.

Some people are still using older browsers that don't support JavaScript at all. Others won't be able to take advantage of many of JavaScript's fancy features because of accessibility programs, text readers, and other add-on software that assists the browsing experience. And some people might just choose to disable JavaScript because they can, because of security concerns (whether perceived or real), or because of the poor reputation JavaScript received as a result of certain annoyances like pop-up ads.

Regardless of the reason, you need to perform some extra work to ensure that the website you're designing is available to those individuals who don't have JavaScript. I can hear your protests already: "But this feature is really [insert your own superlative here: cool, sweet, essential, nice, fantastic]." Regardless of how nice your feature might be, the chances are you will benefit from better interoperability and more site visitors. In the "Tips for using JavaScript" section later in this chapter, I offer some pointers that you can follow for using JavaScript appropriately on your website.

It might be helpful to think of this issue another way. When you build a web application that gets served from Microsoft Internet Information Services (IIS) 6.0, you can assume that the application will usually work when served from an IIS 6.0 server anywhere. Likewise, when you build an application for Apache 2, you can be pretty sure that it will work on other Apache 2 installations. However, the same assumption cannot be made for JavaScript. When you write an application that works fine on your desktop, you can't guarantee that it will work on somebody else's. You can't control how your application will work after it gets sent to the client.

JavaScript can't guarantee data security

Because JavaScript is run wholly on the client, the developer must learn to let go. As you might expect, letting go of control over your program has serious implications. After the program is on the client's computer, the client can do many undesirable things to the data before sending it back to the server. As with any other web programming, you should never trust any data coming back from the client. Even if you've used JavaScript functions to validate the contents of forms, you still must validate this input again when it gets to the server. A client with JavaScript disabled might send back garbage data through a web form. If you believe, innocently enough, that your client-side JavaScript function has already checked the data to ensure that it is valid, you might find that invalid data gets back to the server, causing unforeseen and possibly dangerous consequences.

Important Remember that JavaScript can be disabled on your visitor's computer. You cannot rely on cute tricks to be successful, such as using JavaScript to disable right-clicks or to prevent visitors from viewing the page source, and you shouldn't use them as security measures.

JavaScript can't cross domains

The JavaScript developer also must be aware of the *Same-Origin Policy*, which dictates that scripts running from within one domain neither have access to the resources from another Internet domain, nor can they affect the scripts and data from another domain. For example, JavaScript can be used to open a new browser window, but the contents of that window are somewhat restricted to the calling script. When a page from my website (*braingia.org*) contains JavaScript, that page can't access any JavaScript executed from a different domain, such as *microsoft.com*. This is the essence of the Same-Origin Policy: JavaScript has to be executed in or originate from the same location.

The Same-Origin Policy is frequently a restriction to contend with in the context of frames and AJAX's *XMLHttpRequest* object, where multiple JavaScript requests might be sent to different web servers. With the introduction of Windows Internet Explorer 8, Microsoft introduced support for the *XDomainRequest* object, which allows limited access to data from other domains.

JavaScript doesn't do servers

When developing server-side code such as Visual Basic .NET or PHP (a recursive acronym that stands for *PHP: Hypertext Preprocessor*), you can be fairly sure that the server will implement certain functions, such as talking to a database or giving access to modules necessary for the web application. JavaScript doesn't have access to server-side variables. For example, JavaScript cannot access databases that are located on the server. JavaScript code is limited to what can be done inside the platform on which the script is running, which is typically the browser.

Another shift you need to make in your thinking, if you're familiar with server-side programming, is that with JavaScript, you have to test the code on many different clients to know what a particular client is capable of. When you're programming server-side, if the server doesn't implement a given function, you know it right away because the server-side script fails when you test it. Naughty administrators aside, the back-end server code implementation shouldn't change on a whim, and thus, you more easily know what you can and cannot code. But you can't anticipate JavaScript code that is intended to run on clients, because these clients are completely out of your control.

Note There are server-side implementations of JavaScript, but they are beyond the scope of this book.

Tips for using JavaScript

Several factors go into good web design, and really, who arbitrates what is and is not considered good anyway? One visitor to a site might call the site an ugly hodgepodge of colors and text created as if those elements were put in a sack and shaken until they fell out onto the page; the next visitor might love the design and color scheme.

Because you're reading this book, I assume that you're looking for some help with using JavaScript to enhance your website. I also assume that you want to use this programming language to help people use your site and to make your site look, feel, and work better.

The design of a website is not and will never be an entirely objective process. The goal of one website might be informational, which would dictate one design approach, whereas the goal of another website might be to connect to an application, thus requiring specialized design and functionality. That said, many popular and seemingly well-designed sites have certain aspects in common. I try to break down those aspects here, although I ask you to remember that I didn't create a comprehensive list and that the items reflect only one person's opinions.

A well-designed website does the following:

- **Emphasizes function over form** When a user visits a website, she usually wants to obtain information or perform a task. The more difficult your site is to browse, the more likely the user is to move to another site with better browsing.

Animations and blinking bits come and go, but what remain are sites that have basic information presented in a professional, easily accessible manner. Using the latest cool animation software or web technology makes me think of the days of the HTML *<BLINK>* tag. The *<BLINK>* tag, for those who never saw it in action, caused the text within it to disappear and reappear on the screen. Nearly all web developers seem to hate the *<BLINK>* tag and what it does to a webpage. Those same developers would be wise to keep in mind that today's exciting feature or special effect on a webpage will be tomorrow's *<BLINK>* tag. Successful websites stick to the basics and use these types of bits only when the content requires them.

Use elements like a site map, alt tags, and simple navigation tools, and don't require special software or plug-ins for viewing the site's main content. Too often, I visit a website only to be stopped because I need a plug-in or the latest version of this or that player (which I don't have) to browse it.

Although site maps, alt tags, and simple navigation might seem quaint, they are indispensable items for accessibility. Text readers and other such technologies that enable sites to be read aloud or browsed by individuals with disabilities use these assistive features and frequently have problems with complex JavaScript.

- **Follows standards** Web standards exist to be followed, so ignore them at your own peril. Using a correct DOCTYPE declaration and well-formed HTML helps ensure that your site will display correctly to your visitors. Validation using the W3C's Markup Validator tool is highly recommended. If your site is broken, fix it!

- **Renders correctly in multiple browsers** Even when Internet Explorer had 90 percent market share, it was never a good idea for programmers to ignore other browsers. Doing so usually meant that accessibility was also ignored, so people with text readers or other add-ons couldn't use the site. People using operating systems other than Microsoft Windows might also be out of luck visiting those sites.

  Although Internet Explorer is still the leader among browsers used by web visitors, it isn't the only browser your web visitors will use. Somewhere around 3 or 4 of every 10 visitors will be using a different web browser.

  You never want to turn away visitors because of their browser choice. Imagine the shopkeeper who turned away 3 of every 10 potential customers just because of their shoes. That shop wouldn't be in business too long—or at the very least, it wouldn't be as successful.

  If you strive to follow web standards, chances are that you're already doing most of what you need to do to support multiple browsers. Avoiding the use of proprietary plug-ins for your website is another way to ensure that your site renders correctly. You need to look only as far as the iPad to see a device that is popular but whose use is restricted because it doesn't natively support Flash. For this reason, creating sites that follow standards and avoid proprietary plug-ins ensures that your site is viewable by the widest possible audience.

- **Uses appropriate technologies at appropriate times** Speaking of plug-ins, a well-designed website doesn't overuse or misuse technology. On a video site, playing videos is appropriate. Likewise, on a music site, playing background music is appropriate. On other sites, these features might not be so appropriate. If you feel that your site needs to play background music, go back to the drawing board and examine why you want a website in the first place! I still shudder when I think of an attorney's website that I once visited. The site started playing the firm's jingle in the background, without my intervention. Friends don't let friends use background music on their sites.

Where JavaScript fits

Today's web is still evolving. One of the more popular movements is known as *unobtrusive scripting*. The unobtrusive scripting paradigm contains two components, progressive enhancement and behavioral separation. *Behavioral separation* calls for structure to be separated from style, and for both of these elements to be separated from behavior. In this model, HTML or XHTML provides the structure, whereas the CSS provides the style and JavaScript provides the behavior. Progressive enhancement means adding more features to the page as the browser's capabilities are tested; enhancing the user experience when possible but not expecting that JavaScript or a certain JavaScript function will always be available. In this way, the JavaScript is unobtrusive; it doesn't get in the way of the user experience. If JavaScript isn't available in the browser, the website still works because the visitor can use the website in some other way.

When applied properly, unobtrusive scripting means that JavaScript is not assumed to be available and that JavaScript will fail in a graceful manner. Graceful degradation helps the page function without JavaScript or uses proper approaches to make JavaScript available when it's required for the site.

I'm a proponent of unobtrusive scripting because it means that standards are followed and the resulting site adheres to the four recommendations shared in the previous section. Unfortunately, this isn't always the case. You could separate the HTML, CSS, and JavaScript and still end up using proprietary tags, but when you program in an unobtrusive manner, you tend to pay closer attention to detail and care much more about the end result being compliant with standards.

Throughout this book, I strive to show you not only the basics of JavaScript but also the best way to use JavaScript effectively and, as much as possible, unobtrusively.

A note on JScript and JavaScript and this book

This book covers JavaScript as defined by the ECMA standard, in versions all the way through the latest edition 5. This is distinct from Microsoft's implementation of JScript, which is not covered in this book. For an additional reference on only JScript, I recommend the following site: JScript (Windows Script Technologies) at *http://msdn.microsoft.com/en-us/library/hbxc2t98.aspx*.

Which browsers should the site support?

Downward compatibility has been an issue for the web developer for a long time. Choosing which browser versions to support becomes a trade-off between using the latest functionality available in the newest browsers and the compatible functionality required for older browsers. There is no hard and fast rule for which browsers you should support on your website, so the answer is: it depends.

Your decision depends on what you'd like to do with your site and whether you value visits by people using older hardware and software more than you value the added functionality available in later browser versions. Some browsers are just too old to support because they can't render CSS correctly, much less JavaScript. A key to supporting multiple browser versions is to test with them. All of this means that you need to develop for and test in an appropriate set of browsers before releasing a website for public consumption.

Obtaining an MSDN account from Microsoft will give you access to both new and older verions of products, including Internet Explorer. Additional resources are the Application Compatibility Virtual PC Images, available for free from Microsoft. These allow you to use a time-limited version of Microsoft Windows containing older versions of Internet Explorer, too. For more information, see *http://www.microsoft.com/downloads/details.aspx?FamilyId=21EABB90-958F-4B64-B5F1-73D0A413C8EF&displaylang=en*.

Many web designs and JavaScript functions don't require newer versions of web browsers. However, as already explained, verifying that your site renders correctly in various browsers is always a good idea. See *http://browsers.evolt.org/* for links to archives of many historical versions of web browsers. Even if you can't conduct extensive testing in multiple browsers, you can design the site so that it fails in a graceful manner. You want the site to render appropriately regardless of the browser being used.

And then came Windows 8

Microsoft Windows 8 represents a paradigm shift for JavaScript programmers. In Windows 8, Microsoft has elevated JavaScript to the same level as other client-side languages, such as Visual Basic and C#, for developing Windows 8 applications. Before Windows 8, if you wanted to create an application that ran on the desktop, you'd need to use Visual Basic, C#, or a similar language. With Windows 8, you need only use HTML and JavaScript to create a full-fledged Windows 8–style app.

Windows 8 exposes an Application Programming Interface (API), providing a set of functions that enable the JavaScript programmer to natively access behind-the-scenes areas of the operating system. This means that programming for Windows 8 is slightly different from programming JavaScript for a web browser.

Of course, your web applications will still work in Internet Explorer, which comes with Windows 8. These web applications are distinct and separate from the Windows 8 native applications.

This book will show how to develop for Windows 8 using JavaScript. Before you get there, you'll see how to create JavaScript programs that run in web browsers.

Exercises

1. True or False: JavaScript is defined by a standards body and is supported on all web browsers.
2. True or False: When a visitor whose machine has JavaScript disabled comes to your website, you should block his access to the site because there's no valid reason to have JavaScript disabled.
3. Create a JavaScript definition block that would typically appear on an HTML page within the *<HEAD>* or *<BODY>* block.
4. True or False: It's important to declare the version of JavaScript being used within the DOCTYPE definition block.
5. True or False: JavaScript can appear in the *<HEAD>* block and within the *<BODY>* text of an HTML page.

CHAPTER 4

Working with variables and data types

After completing this chapter, you will be able to

- Understand the primitive data types used in JavaScript.
- Use functions associated with the data types.
- Create variables.
- Define objects and arrays.
- Understand the scope of variables.
- Debug JavaScript using Firebug.

Data types in JavaScript

The *data types* of a language describe the basic elements that can be used within that language. You're probably already familiar with data types, such as strings or integers, from other languages. Depending on who you ask, JavaScript defines anywhere from three to six data types. (The answer depends largely on the definition of a data type.) You work with all these data types regularly, some more than others.

The six data types in JavaScript discussed in this chapter are as follows:

- Numbers
- Strings
- Booleans
- Null
- Undefined
- Objects

The first three data types—numbers, strings, and Booleans—should be fairly familiar to programmers in any language. The latter three—null, undefined, and objects—require some additional explanation. I examine each of the data types in turn and explain objects further in Chapter 8, "Objects in JavaScript."

Additionally, JavaScript has several reference data types, including the *Array*, *Date*, and *RegExp* types. The *Date* and *RegExp* types are discussed in this chapter, and the *Array* type is discussed in Chapter 8.

Working with numbers

Numbers in JavaScript are just what you might expect them to be: numbers. However, what might be a surprise for programmers who are familiar with data types in other languages like C is that integers and floating point numbers do not have special or separate types. All these are perfectly valid numbers in JavaScript:

```
4
51.50
-14
0xd
```

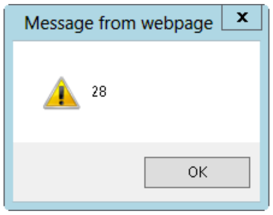
The last example, *0xd*, is a hexadecimal number. Hexadecimal numbers are valid in JavaScript, and you won't be surprised to learn that JavaScript allows math to be performed using all of the listed number formats. Try the following exercise.

Performing hexadecimal math with JavaScript

1. Using Microsoft Visual Studio, Eclipse, or another editor, edit the file example1.html in the Chapter04 sample files folder in the companion content.
2. Within the webpage, replace the TODO comment with the boldface code shown here:

```
<!doctype html>
<html>
<head>
<title>Hexadecimal Numbers</title>
<script type="text/javascript">
var h = 0xe;
var i = 0x2;
var j = h * i;
alert(j);
</script>
</head>
<body>
</body>
</html>
```

3. View the webpage in a browser. You should see a dialog box similar to this one:



```
28
```

The preceding script first defines two variables (you learn about defining variables later in this chapter) and sets them equal to two hexadecimal numbers, *0xe* (14 in base 10 notation) and *0x2*, respectively:

```
var h = 0xe;
var i = 0x2;
```

Then a new variable is created and set to the product of the previous two variables, as follows:

```
var j = h * i;
```

The resulting variable is then passed to the *alert()* function, which displays the dialog box in the preceding step 3. It's interesting to note that even though you multiplied two hexadecimal numbers, the output in the alert dialog box is in base 10 format.

Numeric functions

JavaScript has some built-in functions (and objects, too, which you learn about soon) for working with numeric values. The European Computer Manufacturers Association (ECMA) standard defines several of them. One more common numeric function is the *isNaN()* function. By *common*, I mean that *isNaN()* is a function that I use frequently in JavaScript programming. Your usage might vary, but an explanation follows nonetheless.

NaN is an abbreviation for *Not a Number,* and it represents an illegal number. You use the *isNaN()* function to determine whether a number is legal or valid according to the ECMA-262 specification. For example, a number divided by zero would be an illegal number in JavaScript. The string value "This is not a number" is obviously also not a number. Although people might have a different interpretation of what is and isn't a number, the string "four" is not a number to the *isNaN()* function, whereas the string "4" is. The *isNaN()* function requires some mental yoga at times because it attempts to prove a negative—that the value in a variable *is not* a number. Here are a couple of examples that you can try to test whether a number is illegal.

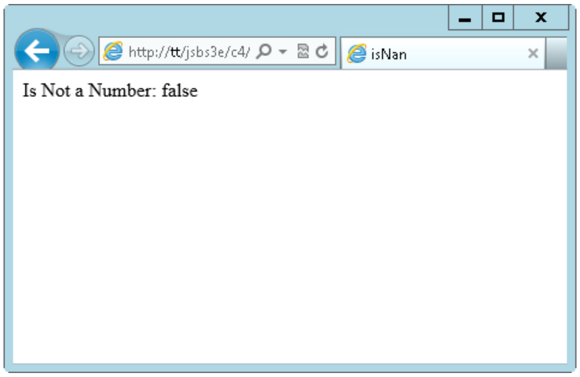
Testing the *isNaN()* function (test 1)

1. In Microsoft Visual Studio, Eclipse, or another editor, create a new HTML file or edit the isnan.html file in the companion content.

2. In the file, place the following markup. If you've created a new file with Vision Studio, delete any existing contents first.

```
<!doctype html>
<html>
<head>
<title>isNaN</title>
</head>
<body>
<script type="text/javascript">
document.write("Is Not a Number: " + isNaN("4"));
</script>
</body>
</html>
```

3. View this page in a browser. In Visual Studio, press F5. You'll see a page like this one:



Is Not a Number: false

The function *isNaN()* returns *false* from this expression because the integer value 4 *is* a number. Remember that the meaning of this function is, "Is 4 Not a Number?" Well, 4 *is* a number, so the result is *false*.

Now consider the next example.

Testing the *isNaN()* function (test 2)

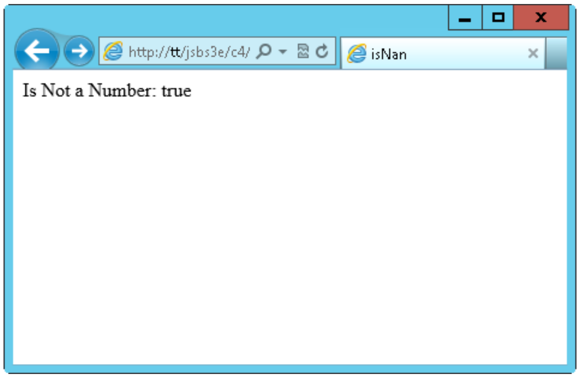
1. If you're running through Microsoft Visual Studio, stop the project. For those not running Visual Studio, close the web browser.

2. Edit isnan.html.

3. Change the *isNaN()* function line to read:

```
document.write("Is Not a Number: " + isNaN("four"));
```

View the page in a browser, or rerun the project in Visual Studio. You'll now see a page like this:



```
Is Not a Number: true
```

In second test case, because the numeral 4 is represented as a string of nonnumeric characters (*four*), the function returns *true*: the string *four* is not a number. I purposefully used double quotation marks in each code example (that is, *"4"* and *"four"*) to show that the quotation marks don't matter for this function. Because JavaScript is smart enough to realize that *"4"* is a number, JavaScript does the type conversion for you. However, this conversion can sometimes be a disadvantage, such as when you're counting on a variable or value to be a certain type.

The *isNaN()* function is used frequently when validating input to determine whether something—maybe a form variable—was entered as a number or as text.

Numeric constants

Other numeric constants are available in JavaScript, some of which are described in Table 4-1. These constants might or might not be useful to you in your JavaScript programming, but they exist if you need them.

TABLE 4-1 Selected numeric constants

Constant	Description
Infinity	Represents positive infinity
Number.MAX_VALUE	The largest number able to be represented in JavaScript
Number.MIN_VALUE	The smallest or most negative number able to be represented in JavaScript
Number.NEGATIVE_INFINITY	A value representing negative infinity
Number.POSITIVE_INFINITY	A value representing positive infinity

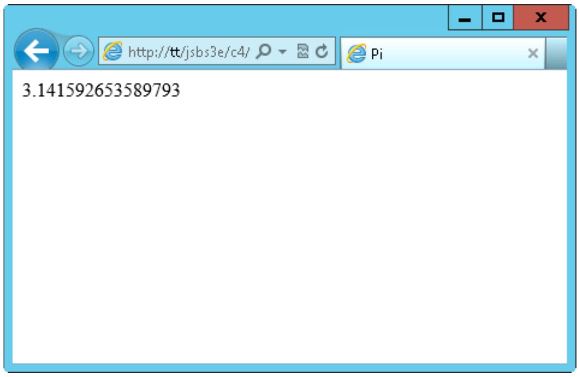
The *Math* object

The *Math* object is a special built-in object used for working with numbers in JavaScript, and it has several properties that are helpful to the JavaScript programmer, including properties that return the value of pi, the square root of a number, a pseudo-random number, and an absolute value.

Some properties are value properties, meaning they return a value, whereas others act like functions and return values based on the arguments sent into them. Consider this example of the *Math.PI* value property. Place this code between the opening *<SCRIPT TYPE="text/javascript">* and closing *</SCRIPT>* tags in your sample page:

```
document.write(Math.PI);
```

The result is shown in Figure 4-1.



```
3.141592653589793
```

FIGURE 4-1 Viewing the value of the *Math.PI* property.

Dot notation

Dot notation is so named because a single period, or *dot*, is used to access the members of an object. The single dot (.) creates an easy visual delineator between elements. For example, to access a property that you might call the "length of a variable *room*," you would write *room.length*. The *dot* operator is used similarly in many programming languages.

Several other properties of the *Math* object can be helpful to your program. Some of them act as functions or methods on the object, several of which are listed in Table 4-2. You can obtain a complete list of properties for the *Math* object in the ECMA-262 specification at *http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf*.

TABLE 4-2 Select function properties of the *Math* object

Property	**Definition**
Math.random()	Returns a pseudo-random number
Math.abs(x)	Returns the absolute value of *x*
Math.pow(x,y)	Returns *x* to the power of *y*
Math.round(x)	Returns *x* rounded to the nearest integer value

Working with strings

Strings are another basic data type available in JavaScript. They consist of one (technically zero) or more characters surrounded by quotation marks. The following examples are strings:

- *"Hello world"*
- *"B"*
- *"This is 'another string'"*

The last example in the preceding list requires some explanation. Strings are surrounded by either single or double quotation marks. Strings enclosed in single quotation marks can contain double quotation marks. Likewise, a string enclosed in double quotation marks, like the ones you see in the preceding example, can contain single quotation marks. So basically, if the string is surrounded by one type of quotation mark, you can use the other type within it. Here are some more examples:

- *'The cow says "moo".'*
- *'The talking clock says the time is "Twelve Noon".'*
- *"'Everyone had a good time' was the official slogan."*

Escaping quotation marks

If you use the same style of quotation mark both within the string and to enclose the string, the quotation marks must be *escaped* so that they won't be interpreted by the JavaScript engine. A single backslash character (\) escapes the quotation mark, as in these examples:

- *'I\'m using single quotation marks both outside and within this example. They\'re neat.'*
- *"This is a \"great\" example of using \"double quotes\" within a string that's enclosed with \"double quotes\" too."*

Other escape characters

JavaScript enables other characters to be represented with specific escape sequences that can appear within a string. Table 4-3 shows those escape sequences.

TABLE 4-3 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

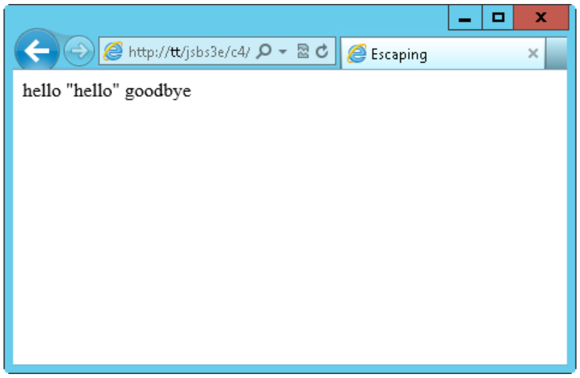
Here's an example of some escape sequences in action.

Using escape sequences

1. In Visual Studio, Eclipse, or another editor, open your sample page.
2. Within the *<SCRIPT>* section, place the following line of JavaScript:

```
document.write("hello\t\t\"hello\"goodbye");
```

3. View the page in a browser. You'll see a page like the following. Notice that the tab characters don't show through because the browser interprets HTML and not tab characters.



```
hello "hello" goodbye
```

This rather contrived example shows escape sequences in action. In the code, the word *hello* is followed by two tabs, represented by their escape sequence of *t*, followed by an escaped double-quote \" and then the word *hello* followed by another escaped double-quote \", finally followed by the word *goodbye*.

String methods and properties

JavaScript defines several properties and methods for working with strings. These properties and methods are accessed using dot notation ("."), explained earlier in this chapter and familiar to many programmers.

Note In the same way I describe only some of the elements of JavaScript in this book, I cover only a subset of the string properties and methods available in the ECMA-262 specification. Refer to the ECMA specification for more information.

The *length* property on a *string* object gives the length of a string, not including the enclosing quotation marks. The *length* property can be called directly on a string literal, as in this example:

```
alert("This is a string.".length);
```

However, it’s much more common to call the *length* property on a variable, like this:

```
var x = "This is a string.";
alert(x.length);
```

Both examples give the same result.

Some commonly used *string* methods, besides *substring,* include *slice*, *substr*, *concat*, *toUpperCase*, *toLowerCase*, and the pattern matching methods of *match*, *search*, and *replace*. I discuss each of these briefly.

Methods that change strings include *slice*, *substring*, *substr*, and *concat*. The *slice* and *substring* methods return string values based on another string. They accept two arguments: the beginning position and an optional end position. Here are some examples:

```
var myString = "This is a string.";
alert(myString.substring(3));  //Returns "s is a string."
alert(myString.substring(3,9));  //Returns "s is a"
alert(myString.slice(3));  //Returns "s is a string."
alert(myString.slice(3,9));  //Returns "s is a"
```

A subtle difference between *slice* and *substring* is how they handle arguments with negative values. The *substring* method will convert any negative values to 0, while *slice* will treat negative arguments as the starting point from the end of the string (counting backwards from the end, essentially).

The *substr* method also accepts two arguments: the first is the beginning position to return, and, in contrast to *substring/slice*, the second argument is the number of characters to return, not the stopping position. Therefore, the code examples for *substring/slice* work a little differently with *substr*:

```
var myString = "This is a string.";
alert(myString.substr(3));  //Returns "s is a string." (The same as substring/slice)
alert(myString.substr(3,9));  //Returns "s is a st" (Different from substring/slice)
```

The *concat* method concatenates two strings together:

```
var firstString = "Hello ";
var finalString = firstString.concat("World");
alert(finalString);  //Outputs "Hello World"
```

It’s somewhat more common to use the plus sign (+) for concatenation, so the same output could be accomplished with this:

```
var finalString = firstString + "World";
```

The *toUpperCase* and *toLowerCase* methods, and their brethren *toLocaleUpperCase* and *toLocaleLowerCase,* convert a string to all uppercase or all lowercase, respectively:

```
var myString = "this is a String";
alert(myString.toUpperCase());  // "THIS IS A STRING"
alert(myString.toLowerCase());  // "this is a string"
```

Note The *toLocale* methods perform conversions in a locale-specific manner.

As I stated, numerous string properties and methods exist. The remainder of the book features other string properties and methods, and you can always find a complete list within the ECMA specification at *http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf*.

Booleans

Booleans are kind of a hidden, or passive, data type in JavaScript. By *hidden,* or *passive*, I mean that you don't work with Booleans in the same way that you work with strings and numbers; you can define and use a Boolean variable, but typically you just use an expression that evaluates to a Boolean value. Booleans have only two values, *true* and *false*, and in practice, you rarely set variables as such. Rather, you use Boolean expressions within tests, such as an *if/then/else* statement.

Consider this statement:

```
If (myNumber > 18) {
    //do something
}
```

A Boolean expression is used within the *if* statement's condition to determine whether the code within the braces will be executed. If the content of the variable *myNumber* is greater than the integer 18, the Boolean expression evaluates to *true*; otherwise, the Boolean evaluates to *false*.

Null

Null is another special data type in JavaScript (as it is in most languages). *Null* is, simply, nothing. It represents and evaluates to *false*. When a value is *null*, it is nothing and contains nothing. However, don't confuse this nothingness with being empty. An empty value or variable is still full; it's just full of emptiness. Emptiness is different from *null*, which is just plain nothing. For example, defining a variable and setting its value to an empty string looks like this:

```
var myVariable = '';
```

The variable *myVariable* is empty, but it is not null.

Undefined

Undefined is a state, sometimes used like a value, to represent a variable that hasn't yet contained a value. This state is different from *null*, although both *null* and *undefined* can evaluate the same way. You'll learn how to distinguish between a null value and an undefined value in Chapter 5, "Using operators and expressions."

Objects

Like functions, objects are special enough to get their own chapter (Chapter 8, to be exact). But I still discuss objects here briefly. JavaScript is an object-based language, as opposed to a full-blown object-oriented language. JavaScript implements some functionality similar to object-oriented functionality, and for most basic usages of JavaScript, you won't notice the difference.

Objects in JavaScript are a collection of properties, each of which can contain a value. These properties—think of them as *keys*—enable access to values. Each value stored in the properties can be a value, another object, or even a function. You can define your own objects with JavaScript, or you can use the several built-in objects.

Objects are created with curly braces, so the following code creates an empty object called *myObject*:

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

Here's an object with several properties:

```
var dvdCatalog = {
    "identifier": "1",
    "name": "Coho Vineyard"
};
```

The preceding code example creates an object called *dvdCatalog,* which holds two properties: one called *identifier* and the other called *name*. The values contained in each property are *1* and *"Coho Vineyard"*, respectively. You could access the *name* property of the *dvdCatalog* object like this:

```
alert(dvdCatalog.name);
```

Here's a more complete example of an object, which can also be found in the sample code in the file object.txt:

```
// Create four new objects
var star = {};
// Create properties  for each of four stars.
star["Polaris"] = new Object;
star["Deneb"] = new Object;
star["Vega"] = new Object;
star["Altair"] = new Object;
```

Examples later in the book show how to add properties to these objects and how to access properties. There's much more to objects, and Chapter 8 gives that additional detail.

Arrays

You've seen in the previous example how to create an object with a name. You can also use array elements that are accessed by a numbered index value. These are the traditional arrays, familiar to programmers in many languages. You just saw several objects, each named for a star. The following code creates an array with four elements.

```
var star = new Array();
star[0] = "Polaris";
star[1] = "Deneb";
star[2] = "Vega";
star[3] = "Altair";
```

The same code can also be written like this, using literal notation, represented by square brackets:

```
var star = ["Polaris", "Deneb", "Vega", "Altair"];
```

Arrays can contain nested values, creating an array of arrays, as in this example that combines the star name with the constellation in which it appears:

```
var star = [["Polaris", "Ursa Minor"],["Deneb","Cygnus"],["Vega","Lyra"],
["Altair","Aquila"]];
```

Finally, although less common, you can call the *Array()* constructor with arguments:

```
var star = new Array("Polaris", "Deneb", "Vega", "Altair");
```

Note Calling the *Array()* constructor with a single numeric argument sets the length of the array rather than the value of the first element, which is what you might expect.

The new ECMA-262 edition 5 specification added several new methods for iterating and working with arrays. Arrays, including methods that iterate through them and work with them, are covered in more detail in Chapter 8.

Defining and using variables

Variables should be familiar to programmers in just about any language. Variables store data that might change during the program's execution lifetime. You've seen several examples of declaring variables throughout the previous chapters of this book. This section formalizes the use of variables in JavaScript.

Declaring variables

Variables are declared in JavaScript with the *var* keyword. The following are all valid variable declarations:

```
var x;
var myVar;
var counter1;
```

Variable names can contain uppercase and lowercase letters as well as numbers, but they cannot start with a number. Variables cannot contain spaces or other punctuation, with the exception of the underscore character (_). The following variable names are invalid:

```
var 1stCounter;
var new variable;
var new.variable;
var var;
```

Take a look at the preceding example. Whereas the first three variable names are invalid because characters are used that aren't valid at all (or aren't valid in that position, as is the case with the first example), the last variable name, *var,* is invalid because it uses a keyword. For more information about keywords or reserved words in JavaScript, refer to Chapter 3, "JavaScript syntax and statements."

You can declare multiple variables on the same line of code, as follows:

```
var x, y, zeta;
```

These can be initialized on the same line, too:

```
var x = 1, y = "hello", zeta = 14;
```

Variable types

Variables in JavaScript are not strongly typed. It's not necessary to declare whether a given variable will hold an integer, a floating point number, or a string. You can also change the type of data being held within a variable through simple reassignment. Consider this example, where the variable *x* first holds an integer but then, through another assignment, it changes to hold a string:

```
var x = 4;
x = "Now it's a string.";
```

Variable scope

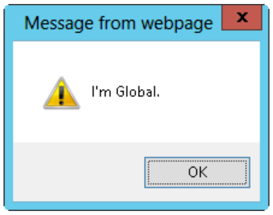
A variable's *scope* refers to the locations from which its value can be accessed. Variables are *globally scoped* when they are used outside a function. A globally scoped variable can be accessed throughout your JavaScript program. In the context of a webpage—or a document, as you might think of it—you can access and use a global variable throughout.

Variables defined within a function are scoped solely within that function. This effectively means that the values of those variables cannot be accessed outside the function. Function parameters are scoped locally to the function as well.

Here are some practical examples of scoping, which you can also find in the companion code in the scope1.html file:

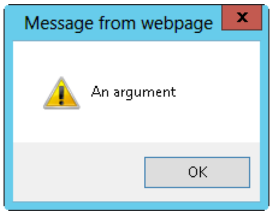
```
<script type="text/javascript">
var aNewVariable = "I'm Global.";
function doSomething(incomingBits) {
    alert(aNewVariable);
    alert(incomingBits);
}
doSomething("An argument");
</script>
```

The code defines two variables: a global variable called *aNewVariable* and a variable called *incomingBits*, which is local to the *doSomething()* function. Both variables are passed to respective *alert()* functions within the *doSomething()* function. When the *doSomething()* function is called, the contents of both variables are sent successfully and displayed on the screen, as depicted in Figures 4-2 and 4-3.



I'm Global.

FIGURE 4-2 The variable *aNewVariable* is globally scoped.



An argument

FIGURE 4-3 The variable *incomingBits* is locally scoped to the function.

Here's a more complex example for you to try.

Examining variable scope

1. Using Visual Studio, Eclipse, or another editor, edit the file scoping.html in the Chapter04 sample files folder, which you can find in the companion content.

2. Within the page, replace the TODO comment with the boldface code shown here (the new code can be found in the scoping.txt file in the companion content):

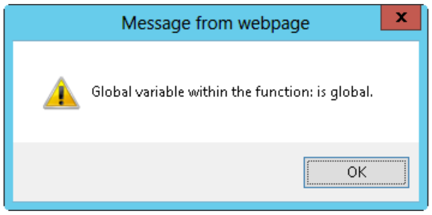
```
<!doctype html>
<html>
<head>
    <title>Scoping Example</title>
    <script type="text/javascript">
    var aNewVariable = "is global.";
    function doSomething(incomingBits) {
        alert("Global variable within the function: " + aNewVariable);
        alert("Local variable within the function: " + incomingBits);
    }
    </script>

</head>
<body>
<script type="text/javascript">
```

```
    doSomething("is a local variable");
    alert("Global var outside the function: " + aNewVariable);
    alert("Local var outside the function: " + incomingBits);

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

3. Save the file.

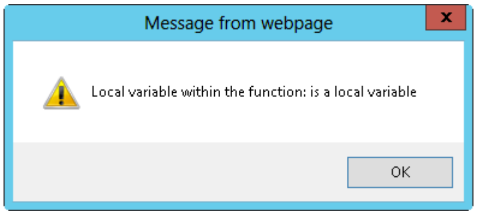
4. View the file in a web browser. The result is three alerts on the screen.

   The first alert is this:



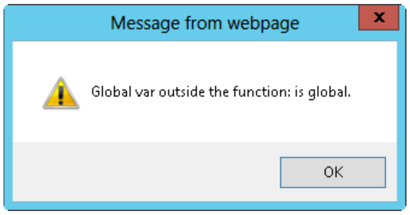
```
Global variable within the function: is global.
```

   The second alert is this:



```
Local variable within the function: is a local variable
```

   The third alert looks like this:



```
Global var outside the function: is global.
```

But wait a minute—examine the code. How many calls to the *alert()* function do you see? Hint: two are in the <*HEAD*> portion, and another two are within the <*BODY*> portion, for a total of four calls to the *alert()* function. So why are there only three alerts on the screen when four calls are made to the *alert()* function in the script?

Because this is a section on variable scoping (and I already explained the answer), you might already have figured it out. But this example demonstrates well how to troubleshoot JavaScript problems when the result isn't what you expect.

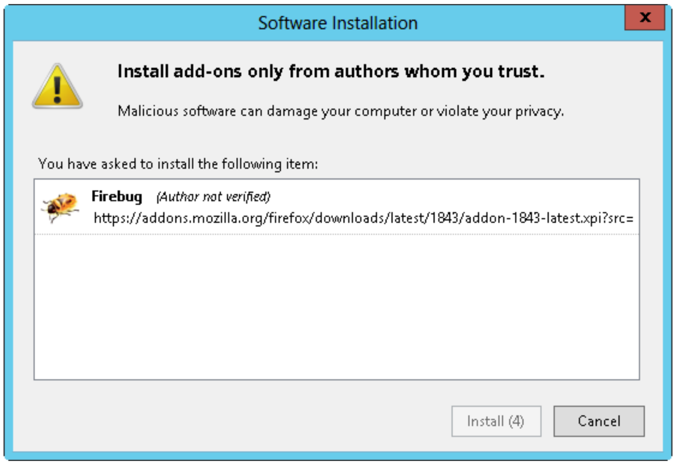
The next procedure requires the use of the Firebug add-on to the Mozilla Firefox web browser. If you don't yet have Firefox, download it from *http://www.mozilla.com/firefox/.*

Installing Firebug

This first procedure walks you through installing Firebug in Firefox. Firebug is very powerful and flexible.

1. With Firefox installed, it's time to get the Firebug add-on. Accomplish this task by going to *http://www.getfirebug.com/.* On that site, click the Install Firebug link. When you do so, you'll be asked to choose the version of Firebug to install. Install the version that corresponds to your version of Firefox (or is as close as possible to the version of Firefox that you have).

2. When you click the install link, you'll be sent to Mozilla's site, where you get to click another button, this one labeled "Add To Firefox." A Software Installation dialog box opens, as shown in the following screen. Click Install Now.


Software Installation

Install add-ons only from authors whom you trust.

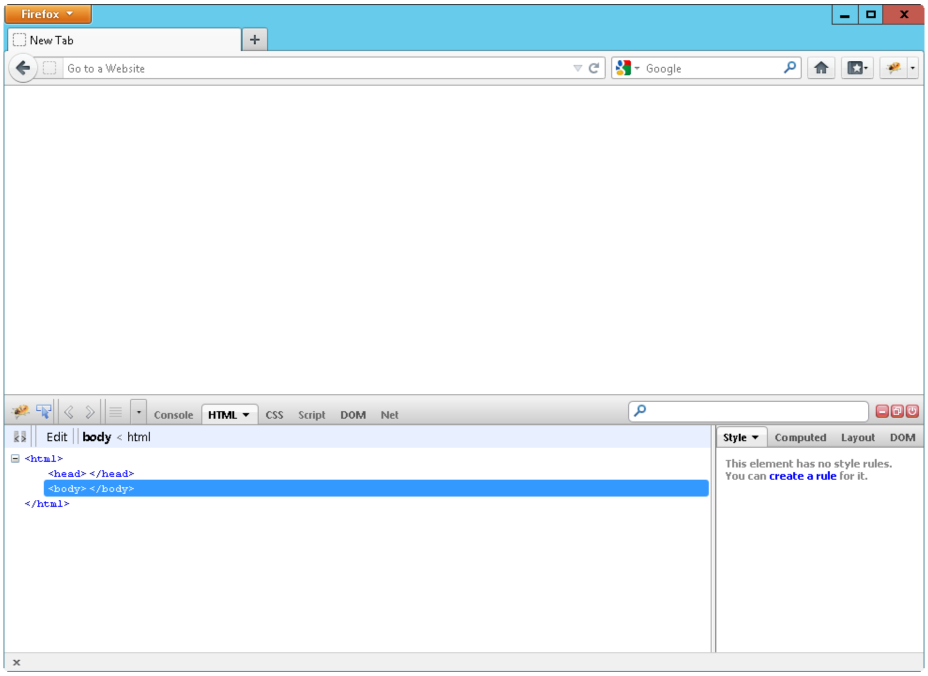
Malicious software can damage your computer or violate your privacy.

You have asked to install the following item:

Firebug *(Author not verified)*
https://addons.mozilla.org/firefox/downloads/latest/1843/addon-1843-latest.xpi?src=

3. The installation completes when you restart Firefox, so click Restart Firefox after the add-on finishes downloading.

4. Firefox closes and opens again, showing the installed add-on. Congratulations! Firebug is installed. Notice a small icon in the upper-right corner of the Firefox browser window. (The Firefox development team keeps moving buttons around, so the Firebug button might not be in the upper right when you read this.) Click the icon to open the Firebug console, shown here:



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

This element has no style rules. You can create a rule for it.

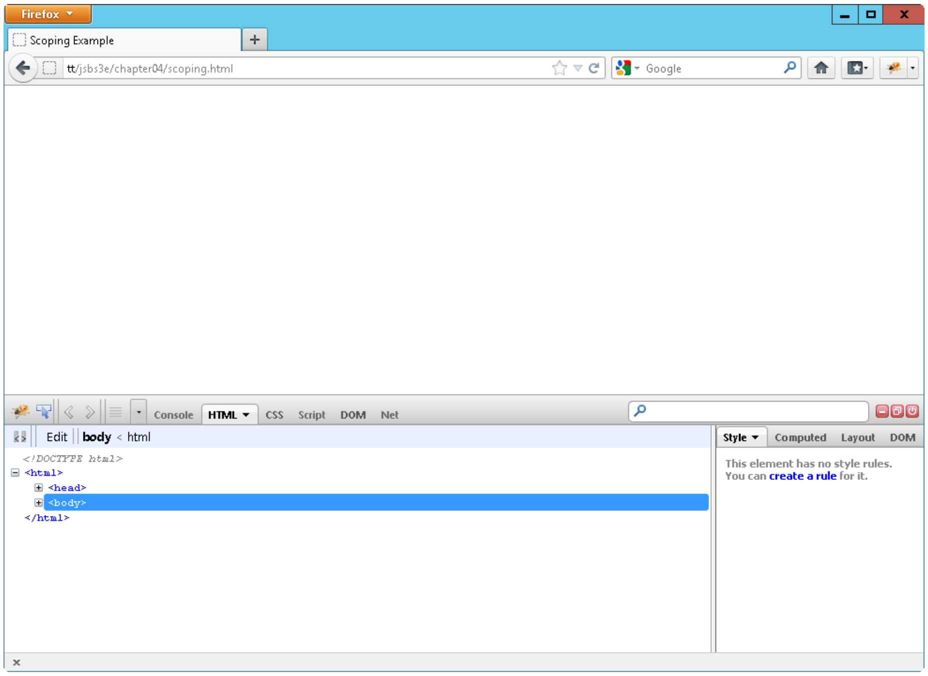
5. Firebug's JavaScript console is disabled, but don't worry—the next procedure walks you through enabling and using it. Feel free to experiment with Firebug by enabling it.

With Firebug installed, you can troubleshoot the earlier problem you encountered in the scoping example of only three of the four expected alerts being displayed.

Troubleshooting with Firebug

1. Open Firefox and open the scoping.html example that was created earlier in this chapter. The JavaScript code again executes as before, showing the three alerts. Close all three alerts. You end up with a blank page loaded in Firefox.

2. Click the Firebug icon in Firefox browser window so that Firebug opens.

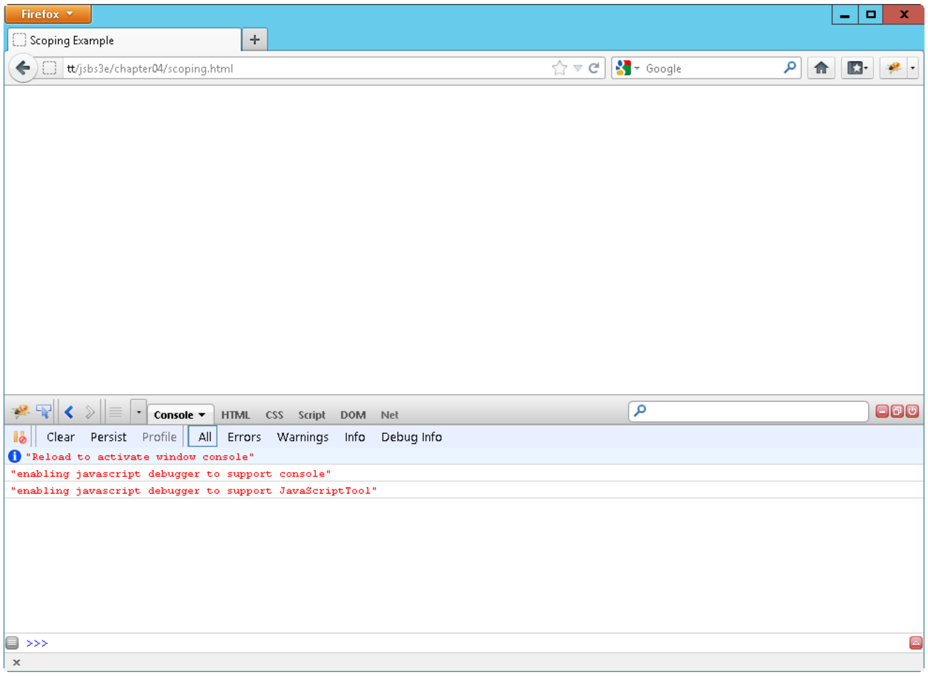


```
<!DOCTYPE html>
<html>
  <head>
  <body>
</html>
```

This element has no style rules. You can create a rule for it.

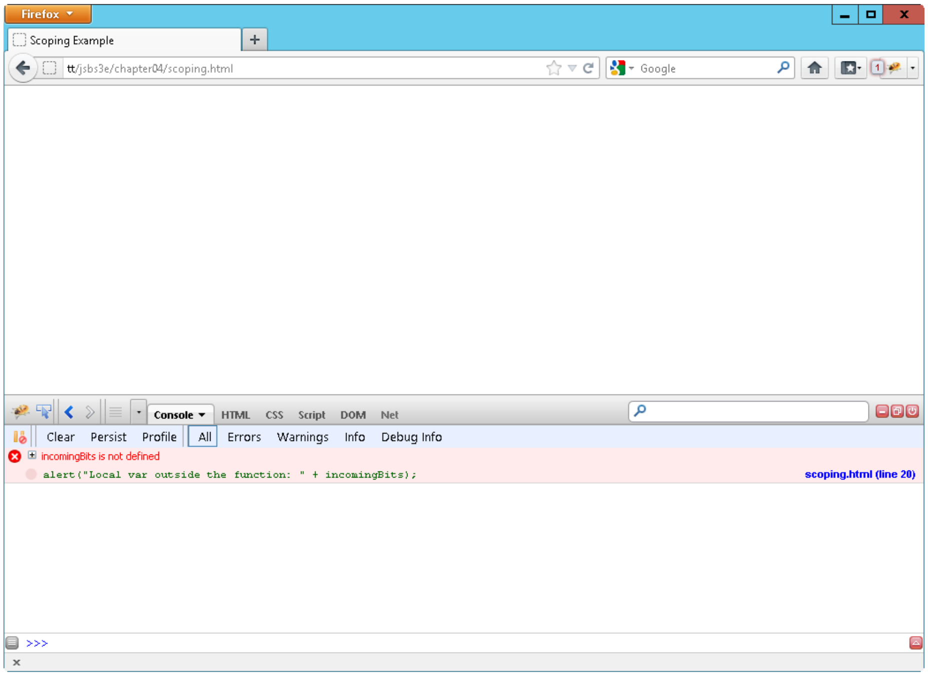
3. Click the Script tab to open the Script pane, and notice that it is disabled. Click the arrow/triangle next to the word *Script*, and click Enabled.

4. Click the Console tab, click the arrow/triangle next to the word *Console*, and click Enabled. You can see here that the Console is now activated:



```
"Reload to activate window console"
"enabling javascript debugger to support console"
"enabling javascript debugger to support JavaScriptTool"
```

5. With both the Console and Script panes enabled, click the Reload button on the main Firefox toolbar. The page reloads, and the JavaScript executes again. All three alerts are displayed again, but notice now that Firebug has discovered an error, denoted by the red X indication in the Firebug Console:



```
incomingBits is not defined
alert("Local var outside the function: " + incomingBits);    scoping.html (line 20)
```

6. The error, as you can see, is that the variable *incomingBits* isn't defined. This window also shows the line number at which the problem occurred. However, notice that because of the way the document is parsed, the line number in your original source code might not always be accurate. Regardless, you can see that *incomingBits* is not defined within the *<BODY>* section of the webpage because its scope is limited to the *doSomething()* function.

This procedure demonstrated not only the use of Firebug but also the effect of local versus global scoping of variables. Firebug is an integral part of JavaScript (and webpage) debugging. I invite you to spend some time with Firebug on just about any site to see how JavaScript, CSS, and HTML all interact.

In this procedure, the fix would be to define the variable *incomingBits* so that it gets instantiated outside the function call. (This new line of code follows and is in the file scoping-fixed.html in the Chapter04 folder in the companion content.) Because this variable was defined only as part of the function definition, the variable didn't exist outside the function's scope.

```
<!doctype html>
<html>
<head>
    <title>Scoping Example</title>
    <script type="text/javascript">
    var aNewVariable = "is global.";
    function doSomething(incomingBits) {
        alert("Global variable within the function: " + aNewVariable);
        alert("Local variable within the function: " + incomingBits);
    }

    </script>

</head>
<body>
<script type="text/javascript">
    var incomingBits = " must be defined if necessary.";
    doSomething("is a local variable");
    alert("Global var outside the function: " + aNewVariable);
    alert("Local var outside the function: " + incomingBits);

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

You can find more information about functions in Chapter 7, "Working with functions."

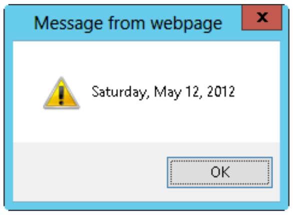
The *Date* object

The *Date* object includes many methods that are helpful when working with dates in JavaScript—too many, in fact, to examine in any depth in a broad-based book such as this—but I do show you some examples that you might incorporate in your projects.

One of the unfortunate aspects of the *Date* object in JavaScript is that the implementation of its methods varies greatly depending on the browser and the operating system. For example, consider this code to return a date for the current time, adjusted for the local time zone and formatted automatically by the *toLocaleDateString()* method:

```
var myDate = new Date();
alert(myDate.toLocaleDateString());
```

When run in Internet Explorer 10 on a computer running Windows 8, the code results in a date like that shown in Figure 4-4.



Saturday, May 12, 2012

FIGURE 4-4 The *toLocaleString()* method of the *Date* object in Internet Explorer 8.

Figure 4-5 shows what happens when that same code is executed in Firefox 12 on a Mac.

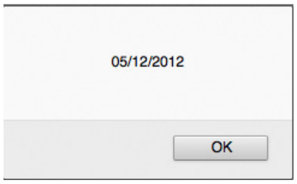

05/12/2012

FIGURE 4-5 The *toLocaleString()* method of the *Date* object displays the message differently in Firefox on Mac.

The difference between these two dialog boxes might seem trivial, but if you were expecting to use the day of the week in your code (Monday, in the examples), you'd be in for a surprise. And don't be fooled into thinking that the implementation issues are merely cross-operating system problems. Differences in the implementation of the *Date* object and its methods exist in browsers on products running Microsoft Windows as well.

The only way to resolve these and other implementation differences in your JavaScript application is to perform both cross-browser and cross-platform tests. Doing so adds time to the application development cycle, but finding and fixing a problem during development is probably less costly than finding and fixing the problem after users discover it in a production environment.

The *Date* object can be handed a number of arguments, ranging from zero arguments to up to seven arguments. When the *Date* object constructor is passed a single string argument, the string is assumed to contain the date. When it is passed a number type of argument, the argument is assumed to be the date in milliseconds since January 1, 1970, and when it is passed seven arguments, they're assumed to be the following:

```
new Date(year, month, day, hours, minutes, seconds, milliseconds)
```

Note Only *year* and *month* are required arguments; the others are optional.

Remember the following points when using a *Date* object:

- The year should be given with four digits unless you want to specify a year between the year 1900 and the year 2000, in which case you'd just send in the two-digit year, 0 through 99, which is then added to 1900. So, 2008 equals the year 2008, but 98 is turned into 1998.
- The month is represented by an integer 0 through 11, with 0 being January and 11 being December.
- The day is an integer from 1 to 31.
- Hours are represented by 0 through 23, where 23 represents 11 P.M.
- Minutes and seconds are both integers ranging from 0 to 59.
- Milliseconds are an integer from 0 to 999.

Although the following procedure uses some items that won't be covered until later chapters, you're looking at the *Date* object now, so it's a good time learn how to write the date and time to a webpage—a popular operation.

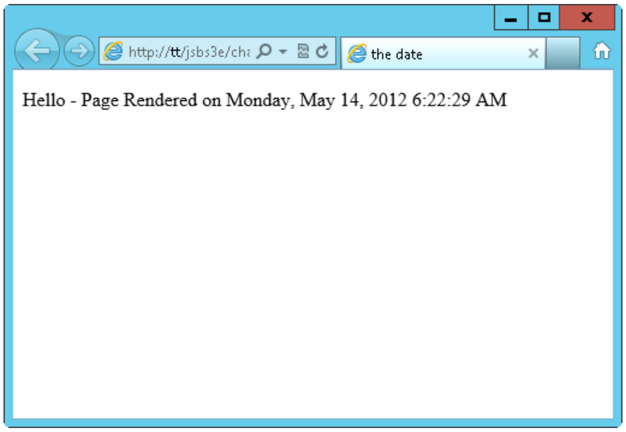
Writing the date and time to a webpage

1. Using Visual Studio, Eclipse, or another editor, edit the file writingthedate.html in the Chapter04 sample files folder in the companion content.

2. Within the page, add the code in boldface type shown here:

```
<!doctype html>
<html>
<head>
    <title>the date</title>
</head>
<body>
    <p id="dateField">&nbsp;</p>
    <script type = "text/javascript">
    var myDate = new Date();
    var dateString = myDate.toLocaleDateString() + " " + myDate.toLocaleTimeString();
    var dateLoc = document.getElementById("dateField");
    dateLoc.innerHTML = "Hello - Page Rendered on " + dateString;
    </script>
</body>
</html>
```

3. When saved and viewed in a web browser, you should receive a page like this (although the date you see will be different from what's shown here):



Hello - Page Rendered on Monday, May 14, 2012 6:22:29 AM

The relevant JavaScript from the preceding steps is repeated here:

```
var myDate = new Date();
var dateString = myDate.toLocaleDateString() + " " + myDate.toLocaleTimeString();
var dateLoc = document.getElementById("dateField");
dateLoc.innerHTML = "Hello - Page Rendered on " + dateString;
```

The JavaScript related to the *Date* object is rather simple. It takes advantage of the *toLocaleDateString()* method, which you've already seen, and its cousin, *toLocaleTimeString()*, which returns the local time. These two methods are concatenated together with a single space and placed into the *dateString* variable, like this:

```
var dateString = myDate.toLocaleDateString() + " " + myDate.toLocaleTimeString();
```

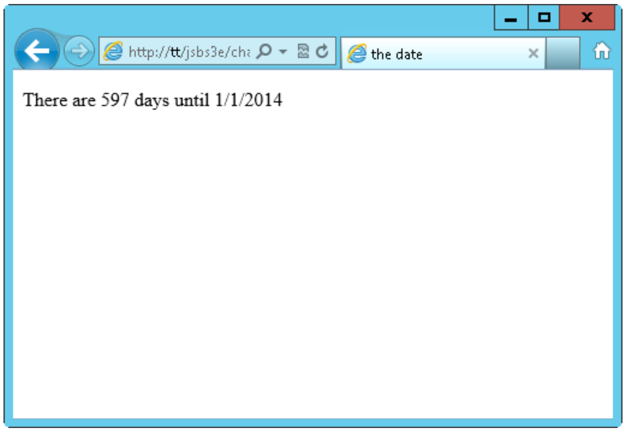
The remainder of the code writes the contents of the *dateString* variable to the webpage, which is covered in more detail in Part 2.

Counting down to a certain date in the future

1. Using Visual Studio, Eclipse, or another editor, edit the file countdown.html in the Chapter04 sample files folder, which you can find in the companion content.
2. Add the following code shown in boldface type to the page:

```
<!doctype html>
<html>
<head>
    <title>the date</title>
</head>
<body>
    <p id="dateField">&nbsp;</p>
    <script type = "text/javascript">
    var today = new Date();
    var then = new Date();
    // January 1, 2014
    then.setFullYear(2014,0,1);
    var diff = then.getTime() - today.getTime();
    diff = Math.floor(diff / (1000 * 60 * 60 * 24));
    var dateLoc = document.getElementById("dateField");
    dateLoc.innerHTML = "There are " + diff + " days until 1/1/2014";
    </script>

</body>
</html>
```

3. Save the page, and view it in a web browser. Depending on the date on your computer, the number of days represented will be different, but the general appearance of the page should look like this:



```
There are 597 days until 1/1/2014
```

Tip Be careful when using JavaScript dates for anything other than displaying them. Because the dates are dependent on the visitor's time, don't rely on them when an accurate time might be important—for example, in an ordering process.

The exercise you just completed used some additional functions of both the *Math* and *Date* objects, namely *floor()* and *getTime()*. While this book covers a lot of ground, it's not a complete JavaScript language reference. For that and even more information, refer to the ECMA-262 standard at *http://www.ecma-international.org/publications/standards/Ecma-262.htm*.

The next procedure shows how to calculate (or better yet, roughly estimate) the time it takes for a webpage to load in a person's browser.

Note The next procedure isn't accurate because it doesn't take into consideration the time required for the loading and rendering of images (or other multimedia items), which are external to the text of the webpage. A few more bits load after the script is finished running.

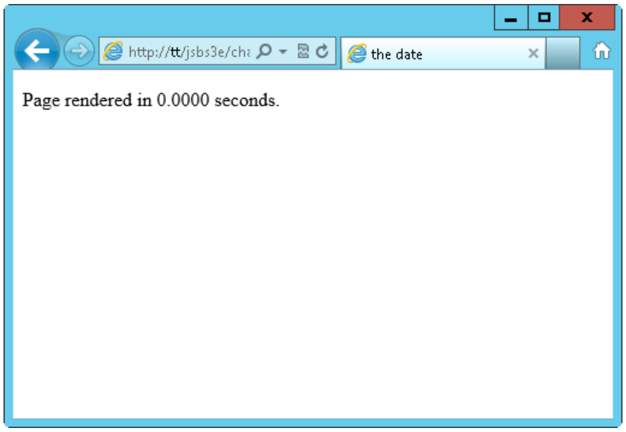
Calculating render time

1. Using Visual Studio, Eclipse, or another editor, edit the file render.html in the Chapter04 sample files folder, which you can find in the companion content.

2. Add the following code shown in boldface type to the page:

```
<!doctype html>
<html>
<head>
    <title>the date</title>
    <script type = "text/javascript">
    var started = new Date();
    var now = started.getTime();
    </script>
</head>
<body>
    <p id="dateField">&nbsp;</p>
    <script type = "text/javascript">
    var bottom = new Date();
    var diff = (bottom.getTime() - now)/1000;
    var finaltime = diff.toPrecision(5);
    var dateLoc = document.getElementById("dateField");
    dateLoc.innerHTML = "Page rendered in " + finaltime + " seconds.";
    </script>

</body>
</html>
```

3. Save the page, and view it in a web browser. Depending on the speed of your computer, web server, and network connection, you might receive a page that indicates only 0 seconds for the page load time, like this:



Page rendered in 0.0000 seconds.

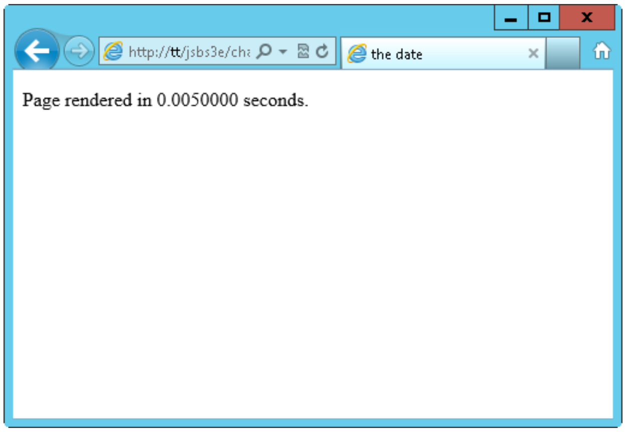
4. If your page takes 0.0000 seconds, as mine did, you can introduce a delay into the page so that you can test it. (I'd never recommend doing this on a live site because I can't think of a reason you'd want to slow down the rendering of your page! But introducing a delay can come in handy for testing purposes.) Using a *for* loop is a cheap and easy way to slow down the JavaScript execution:

    ```
    for (var i = 0; i < 1000000; i++) {
       //delay

    }
    ```

    The value I chose, 1000000, is arbitrary. You might need to choose a larger or smaller number to cause the desired delay. The final code looks like this:

    ```
    <!doctype html>
    <html>
    <head>
        <title>the date</title>
        <script type = "text/javascript">
        var started = new Date();
        var now = started.getTime();
        for (var i = 0; i < 1000000; i++) {
           //delay
        }
        </script>
    </head>
    <body>
        <p id="dateField">&nbsp;</p>
        <script type = "text/javascript">
        var bottom = new Date();
        var diff = (bottom.getTime() - now)/1000;
        var finaltime = diff.toPrecision(5);
        var dateLoc = document.getElementById("dateField");
        dateLoc.innerHTML = "Page rendered in " + finaltime + " seconds.";
        </script>

    </body>
    </html>
    ```

5. Save the page, and view it again in a web browser. You should see some delay in the page load, which causes the value to be a positive number:



```
Page rendered in 0.0050000 seconds.
```

When using this or similar functions to determine the page load times, to calculate the most accurate value, place the initial variable near the top of the page or script, and then place another one near the bottom of the page.

Note The *now()* method of the *Date()* object can also be used as a substitute for *getTime()*.

You just learned about a few of the more than 40 methods of the *Date* object. Many of these methods have UTC (Coordinated Universal Time) counterparts, meaning that they can get or set the date and time in UTC rather than local time. Table 4-4 lists the methods that return dates. With the exception of *getTime()* and *getTimezoneOffset()*, all these methods have UTC counterparts that are called using the format *getUTCDate()*, *getUTCDay()*, and so on.

TABLE 4-4 The *get* methods of the *Date* object

Method	Description
getDate()	Returns the day of the month
getDay()	Returns the day of the week
getFullYear()	Returns the four-digit year and is recommended in most circumstances over the *getYear()* method
getHours()	Returns the hours of a date
getMilliseconds()	Returns the milliseconds of a date
getMinutes()	Returns the minutes of a date
getMonth()	Returns the month of a date
getSeconds()	Returns the seconds of a date
getTime()	Returns the milliseconds since January 1, 1970
getTimezoneOffset()	Returns the number of minutes calculated as the difference between UTC and local time

Many of the *get...()* methods have siblings prefixed with *set*, as shown in Table 4-5. And like their *get* brethren, most of the *set...()* methods have UTC counterparts, except for *setTime()*.

TABLE 4-5 The *set* methods of the *Date* object

Method	Description
setDate()	Sets the day of the month of a date
setFullYear()	Sets the four-digit year of a date; also accepts the month and day-of-month integers
setHours()	Sets the hour of a date
setMilliseconds()	Sets the milliseconds of a date
setMinutes()	Sets the minutes of a date
setMonth()	Sets the month as an integer of a date
setSeconds()	Sets the seconds of a date
setTime()	Sets the time using milliseconds since January 1, 1970

The *Date* object also has several methods for converting the date to a string in a different format. You already reviewed some of these methods, such as *toLocaleDateString()*. Other similar methods include *toLocaleString()*, *toLocaleTimeString()*, *toString()*, *toISOString()*, *toDateString()*, *toUTCString()*, and *toTimeString()*. Feel free to experiment with these, noting that *toISOString()* is a new method in the ECMA-262 version 5 specification and support for it might not be available in all browsers. (It's notably missing from most versions of Internet Explorer.) The following simple one-line code examples will get you started experimenting. Try typing them in the address bar of your browser:

```
javascript:var myDate = new Date(); alert(myDate.toLocaleDateString());

javascript:var myDate = new Date(); alert(myDate.toLocaleString());

javascript:var myDate = new Date(); alert(myDate.toGMTString());

javascript:var myDate = new Date(); alert(myDate.toLocaleTimeString());

javascript:var myDate = new Date(); alert(myDate.toString());

javascript:var myDate = new Date(); alert(myDate.toISOString());

javascript:var myDate = new Date(); alert(myDate.toDateString());

javascript:var myDate = new Date(); alert(myDate.toUTCString());

javascript:var myDate = new Date(); alert(myDate.toTimeString());
```

You can also write these code samples without creating the *myDate* variable, like so:

```
javascript: alert(new Date().toUTCString());
```

Using the *RegExp* object

Regular expressions are the syntax you use to match and manipulate strings. If you've heard of or worked with regular expressions before, don't be alarmed. Regular expressions have an unnecessarily bad reputation solely because of their looks. And, lucky for me, we shouldn't judge things solely on looks alone. With that said, if you've had a bad experience with regular expressions, I'd ask that you read through this section with an open mind and see whether my explanation helps clear up some confusion.

The primary reason that I have confidence in your ability to understand regular expressions is that you're a programmer, and programmers use logic to reduce problems to small and simple pieces. When writing or reading a regular expression, the key is to reduce the problem to small pieces and work through each.

Another reason to have confidence is that you've probably worked with something close to regular expressions before, so all you need to do is extend what you already know. If you've worked with a command prompt in Microsoft Windows or with the shell in Linux/Unix, you might have looked for files by trying to match all files using an asterisk, or star (*) character, as in:

```
dir *.*
```

or:

```
dir *.txt
```

If you've used a wildcard character such as the asterisk, you've used an element akin to a regular expression. In fact, the asterisk is also a character used in regular expressions.

In JavaScript, regular expressions are used with the *RegExp* object and some syntax called *regular expression literals*. These elements provide a powerful way to work with strings of text or alphanumerics. The ECMA-262 implementation of regular expressions is largely borrowed from the Perl 5 regular expression parser. Here's a regular expression to match the word *JavaScript*:

```
var myRegex = /JavaScript/;
```

The regular expression shown would match the string "JavaScript" anywhere that it appeared within another string. For example, the regular expression would match in the sentence "This is a book about JavaScript," and it would match in the string "ThisIsAJavaScriptBook," but it would not match "This is a book about javascript," because regular expressions are case sensitive. (You can change this, as you'll see later in this chapter.)

With that short introduction you're now prepared to look at regular expressions in more detail. The knowledge you gain here will prepare you for the remainder of the book, helping you not only understand how to work with strings in JavaScript but also understand how to use regular expressions in other languages. This section provides a reference for regular expression syntax and shows a couple simple examples.

The syntax of regular expressions

Regular expressions have a terse—and some would argue cryptic—syntax. But don't let terse syntax scare you away from regular expressions, because in that syntax is power. This is a brief introduction to regular expressions. It's not meant to be exhaustive. (There are entire books on regular expressions.) However, you'll find that this gentle introduction will serve you well for the remainder of the book. Don't worry if this material doesn't sink in on the first read through. There are multiple tables that make it easy to use as a reference later.

The syntax of regular expressions includes several characters that have special meaning, including characters that anchor the match to the beginning or end of a string, a wildcard, and groups of characters, among others.

Table 4-6 shows several of the special characters.

TABLE 4-6 Common special characters in JavaScript regular expressions

Character	Description
^	Sets an anchor to the beginning of the input.
$	Sets an anchor to the end of the input.
.	Matches any character.
*	Matches the previous character zero or more times. Think of this as a wildcard.
+	Matches the previous character one or more times.
?	Matches the previous character zero or one time.
()	Places any matching characters inside the parentheses into a group. This group can then be referenced later, such as in a replace operation.
{n, }	Matches the previous character at least *n* times.
{n,m}	Matches the previous character at least *n* but no more than *m* times.
[ ]	Defines a character class to match any of the characters contained in the brackets. This character can use a range like 0–9 to match any number or like a–z to match any letter.
[^]	The use of a caret within a character class negates that character class, meaning that the characters in that class cannot appear in the match.
\	Typically used as an escape character, and meaning that whatever follows the backslash is treated as a literal character instead of as having its special meaning. Can also be used to define special character sets, which are shown in Table 4-7.

In addition to the special characters, several sequences exist to match groups of characters or nonalphanumeric characters. Some of these sequences are shown in Table 4-7.

TABLE 4-7 Common character sequences in JavaScript regular expressions

Character	Match
\b	Word boundary.
\B	Nonword boundary.
\c	Control character when used in conjunction with another character. For example, \cA is the escape sequence for Control-A.
\d	Digit.

Character	Match
\D	Nondigit.
\n	Newline.
\r	Carriage return.
\s	Single whitespace character such as a space or tab.
\S	Single nonwhitespace character.
\t	Tab.
\w	Any alphanumeric character, whether number or letter.
\W	Any nonalphanumeric character.

And finally, in addition to the characters in Table 4-7, you can use the modifiers *i*, *g*, and *m*. The *i* modifier specifies that the regular expression should be parsed in a case-insensitive manner, while the *g* modifier indicates that the parsing should continue after the first match, sometimes called *global* or *greedy* (thus the *g*). The *m* modifier is used for multiline matching. You'll see an example of modifier use in an upcoming example.

The *RegExp* object has its own methods, including *exec* and *test*, the latter of which tests a regular expression against a string and returns *true* or *false* based on whether the regular expression matches that string. However, when working with regular expressions, using methods native to the *String* type, such as *match*, *search*, *split*, and *replace*, is just as common.

The *exec()* method of the *RegExp* object is used to parse the regular expression against a string and return the result. For example, parsing a simple URL and extracting the domain might look like this:

```
var myString = "http://www.braingia.org";
var myRegex = /http:\/\/\w+\.(.*)/i;
var results = myRegex.exec(myString);
alert(results[1]);
```

The output from this code is an alert showing the domain portion of the address, as shown in Figure 4-6.

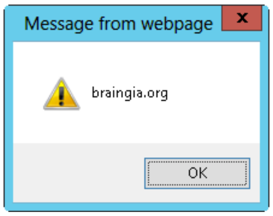


FIGURE 4-6 Parsing a typical web URL using a regular expression.

A breakdown of this code is helpful. First you have the string declaration:

```
var myString = "http://www.braingia.org";
```

This is followed by the regular expression declaration and then a call to the *exec()* method, which parses the regular expression against the string found in *myString* and places the results into a variable called *results*.

```
var myRegex = /http:\/\/\w+\.(.*)/i;
var results = myRegex.exec(myString);
```

The regular expression contains several important elements. It begins by looking for the literal string *http:*. The two forward slashes follow, but because forward slashes (/) are special characters in regular expressions, you must escape them by using backslashes (\),making the regular expression *http:\/\/* to this point.

The next part of the regular expression, *\w*, looks for any single alphanumeric character. Web addresses are typically *www*, so don't be confused into thinking that the expression is looking for three literal *w*s—the host in this example could be called *web*, *host1*, *myhost*, or *www*, as shown in the code you're examining. Because *\w* matches any single character, and web hosts typically have three characters (*www*), the regular expression adds a special character + to indicate that the regular expression must find an alphanumeric character at least once and possibly more than once. So now the code has *http:\/\/\w+*, which matches the address *http://www* right up to the *.braingia.org* portion.

You need to account for the dot character between the host name (*www*) and the domain name (*braingia.org*). You accomplish this by adding a dot character (.), but because the dot is also a special character, you need to escape it with \.. You now have *http:\/\/\w+\.*, which matches all the elements of a typical address right up to the domain name.

Finally, you need to capture the domain and use it later, so place the domain inside parentheses. Because you don't care what the domain is or what follows it, you can use two special characters: the dot, to match any character; and the asterisk, to match any and all of the previous characters, which is any character in this example. You're left with the final regular expression, which is used by the *exec()* method. The result is placed into the *results* variable. Also note the use of the *i* modifier, to indicate that the regular expression will be parsed in a case-insensitive manner.

If a match is found, the output from the *exec()* method is an array containing the last characters matched as the first element of the array and an index for each captured portion of the expression.

In the example shown, the second element of the array (*1*) is sent to an alert, which produces the output shown in Figure 4-6.

```
alert(results[1]);
```

That's a lot to digest, and I admit this regular expression could be vastly improved with the addition of other characters to anchor the match and to account for characters after the domain as well as non-alphanumerics in the host name portion. However, in the interest of keeping the example somewhat simpler, the less-strict match is shown.

The *String* object type contains three methods for both matching and working with strings and uses regular expressions to do so. The *match*, *replace*, and *search* methods all use regular expression pattern matching. Because you've learned about regular expressions, it's time to introduce these methods.

The *match* method returns an array with the same information as the *Regexp* data type's *exec()* method. Here's an example:

```
var emailAddr = "suehring@braingia.com";
var myRegex = /\.com/;
var checkMatch = emailAddr.match(myRegex);
alert(checkMatch[0]); //Returns .com
```

This can be used in a conditional to determine whether a given email address contains the string *.com*:

```
var emailAddr = "suehring@braingia.com";
var myRegex = /\.com/;
var checkMatch = emailAddr.match(myRegex);
if (checkMatch !== null) {
    alert(checkMatch[0]); //Returns .com
}
```

The *search* method works in much the same way as the *match* method but sends back only the index (position) of the first match, as shown here:

```
var emailAddr = "suehring@braingia.com";
var myRegex = /\.com/;
var searchResult = emailAddr.search(myRegex);
alert(searchResult); //Returns 17
```

If no match is found, the *search* method returns *-1*.

The *replace* method does just what its name implies—it replaces one string with another when a match is found. Assume in the email address example that I want to change any .com email address to a .net email address. You can accomplish this by using the *replace* method, like so:

```
var emailAddr = "suehring@braingia.com";
var myRegex = /\.com$/;
var replaceWith = ".net";
var result = emailAddr.replace(myRegex,replaceWith);
alert(result); //Returns suehring@braingia.net
```

If the pattern doesn't match, the original string is placed into the *result* variable; if it does, the new value is returned.

Note You can use several special characters to help with substitutions. Please see the ECMA-262 specification for more information about these methods.

Later chapters show more examples of *string* methods related to regular expressions. Feel free to use this chapter as a reference for the special characters used in regular expressions.

References and garbage collection

Some types of variables or the values they contain are primitive, whereas others are reference types. The implications of this might not mean much to you at first glance—you might not even think you'll ever care about this. But you'll change your mind the first time you encounter odd behavior with a variable that you just copied.

First, some explanation: objects, arrays, and functions operate as *reference types*, whereas numbers, Booleans, *null*, and *undefined* are known as *primitive types*. According to the ECMA-262 specification, other primitive types exist, such as *Numbers* and *Strings*, but *Strings* aren't relevant to this discussion.

When a number is copied, the behavior is what you'd expect: The original and the copy both get the same value. However, if you change the original, the copy is unaffected. Here's an example:

```
// Set the value of myNum to 20.
var myNum = 20;
// Create a new variable, anotherNum, and copy the contents of myNum to it.
// Both anotherNum and myNum are now 20.
var anotherNum = myNum;
// Change the value of myNum to 1000.
myNum = 1000;
// Display the contents of both variables.
// Note that the contents of anotherNum haven't changed.
alert(myNum);
alert(anotherNum);
```

The alerts display *1000* and *20*, respectively. When the variable *anotherNum* gets a copy of *myNum*'s contents, it holds on to the contents no matter what happens to the variable *myNum* after that. The variable does this because numbers are primitive types in JavaScript.

Contrast that example with a variable type that's a reference type, as in this example:

```
// Create an array of three numbers in a variable named myNumbers.
var myNumbers = [20, 21, 22];
// Make a copy of myNumbers in a newly created variable named copyNumbers.
var copyNumbers = myNumbers;
// Change the first index value of myNumbers to the integer 1000.
myNumbers[0] = 1000;
// Alert both.
alert(myNumbers);
alert(copyNumbers);
```

In this case, because arrays are reference types, both alerts display *1000,21,22*, even though only *myNumbers* was directly changed in the code. The moral of this story is to be aware that object, array, and function variable types are reference types, so any change to the original changes all copies.

Loosely related to this discussion of differences between primitive types and reference types is the subject of garbage collection. *Garbage collection* refers to the destruction of unused variables by the JavaScript interpreter to save memory. When a variable is no longer used within a program, the interpreter frees up the memory for reuse. It also does this for you if you're using Java Virtual machine or .NET Common Language Runtime.

This automatic freeing of memory in JavaScript is different from the way in which other languages, such as C++, deal with unused variables. In those languages, the programmer must perform the garbage collection task manually. This is all you really need to know about garbage collection.

Learning about type conversions

Before finishing the discussion on data types and variables, you should know a bit about *type conversions*, or converting between data types. JavaScript usually performs implicit type conversion for you, but in many cases, you can explicitly cast, or convert, a variable from one type to another.

Number conversions

You've already seen a conversion between two number formats, hexadecimal to base 10, in the example discussed in the section "Data types in JavaScript" earlier in this chapter. However, you can convert numbers to strings as well. JavaScript implicitly converts a number to a string when the number is used in a string context.

To explicitly convert a number to a string, cast the number as a string, as in this example:

```
// Convert myNumString as a string with value of 100
var myNumString = String(100);
```

String conversions

In the same way that you can convert numbers to strings, you can convert strings to numbers. You do this by casting the string as a number.

```
var myNumString = "100";
var myNum = Number(myNumString);
```

Tip JavaScript converts strings to numbers automatically when those strings are used in a numeric context. However, in practice, I've had hit-or-miss luck with this implicit conversion, so I usually just convert to a number whenever I want to use a number. The downside of doing this is that you have to execute some extra code, but doing that is better than the uncertainty inherent in leaving it up to a JavaScript interpreter.

Boolean conversions

Booleans are converted to numbers automatically when used in a numeric context. The value of *true* becomes *1*, and the value of *false* becomes *0*. When used in a string context, *true* becomes "true", and *false* becomes "false". The *Boolean()* function exists if you need to explicitly convert a number or string to a Boolean value.

Exercises

1. Declare three variables—one number and two strings. The number should be *120*, and the strings should be “5150” and “Two Hundred Thirty”.
2. Create a new array with three numbers and two strings or words.
3. Use the *alert()* function to display the following string, properly escaped: Steve’s response was “Cool!”
4. Use Internet Explorer to examine three of your favorite websites, and debug the errors using Interne Explorer tools. Look closely for any JavaScript errors reported. Bonus: Try using Firebug to examine those same three websites.

Index

About the Author

STEVE SUEHRING is a technology architect who specializes in finding simple solutions to complex problems and complex solutions to simple problems. When not writing technology books, Steve enjoys playing several musical instruments. You can follow Steve on Twitter, @stevesuehring.