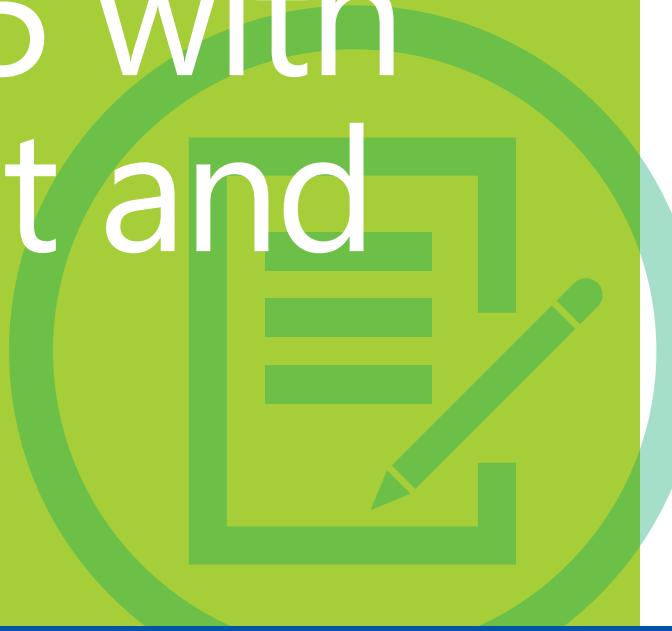




Programming in HTML5 with JavaScript and CSS3



Training Guide

Glenn Johnson

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2013 by Glenn Johnson

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2013933428
ISBN: 978-0-7356-7438-7

Printed and bound in the United States of America.

Second Printing

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

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Devon Musgrave

Developmental Editor: Devon Musgrave

Project Editor: Carol Dillingham

Editorial Production: nSight, Inc.

Technical Reviewer: Pierce Bizzaca; Technical Review services provided by Content Master, a member of CM Group, Ltd.

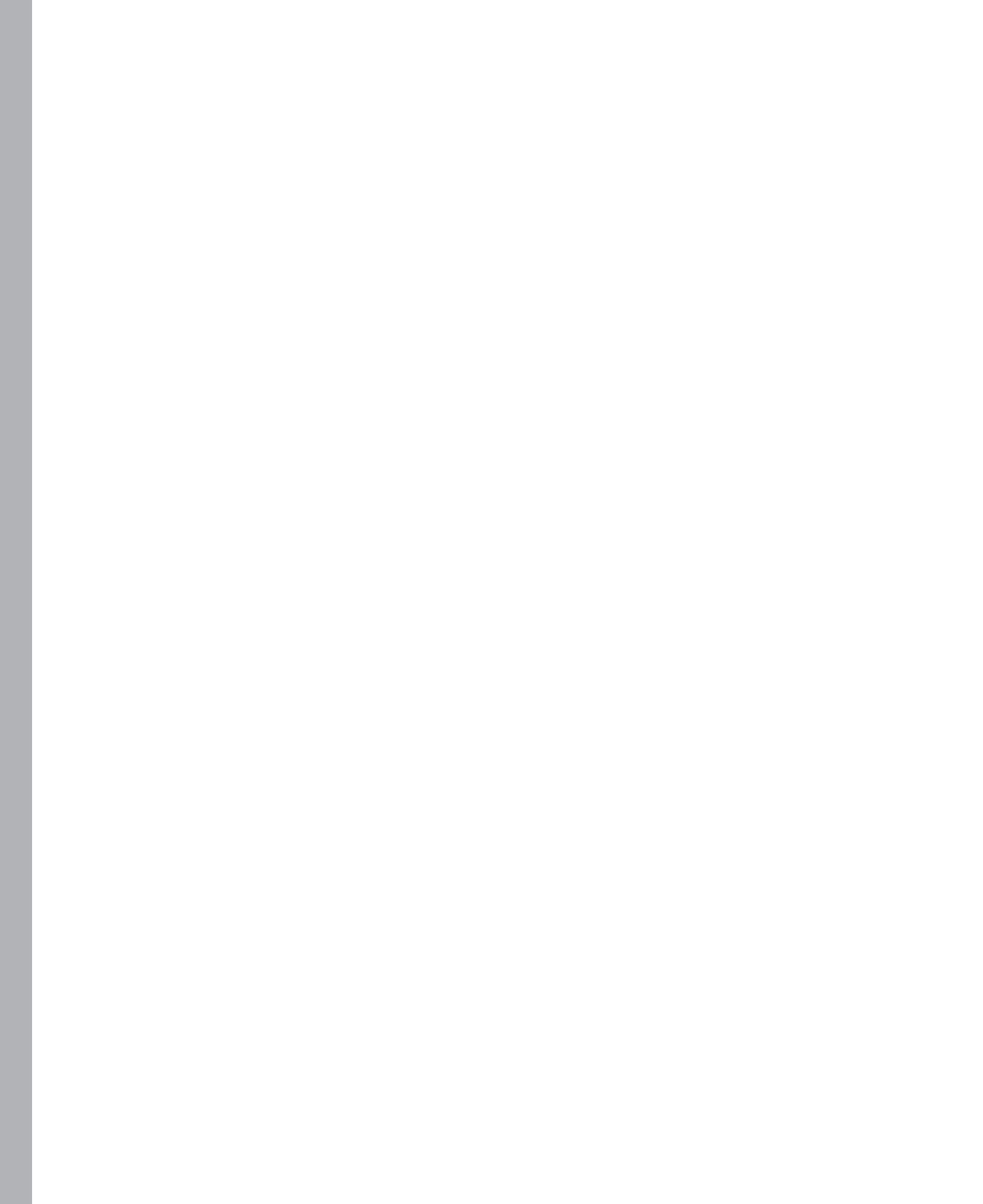
Copieditor: Kerin Forsyth

Indexer: Lucie Haskins

Cover: Twist Creative • Seattle

Contents at a glance

<i>Introduction</i>	<i>xxi</i>	
CHAPTER 1	Getting started with Visual Studio 2012 and Blend for Visual Studio 2012	1
CHAPTER 2	Getting started with HTML5	29
CHAPTER 3	Getting started with JavaScript	65
CHAPTER 4	Getting started with CSS3	137
CHAPTER 5	More HTML5	205
CHAPTER 6	Essential JavaScript and jQuery	261
CHAPTER 7	Working with forms	311
CHAPTER 8	Websites and services	341
CHAPTER 9	Asynchronous operations	393
CHAPTER 10	WebSocket communications	415
CHAPTER 11	HTML5 supports multimedia	437
CHAPTER 12	Drawing with HTML5	459
CHAPTER 13	Drag and drop	507
CHAPTER 14	Making your HTML location-aware	539
CHAPTER 15	Local data with web storage	555
CHAPTER 16	Offline web applications	581
<i>Index</i>	<i>621</i>	



Contents

Introduction	xix
<i>Backward compatibility and cross-browser compatibility</i>	xx
<i>System requirements</i>	xx
<i>Practice exercises</i>	xxi
<i>Acknowledgments</i>	xxi
<i>Errata and book support</i>	xxi
<i>We want to hear from you</i>	xxii
<i>Stay in touch</i>	xxii
Chapter 1 Getting started with Visual Studio 2012 and Blend for Visual Studio 2012	4
Lesson 1: Visual Studio 2012.....	5
Visual Studio 2012 editions	5
Visual Studio 2012 support for HTML5	6
CSS3 support	7
JavaScript support	7
Exploring Visual Studio Express 2012 for Windows 8	8
Exploring Visual Studio Express 2012 for Web	12
Lesson summary	14
Lesson review	15
Lesson 2: Blend for Visual Studio 2012	16
Exploring Blend	16
Lesson summary	22
Lesson review	23
Practice exercises	23

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:

www.microsoft.com/learning/booksurvey/

Exercise 1: Hello World with Visual Studio Express 2012 for Windows 8	24
Exercise 2: Hello World with Visual Studio Express 2012 for Web	25
Exercise 3: Hello World with Blend	27
Suggested practice exercises	28
Answers.....	29
Chapter 2 Getting started with HTML5	32
Lesson 1: Introducing HTML5.....	32
Understanding HTML, XHTML, and HTML5	33
Introducing semantic markup	34
Working with elements	35
Creating an HTML document	43
Lesson summary	45
Lesson review	46
Lesson 2: Embedding content.....	47
Embedding HTML by using inline frames	47
Working with hyperlinks	49
Adding images to your HTML document	50
Embedding plug-in content	53
Lesson summary	55
Lesson review	56
Practice exercises	56
Exercise 1: Create a simple website by using Visual Studio Express for Web	56
Exercise 2: Create additional pages	59
Exercise 3: Embedding Content	61
Suggested practice exercises	64
Answers.....	65
Chapter 3 Getting started with JavaScript	65
Lesson 1: Introducing JavaScript	66
Understanding JavaScript	66

Understanding the role of data	67
Using statements	71
Working with functions	73
Scoping variables	77
Nesting functions and nested local variable scoping	78
Converting to a different type	78
Conditional programming	80
Implementing code loops	84
Handling errors	87
Lesson summary	88
Lesson review	88
Lesson 2: Writing, testing, and debugging JavaScript	89
Hello World from JavaScript	90
Using the script tag	100
Handling browsers that don't support JavaScript	101
Inline JavaScript vs. external JavaScript files	102
Placing your script elements	102
Using the Visual Studio .NET JavaScript debugger	103
Lesson summary	107
Lesson review	107
Lesson 3: Working with objects	108
Working with arrays	108
Accessing DOM objects	112
Lesson summary	120
Lesson review	121
Practice exercises	121
Exercise 1: Create a calculator webpage	121
Exercise 2: Add the QUnit testing framework	123
Suggested practice exercises	133
Answers.....	134

Chapter 4 Getting started with CSS3 **137**

Lesson 1: Introducing CSS3.....	137
Defining and applying a style	139

Adding comments within a style sheet	139
Creating an inline style	140
Creating an embedded style	140
Creating an external style sheet	141
Lesson summary	144
Lesson review	145
Lesson 2: Understanding selectors, specificity, and cascading.....	145
Defining selectors	146
Understanding the browser's built-in styles	159
Extending browser styles with user styles	159
Working with important styles	159
How do styles cascade?	160
Using specificity	161
Understanding inheritance	162
Lesson summary	163
Lesson review	164
Lesson 3: Working with CSS properties.....	165
Working with CSS colors	166
Working with text	173
Working with the CSS box model	175
Setting the border, padding, and margin properties	176
Positioning <code><div></code> elements	178
Using the float property	186
Using the clear property	189
Using the box-sizing property	190
Centering content in the browser window	193
Lesson summary	193
Lesson review	194
Practice exercises	194
Exercise 1: Add a style sheet to the calculator project	195
Exercise 2: Clean up the web calculator	197
Suggested practice exercises	201
Answers.....	202

Chapter 5 More HTML5	205
Lesson 1: Thinking HTML5 semantics	205
Why semantic markup?	206
Browser support for HTML5	206
Creating semantic HTML5 documents	207
Creating an HTML5 layout container	207
Controlling format by using the <code><div></code> element	213
Adding thematic breaks	213
Annotating content	213
Working with lists	221
Lesson summary	228
Lesson review	229
Lesson 2: Working with tables	229
Table misuse	230
Creating a basic table	230
Adding header cells	231
Styling the table headers	232
Declaring the header, footer, and table body	233
Creating irregular tables	238
Adding a caption to a table	241
Styling columns	241
Lesson summary	242
Lesson review	243
Practice exercises	243
Exercise 1: Add a page layout to the calculator project	244
Exercise 2: Add styles to the calculator layout	246
Exercise 3: Cleaning up the web calculator	252
Suggested practice exercises	257
Answers.....	258
Chapter 6 Essential JavaScript and jQuery	261
Lesson 1: Creating JavaScript objects	262
Using object-oriented terminology	262

Understanding the JavaScript object-oriented caveat	263
Using the JavaScript object literal pattern	263
Creating dynamic objects by using the factory pattern	265
Creating a class	266
Using the prototype property	271
Debating the prototype/private compromise	274
Implementing namespaces	276
Implementing inheritance	278
Lesson summary	283
Lesson review	284
 Lesson 2: Working with jQuery.....	285
Introducing jQuery	285
Getting started with jQuery	286
Using jQuery	287
Enabling JavaScript and jQuery IntelliSense	291
Creating a jQuery wrapper for a DOM element reference	294
Adding event listeners	295
Triggering event handlers	295
Initializing code when the browser is ready	295
Lesson summary	296
Lesson review	296
 Practice exercises	297
Exercise 1: Create a calculator object	297
Suggested practice exercises	307
Answers.....	308

Chapter 7 Working with forms **311**

Lesson 1: Understanding forms	311
Understanding web communications	312
Submitting form data to the web server	316
Sending data when submitting a form	316
Using the <code><label></code> element	318
Specifying the parent forms	319
Triggering the form submission	319

Serializing the form	321
Using the autofocus attribute	321
Using data submission constraints	322
Using POST or GET	322
Lesson summary	323
Lesson review	324
Lesson 2: Form validation	324
Required validation	325
Validating URL input	327
Validating numbers and ranges	329
Styling the validations	330
Lesson summary	330
Lesson review	330
Practice exercises	331
Exercise 1: Create a Contact Us form	331
Exercise 2: Add validation to the Contact Us form	335
Suggested practice exercises	337
Answers.....	338

Chapter 8 Websites and services 341

Lesson 1: Getting started with Node.js.....	341
Installing Node.js	342
Creating Hello World from Node.js	342
Creating a Node.js module	344
Creating a Node.js package	345
Fast forward to express	354
Starting with express	354
Lesson summary	363
Lesson review	363
Lesson 2: Working with web services	364
Introducing web services	364
Creating a RESTful web service by using Node.js	366
Using AJAX to call a web service	368
Cross-origin resource sharing	380

Lesson summary	381
Lesson review	382
Practice exercises	382
Exercise 1: Create a website to receive data	382
Exercise 2: Create a web service to receive data	386
Suggested practice exercises	390
Answers.....	391
Chapter 9 Asynchronous operations	393
Lesson 1: Asynchronous operations using jQuery and WinJS	393
Using a promise object	394
Creating jQuery promise objects by using \$.Deferred()	395
Handling failure	397
Handling completion cleanup	397
Subscribing to a completed promise object	398
Chaining promises by using the pipe method	398
Parallel execution using \$.when().then()	400
Updating progress	400
Conditional asynchronous calls	401
Lesson summary	402
Lesson review	403
Lesson 2: Working with web workers	404
Web worker details	404
Lesson summary	405
Lesson review	406
Practice exercises	406
Exercise 1: Implement asynchronous code execution	406
Suggested practice exercises	412
Answers.....	413
Chapter 10 WebSocket communications	415
Lesson 1: Communicating by using WebSocket	415
Understanding the WebSocket protocol	416
Defining the WebSocket API	416

Implementing the WebSocket object	417
Dealing with timeouts	420
Handling connection disconnects	422
Dealing with web farms	422
Using WebSocket libraries	423
Lesson summary	424
Lesson review	424
Practice exercises	425
Exercise 1: Create a chat server	425
Exercise 2: Create the chat client	429
Suggested practice exercises	435
Answers.....	436
Chapter 11 HTML5 supports multimedia	437
Lesson 1: Playing video	437
Video formats	438
Implementing the <video> element	438
Setting the source	439
Configuring the <video> element	441
Accessing tracks	441
Lesson summary	442
Lesson review	443
Lesson 2: Playing audio	443
Audio formats	444
The <audio> element	444
Setting the source	445
Configuring the <audio> element	445
Lesson summary	446
Lesson review	446
Lesson 3: Using the HTMLMediaElement object.....	447
Understanding the HTMLMediaElement methods	447
Using HTMLMediaElement properties	447
Subscribing to HTMLMediaElement events	449
Using media control	450

Lesson summary	451
Lesson review	451
Practice exercises	452
Exercise 1: Create a webpage that displays video	452
Suggested practice exercises	455
Answers.....	456
Chapter 12 Drawing with HTML5	459
Lesson 1: Drawing by using the <code><canvas></code> element	460
The <code><canvas></code> element reference	460
CanvasRenderingContext2D context object reference	460
Implementing the canvas	462
Drawing rectangles	463
Configuring the drawing state	465
Saving and restoring the drawing state	474
Drawing by using paths	475
Drawing text	488
Drawing with images	490
Lesson summary	494
Lesson review	495
Lesson 2: Using scalable vector graphics	495
Using the <code><svg></code> element	496
Displaying SVG files by using the <code></code> element	499
Lesson summary	501
Lesson review	502
Practice exercises	502
Exercise 1: Create a webpage by using a canvas	502
Suggested practice exercises	505
Answers.....	506
Chapter 13 Drag and drop	507
Lesson 1: Dragging and dropping	507
Dragging	509
Understanding drag events	510

Dropping	511
Using the DataTransfer object	513
Lesson summary	516
Lesson review	516
Lesson 2: Dragging and dropping files.....	517
Using the FileList and File objects	517
Lesson summary	521
Lesson review	521
Practice exercises	522
Exercise 1: Create a number scramble game	522
Exercise 2: Add drag and drop to the game	526
Exercise 3: Add scramble and winner check	530
Suggested practice exercises	535
Answers.....	536
Chapter 14 Making your HTML location-aware	539
Lesson 1: Basic positioning	540
Geolocation object reference	540
Retrieving the current position	541
Handling errors	543
Addressing privacy	544
Specifying options	544
Lesson summary	545
Lesson review	546
Lesson 2: Monitored positioning	546
Where are you now? How about now?	546
Calculating distance between samples	548
Lesson summary	549
Lesson review	549
Practice exercises	550
Exercise 1: Map your current positions	550
Suggested practice exercises	553
Answers.....	554

Chapter 15 Local data with web storage	555
Lesson 1: Introducing web storage	555
Understanding cookies	556
Using the jQuery cookie plug-in	556
Working with cookie limitations	557
Alternatives to cookies prior to HTML5	557
Understanding HTML5 storage	558
Exploring localStorage	560
Using short-term persistence with <i>sessionStorage</i>	562
Anticipating potential performance pitfalls	563
Lesson summary	564
Lesson review	564
Lesson 2: Handling storage events	565
Sending notifications only to other windows	566
Using the StorageEvent object reference	566
Subscribing to events	567
Using events with <i>sessionStorage</i>	568
Lesson summary	568
Lesson review	568
Practice exercises	569
Exercise 1: Create a contact book by using <i>localStorage</i>	569
Suggested practice exercises	578
Answers.....	579
Lesson 1	579
Lesson 2	580
Chapter 16 Offline web applications	581
Lesson 1: Working with Web SQL	582
Considering the questionable longevity of Web SQL	582
Creating and opening the database	582
Performing schema updates	583
Using transactions	584

Lesson summary	588
Lesson review	589
Lesson 2: Working with IndexedDB	589
Using browser-specific code	590
Creating and opening the database	590
Using object stores	591
Using transactions	593
Inserting a new record	594
Updating an existing record	594
Deleting a record	595
Retrieving a record	595
Understanding cursors	596
Dropping a database	599
Lesson summary	599
Lesson review	600
Lesson 3: Working with the FileSystem API	600
Assessing browser support	601
Opening the file system	601
Creating and opening a file	602
Writing to a file	602
Reading a file	603
Deleting a file	604
Creating and opening a directory	604
Writing a file to a directory	605
Deleting a directory	605
Lesson summary	606
Lesson review	606
Lesson 4: Working with the offline application HTTP cache.....	607
Browser support	608
The cache manifest file	608
Updating the cache	609
Understanding events	610

Lesson summary	610
Lesson review	611
Practice exercises	611
Exercise 1: Modify a contact book to use IndexedDB	611
Suggested practice exercises	616
Answers.....	617
<i>Index</i>	621

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:

www.microsoft.com/learning/booksurvey/

Introduction

This training guide is designed for information technology (IT) professionals who develop or plan to develop HTML documents such as webpages or Windows Store applications. It is assumed that, before you begin using this guide, you are familiar with web development and common Internet technologies.

This book covers some of the topics and skills that are the subject of the Microsoft certification exam 70-480. If you are using this book to complement your study materials, you might find this information useful. Note that this book is designed to help you in the job role; it might not cover all exam topics. If you are preparing for the exam, you should use additional study materials to help bolster your real-world experience. For your reference, a mapping of the topics in this book to the exam objectives is included in the back of the book.

By using this training guide, you will learn how to do the following.

- Create a project using Visual Studio Express 2012 for Web.
- Create a project using Blend for Visual Studio 2012.
- Create a project using Visual Studio Express 2012 for Windows 8.
- Create an HTML document using semantic markup.
- Implement JavaScript functionality with your HTML documents.
- Use test-driven development techniques for writing JavaScript code.
- Create Cascading Style Sheets (CSS) that visually format your HTML document.
- Create HTML tables.
- Create JavaScript objects.
- Use jQuery to simplify JavaScript programming.
- Create HTML forms with validation.
- Create a Node.js website and web service.
- Call web services from JavaScript.
- Perform asynchronous JavaScript operations.
- Perform WebSocket communications.
- Play audio and video on a webpage.
- Draw with an HTML5 canvas.
- Use SVG image files.
- Perform drag and drop operations.
- Make your HTML location aware.
- Persist data on the browser client.

Backward compatibility and cross-browser compatibility

This book does not attempt to cover every difference between every version of every browser. Such a comprehensive discussion could easily yield a library of books.

Most of the code in this book is written using Internet Explorer 10, which is installed with Windows 8. In addition, many but not all the code examples were tested using the following browsers.

- Firefox 17.0.1
- Google Chrome 23.0.1271.97 m
- Opera 12.11
- Apple Safari 5.1.7

In most cases, if the other browsers were not compatible, there is a note stating so. This is especially true in the last chapters because web storage is still relatively new, and the requirements are still fluid.

The best way to see which features are available among browsers is to visit a website that is updated when new browser versions are released and HTML5 features are updated. The website <http://caniuse.com> is particularly good.

System requirements

The following are the minimum system requirements your computer needs to meet to complete the practice exercises in this book.

- Windows 8 or newer. If you want to develop Windows Store applications, you need Windows 8 on your development computer.

Hardware requirements

This section presents the hardware requirements for using Visual Studio 2012.

- 1.6 GHz or faster processor
- 1 GB of RAM (more is always recommended)
- 10 GB (NTFS) of available hard disk space
- 5400 RPM hard drive
- DirectX 9–capable video card running at 1024 × 768 or higher display resolution.
- Internet connectivity

Software requirements

The following software is required to complete the practice exercises.

- Visual Studio 2012 Professional, Visual Studio 2012 Premium, or Visual Studio 2012 Ultimate. You must pay for these versions, but in lieu of one of these versions, you can install the following free express versions.
 - Visual Studio Express 2012 for Web. Available from <http://www.microsoft.com/visualstudio/eng/downloads#d-express-web>.
 - Visual Studio Express 2012 for Windows 8. This installation also installs Blend for Visual Studio 2012. Available from <http://www.microsoft.com/visualstudio/eng/downloads#d-express-web>.

Practice exercises

This book features practices exercises to reinforce the topics you've learned. These exercises are organized by chapter, and you can download them from <http://aka.ms/TGProgHTML5/files>.

Acknowledgments

Thanks go to the following people for making this book a reality.

- To Carol Dillingham for your constructive feedback throughout the entire process of writing this book. Thanks for also having patience while the winter holiday months were passing, and my desire and ability to write was constantly interrupted.
- To Devon Musgrave for providing me the opportunity to write this book.
- To Kerin Forsyth for your hard work in making this book consistent with other Microsoft Press books and helping me with the delivery of this book.
- To Pierce Bizzaca for your technical reviewing skills.

To all the other editors and artists who played a role in getting my book to the public, thank you for your hard work and thanks for making this book venture a positive experience for me.

Errata and book support

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

<http://aka.ms/TGProgHTML5/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, send an email to Microsoft Press Book Support at mspinput@microsoft.com.

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

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://aka.ms/tellpress>

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 at <http://twitter.com/MicrosoftPress>.

CHAPTER 5

More HTML5

The previous chapters covered a lot of material you need to know. Much, but not all, of the content in the previous chapters existed before the HTML5 technologies came to be. This chapter provides a transition, moving you from old to new topics. Lesson 1, “Thinking HTML5 semantics,” discusses many aspects of HTML5 semantics that are primarily new topics. Lesson 2, “Working with tables,” explains tables, which is an older but relevant topic, and describes added features in HTML5.

Lessons in this chapter:

- Lesson 1: Thinking HTML5 semantics **205**
- Lesson 2: Working with tables **229**

Before you begin

To complete this chapter, you must have some understanding of web development. This chapter requires the hardware and software listed in the “System requirements” section in the book’s Introduction.

Lesson 1: Thinking HTML5 semantics

The previous chapter covered CSS positioning; all the examples used the `<div>` element. The `<div>` element has been the preferred element to use for positioning content when creating a page layout.

Many developers have also used the `<table>` element, but that element is much more difficult to use, especially to maintain a website. Lesson 2 explains the `<table>` element but doesn’t use a `<table>` element for page layout.

This lesson provides a different approach to creating a page layout; it covers semantic elements and explains why you should use them.

After this lesson, you will be able to:

- Create a semantic layout.
- Create an HTML5 document.
- Annotate content.
- Display various forms of semantic content.

Estimated lesson time: 30 minutes

Why semantic markup?

One of the problems with using `<div>` and `` elements is that they have little meaning other than “I need to do something with this content.” For `<div>` elements, you typically need to position the content on the page. For `` elements, you need to apply special formatting to the content.

You might be wondering what kind of meaning the `<div>` and `` elements can provide. For `<div>` elements, it might be better to have an element that represents the page header and can be positioned. You might want a different element that represents the page footer and can be positioned.

Are your users reading your HTML source? If the `<div>` element is the all-purpose tool to position elements, why use these new semantic elements?

These are good questions and thoughts. In fact, if you search the web for semantic markup, you’ll see plenty of discussions, some quite heated, about this topic.

Developers have been using `<div>` elements for page layout, and the developer usually provides the meaning of each `<div>` element based on its id or CSS class. The W3C analyzed thousands of webpages and found the most common id and class names. Rather than start over, the W3C made these names into new elements. Obvious examples are the `<header>` and `<footer>` elements.

Browser support for HTML5

Your users typically don’t read your HTML source when they browse to your website, but many machines are reading your HTML source with the goal of interpreting your webpage. Web crawlers are constantly surfing the Internet, reading webpages and building indexed searchable content that can be used to find your website. Many people have Nonvisual Desktop Access (NVDA) devices, which provide an alternate means of viewing, reading, and processing webpages. Some NVDA devices implement voice synthesis to read webpages to visually impaired people; others provide a Braille-like interface so the user can read your webpages by touch, as shown in Figure 5-1.

NVDA devices need your help to interpret your webpage content properly. They need you to use meaningful HTML tags that define the purpose of each element's contents. Doing so helps crawlers produce better matches to search queries, and NVDA devices that read your webpages to users can provide a more meaningful experience. For more information, visit <http://www.nvda-project.org/>.



FIGURE 5-1 The refreshable Braille display

Creating semantic HTML5 documents

Now that you understand the importance of using semantic markup, you might decide that you'll never use a `<div>` or `` element again. However, you will come across content that needs to be styled but doesn't clearly fit the meaning of any semantic elements. You can and should use `<div>` and `` elements in these scenarios.

Throughout this book, you will find many HTML5 tags. As you create your HTML pages, you will be faced with the sometimes daunting task of providing meaning to your content by supplying semantic tags. Use semantics carefully so you use an element only for its intended purpose. If you need a custom element, use the `<div>` or `` tag and add a class name or id that conveys the semantics you desire. Be pragmatic and not too much of a purist.

Creating an HTML5 layout container

The previous chapter showed many examples that demonstrate the use of `<div>` elements to provide positioning of content on a webpage. If you were creating a webpage to display blog posts, you might create a layout container for your page that looks like the example in Figure 5-2.

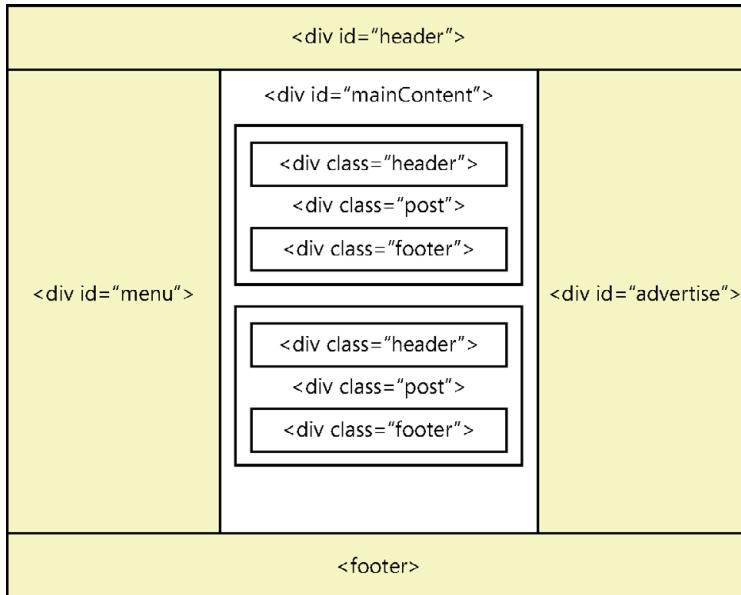


FIGURE 5-2 A blog site layout container using `<div>` elements

 A *layout container* lays out its children in a way that is flexible and reusable. For the developer, the purpose of the `<div>` elements is easy to understand based on the id and CSS class names. How can the browser understand the elements? If you want the browser to give the user the ability to focus automatically on the first element in the main content when the page opens, how would you do this? If you want the browser to give the user special quick-launch buttons for the menu items, how could you accomplish this?

By using HTML5 semantic elements, you can create a layout container that uses elements that are meaningful to both the developer and the browser. The following are common elements by which to create an HTML5 layout container.

- **<header>** Defines a section that provides a header. You can use the `<header>` element at the top of your HTML document as a page header. You can also use the `<header>` element in the `<article>` element.
- **<footer>** Defines a section that provides a footer. You can use the `<footer>` element at the bottom of your HTML document as a page footer. You can also use the `<footer>` element in the `<article>` element.
- **<nav>** Defines a section that houses a block of major navigational links.
- **<aside>** Defines a section of content that is separate from the content the `<aside>` element is in. This is typically used for sidebars.
- **<section>** Part of the whole that is typically named with an `<h1>` to `<h6>` element internal element.

- **<article>** A unit of content that can stand on its own and can be copied to other locations. A blog post is a good example of an article.

Figure 5-3 shows how these elements might be applied to create a layout container.

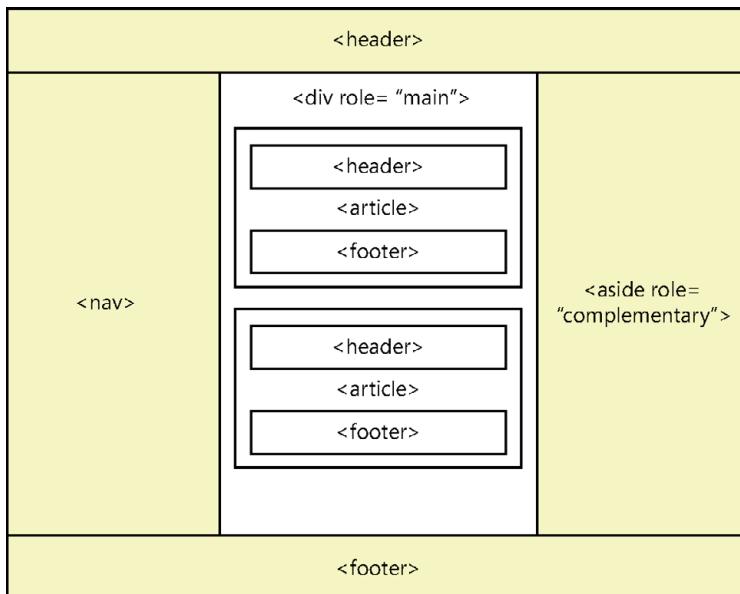


FIGURE 5-3 Layout container example, using the new HTML5 elements

In Figure 5-3, all *<div>* elements have been replaced with the new HTML5 elements.

Using the *<header>* element

The *<header>* elements in Figure 5-3 replace the *<div id="header">* elements in Figure 5-2, which cleans up the page a bit and provides a section meaning to browsers and other devices. Don't confuse the HTML5 *<header>* element that is within a *<body>* element, which is visible, with the HTML *<head>* element for the HTML document, which is invisible.

The *<header>* element should contain *<h1>* to *<h6>*, referred to as an *<hn>* moving forward element, containing your visible heading. You can also have other content with your *<header>* element, such as company logos and navigational links, as in the following example.

```
<header>
  <h1>Contoso Ltd.</h1>
  
  <p>Other supplementary information</p>
</header>
```

You can have multiple *<header>* elements in an HTML document. There are multiple *<header>* elements within this layout container because each *<article>* element has a *<header>*.

The `<header>` element can also contain the `<hgroup>` element, which provides the ability to group one or more `<hn>` elements within a `<header>` element, as shown in the following code example.

```
<header>
  <hgroup>
    <h1>Primary Header</h1>
    <h2>seconday header</h2>
  </hgroup>
  <img src='logo.jpg' alt='Company Logo' />
</header>
```

The `<hgroup>` is a wrapper for one or more related header elements. The `<hgroup>` element can be in a `<header>` element but doesn't need to be in a `<header>` element. The `<hgroup>` is most commonly used for subtitles and alternative titles.

The biggest reason for using the `<hgroup>` element has to do with using HTML5 outliner tools. If you have an `<h1>` header element and an `<h2>` subheader element, and you don't use the `<hgroup>` to connect a header and subheader, the outline treats the `<h2>` as a second level, but you really want the subheading to be ignored. The `<hgroup>` will expose only the first element and hide the other `<hn>` elements in the `<hgroup>`.

If your header is just an `<hn>` and contains no other content, you don't need a `<header>` element. As soon as you have more than a single `<hn>`, such as `` elements and `<p>` elements, wrap your content in a `<header>` element. When you have multiple `<hn>` elements, such as a header and subheader, wrap the `<hn>` elements in the `<hgroup>` element. A `<header>` element should not be nested inside a `<header>` element.

By looking at the difference between Figure 5-2 and Figure 5-3, you can see that the `<header>` element replaced the `<div id="header">` element. Be careful, because by default the `<header>` element on some browsers does not render as a rectangular block as the `<div>` element does. You can fix this by adding the following style rule to provide matching behavior.

```
header { display:block; }
```

Using the `<footer>` element

The `<footer>` elements in Figure 5-3 replace the `<div id="footer">` elements in Figure 5-2, which cleans up the page a bit and provides a section meaning to browsers and other devices.

The `<footer>` element typically contains information about the section it's in, such as who wrote it, copyright information, and links to related documents. The `<footer>` element is much like the `<header>` element except that it typically belongs at the bottom of a section instead of at the top. Like the `<header>` element, it can be used many times within an HTML document to provide ending content for the HTML document and ending content for articles and sections within the HTML document. The `<footer>` element should not be nested inside a `<footer>` element. The following is an example of a `<footer>` element.

```
<footer>
  <ul>
    <li>Copyright (C) 2012, Contoso Ltd., All rights reserved</li>
    <li><a href="default.html">Home</a></li>
  </ul>
</footer>
```

Using the *<nav>* element

The *<nav>* element in Figure 5-3 replaces the *<div id="menu">* element in Figure 5-2, which provides a section meaning to browsers and devices. The *<nav>* element wraps a group of major links that are on your page. Menus are the most common candidates for the *<nav>* element.

Like menus, footers commonly have groups of links, but you don't need to use the *<nav>* element if you are using the *<footer>* element, and the *<nav>* element is not required for links within your content. You can have many *<nav>* elements in an HTML document. For example, in addition to the menu that is normally on the left side or across the top of the page, you might have a group of links above the footer that link to the next page of blog posts or to other major areas of your site.

Think of a screen reader when implementing the *<nav>* element. It will be looking for the primary navigation area on the webpage so it can present these links to the user as menu items that have links to other areas within the current website. Links to off-site locations should not be part of the *<nav>* element. Footer links to secondary areas of your website also don't require a *<nav>* element.

Using the *<aside>* element

The *<aside>* element in Figure 5-3 replaces the *<div id="advertise">* element in Figure 5-2, which provides a section meaning to browsers and devices.

The *<aside>* element wraps secondary content when used for sidebars. In many cases, this is where the advertising and other site-related content goes. In addition, when the *<aside>* element is in an article, it should contain content tangentially related to the content within the article. The use of the *<aside>* element differs based on the context, as shown in the following example.

```
<body>
  <header>
    <h1>Blogging for fun</h1>
  </header>
  <article>
    <h1>Blog of the day</h1>
    <p>This is today's blog post. La, la, la, la, la,
    la, la, la, la, la, la</p>
    <aside>
      <!-- Inside the article, so it's related to the article -->
      <h1>What's this all about?</h1>
      <p>This article talks about la, la...</p>
    </aside>
```

```
</article>
<aside>
    <!-- Outside the article, so it's related to the sites -->
    <h2>Blog Advertising</h2>
    <p>You too can have your own blog...</p>
</aside>
</body>
```

The two meanings make sense when you consider that an article should be a complete unit that can be shared.

Using roles

In Figure 5-3, the `<aside>` element and the `<div>` element implement the role attribute, specified by the Web Accessible Initiative (WAI), which specifies the Accessible Rich Internet Applications (ARIA) suite, called WAI-ARIA.

WAI-ARIA defines the role class hierarchy and how roles are used to provide specific meaning to screen readers for accessibility purposes. There are many parent role classes, and there are child role classes that inherit from role classes. One such parent role class is called the *landmark* role class, which represents regions of the page intended as navigational landmarks. The following are child classes of the landmark role class.

- **application** An area declared as a web application as opposed to a web document.
- **banner** An area on a webpage that has site-specific content, such as site name and logo, instead of page-specific content; maximum one per webpage, usually header content.
- **complementary** An area on a webpage that complements the page but still has meaning if separated from the page.
- **contentinfo** An area that contains information about the parent document such as copyright notices and links; maximum one per webpage, usually footer content.
- **form** An area on a webpage that contains a collection of input controls for gathering data to be sent to the web server; search forms should use the search role.
- **main** An area that contains the main content of the document; maximum one per webpage.
- **navigation** An area that contains navigational links.
- **search** An area on a webpage that contains a collection of input controls for entering and displaying search information.

You can use these roles to provide meaning to an area of the webpage, but the new HTML5 elements already provide meaning. However, the HTML5 elements don't provide a new element to identify the main content of the webpage. Instead, all known content is not the main content, and what's left over must be the main content. Furthermore, the `<aside>` element is used as a sidebar, and you might want to provide more meaning. Why not use the WAI-ARIA role to provide meaning to other developers and to assistive devices? That is what is illustrated in Figure 5-3.



Controlling format by using the `<div>` element

Don't forget that the `<div>` element can be placed around content, enabling you to control its format. The `<div>` element is invisible and has no meaning, so when using HTML5, it's generally better to use a semantic element such as `article` or `section` to provide context that has meaning. If all you need is formatting, the use of the `<div>` element is perfect.

Adding thematic breaks

Use the `<hr/>` element to add a thematic break. It is a void element, so it cannot have any content. You can use the `<hr/>` element to provide a thematic break when there is a scene change in a story or to denote a transition to another topic within a section of a reference book.

Annotating content

When annotating content by using HTML5 elements, be aware that the `` and `<i>` elements that have been around since the beginning are still available but now have new meaning. This section describes the use of the `` and `<i>` elements and many other elements that can be used to annotate content.

Using the `` element

The `` element was used to produce bold text, but now elements should have meaning, not style. To keep the `` element but also have semantic elements, the meaning needed to change.

According to the W3C, the `` element represents a span of text to which attention is being drawn for utilitarian purposes without conveying any extra importance and with no implication of an alternate voice or mood, such as keywords in a document abstract; product names in a review; actionable words in interactive, text-driven software; or an article lede. Therefore, you can apply any style you want to the `` element, although keeping the bold style makes the most sense.

The `` element is the element of last resort because headings should be denoted with the `<h1>` element, emphasized text should be denoted with the `` element, important text should be denoted with the `` element, and marked or highlighted text should use the `<mark>` element. Refrain from using the `` element except to denote product names in a review, keywords in a document extract, or an article lede, as shown in the following example.

```
<article>
  <h1>PolyWannaWidget Review</h1>
  The <b>PolyWannaWidget</b> is the best product
    to use for creating crackers from nothing
    other than a hammer.
</article>
```

Using the `` element

Closely related to the `` element is the `` element, which represents strong importance for its contents. You can show relative importance by nesting `` elements within `` elements. Note that changing the importance of part of the text in a sentence does not change the meaning of the sentence. The following is an example that is in response to the question, "Should I take a left turn?"

```
<p>
    You need to turn <strong>right</strong>.
</p>
```

Note that the default styles for `` and `` elements look the same.

Using the `<i>` element

The `<i>` element was used to produce italic text, but like the `` element, the element should provide meaning, not style.

According to the W3C, the `<i>` element represents a span of text that is in an alternate voice or mood or is otherwise offset from the normal prose in a manner indicating a different quality of text, such as a taxonomic designation, a technical term, an idiomatic phrase from another language, a thought, or a ship name in Western texts.

This means that you can apply any style to the `<i>` element, although, like the `` element, you probably should keep the default style.

The `` element

Use the `` element for emphatic stress. Use it to designate text you'd pronounce somewhat differently, with emphasis. The following is an example that is in response to the question, "Can you find a developer?"

```
<p>
    I <em>am</em> a developer.
</p>
```

Note that the default styles for `` and `<i>` elements look the same.

Using the `<abbr>` element for abbreviations and acronyms

In earlier versions of HTML, you could indicate an acronym by using the `<acronym>` element, but in HTML5, the `<acronym>` element is obsolete. Use the `<abbr>` element to indicate an abbreviation or acronym and use the title attribute to provide the full text.

The `<abbr>` element is an inline element and can be used with `` or other inline elements as necessary. The following is an example of denoting an abbreviation and acronym by using the `<abbr>` element.

```
<p>
    The <abbr title='radio detection and ranging'>radar</abbr>
    must be repaired <abbr title='as soon as possible'>ASAP</abbr>
```

```
by Contoso, <abbr title='Incorporated'>Inc.</abbr>
</p>
```

Note that the title is not required, especially when you know that everyone will know the meanings of the abbreviations and acronyms.

The *<address>* element

Use the *<address>* element to define contact information for the author/owner of a document. You may include email address, postal address, or any contact address that references the author/owner of the document.

Be careful to use the *<address>* element only when referencing the author/owner of the document. Do not use it for arbitrary address information on your webpage. For example, if you are displaying your customer's address on a webpage, it should not be wrapped in an *<address>* element.

The following is an example of the use of the *<address>* element in the footer of a webpage.

```
<footer>
    Copyright (C) 2012
    <address>
        Contoso, Inc.
        <a href="mailto:WebMaster@Contoso.com">
            WebMaster@Contoso.com
        </a>
    </address>
</footer>
```

Quotations and citations

When it's time to start quoting, you can use the *<blockquote>* element to create a long, running quotation and the *<q>* element for an inline quotation. Both these elements have a *cite* attribute that names the source work of the quote.

The *<blockquote>* element is a block-level element; it can contain almost anything, including headers, footers, tables, and paragraphs. The *<blockquote>* element is a *sectioning root*, which means that any *<hn>* elements within the *<blockquote>* element will not be included in an outline of the HTML document. In addition, a single paragraph does not need to be included in a *<p>* element.

The *<blockquote>* and *<q>* elements have a *cite* attribute that names the source work, but as an attribute, this is hidden data. A better approach is to use the *<cite>* element, which you can place in the *<footer>* element of your *<blockquote>* and *<q>* elements. The citation should always contain the name of the work, not the author name. The following is an example of the *<blockquote>* element.

```
<blockquote>
    O Romeo, Romeo, wherefore art thou Romeo?<br />
    Deny thy father and refuse thy name;<br />
```



```
Or if thou wilt not, be but sworn my love<br />
And I'll no longer be a Capulet.<br />
<footer>
    <p>
        by William Shakespeare,
        <cite>Romeo and Juliet</cite> Act 2, scene 2
    </p>
</footer>
</blockquote>
```

The `<cite>` element contains only the name of the work, not the author or the location within the work.

When you want to add an inline quotation, use the `<q>` element instead of using quotation marks. The browser will insert the quotation marks for you. You can add the `cite` attribute to the `<q>` element, which should contain only the name of the work. Furthermore, the `<q>` element can be nested within another `<q>` element. The following is an example of the `<q>` element.

```
<p>
    John said to the audience <q>Sally was crying when she
    shouted <q>Leave me alone</q> and then she ran away.</q>
</p>
```

This example renders the first quotation by using double quotes and the second quotation by using single quotes.

Documenting code by using the `<code>` and `<samp>` elements

When you're documenting code and code examples in your HTML document, the `<code>` and `<samp>` elements provide a means for adding semantic meaning to your code and code output.

When you want to display source code of any type in the HTML document, use the `<code>` element, as shown in the following example.

```
<code class="keepWhiteSpace">
sayHello('Mom');
function sayHello(name)
{
    alert('Hello ' + name + '!');
}
</code>
```

After you run the sample code, you can document the output of the code by using the `<samp>` element, as shown in the following example.

```
<samp class="keepWhiteSpace">
    Hello Mom!
</samp>
```

Remember that the `<code>` and `<samp>` elements provide semantic meaning to the HTML, but they don't preserve the white space. For example, the preceding sample code will

display on one line, but the `keepWhiteSpace` class preserves the white space by using the following style rule.

```
.keepWhiteSpace {  
    white-space: pre;  
}
```

This style rule is not compatible with all browsers, so you might want to use the `<pre>` element to prevent white space normalization, as described next.

Displaying preformatted content by using the `<pre>` element

The browser typically normalizes the HTML content by removing extra white space, line feeds, and paragraphs from the rendered page. You will often need to provide blocks of text where you want to maintain the existing format when it's rendered. Use the `<pre>` element to prevent the normalization of the HTML document, as shown in the following example.

```
<pre>  
<code>  
sayHello('Mom');  
function sayHello(name)  
{  
    alert('Hello ' + name + '!');  
}  
</code>  
</pre>
```

In this example, the `<code>` element provides semantic meaning to the content, and the `<pre>` element prevents white-space normalization.

Using the `<var>` element

The `<var>` element denotes a variable in a mathematical equation, as shown in the following example.

```
<p>  
The resistance <var>r</var> of a piece of wire is equal to the voltage <var>v</var>  
divided by the current <var>i</var>.   
</p>
```

Using the `
` and `<wbr />` elements

The `
` and `<wbr />` elements are void elements, meaning that they cannot have any content and provide only a line break in your HTML document.

The `
` element provides an immediate line break, which continues the document flow on the next line of the browser.

The `<wbr />` element, which is a *word break*, provides an indication to the browser that it may insert a line break at this location. The browser decides whether to insert the break.

Using the *<dfn>* element to define a term

The *<dfn>* element denotes the definition of a term, also known as the defining instance of the term. The *<dfn>* element can contain a title attribute, which, if it exists, must contain the term being defined.

If the *<dfn>* element contains exactly one element child node and no child text nodes, and that child element is an *<abbr>* element with a title attribute, that attribute is the term being defined.

Consider the following example that uses the *<dfn>* element with the *<abbr>* element to provide a definition.

```
<p>
    A motor vehicle has a <dfn id="vin">
        <abbr title="Vehicle Identification Number">VIN</abbr></dfn>
        that is unique. Over the years, the
        <abbr title="Vehicle Identification Number">VIN</abbr>
        has had different formats,
        based on the vehicle manufacturer.
    </p>
<p>
    In the United States, the <a href="#vin">
        <abbr title="Vehicle Identification Number">VIN</abbr></a>
    was standardized to a 17 character format where
    the 10th character of the
    <abbr title="Vehicle Identification Number">VIN</abbr>
    represents the year of the vehicle.
    </p>
```

In this example, the *<dfn>* element is used once where the first instance of VIN is being presented. Inside the *<dfn>* element is an *<abbr>* element, which provides the meaning of VIN in its title attribute. The default style of the *<dfn>* element is italic text, as shown in Figure 5-4. The use of the *<a>* element provides a hyperlink to the definition.

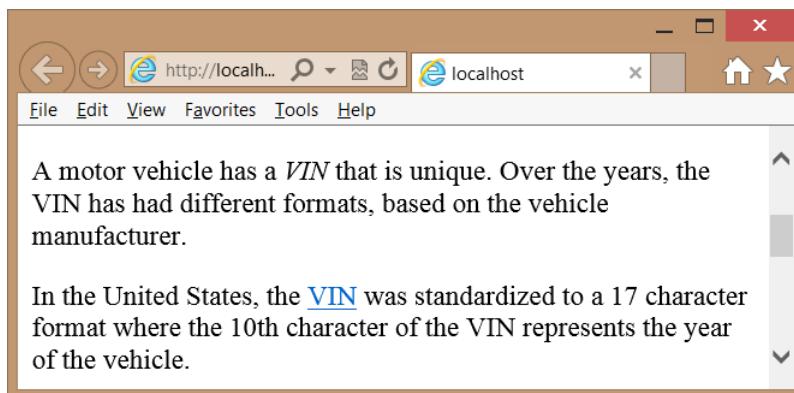


FIGURE 5-4 The *<dfn>* element italicizing its text by default



Working with figures

A *figure* is a unit of content that might have a caption and is referenced from the main document. Use the `<figure>` element to denote a figure that can be one or more photos, one or more drawings, one or more illustrations, or other content that is referred to as a unit. Use the `<figcaption>` element to denote an optional caption.

When using the `<figure>` element, remember that the figure is related to the main content of the page, and the figure's location is not important. This is different from the `<aside>` element, which is more related to the site than to the page's document. If placement is important, don't use the `<figure>` element; use the `<div>` element.

The following example shows the use of the `<figure>` and `<figcaption>` elements (bolded) to display an image that is referred to in the main document of the webpage.

```
<div role="main">
  <p>
    The peanut butter and jelly
    <abbr title="peanut butter and jelly">PB&J</abbr>
    sandwich has been a staple food of many families
    due to its health benefits, its cost, and its
    wonderful flavor.
  </p>
  <p>
    When assembling a peanut butter and jelly sandwich,
    you need to gather all the required materials as
    shown in <a href="#figure1">Figure 1</a>.
  </p>
  <figure id="figure1">
    
    <figcaption>Figure 1 The PB&J sandwich requirements.</figcaption>
  </figure>
</div>
```

Working with the `<summary>` and `<details>` elements

Use the `<details>` element with the `<summary>` element to create collapsible details content under the summary. The `<details>` and `<summary>` elements currently work with the Google Chrome browser only, but more support is expected.

In the `<details>` element, nest a `<summary>` element that contains the content that will always be displayed. The details content is placed inside the `<details>` element following the `<summary>` element. When the page is rendered, only the content of the `<summary>` element is displayed. Clicking the summary content causes the details content to be displayed. Clicking again causes the details content to be hidden.

```
<div role="main">
  <details>
    <summary>Make a peanut butter and jelly sandwich</summary>
    <p>
      The peanut butter and jelly
      <abbr title="peanut butter and jelly">PB&J</abbr>
    </p>
  </details>
</div>
```

```

        sandwich has been a staple food of many American families
        due to its health benefits, its cost, and its
        wonderful flavor.

    </p>
    <p>
        When assembling a peanut butter and jelly sandwich,
        you need to gather all the required materials as
        shown in <a href="#figure1">Figure 1</a>.

    </p>
    <figure id="figure1">
        
        <figcaption>The PB&J sandwich requirements.</figcaption>
    </figure>
    </details>
</div>

```

In this example, the previous example content is placed in the `<details>` element, and the `<summary>` element contains a general description of the content. Clicking the summary content toggles the display of the details.

Understanding other annotations

In addition to the annotation elements already discussed, the following is a list of annotations you might use in your HTML document.

- **`<s>`** Denotes strike-out text, text that is no longer valid.
- **`<u>`** Offsets a span of text without implying a difference of importance. The default behavior is to underline the text, but this could be accommodated better by using a span tag with the appropriate style.
- **`<mark>`** Marks, or highlights, a span of text.
- **`<ins>`** Indicates inserted text.
- **``** Indicates deleted text.
- **`<small>`** Indicates fine print.
- **`<sub>`** Indicates subscript.
- **`<sup>`** Indicates superscript.
- **`<time>`** Denotes a time of day or a date in the text.
- **`<kbd>`** Indicates user input.

Using language elements

You might need to provide content that uses characters of Chinese origin, which are called *kanji*. These characters are used in Chinese, Japanese, and Korean (CJK) languages. To indicate the pronunciation of *kanji*, you can use small phonetic characters, which are commonly called *ruby* or *furigana*. The term “*ruby*” has English roots from when printers used this term to refer to small type used for this purpose.



Use the `<ruby>` element to place a notation above or to the right of characters. Use the `<rt>` and `<rp>` elements with the `<ruby>` element to place the notation or to place parentheses around the ruby. Use the `<bdo>` element to define the text direction and use the `<bdi>` element to isolate a block of text to set the text direction.

Working with lists

HTML5 defines various semantic elements that can be used to create ordered, unordered, and descriptive lists. All lists have list items, which are implemented by using the `<i>` element. All lists support nesting of lists. This section describes each of these lists.

Ordered lists



An *ordered list* is a numbered list. Use the `` element when you want auto-numbering of the list items. The following example shows three favorite fruits.

```
<h3>Favorite Fruit</h3>
<ol>
  <li>Apples</li>
  <li>Oranges</li>
  <li>Grapes</li>
</ol>
```

This list is automatically rendered with numbers beside each fruit list item, as shown in Figure 5-5.

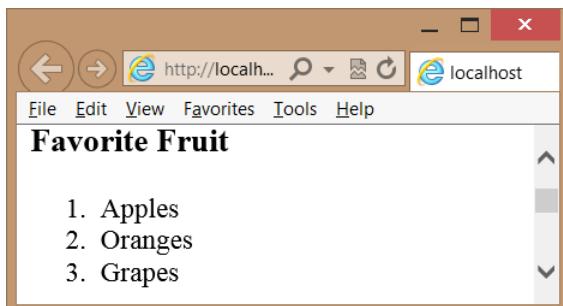


FIGURE 5-5 The ordered list numbering its items automatically

The `` element supports the following attributes.

- **reversed** Reverses the number order to be descending instead of ascending
- **start** Sets the starting number
- **type** Sets the list type; can be "1", "A", "a", or "I"

The `reversed` attribute currently does not work in most browsers, but you might find JavaScript libraries, such as *modernizr.js*, that emulate that functionality until the feature is implemented by the browser manufacturer. Even if you set the type to a value such as "A",

you still set the start as a number. The following is an example of the type and start attributes, using the favorite fruit list.

```
<h3>Favorite Fruit</h3>
<ol type="A" start="6" >
  <li>Apples</li>
  <li>Oranges</li>
  <li>Grapes</li>
</ol>
```

Figure 5-6 shows the rendered list. The start value of “6” translates to the letter “F” when rendered.

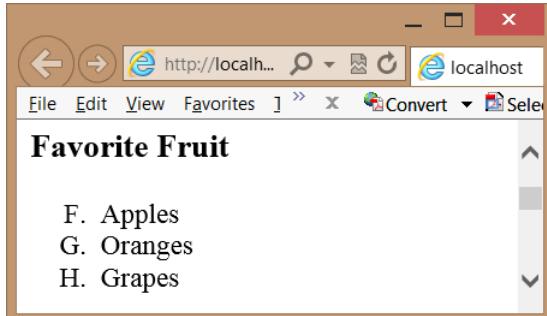


FIGURE 5-6 The ordered list with its type and start attributes set

Unordered lists

An *unordered list* is not auto-numbered. Use the `` element to create an unordered list of items. When the unordered list is rendered, it produces bullet points before each list item, as shown in the following example that describes the items required to repair a flat tire.

```
<h3>Items required to change a flat tire</h3>
<ul>
  <li>A jack</li>
  <li>A lug wrench with a socket on one end and a pry bar on the other</li>
  <li>A spare tire</li>
</ul>
```

Each item is rendered with a bullet, and where the text wraps to the next line, the text aligns itself properly with the text of the previous line, as shown in Figure 5-7.

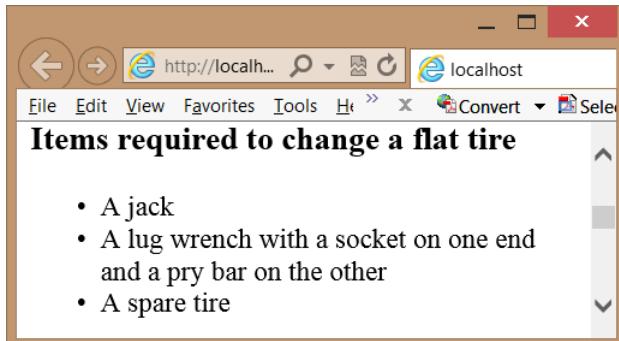


FIGURE 5-7 The unordered list rendering each list item as a bullet

Description lists

Use the `<dl>` element to create a description list, which consists of zero or more term-description groupings, also known as name-value or key-value pairs. Each grouping associates one or more terms or names, which are the contents of `<dt>` elements, with one or more descriptions or values, which are the contents of `<dd>` elements, as shown in the following example.

```
<h3>Common Vehicles</h3>
<dl>
  <dt>Boat</dt>
  <dd>A small vehicle propelled on water by oars, sails, or an engine</dd>
  <dt>Car</dt>
  <dd>An automobile</dd>
  <dd>A passenger vehicle designed for operation on ordinary roads
      and typically having four wheels and an engine</dd>
  <dt>Bicycle</dt>
  <dt>Bike</dt>
  <dd>A vehicle with two wheels in tandem, typically propelled by pedals
      connected to the rear wheel by a chain, and having handlebars
      for steering and a saddlelike seat</dd>
</dl>
```

In this example, the boat is associated with a single definition. The car is associated with two definitions. The bicycle and bike are both associated with the same definition. The rendered output is shown in Figure 5-8.

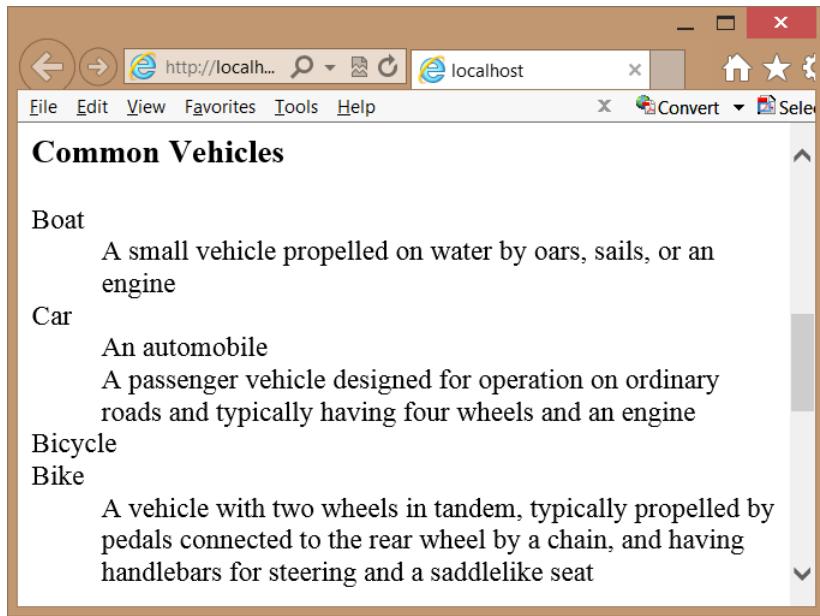


FIGURE 5-8 The definition list with its terms and definitions

Custom lists

You can create custom lists by using the CSS3 styles, and you can use the CSS3 counter and the :before and :after selectors. Consider the following list, which contains nested lists.

```
<ul class="level1">
    <li>Automobiles
        <ul class="level2">
            <li>BMW
                <ul class="level3">
                    <li>X1</li>
                    <li>X3</li>
                    <li>Z4</li>
                </ul>
            </li>
            <li>Chevrolet
                <ul class="level3">
                    <li>Cobalt</li>
                    <li>Impala</li>
                    <li>Volt</li>
                </ul>
            </li>
            <li>Ford
                <ul class="level3">
                    <li>Edge</li>
                    <li>Focus</li>
                    <li>Mustang</li>
                </ul>
            </li>
        </ul>
    </li>
```

```

</ul>
</li>
<li>Boats
    <ul class="level2">
        <li>Sea Ray</li>
        <li>Cobalt</li>
    </ul>
</li>
</ul>

```

Figure 5-9 shows the rendered list with the default styles. The bullet shapes change with each level of nesting, and each level of nesting is automatically indented.

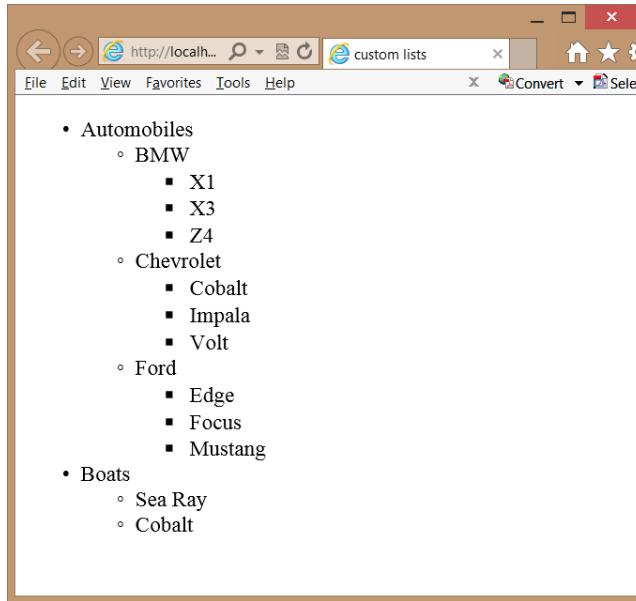


FIGURE 5-9 The rendered output with default styles

In addition to font-related styles and color-related styles, there are also list-related styles that you can alter to change the presentation of your list. In the rendered example, the first-level list-style-type CSS property is set to *disc*, which displays as a filled-in circle. The second-level list-style-type is set to *circle*, and the third-level *list-style-type* is set to *square*. In addition, each of the levels' list-style-position CSS property is set to *outside*, which means that when the text wraps, the first character of the next line will align with the first character of the previous line. If you set the list-style-position to *inside*, the first character of the next line will align with the bullet symbol of the first line.

In Visual Studio Express 2012 for Web, you can open the CSS file and enter the selector as follows.

```

li {
}

```

After the selector is added, you can right-click the style rule and choose Build Style to display a menu of styles to apply. Clicking List in the Category menu displays the styles that can be applied to a list. Figure 5-10 shows the Modify Style window.

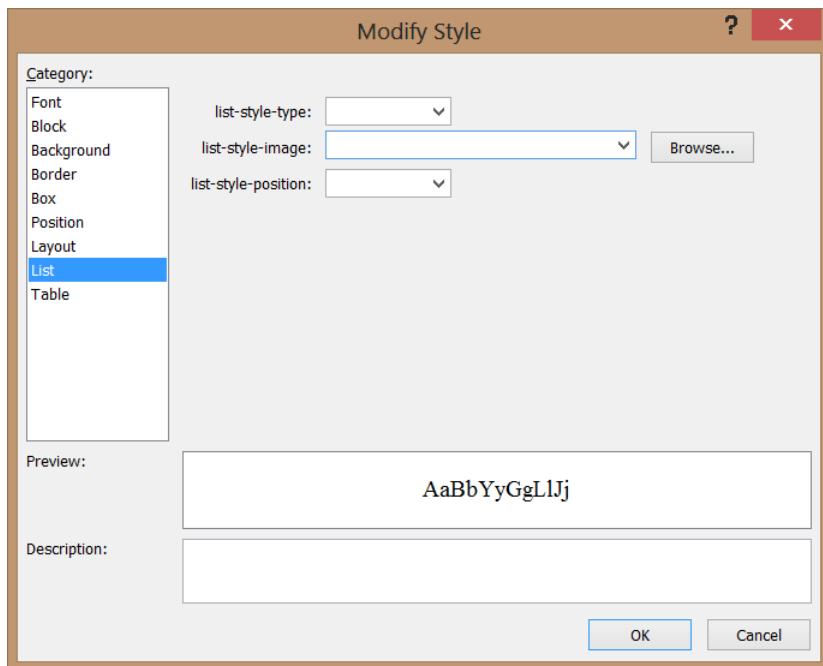


FIGURE 5-10 The Modify Style window showing menu-based style settings

By using the Modify Style window, you can easily override the default setting of the *list-style-type* and *list-style-position*. In addition, you can provide a *list-style-image* when the bullet symbols are not what you want. In this example, set the *list-item-style* to *none* and click OK. The style sheet now contains the modified style rule as follows.

```
li {  
    list-style-type: none;  
}
```

Try rendering the webpage and note that no bullets are displayed. Try many of the other settings to see how they render.

Instead of using the Modify Style window, you can type the style rules. When you're typing the rules, IntelliSense helps reduce the number of keystrokes. When the IntelliSense menu appears, you can select an item and press the tab key. In the CSS file, insert the following style rules.

```
body {  
    counter-reset: section;  
}
```

```

ul.level1 > li:before {
    counter-increment: section;
    content: "Section " counter(section) ". ";
    counter-reset: subsection;
}

ul.level2 > li:before {
    counter-increment: subsection;
    content: counter(section) "(" counter(subsection, lower-alpha) "- ";
}

ul.level1 > li, ul.level2 > li {
    list-style-type: none;
}

ul.level3 > li {
    list-style-type: disc;
}

```

The following is a description of each of the style rules in this example.

- The first style rule resets a user-defined section counter to one when the `<body>` element is styled. The section counter will be set to one only after the page is loaded, but it will be incremented in a different style rule.
- The second style rule is executed when a `</i>` element that is a child of a `` element with a CSS class of `level1` is rendered. It increments the section counter by one. It then inserts the content property before the `</i>` element, which outputs the “Section” string, followed by the value of the section counter and then followed by the “.” string. Finally, the rule resets a user-defined subsection counter to one. This style rule executes twice, before Automobiles and before Boats.
- The third style rule is executed when a `</i>` element that is a child of a `` element with a CSS class of `level2` is rendered. It increments the subsection counter by one. It then inserts the content property before the `</i>` element, which outputs the value of the section counter, followed by the “(” string and then followed by the value of the subsection counter, but this value is converted to lowercase alpha representation. After the subsection is rendered, the “) – ” string is rendered. This style rule executes five times.
- The fourth style rule sets the `list-style-type` to `none` for `level1` and `level2` list items.
- The fifth style rule sets the `list-style-type` to `disc` for `level3` list items.

The rendered output is shown in Figure 5-11. This should give you a good idea of the capabilities of HTML5 when working with lists.

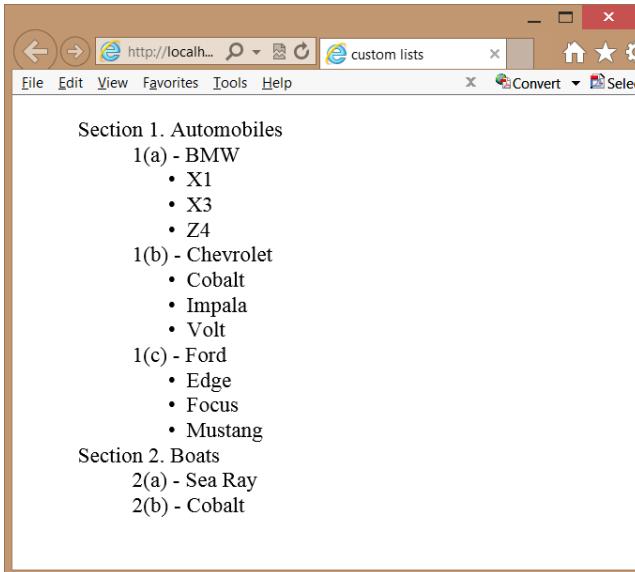


FIGURE 5-11 The rendered custom list

Lesson summary

- Semantic markup provides meaning to HTML elements to aid devices that consume HTML content.
- Nonvisual Desktop Access (NVDA) devices read and process webpages.
- Content that needs to be styled but doesn't clearly fit the meaning of any semantic elements can be styled by wrapping it with a `<div>` or `` element.
- The `<header>` element defines a section that provides a header. The `<footer>` element defines a section that provides a footer. The `<nav>` element defines a section that houses a block of major navigational links. The `<aside>` element defines a section of content that is separate from the content the `<aside>` element is in. The `<section>` element defines part of the whole and is typically named with an `<h1>` to `<h6>` internal element.
- The `<article>` element is a unit of content that can stand on its own and be copied to other locations. A blog post is a good example of an article.
- The Web Accessible Initiative (WAI) specifies the Accessible Rich Internet Applications (ARIA) suite, which is called WAI-ARIA. Use the WAI-ARIA role attribute to provide meaning to elements that are not semantically defined.
- In lieu of the `` element, use the `` element. In lieu of the `<i>` element, use the `` element.

Lesson review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. You are creating a webpage that will display short stories, and you want the stories to be shareable on other sites. Which element should each story be wrapped with?
 - A. `<section>`
 - B. `<pre>`
 - C. `<aside>`
 - D. `<article>`
2. You want to indicate an important item in your content text. Which element do you use?
 - A. ``
 - B. ``
 - C. ``
 - D. `<i>`
3. You want to identify the author of webpages on your website by providing the author name and email address in the footer of each page. What is the proper way to do this?
 - A. `<address>Author Name</address>`
 - B. `<contact>Author Name</contact>`
 - C. `<author>Author Name</author>`
 - D. `<name>Author Name</name>`

Lesson 2: Working with tables

Tables are the way to lay out data in your HTML document in rows and columns. A table displays a two-dimensional grid of data. Use the `<table>` element with the `<tr>` element to create table rows and the `<td>` element to create table details, which are better known as table cells. This lesson discusses tables in detail.

After this lesson, you will be able to:

- Create a basic table.
- Add a header and footer to a table.
- Create an irregular table.
- Access column data.
- Apply style rules to table elements.

Estimated lesson time: 30 minutes

Table misuse

HTML tables are powerful and, due to their flexibility, they are often misused. It's important to understand both proper table implementation and where it's inappropriate to implement a table.

Over the years, many developers have used the `<table>` element to create a page layout. Here are some reasons you should not use the `<table>` element to create a page layout.

- The table will not render until the `</table>` tag has been read. Webpages should be written with semantic markup, and the main `<div role="main">` element should be as close to the top of the HTML document as possible. The `<div>` element will render its content as the browser receives it. This enables the user to read the content as it's being loaded into the browser.
- Using a table forces you into a deeply nested HTML structure that is difficult to maintain.
- Using a table confuses accessibility devices.

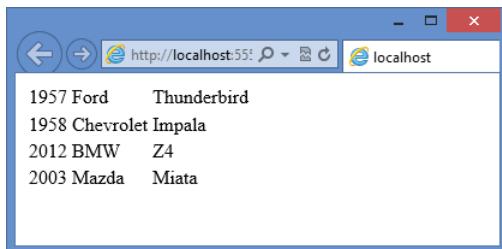
Remember that using a `<table>` element for anything other than tabular layout of data will be much more difficult to maintain than using `<div>` elements with positioning.

Creating a basic table

You can create a basic table by using the `<table>` element to denote the table. Inside the `<table>` element, you can add a `<tr>` element for each row that you require. Inside each `<tr>` element, add `<td>` elements for each cell that you need. The following is a simple table of vehicle information.

```
<table>
  <tr>
    <td>1957</td>
    <td>Ford</td>
    <td>Thunderbird</td>
  </tr>
  <tr>
    <td>1958</td>
    <td>Chevrolet</td>
    <td>Impala</td>
  </tr>
  <tr>
    <td>2012</td>
    <td>BMW</td>
    <td>Z4</td>
  </tr>
  <tr>
    <td>2003</td>
    <td>Mazda</td>
    <td>Miata</td>
  </tr>
</table>
```

Figure 5-12 shows the rendered output as four rows with three columns in each row. It's not obvious that there are columns in each row, however, and there is no header or footer. You might also want to see a border around all cells to make the table more obvious. This table needs improvement. Would alternating column colors improve it?



The screenshot shows a Microsoft Edge browser window with the URL <http://localhost:551>. The page displays a table with four rows. Each row contains three cells separated by vertical lines. The data is as follows:

1957	Ford	Thunderbird
1958	Chevrolet	Impala
2012	BMW	Z4
2003	Mazda	Miata

FIGURE 5-12 The rendered table with rows and columns

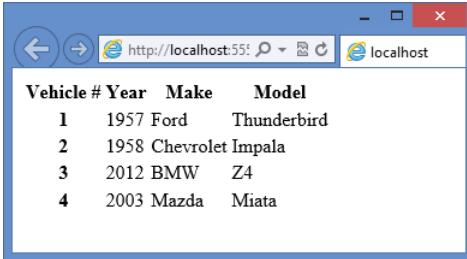
Adding header cells

Use the `<th>` element instead of the `<td>` element to display a header. The header can be horizontal or vertical. For example, you might want a header across the top to label each column and a header down the left side (in the first column) to label each row. The following is the modified table.

```
<table>
  <tr>
    <th>Vehicle #</th>
    <th>Year</th>
    <th>Make</th>
    <th>Model</th>
  </tr>
  <tr>
    <th>1</th>
    <td>1957</td>
    <td>Ford</td>
    <td>Thunderbird</td>
  </tr>
  <tr>
    <th>2</th>
    <td>1958</td>
    <td>Chevrolet</td>
    <td>Impala</td>
  </tr>
  <tr>
    <th>3</th>
    <td>2012</td>
    <td>BMW</td>
    <td>Z4</td>
  </tr>
  <tr>
    <th>4</th>
    <td>2003</td>
```

```
<td> Mazda</td>
<td>Miata</td>
</tr>
</table>
```

The rendered table is shown in Figure 5-13. This revised table now has horizontal and vertical headers. Notice that the default style of the *<th>* element is bold.



A screenshot of a web browser window titled "localhost". The address bar shows "http://localhost:5555". The main content area displays a table with four rows and three columns. The columns are labeled "Vehicle #", "Year", "Make", and "Model". The first row is a header row with bolded column titles. The subsequent three rows contain data: Row 2 (Vehicle # 1) has Ford Thunderbird, Row 3 (Vehicle # 2) has Chevrolet Impala, and Row 4 (Vehicle # 3) has BMW Z4. All cells in the table have a white background.

Vehicle #	Year	Make	Model
1	1957	Ford	Thunderbird
2	1958	Chevrolet	Impala
3	2012	BMW	Z4
4	2003	Mazda	Miata

FIGURE 5-13 The revised table with horizontal and vertical headers

Styling the table headers

Now that you have *<th>* elements for the headers, add a style to the *<th>* elements as follows.

```
th {  
    background-color: #BDEAFF;  
    width: 100px;  
}
```

This adds a pale blue background to all the *<th>* elements and sets the width of all columns to 100 pixels. What changes can you make to give the horizontal header and vertical header different styles? The following example can accomplish this task.

```
th {  
    background-color: #BDEAFF;  
    width: 100px;  
}  
  
th:only-of-type {  
    background-color: #FFFF99;  
}
```

The first style rule sets the color of all *<th>* elements to a pale blue and sets the width to 100 pixels. The second style rule has a higher priority, so it overrides the first style rule and applies a pale yellow color to the vertical header.

Declaring the header, footer, and table body

Most browsers automatically wrap all `<tr>` elements with a `<tbody>` element to indicate the body of the table. What would happen if you had a CSS style selector of `table > tr`? You wouldn't get a match because the browser adds the `<tbody>` element. The selector can be rewritten as `table > tbody > tr` instead, or maybe `tbody > tr` is all you need. It's good practice to define the `<tbody>` element explicitly in every table.

You might also have multiple rows that are to be used as horizontal headers or footers. You can use the `<thead>` element to identify rows that are header rows and use the `<tfoot>` element to identify rows that are footer rows. The following is an example of the addition of the `<thead>`, `<tfoot>`, and `<tbody>` elements.

```
<table>
  <thead>
    <tr>
      <th>Vehicle #</th>
      <th>Year</th>
      <th>Make</th>
      <th>Model</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>1</th>
      <td>1957</td>
      <td>Ford</td>
      <td>Thunderbird</td>
      <td>14,000</td>
    </tr>
    <tr>
      <th>2</th>
      <td>1958</td>
      <td>Chevrolet</td>
      <td>Impala</td>
      <td>3,000</td>
    </tr>
    <tr>
      <th>3</th>
      <td>2012</td>
      <td>BMW</td>
      <td>Z4</td>
      <td>40,000</td>
    </tr>
    <tr>
      <th>4</th>
      <td>2003</td>
      <td>Mazda</td>
      <td>Miata</td>
      <td>5,000</td>
    </tr>
  </tbody>
  <tfoot>
```

```

<tr>
    <th>Total:</th>
    <th></th>
    <th></th>
    <th></th>
    <th>62,000</th>
</tr>
</tfoot>
</table>

```

In addition to adding structure to the table, you can use the `<thead>`, `<tbody>`, and `<tfoot>` elements to control the styling of the `<th>` elements better. Without these elements, how would you provide a different style to the header and footer? The following style rules provide an example of such styling.

```

thead th {
    background-color: #BDEAFF;
    width: 100px;
}

tbody th {
    background-color: #FFFF99;
}

tfoot th {
    background-color: #C2FE9A;
}

tfoot th:last-of-type {
    text-align: right;
}

td {
    text-align: center;
}

td:last-of-type {
    text-align: right;
}

```

The rendered table is shown in Figure 5-14. The following is a description of the style rules applied.

- The first style rule applies a blue background color to the header and sets the width of all columns to 100 pixels.
- The second style rule applies a yellow background color to the vertical header.
- The third style rule applies a green background color to the footer.
- The fourth style rule applies right alignment to the price in the footer.
- The fifth style rule centers the text of all table cells.
- The last style rule applies right alignment to the price cells.

Vehicle #	Year	Make	Model	Price
1	1957	Ford	Thunderbird	14,000
2	1958	Chevrolet	Impala	3,000
3	2012	BMW	Z4	40,000
4	2003	Mazda	Miata	5,000
Total:				\$62,000

FIGURE 5-14 The styled table

Although you can have a maximum of one *<thead>* element and one *<tfoot>* element, you can have many *<tbody>* elements within a *<table>* element. The benefit of having multiple *<tbody>* elements is that you can group rows to apply styles. You can even display or hide groups of rows by setting the style display property to none (to hide) or by clearing the display property (to show). The following example extends the previous example by using multiple *<tbody>* elements, adding one for Antique Cars and one for Non-Antique Cars.

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Vehicles</title>
    <link href="Content/vehicles.css" rel="stylesheet" />
    <script src="Scripts/vehicles.js"></script>
</head>
<body>
    <div role="main">
        <button id="showAntique">Antique Cars</button>
        <button id="showNonAntique">Non-Antique Cars</button>
        <table>
            <thead>
                <tr>
                    <th>Vehicle #</th>
                    <th>Year</th>
                    <th>Make</th>
                    <th>Model</th>
                    <th>Price</th>
                </tr>
            </thead>
            <tbody id="antiqueCars">
                <tr>
                    <td>1</td>
                    <td>1957</td>
                    <td>Ford</td>
                    <td>Thunderbird</td>
                    <td>14,000</td>
                </tr>
                <tr>
                    <td>2</td>
                    <td>1958</td>

```

```

        <td>Chevrolet</td>
        <td>Impala</td>
        <td>3,000</td>
    </tr>
</tbody>
<tbody id="nonAntiqueCars">
    <tr>
        <th>3</th>
        <td>2012</td>
        <td>BMW</td>
        <td>Z4</td>
        <td>40,000</td>
    </tr>
    <tr>
        <th>4</th>
        <td>2003</td>
        <td>Madza</td>
        <td>Miata</td>
        <td>5,000</td>
    </tr>
</tbody>
<tfoot>
    <tr>
        <th>Total:</th>
        <th></th>
        <th></th>
        <th></th>
        <th>62,000</th>
    </tr>
</tfoot>
</table>
</div>
</body>
</html>

<script>
    init();
</script>
```

This example shows the complete HTML document, so you can see the inclusion of the CSS file and JavaScript file. The HTML has been extended to include two buttons at the top so you can filter by Antique Cars or Non-Antique Cars. There are two `<tbody>` elements, each having an explicit id of antiqueCars and nonAntiqueCars, respectively, and a `<script>` element at the bottom that initializes the JavaScript, which will attach event handlers to the click event of the buttons. The CSS file is slightly modified from the previous example as follows.

```

thead th {
    background-color: #BDEAFF;
    width: 100px;
}

tbody th {
    background-color: #FFFF99;
```

```

}

tfoot th {
    background-color: #C2FE9A;
}

tfoot th:last-of-type {
    text-align: right;
}

td {
    text-align: center;
}

td:last-of-type {
    text-align: right;
}

.hidden {
    display: none;
}

.visible {
    display: normal;
}

```

The CSS file now has the `.hidden` and `.visible` selectors. These are used to show or hide the `<tbody>` elements, including their contents. The JavaScript file contains the following code.

```

function init() {
    document.getElementById('showAntique').addEventListener('click', showAntiqueCars);
    document.getElementById('showNonAntique').addEventListener('click', showNonAntiqueCars);
}

function showAntiqueCars() {
    document.getElementById('antiqueCars').className = "visible";
    document.getElementById('nonAntiqueCars').className = "hidden";
}

function showNonAntiqueCars() {
    document.getElementById('antiqueCars').className = "hidden";
    document.getElementById('nonAntiqueCars').className = "visible";
}

```

The JavaScript code contains an `init` function that is called when the HTML document is loaded. The `init` function attaches event handlers to the `click` event of the two buttons. The additional functions set the CSS class to display or hide the `<tbody>` elements.

When the webpage is displayed, all vehicles are displayed. Clicking the Antique Cars button displays the antique cars and hides the non-antique cars. Clicking the Non-Antique Cars button displays the non-antique cars and hides the antique cars.

Creating irregular tables

Tables need to be rectangular to work properly, but you'll often need to present tables that don't contain the same number of cells in each row. In the case of the previous examples, the footer contained the same number of cells as the other rows, but you only need to have two cells, one for "Total:" and one for the total price. You might also want to add a column that indicates Antique Cars versus Non-Antique Cars, but you don't want a cell on every row that says "Antique Car" or "Non-Antique Car". You want to add a single cell that says "Antique Cars" and is the combined height of all Antique Car rows. You want to add a single cell that says "Non-Antique Cars" and is the combined height of all Non-Antique Car rows. Use the *rowspan* or *colspan* attributes on the *<td>* or *<th>* element to solve this problem.

The *colspan* attribute tells the browser that a *<td>* or *<th>* element should be the size of multiple horizontal cells. In the previous example, where you want the "Total:" text to span the footer row, use *<th colspan="4">* as follows.

```
<tfoot>
  <tr>
    <th colspan="4">Total:</th>
    <th>62,000</th>
  </tr>
</tfoot>
```

The default style for the *<th>* element is bold and centered. When "Total:" is displayed, it's centered within the four cells it spans. The CSS style rule is changed to right-align "Total:" as follows.

```
tfoot th {
  background-color: #C2FE9A;
}

tfoot th:first-of-type {
  text-align: right;
}

tfoot th:last-of-type {
  text-align: right;
}
```

You could just right-align all *<th>* elements in the footer by eliminating the last two style rules in this example and adding the *text-align* style to the first style rule. The rendered output is shown in Figure 5-15.

Antique Cars				
Vehicle #	Year	Make	Model	Price
1	1957	Ford	Thunderbird	14,000
2	1958	Chevrolet	Impala	3,000
3	2012	BMW	Z4	40,000
4	2003	Madza	Miata	5,000
Total:		62,000		

FIGURE 5-15 The rendered page with the footer containing only two cells

The rowspan attribute tells the browser that a `<td>` or `<th>` element should be the size of multiple vertical cells. In the previous example, when you want to add a column with only two cells, use `<td rowspan="n">` where *n* equals the number of rows to span, in this case, 2. Remember that adding a column also requires you to add the column to the header and to modify the colspan attribute in the footer. The following is the modified table.

```


| Vehicle #     | Category  | Year          | Make   | Model       | Price  |
|---------------|-----------|---------------|--------|-------------|--------|
| 1             | Antique   | 1957          | Ford   | Thunderbird | 14,000 |
| 2             | Chevrolet | 1958          | Impala |             | 3,000  |
| 3             | BMW       | 2012          | Z4     |             | 40,000 |
| 4             | Madza     | 2003          | Miata  |             | 5,000  |
| <b>Total:</b> |           | <b>62,000</b> |        |             |        |


```

```

        <td>2012</td>
        <td>BMW</td>
        <td>Z4</td>
        <td>40,000</td>
    </tr>
    <tr>
        <th>4</th>
        <td>2003</td>
        <td>Mazda</td>
        <td>Miata</td>
        <td>5,000</td>
    </tr>
</tbody>
<tfoot>
    <tr>
        <th colspan="5">Total:</th>
        <th>62,000</th>
    </tr>
</tfoot>
</table>

```

To help illustrate the rowspan and colspan attributes, a black border is added to the table cells. The following is the complete CSS file.

```

table {
    border: medium solid #000000;
}

thead th {
    background-color: #BDEAFF;
    width: 100px;
}

tbody th {
    background-color: #FFFF99;
}

tfoot th {
    background-color: #C2FE9A;
}

tfoot th:first-of-type {
    text-align: right;
}

tfoot th:last-of-type {
    text-align: right;
}

td {
    text-align: center;
    border: thin solid #000000;
}

td:last-of-type {

```

```

        text-align: right;
    }

th {
    border: thin solid #000000;
}

.hidden {
    display: none;
}

.visible {
    display: normal;
}

```

The results are displayed in Figure 5-16.

Vehicle #	Category	Year	Make	Model	Price
1	Antique	1957	Ford	Thunderbird	14,000
2		1958	Chevrolet	Impala	3,000
3	Non-Antique	2012	BMW	Z4	40,000
4		2003	Mazda	Miata	5,000
Total:					62,000

FIGURE 5-16 The rendered page with borders set, clearly showing the rowspan and colspan attributes

Adding a caption to a table

You can use the `<caption>` element to define and associate a caption with a table. The default style of the caption is centered and located above the table. You can use the CSS `text-align` and `caption-side` properties to override the default style. If you use the `<caption>` element, it must be the first element within the `<table>` element.

Styling columns

Styling columns is a common difficulty because tables are row-centric, not column-centric. It's relatively easy to apply a style to a row because you can apply a `<tr>` element to the style, but there isn't a `<tc>` element for a column. Remember that the `<td>` element represents a cell, not a column. Columns are actually created implicitly by creating the cells. Use the `<colgroup>` and `<col>` elements to style columns.

The `<colgroup>` element is placed inside the `<table>` element to define columns that can be styled. Remember that styling includes hiding and displaying the columns. Inside the

`<colgroup>` element, `<col>` elements are added for each column to be styled. The `<col>` element has a `span` attribute that identifies multiple columns that will have the same style.

In the previous examples, the `<colgroup>` and `<col>` elements can provide a style for the vertical headers, but this time, you want to apply a style to the first two columns. You can define the columns as follows.

```
<colgroup>
  <col span="2" class="verticalHeader" />
</colgroup>
```

This example defines the first two columns to have a style of `verticalHeader`. The `verticalHeader` class is set to apply a gray background color as follows.

```
.verticalHeader {
  background-color: #C0C0C0;
}
```

In addition, the existing style for the first column has been removed. Figure 5-17 shows the rendered webpage.

Antique Cars		Non-Antique Cars		
Vehicle #	Year	Make	Model	Price
1	1957	Ford	Thunderbird	14,000
2	1958	Chevrolet	Impala	3,000
3	2012	BMW	Z4	40,000
4	2003	Madza	Miata	5,000
Total:				62,000

FIGURE 5-17 Using the `<colgroup>` and `<col>` elements to apply a style to multiple columns

Lesson summary

- Refrain from using the `<table>` element for page layout.
- A `<tr>` element creates a table row. A `<td>` element creates a table cell in a table row.
- To identify a header cell, use the `<th>` element instead of using the `<td>` element.
- Use the `<thead>` element to specify table rows that comprise the table header. Use the `<tfoot>` element to specify table rows that comprise the table footer. Use the `<tbody>` element to specify data rows. You can group data rows by specifying many `<tbody>` elements.
- Use the `rowspan` and `colspan` attributes on the `<th>` and `<td>` elements to create irregular tables.

- Use the `<caption>` element directly after the `<table>` element to specify a caption for your table.
- Use the `<colgroup>` and `<col>` elements to apply styles to a column.

Lesson review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. You are creating a webpage that will be used to display a list of salespeople with their sales statistics for the years of 2010, 2011, and 2012 in two categories: sales of products and sales of services. You want to each of the years to be in the horizontal header, and under each year, you will have a “Products” column and a “Services” column. How will you define the element for year 2011?
 - A. `<th>2011</th>`
 - B. `<th colspan="2">2011</th>`
 - C. `<th span="2">2011</th>`
 - D. `<th style="2">2011</th>`
2. You want to provide the ability to display or show columns, but you don’t want to add a style or other marking to each `<td>` element. How can you accomplish this?
 - A. Add a `<colgroup>` element to the `<table>` element and define each column by using a `<col>` element inside the `<colgroup>` element.
 - B. Add an id to each `<td>` element and provide a unique id for each; use the ids in your style sheet rules to obtain the desired style.
 - C. Add a `<col>` element to the `<table>` element and define each column by using a `<id>` element inside the `<col>` element.
 - D. Add a `<hidden>` element to the `<table>` element and define each column by using a `<col>` element inside the `<hidden>` element.
3. Which element can you add to the `<table>` element to provide a table caption?
 - A. `<thead>`
 - B. `<colgroup>`
 - C. `<caption>`
 - D. `<th>`

Practice exercises

If you encounter a problem completing any of these exercises, the completed projects can be installed from the Practice Exercises folder that is provided with the companion content.

Exercise 1: Add a page layout to the calculator project

In this exercise, you apply your knowledge of semantic markup by adding a page layout to the WebCalculator project that you worked on in Chapter 4, “Getting started with CSS3,” and then you add style rules to improve the look of the webpage.

This exercise continues with the goal of adding style rules with a minimum of modifications to the default.html file.

1. Start Visual Studio Express 2012 for Web. Click File, choose Open Project, and then select the solution you created in Chapter 4.
2. Select the WebCalculator.sln file and click Open. You can also click File, choose Recent Projects And Solutions, and then select the solution.

If you didn’t complete the exercises in Chapter 4, you can use the solution in the Chapter 5 Exercise 1 Start folder.

3. In the Solution Explorer window, right-click the default.html file and choose Set As Start Page. Press F5 to verify that your home page is displayed.
4. Open the default.html page and wrap the `<div>` element whose id is calculator with a `<div>` element, and then set the id to container.

This `<div>` element will contain the complete page layout.

5. In the container `<div>` element, insert a `<header>` element containing an `<hgroup>` element with an id of headerText. In the `<hgroup>` element, insert an `<h1>` element containing the text, “Contoso, Ltd.” After the `<h1>` element, insert an `<h2>` element containing the text, “Your success equals our success.”

The header should look like the following.

```
<header>
  <hgroup id="headerText">
    <h1>Contoso Ltd.</h1>
    <h2>Your success equals our success</h2>
  </hgroup>
</header>
```

6. After the `<header>` element, insert a `<nav>` element.

The `<nav>` element typically contains the primary links on the page, but there are no other pages in this site.

7. Insert a dummy link to the home page, which is the current page.

This will display on the page to give you an idea of what the `<nav>` element is used for. The `<nav>` element should look like the following.

```
<nav>
  <a href="default.html">Home</a>
</nav>
```

8. After the `<nav>` element, wrap the calculator `<div>` element with a `<div>` element whose role is set to main.

The main `<div>` element with the calculator `<div>` element should look like the following.

```
<div role="main">
    <div id="calculator">
        <input id="txtResult" type="text" readonly="readonly" /><br />
        <input id="txtInput" type="text" /><br />
        <button id="btn7">7</button>
        <button id="btn8">8</button>
        <button id="btn9">9</button><br />
        <button id="btn4">4</button>
        <button id="btn5">5</button>
        <button id="btn6">6</button><br />
        <button id="btn1">1</button>
        <button id="btn2">2</button>
        <button id="btn3">3</button><br />
        <button id="btnClear">C</button>
        <button id="btn0">0</button>
        <button id="btnClearEntry">CE</button><br />
        <button id="btnPlus">+</button>
        <button id="btnMinus">-</button>
    </div>
</div>
```

9. After the main `<div>` element, insert an `<aside>` element, which will contain the advertisements. Because there are no advertisements, insert a `<p>` element with the word Advertisements so you can see where the `<aside>` element renders.

The completed `<aside>` element should look like the following.

```
<aside>
    <p>Advertisements</p>
</aside>
```

10. After the `<aside>` element, insert a `<footer>` element. In the `<footer>` element, add a `<p>` element with the following content: Copyright © 2012, Contoso Ltd., All rights reserved.

The completed `<footer>` element should look like the following.

```
<footer>
    <p>
        Copyright © 2012, Contoso Ltd., All rights reserved
    </p>
</footer>
```

The following is the complete default.html webpage.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Web Calculator</title>
    <link href="Content/default.css" rel="stylesheet" />
    <script type="text/javascript" src="Scripts/CalculatorLibrary.js"></script>
</head>
<body>
    <div id="container">
```

```

<header>
    <hgroup id="headerText">
        <h1>Contoso Ltd.</h1>
        <h2>Your success equals our success</h2>
    </hgroup>
</header>
<nav>
    <a href="default.html">Home</a>
</nav>
<div role="main">
    <div id="calculator">
        <input id="txtResult" type="text" readonly="readonly" /><br />
        <input id="txtInput" type="text" /><br />
        <button id="btn7">7</button>
        <button id="btn8">8</button>
        <button id="btn9">9</button><br />
        <button id="btn4">4</button>
        <button id="btn5">5</button>
        <button id="btn6">6</button><br />
        <button id="btn1">1</button>
        <button id="btn2">2</button>
        <button id="btn3">3</button><br />
        <button id="btnClear">C</button>
        <button id="btn0">0</button>
        <button id="btnClearEntry">CE</button><br />
        <button id="btnPlus">+</button>
        <button id="btnMinus">-</button>
    </div>
</div>
<aside>
    <p>Advertisements</p>
</aside>
<footer>
    <p>
        Copyright © 2012, Contoso Ltd., All rights reserved
    </p>
</footer>
</div>
<script type="text/javascript">
    window.addEventListener('load', initialize, false);
</script>
</body>
</html>

```

Exercise 2: Add styles to the calculator layout

Now that you have completed the layout container, add and modify styles in the default.css file.

1. Open the default.css file and, at the top of the file, add a style rule to set the margin and padding of all elements to 0 pixels.

Your style rule should look like the following.

```
* { margin : 0; padding : 0; }
```

- After that style rule, insert another style rule that sets the `<aside>`, `<footer>`, `<header>`, `<hgroup>`, and `<nav>` elements to display as a block to ensure that all browsers render these elements as blocks.

Your style rule should look like the following.

```
aside, footer, header, hgroup, nav {  
    display: block;  
}
```

The current page background color is a dark blue. This background color needs to be lightened, and the font-family needs to be set to Cambria with a backup font of Times New Roman and final fallback of serif font. The font color needs to be set to match the Contoso standard blue. Your body style rule should be modified to match the following.

```
body {  
    background-color: hsl(255, 95%, 95%);  
    font-family: Cambria, 'Times New Roman' , serif;  
    color: #0068AC;  
}
```

- To add a new folder called **Images** to the project, right-click the project in the Solution Explorer window. Click Add, choose New Folder, and name the folder **Images**.
- To add the image from the Resource folder, in the Solution Explorer window, right-click the **Images** folder that you just added.
- Click Add, choose Existing Item, and select the `ContosoLogo.png` file that is located in the Chapter05 Resources folder.
- After the `body` style rule, insert a `header` style rule that sets the height to 100 pixels and set the background image to the `ContosoLogo.png` file.
- Set the background-repeat to no-repeat and set the top margin to 10 pixels by adding the `header` selector with the curly braces, right-clicking in the style rule, and clicking Build Style. When the Modify Style window is displayed, set the properties.

The completed style rule should look like the following.

```
header {  
    height: 100px;  
    background-image: url('../Images/ContosoLogo.png');  
    background-repeat: no-repeat;  
    margin-top: 10px;  
}
```

- Add a style rule based on the element id equal to `headerText`. Set the position to absolute, set the top to 0 pixels, and set the left to 80 pixels, which will locate the header text to the right of the Contoso logo.
- Set the width to 100 percent and set the margin top to 10 pixels as follows.

```
#headerText {  
    position: absolute;
```

```
    top: 0px;
    left: 80px;
    width: 100%;
    margin-top: 10px;
}
```

10. After the headerText style rule, insert a text rule for the `<h1>` element. Add styles to set the font size to 64 pixels and set the line height to 55 pixels as follows.

```
h1 {
    font-size: 64px;
    line-height: 55px;
}
```

11. After the h1 style rule, insert a text rule for the `<h2>` element. Add styles to set the font size to 18 pixels, set the line height to 20 pixels, and set the font style to italic as follows.

```
h2 {
    font-size: 18px;
    line-height: 20px;
    font-style: italic;
}
```

12. After the h2 style rule, create a style rule for the `<nav>` element. Set the styles to float the `<nav>` element to the left, set the width to 20 percent, and set the minimum width to 125 pixels as follows.

```
nav {
    float: left;
    width: 20%;
    min-width: 125px;
}
```

13. After the nav style rule, add a style rule for the main `<div>` element. Set the styles to float to the left, beside the `<nav>` element, and set the width to 60 percent as follows.

```
div[role="main"] {
    float: left;
    width: 60%;
}
```

14. After the main div style rule, add a style rule for the `<aside>` element. Set the styles to float to the left, beside the main `<div>` element, set the width to 20 percent, and set the minimum width to 125 pixels as follows.

```
aside {
    float: left;
    width: 20%;
    min-width: 125px;
}
```

- 15.** After the aside style rule, add a style rule for the `<footer>` element. Set the styles to position the footer clear after the `<aside>` element, set the width to 100 percent, set the height to 70 pixels, and set the font size to small as follows.

```
footer {  
    clear: both;  
    width: 100%;  
    height: 70px;  
    font-size: small;  
}
```

- 16.** After the footer style rule, add a style rule for the `<div>` element whose id is container.

The purpose of this style is to ensure that the float: left styles you've added don't wrap when the browser window is resized to a small size.

- 17.** Set the minimum width to 800 pixels as follows.

```
#container {  
    min-width: 800px;  
}
```

- 18.** In the existing style rule for the calculator `<div>` element, change the height and width to 400 pixels as follows.

```
#calculator {  
    border: solid;  
    background-color: hsl(255, 100%, 60%);  
    width: 400px;  
    height: 400px;  
    margin-left: auto;  
    margin-right: auto;  
    text-align: center;  
    padding: 10px;  
}
```

- 19.** In the existing style rule for the input button, change the font size to 20 point as follows.

```
input, button {  
    font-family: Arial;  
    font-size: 20pt;  
    border-width: thick;  
    border-color: hsl(255, 100%, 100%);  
    margin: 5px;  
}
```

The following is the completed style sheet for your reference.

```
* { margin : 0; padding : 0; }  
  
aside, footer, header, hgroup, nav {  
    display: block;  
}
```

```
body {  
    background-color: hsl(255, 95%, 95%);  
    font-family: Cambria,'Times New Roman' , serif;  
    color: #0068AC;  
}  
  
header {  
    height: 100px;  
    background-image: url('../Images/ContosoLogo.png');  
    background-repeat: no-repeat;  
    margin-top: 10px;  
}  
  
#headerText {  
    position: absolute;  
    top: 0px;  
    left: 80px;  
    width: 100%;  
    margin-top: 10px;  
}  
  
h1 {  
    font-size: 64px;  
    line-height: 55px;  
}  
  
h2 {  
    font-size: 18px;  
    line-height: 20px;  
    font-style: italic;  
}  
  
nav {  
    float: left;  
    width: 20%;  
    min-width:125px;  
}  
  
div[role="main"] {  
    float: left;  
    width: 60%;  
}  
  
aside {  
    float: left;  
    width: 20%;  
    min-width:125px;  
}  
  
footer {  
    clear: both;  
    width: 100%;  
    height: 70px;  
    font-size: small;
```

```
}

#container {
    min-width: 800px;
}

#calculator {
    border: solid;
    background-color: hsl(255, 100%, 60%);
    width: 400px;
    height: 400px;
    margin-left: auto;
    margin-right: auto;
    text-align: center;
    padding: 10px;
}

input {
    width: 85%;
    height: 7%;
    text-align: right;
    padding: 10px;
    border: inset;
}

button {
    background-color: hsl(255, 50%, 80%);
    width: 25%;
    height: 10%;
    border: outset;
}

button:hover {
    background-color: hsl(255, 50%, 90%);
}

button:active {
    border: inset;
    border-width: thick;
    border-color: hsl(255, 100%, 100%);
    background-color: hsl(255, 50%, 50%);
}

input, button {
    font-family: Arial;
    font-size: 20pt;
    border-width: thick;
    border-color: hsl(255, 100%, 100%);
    margin: 5px;
}

[readonly] {
    background-color: hsl(255, 50%, 80%);
```

- 20.** To see your results, press F5 to start debugging the application.

You should see a nicer-looking calculator interface with a page layout as shown in Figure 5-18.

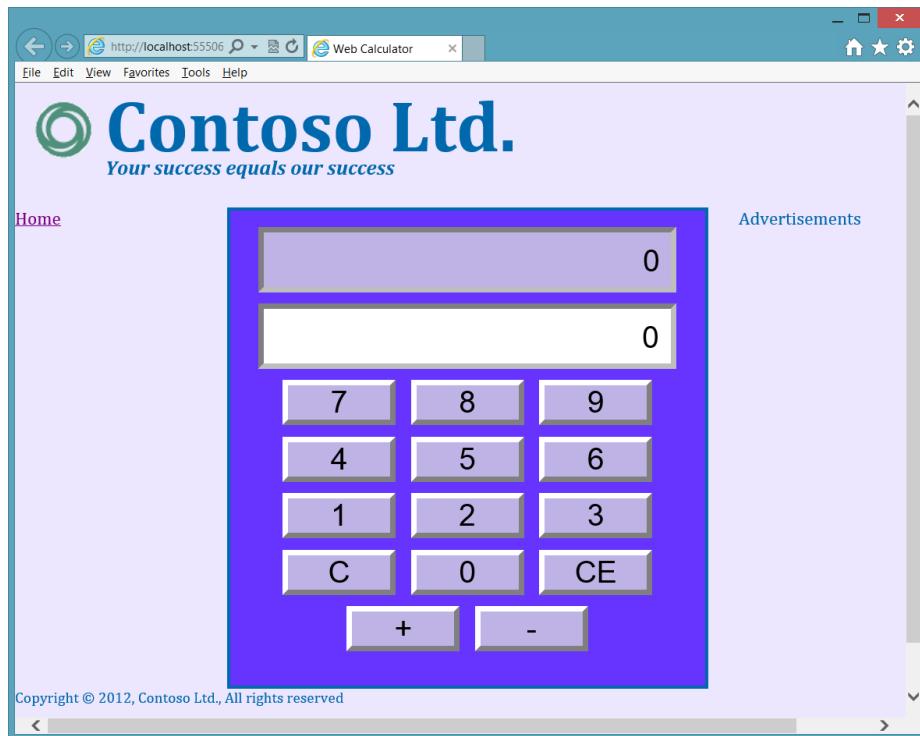


FIGURE 5-18 The web calculator with its page layout

Exercise 3: Cleaning up the web calculator

The calculator's buttons are positioned by keeping them the same size and using `
` elements for each line of buttons. Although the calculator doesn't look too bad, the buttons aren't in their traditional locations. For example, the clear and clear entry buttons are normally at the top, whereas the plus and minus buttons are typically on the right. The goal of this lesson is to reposition the buttons.

In this exercise, you continue with the project from Exercise 2 and modify the default.html file. The elements of the calculator will be positioned by placing them in a table. There will be seven rows and four columns.

1. Open the project from Exercise 2.

If you didn't perform Exercise 2, you can use the project located in the Exercise 2 Start folder.

2. Open the default.html file.

3. Surround the inputs and buttons with a `<table>` element.
4. Remove all `
` elements from the default.html file.
5. Surround the txtResult text box with a table cell that spans four columns. Surround the table cell with a table row.

The table row should look like the following.

```
<tr>
    <td colspan="4">
        <input id="txtResult" type="text" readonly="readonly" />
    </td>
</tr>
```

6. With the txtInput text box, repeat the previous step as follows.

```
<tr>
    <td colspan="4">
        <input id="txtInput" type="text" />
    </td>
</tr>
```

The next table row will have two empty columns, for future buttons, and then a column for the clear entry button and another column for the clear button as follows.

```
<tr>
    <td></td>
    <td></td>
    <td><button id="btnClearEntry">CE</button></td>
    <td><button id="btnClear">C</button></td>
</tr>
```

The next table row will have buttons 7, 8, 9, and the plus button as follows.

```
<tr>
    <td>
        <button id="btn7">7</button></td>
    <td>
        <button id="btn8">8</button></td>
    <td>
        <button id="btn9">9</button></td>
    <td>
        <button id="btnPlus">+</button>
    </td>
</tr>
```

The next table row will have buttons 4, 5, 6, and the minus button as follows.

```
<tr>
    <td>
        <button id="btn4">4</button>
    </td>
    <td>
        <button id="btn5">5</button>
    </td>
    <td>
```

```

        <button id="btn6">6</button>
    </td>
    <td>
        <button id="btnMinus">-</button>
    </td>
</tr>

```

The next table row will have buttons 1, 2, and 3 and an empty column as follows.

```

<tr>
    <td>
        <button id="btn1">1</button>
    </td>
    <td>
        <button id="btn2">2</button>
    </td>
    <td>
        <button id="btn3">3</button>
    </td>
    <td>
    </td>
</tr>

```

The last table row will have an empty column, the 0 button, and two more empty columns as follows.

```

<tr>
    <td></td>
    <td>
        <button id="btn0">0</button>
    </td>
    <td></td>
    <td></td>
</tr>

```

The following is the completed main `<div>` element.

```

<div role="main">
    <div id="calculator">
        <table>
            <tr>
                <td colspan="4">
                    <input id="txtResult" type="text" readonly="readonly" />
                </td>
            </tr>
            <tr>
                <td colspan="4">
                    <input id="txtInput" type="text" />
                </td>
            </tr>
            <tr>
                <td></td>
                <td></td>
                <td>
                    <button id="btnClearEntry">CE</button>
                </td>
            </tr>

```

```

        <td>
            <button id="btnClear">C</button>
        </td>
    </tr>
    <tr>
        <td>
            <button id="btn7">7</button></td>
        <td>
            <button id="btn8">8</button></td>
        <td>
            <button id="btn9">9</button></td>
        <td>
            <button id="btnPlus">+</button>
        </td>
    </tr>
    <tr>
        <td>
            <button id="btn4">4</button>
        </td>
        <td>
            <button id="btn5">5</button>
        </td>
        <td>
            <button id="btn6">6</button>
        </td>
        <td>
            <button id="btnMinus">-</button>
        </td>
    </tr>
    <tr>
        <td>
            <button id="btn1">1</button>
        </td>
        <td>
            <button id="btn2">2</button>
        </td>
        <td>
            <button id="btn3">3</button>
        </td>
        <td></td>
    </tr>
    <tr>
        <td></td>
        <td>
            <button id="btn0">0</button>
        </td>
        <td></td>
        <td></td>
    </tr>
</table>
</div>
</div>

```

- Now that the default.html file is completed, modify the style sheet by opening the default.css file and, at the bottom, adding the table selector and setting the width to 100 percent as follows.

```
table {  
    width: 100%;  
}
```

- Add a td selector and set the width to 25 percent as follows.

```
td {  
    width: 25%;  
}
```

- Locate the existing button selector. Change the width to 90 percent as follows.

```
button {  
    background-color: hsl(255, 50%, 80%);  
    width: 90%;  
    height: 10%;  
    border: outset;  
}
```

- Locate the existing input selector. Change the padding to 5 pixels as follows.

```
input {  
    width: 85%;  
    height: 7%;  
    text-align: right;  
    padding: 5px;  
    border: inset;  
}
```

- Press F5 to run the application.

Figure 5-19 shows the completed calculator.



FIGURE 5-19 The completed calculator

Suggested practice exercises

The following additional exercises are designed to give you more opportunities to practice what you've learned and to help you successfully master the lessons presented in this chapter.

- **Exercise 1** Learn more about semantic markup by adding additional sections to your webpage.
- **Exercise 2** Learn more about tables by adding more rows and cells to the table to hold future buttons.

Answers

This section contains the answers to the lesson review questions in this chapter.

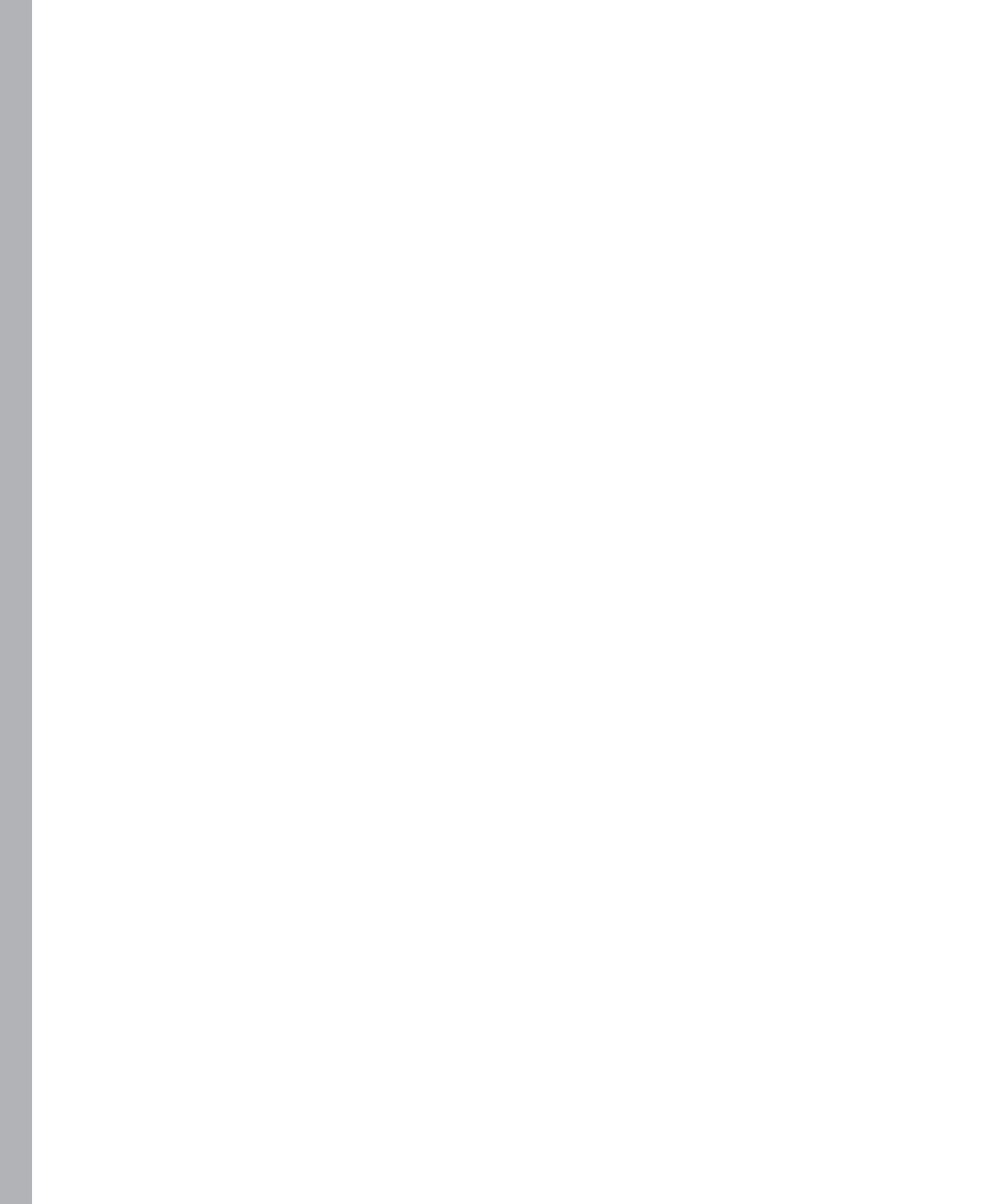
Lesson 1

1. **Correct answer: D**
 - A. **Incorrect:** The `<section>` element denotes a part of something.
 - B. **Incorrect:** The `<pre>` element displays preformatted content.
 - C. **Incorrect:** The `<aside>` element displays content that is related to the site.
 - D. **Correct:** An article wraps stand-alone items that can be shared.
2. **Correct answer: C**
 - A. **Incorrect:** You should refrain from using the `` element.
 - B. **Incorrect:** The `` element indicates emphatic stress but not necessarily importance.
 - C. **Correct:** The `` element indicates importance.
 - D. **Incorrect:** You should refrain from using the `<i>` element.
3. **Correct answer: A**
 - A. **Correct:** The `<address>` element provides contact information for the author of the webpage.
 - B. **Incorrect:** The `<contact>` element is not valid.
 - C. **Incorrect:** The `<author>` element is not valid.
 - D. **Incorrect:** The `<name>` element is not valid.

Lesson 2

1. **Correct answer: B**
 - A. **Incorrect:** The column needs to span two columns.
 - B. **Correct:** The column needs the `colspan="2"` attribute to span the Products and Services columns.
 - C. **Incorrect:** The span attribute is used with the `<col>` element but not with the `<th>` element.
 - D. **Incorrect:** The style attribute cannot be used to cause spanning across two columns.
2. **Correct answer: A**
 - A. **Correct:** You can assign styles to the `<col>` element, which will apply the style to the corresponding table column.
 - B. **Incorrect:** Adding an id to each `<td>` element does not satisfy the criteria.

- C. Incorrect: The `<col>` element must be inside a `<colgroup>` element.
 - D. Incorrect: The `<hidden>` element is not valid.
3. Correct answer: C
- A. Incorrect: The `<thead>` element specifies heading rows.
 - B. Incorrect: The `<colgroup>` element specifies columns.
 - C. Correct: The `<caption>` element adds a caption to the top of a table.
 - D. Incorrect: The `<th>` element specifies header cells.



CHAPTER 6

Essential JavaScript and jQuery

The flexibility of JavaScript is amazing. In the previous chapters, you learned how to add JavaScript code to your webpage to provide dynamic changes to the page when an event is triggered.

One of the biggest difficulties with webpage development is the differences among different browsers, but this book is primarily focused on HTML5, CSS3, and JavaScript (ECMAScript5.1). A completely separate book could be written that deals just with the differences among browsers and browser versions.

In this chapter, you learn how to create objects, which are an important aspect of JavaScript. You use objects to create entities, which are passed to and from the server, and to encapsulate functionality that you want to modularize. You also need to extend objects that others have created.

This chapter also introduces jQuery, the answer to writing browser-compatible code. Although jQuery doesn't solve all browser-compatibility issues, it does solve most of the day-to-day issues that you encounter among browsers. In addition, jQuery is fun and easy to use.

Lessons in this chapter:

- Lesson 1: Creating JavaScript objects **262**
- Lesson 2: Working with jQuery **285**

Before you begin

To complete this book, you must have some understanding of web development. This chapter requires the hardware and software listed in the "System requirements" section in the book's Introduction.

Lesson 1: Creating JavaScript objects

In JavaScript, everything is an object. Strings, numbers, and functions are all objects. You have learned how to create functions, so you already have exposure to creating objects, as you see in this lesson.

After this lesson, you will be able to:

- Understand basic object-oriented terminology.
- Create JavaScript objects.

Estimated lesson time: 20 minutes

Using object-oriented terminology

 In many object-oriented languages, when you want to create objects, you start by creating a *class*, which is a blueprint for an object. Like a blueprint for a house, the blueprint isn't the house; it's the instructions that define the *type* of object that you will be constructing, which is the house. By using a house blueprint, you can create, or *construct*, many houses that are based on the blueprint. Each house is an *object* of type house, also known as an *instance* of the house type.

The developer writes the class, which is then used to construct objects. In a baseball application, you might create a Player (classes are normally capitalized) class that has properties for first and last name, batting average, error count, and so on. When you create your team, you might use the Player class to create nine Player objects, each having its own properties. Each time you construct a Player object, memory is allocated to hold the data for the player, and each piece of data is a property, which has a name and a value.

 The three pillars of object-oriented programming are *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation means that you hide all details except those that are required to communicate with your object in order to simplify the object for anyone using the object. Inheritance means that you can create an "is a" relationship between two classes, in which the child class automatically inherits everything that is in the parent class. Polymorphism means that you can execute a function on the parent class, but the behavior changes (morphs) because your child class has a function that overrides the function in the parent class.

 The *parent class* is also known as the *base class*, the *super class*, or the *generalized class*. The *child class* is also known as the *derived class*, the *subclass*, or the *specialized class*. Because it's easy to think of actual children inheriting from parents, the terms parent and child are usually used, but you should remember the other terms for these classes to communicate effectively with others about object-oriented programming.

 In object-oriented programming, objects can have data implemented as properties and behaviors implemented as methods. A *property* is essentially a variable that is defined on

an object and owned by the object. A *method* is a function that is defined on an object and owned by the object.

Understanding the JavaScript object-oriented caveat

JavaScript is a very flexible language. You can create objects, but the relationship between the JavaScript language and class-based, object-oriented programming is not direct. The most glaring example is that there is no *class* keyword in JavaScript. If you're familiar with class-based, object-oriented programming, you'll be struggling to find the "class."

JavaScript is a prototype-based, object-oriented programming language. In JavaScript, everything is an object, and you either create a new object from nothing, or you create an object from a clone of an existing object, known as a *prototype*.

Conceptually, you can simulate class creation by using a function. Class-based, object-oriented purists dislike the idea of a function being used to simulate a class. Keep an open mind as patterns are presented. This lesson should give you what you need to accomplish your tasks.

The problem you typically encounter is finding one correct solution for all scenarios. As you read on, you'll find that achieving proper encapsulation of private data requires you to create copies of the functions that can access the private data for each object instance, which consumes memory. If you don't want to create copies of the method for each object instance, the data needs to be publicly exposed, thus losing the benefits of encapsulation, by which you hide object details that users shouldn't need to see.

The general consensus of this issue of encapsulation versus wasteful memory consumption is that most people would rather expose the data to minimize memory consumption. Try to understand the benefits and drawbacks of each pattern when deciding which option to implement in your scenario.

Using the JavaScript object literal pattern

Probably the simplest way to create an object in JavaScript is to use the object literal syntax. This starts with a set of curly braces to indicate an object. Inside the curly braces is a comma-separated list of name/value pairs to define each property. Object literals create an object from nothing, so these objects contain precisely what you assign to them and nothing more. No prototype object is associated with the created object. The following example demonstrates the creation of two objects that represent vehicles.

```
var car1 = {
    year: 2000,
    make: 'Ford',
    model: 'Fusion',
    getInfo: function () {
        return 'Vehicle: ' + this.year + ' ' + this.make + ' ' + this.model;
    }
};
```

```

var car2 = {
  year: 2010,
  make: 'BMW',
  model: 'Z4',
  getInfo: function () {
    return 'Vehicle: ' + this.year + ' ' + this.make + ' ' + this.model;
  }
};

```

In this example, public properties are created for *year*, *make*, *model*, and *getInfo*. The *getInfo* property doesn't contain data; it references an anonymous function instead, so *getInfo* is a method. The method uses the *this* keyword to access the data. Remember that the *this* keyword references the object that owns the code where the *this* keyword is. In this case, the object is being created. If the *this* keyword were omitted, the code would look in the global namespace for *year*, *make*, and *model*.

To test this code, the following QUnit test checks to see whether each object contains the data that is expected.

```

test("Object Literal Test", function () {
  expect(2);
  var expected = 'Vehicle: 2000 Ford Fusion';
  var actual = car1.getInfo();
  equal(actual, expected, 'Expected value: ' + expected +
    ' Actual value: ' + actual);
  var expected = 'Vehicle: 2010 BMW Z4';
  var actual = car2.getInfo();
  equal(actual, expected, 'Expected value: ' + expected +
    ' Actual value: ' + actual);
});

```

This test performs an assertion by using the *car1* variable and then performs another assertion by using the *car2* variable. The successful test is shown in Figure 6-1.

If you want to define an array of items and assign it to a property, you can use square brackets as shown in the following example.

```

var car1 = {
  year: 2000,
  make: 'Ford',
  model: 'Fusion',
  repairs: ['repair1', 'repair2', 'repair3'],
  getInfo: function () {
    return 'Vehicle: ' + this.year + ' ' + this.make + ' ' + this.model;
  }
};

```

Because this is one of the easiest ways to create an object, you'll probably use it to gather data to send to other code. In this example, two instances of a type Object are created, and properties are dynamically added to each instance. This does not create a Vehicle type.

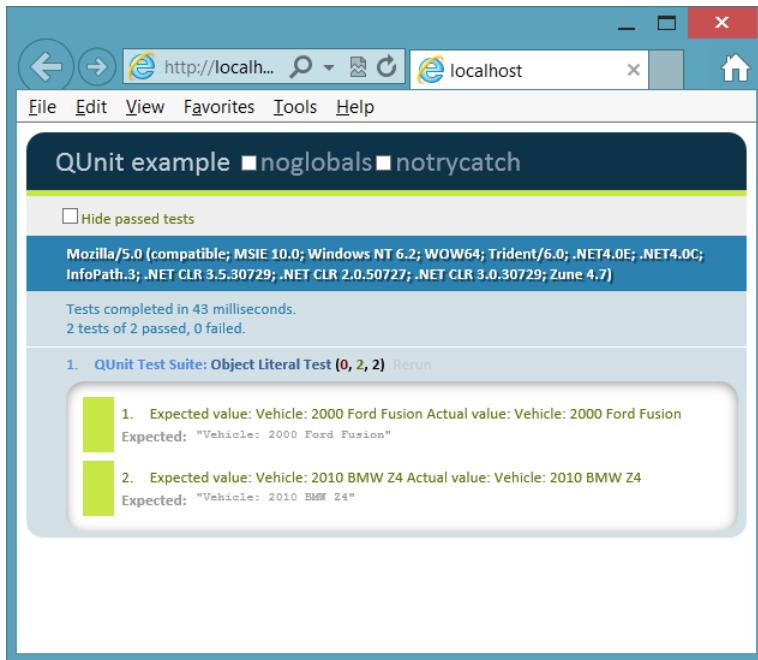


FIGURE 6-1 The JavaScript object literal test

Creating dynamic objects by using the factory pattern

In addition to using the JavaScript literal object syntax, JavaScript has an `Object` type, and you can use it to create an object programmatically. `Object` has a prototype object that is cloned when you use the `new` keyword to create a new `Object` instance. The prototype object has the following inherited methods.

- **constructor** The function that is called to initialize a new object
- **hasOwnProperty** Returns a Boolean indicator of whether the current object has the specified property
- **isPrototypeOf** Returns a Boolean indicator of whether the current object is in the specified object's prototype object chain
- **propertyIsEnumerable** Returns true if the object can be enumerated in a `for...in` loop
- **toLocaleString** Converts a date to a string value based on the current local
- **toString** Returns the string representation of the current object
- **valueOf** Returns the value of the current object converted to its most meaningful primitive value

After the object is created, you can dynamically add properties to it that hold the data and reference functions. You can wrap this code in a function that returns the object as shown in the following code example.

```
function getVehicle(theYear, theMake, theModel) {  
    var vehicle = new Object();  
    vehicle.year = theYear;  
    vehicle.make = theMake;  
    vehicle.model = theModel;  
    vehicle.getInfo = function () {  
        return 'Vehicle: ' + this.year + ' ' + this.make + ' ' + this.model;  
    };  
    return vehicle;  
}
```

This code takes advantage of JavaScript's dynamic nature to add *year*, *make*, *model*, and *getInfo* to the object and then returns the object. Placing this code in a function makes it easy to call the *getVehicle* function to get a new object. The encapsulation of the code to create an object is commonly referred to as using the *factory pattern*. Can you create multiple instances of vehicle? You can create multiple instances of Object and add properties dynamically to each instance, but the actual type is Object, not vehicle. The following QUnit test demonstrates the creation of multiple instances.

```
test("Create Instances Test Using Factory Pattern", function () {  
    expect(2);  
    var car1 = getVehicle(2000, 'Ford', 'Fusion');  
    var car2 = getVehicle(2010, 'BMW', 'Z4');  
    var expected = 'Vehicle: 2000 Ford Fusion';  
    var actual = car1.getInfo();  
    equal(actual, expected, 'Expected value: ' + expected +  
        ' Actual value: ' + actual);  
    var expected = 'Vehicle: 2010 BMW Z4';  
    var actual = car2.getInfo();  
    equal(actual, expected, 'Expected value: ' + expected +  
        ' Actual value: ' + actual);  
});
```

This might be all you need when you are gathering some data to put into an object structure and pass to some other code or service. Although the *getVehicle* function encapsulates the object creation, the properties are all public. This can be desirable in some scenarios, but if you want the data to be private, this approach won't work. Like when using the literal object syntax, you might encounter the problem that every vehicle's type is Object, and you might want to create a Vehicle class to have a named Vehicle type.

Creating a class

There is no *class* keyword in JavaScript, but you can simulate a class by starting with a function, which is actually the *constructor function* of the object. Consider the following function.

```
function Vehicle(theYear, theMake, theModel) {  
    year = theYear;
```

```

make = theMake;
model = theModel;
 getInfo = function () {
    return 'Vehicle: ' + year + ' ' + make + ' ' + model;
};
}

```

There are several problems with this code. All the variables are defined without the `var` keyword, so `year`, `make`, `model`, and `getInfo` are automatically defined in the global scope and are accessible from anywhere. The following is a passing QUnit test that initializes `Vehicle` and calls the `getInfo` method to retrieve the data.

```

test("Function Test", function () {
    expect(2);
    Vehicle(2000, 'Ford', 'Fusion');
    var expected = 'Vehicle: 2000 Ford Fusion';
    var actual = getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    expected = 2000;
    actual = year;
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});

```

The `Vehicle` function accepts three parameters and doesn't return anything. Instead, it is setting global variables, and there is no provision for multiple instances. To prove that global variables are being set, the second assertion is checking to see whether there is a global variable named `year` that equals 2,000. This assertion succeeds, which proves that the data is not encapsulated, and there is only one copy of the data. For example, the following QUnit test fails.

```

test("Failing Function Test", function () {
    expect(1);
    Vehicle(2000, 'Ford', 'Fusion');
Vehicle(2010, 'BMW', 'Z4');
    var expected = 'Vehicle: 2000 Ford Fusion';
    var actual = getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    expected = 2000;
    actual = year;
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});

```

Figure 6-2 shows the failures. The problem is that `year`, `make`, and `model` of the second vehicle replaced `year`, `make`, and `model` of the first vehicle. The variable `getInfo` was also replaced, but instead of holding data, it holds a reference to the function code. The `getInfo` variable's value was replaced with new function code; it just happened to be the same code. Once again, there is no encapsulation.

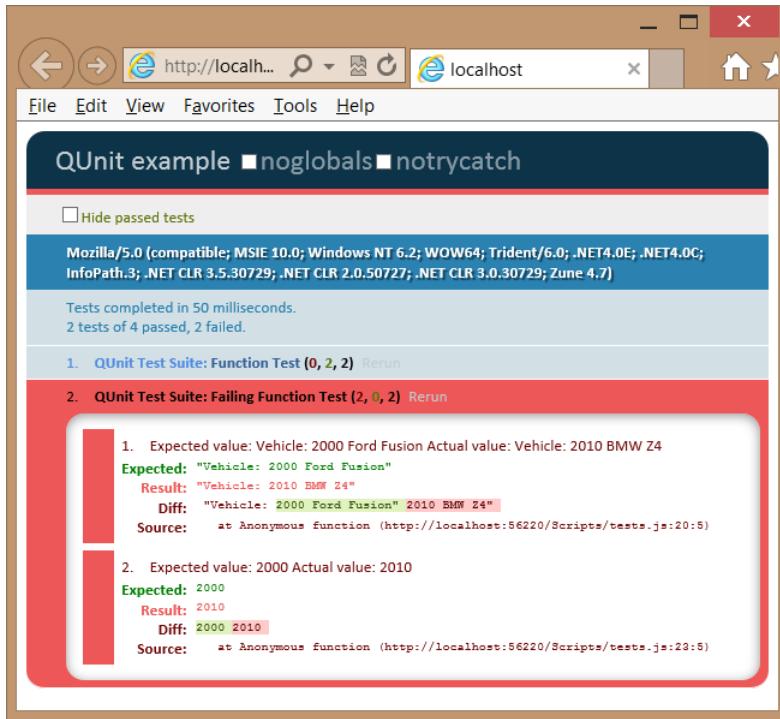


FIGURE 6-2 The failing test assertions after a second vehicle is used

To solve the problem, you want to implement encapsulation. Then you need to create objects, each with its own data. To implement encapsulation, use the `var` keyword for the `year`, `make`, and `model`. This will make these variables private to the function. Notice that the `var` keyword is not used with `getInfo` because the `getInfo` variable needs to be public to be called from outside the object, but you don't want the `getInfo` variable to be global. Assign `getInfo` to the current object by using the `this` keyword. The result is a class that encapsulates the data and exposes `getInfo` to retrieve the data in a controlled way as follows.

```
function Vehicle(theYear, theMake, theModel) {  
    var year = theYear;  
    var make = theMake;  
    var model = theModel;  
    this.getInfo = function () {  
        return 'Vehicle: ' + year + ' ' + make + ' ' + model;  
    };  
}
```

IMPORTANT PRIVATE DATA ISN'T SECURE

In object-oriented programming, private data is not intended to be secure. Private data provides encapsulation so the details can be hidden; the user sees only what is necessary and isn't bogged down in the details.

Remember that the *this* keyword references the object that owns the current code. The way the test is currently written, the *this* keyword references the global object, and *getInfo* will still be a global variable. To solve the problem, the *new* keyword must be used to create an object from this class, as shown in the modified test code.

```
test("Encapsulation Test", function () {
  expect(2);
  var car1 = new Vehicle(2000, 'Ford', 'Fusion');
  var car2 = new Vehicle(2010, 'BMW', 'Z4');
  var expected = 'Vehicle: 2000 Ford Fusion';
  var actual = car1.getInfo();
  equal(actual, expected, 'Expected value: ' + expected +
    ' Actual value: ' + actual);
  expected = 2000;
  actual = year;
  equal(actual, expected, 'Expected value: ' + expected +
    ' Actual value: ' + actual);
});
```

Notice that a new variable is defined, *car1*, and it is assigned the object that is created by using the *new* keyword. After that, another new variable is defined, *car2*, and it is assigned the second *Vehicle* object created by using the *new* keyword. Two instances of the *Vehicle* class are being created, which means that two *Vehicle* objects are being constructed. Each instance has its own data and its own copy of the *getInfo* method. The *getInfo* method is public but has access to the private data. A method that is public but has access to private data is called a *privileged method*.

Figure 6-3 shows the test results. Notice that the first assertion passed, which proves that there are separate object instances, each having its own data. The second assertion failed. The failure message states that the *year* is undefined, which proves that the *year* is not directly accessible from the test, which is in the global namespace. Instead, *year*, in addition to *make* and *model*, is encapsulated in the object.

You have now created a class and constructed objects from the class, but there's more to cover in the *Vehicle* function that is being used as a class. The *Vehicle* function is known as a *constructor function*. The *new* keyword created an object and executed the constructor function to initialize the object by creating the *year*, *make*, and *model* private variables and the public *getInfo* variable. Each instance has these four variables, and memory is allocated for them. That's what you want for the data, but is that what you want for the *getInfo* variable that references a function? The answer is that it depends on what you are trying to accomplish with your code.



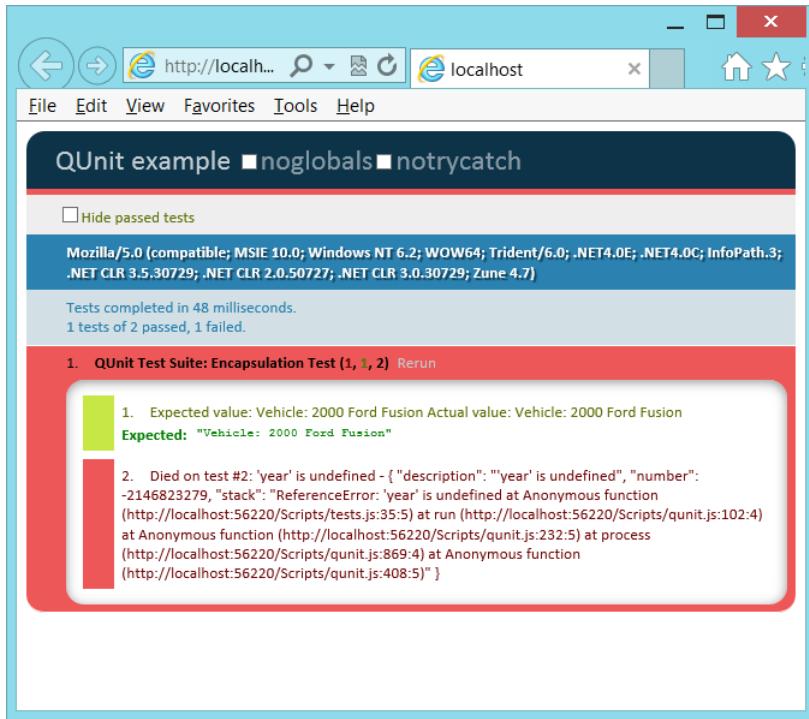


FIGURE 6-3 Successful first assertion and failed second assertion

Consider the following test code that creates two Vehicle objects, but then replaces the code in *getInfo* of the first Vehicle object with different code. Does this replace the code in the second Vehicle object?

```
test("Function Replacement Test", function () {
    expect(2);
    var car1 = new Vehicle(2000, 'Ford', 'Fusion');
    var car2 = new Vehicle(2010, 'BMW', 'Z4');
    car1.getInfo = function () {
        return 'This is a Car';
    };
    var expected = 'This is a Car';
    var actual = car1.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var expected = 'This is a Car';
    var actual = car2.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});
```

The test result is shown in Figure 6-4. The first assertion succeeded, which proves that the function was successfully replaced on the first Vehicle object. The second assertion failed, which proves that the second Vehicle object's *getInfo* function was not replaced. Is that what

you expected? Is that what you wanted? You can see that in some scenarios, this behavior is desirable, but in other scenarios, you might have wanted to replace the function across all objects. To do this, you use the *prototype* pattern.

NOTE ACCESS TO PRIVATE DATA

In the example, the replacement function cannot access the private data because the replacement is executed externally to the Vehicle.

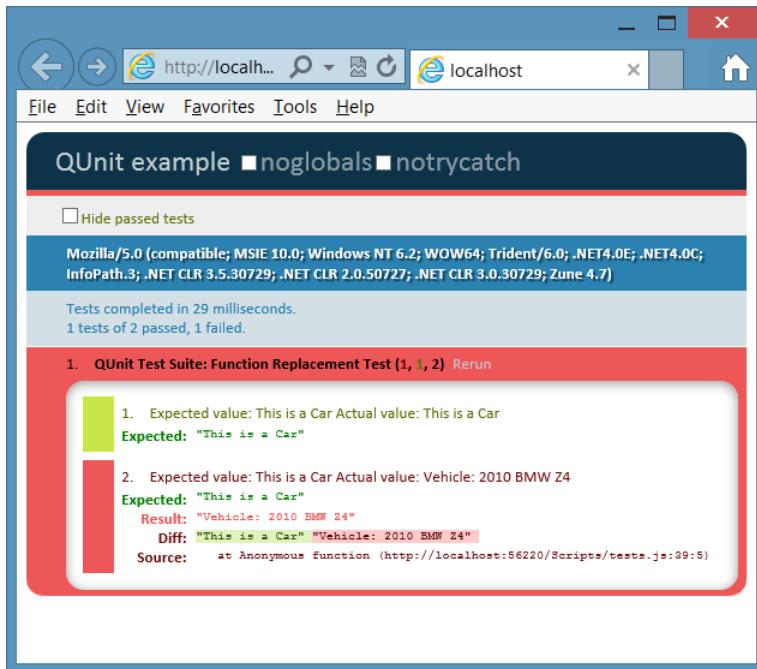


FIGURE 6-4 Successful first assertion, proving that the function was replaced; failed second assertion, proving that the second Vehicle's function was not replaced

Using the prototype property

In JavaScript, everything, including the function, is an Object type, which has a *prototype* property. The prototype itself is an object containing properties and methods that should be available to all instances of the type you're working with. However, this prototype is typically specified externally to the constructor function, so the prototype doesn't have access to private variables. Therefore, you must expose the data for the prototype to work. The following is an example of using the *prototype* property to create a single *getInfo* method that is shared across all instances.

```
function Vehicle(theYear, theMake, theModel) {  
    this.year = theYear;
```

```

        this.make = theMake;
        this.model = theModel;
    }
Vehicle.prototype.getInfo = function () {
    return 'Vehicle: ' + this.year + ' ' + this.make + ' ' + this.model;
}

```

By using this class and the prototype, you can write the following test to ensure that each instance has its own data and that the getInfo function works properly.

```

test("Instance Test Using Prototype", function () {
    expect(2);
    var car1 = new Vehicle(2000, 'Ford', 'Fusion');
    var car2 = new Vehicle(2010, 'BMW', 'Z4');
    var expected = 'Vehicle: 2000 Ford Fusion';
    var actual = car1.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
          ' Actual value: ' + actual);
    var expected = 'Vehicle: 2010 BMW Z4';
    var actual = car2.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
          ' Actual value: ' + actual);
});

```

In this test, two instances of the Vehicle class are created, each having different data. The first assertion calls getInfo on *car1* and verifies that the proper result is returned. The second assertion calls getInfo on *car2* and verifies that the proper result is returned. The result is shown in Figure 6-5.

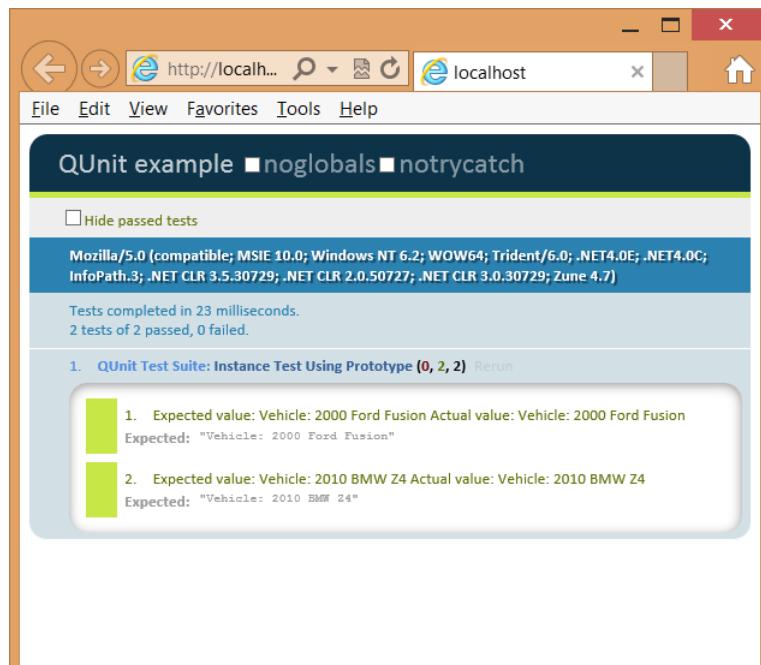


FIGURE 6-5 The modified class using the prototype property to create the getInfo function

Now that you have a functioning class, change the prototype to see whether it can be changed across all instances.

```
test("Instance Test Using Prototype Replace Function", function () {
    expect(2);
    var car1 = new Vehicle(2000, 'Ford', 'Fusion');
    var car2 = new Vehicle(2010, 'BMW', 'Z4');
    Vehicle.prototype.getInfo = function () {
        return 'Car: ' + this.year + ' ' + this.make + ' ' + this.model;
    }
    var expected = 'Car: 2000 Ford Fusion';
    var actual = car1.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var expected = 'Car: 2010 BMW Z4';
    var actual = car2.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});
```

This test creates two Vehicle instances and then changes getInfo. Next, the two assertions are modified to check both instances to see whether they are using the updated getInfo. The result is shown in Figure 6-6.

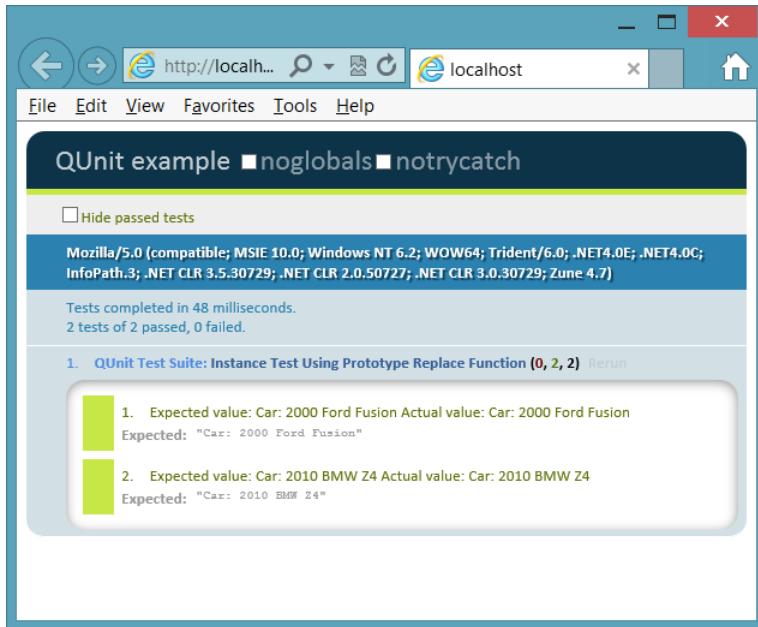


FIGURE 6-6 The modification of the getInfo prototype affected all instances

You might use the prototype property when creating functions that will be shared across all instances, but remember that the prototype is defined externally to the constructor function, so all properties must be public when using the *this* keyword. If you don't need to

replace individual instance functions and you don't mind making your data public, the prototype is efficient.

Quick check

- You want to add a method to all instances of Vehicle. How do you do this?

Quick check answer

- Add the method by using the Vehicle object's prototype method.

Debating the prototype/private compromise

You've learned the primary patterns for creating a JavaScript object, but there can be a compromise in which you can have private data that is readable by creating a method for retrieving the data, also known as a *getter*, which has no *setter*, a method for setting the value. This would require you to write a function that is copied for each object, but you should keep the function as small as possible, as shown in the following code example.

```
function Vehicle(theYear, theMake, theModel) {  
    var year = theYear;  
    var make = theMake;  
    var model = theModel;  
    this.getYear = function () { return year; };  
    this.getMake = function () { return make; };  
    this.getModel = function () { return model; };  
}  
Vehicle.prototype.getInfo = function () {  
    return 'Vehicle: ' + this.getYear() +  
        ' ' + this.getMake() +  
        ' ' + this.getModel();  
}
```

The QUnit test for this code creates two instances of Vehicle and, for each assertion, executes the getInfo method of each object and checks for the proper value. The test is as follows.

```
test("Instance Test Using Prototype and getters", function () {  
    expect(2);  
    var car1 = new Vehicle(2000, 'Ford', 'Fusion');  
    var car2 = new Vehicle(2010, 'BMW', 'Z4');  
    var expected = 'Vehicle: 2000 Ford Fusion';  
    var actual = car1.getInfo();  
    equal(actual, expected, 'Expected value: ' + expected +  
        ' Actual value: ' + actual);  
    var expected = 'Vehicle: 2010 BMW Z4';  
    var actual = car2.getInfo();  
    equal(actual, expected, 'Expected value: ' + expected +  
        ' Actual value: ' + actual);  
});
```

This test is successful, so replace the getInfo method and add more tests. The following test code does this.

```
test("Instance Test Using Prototype and getters", function () {
    expect(4);
    var car1 = new Vehicle(2000, 'Ford', 'Fusion');
    var car2 = new Vehicle(2010, 'BMW', 'Z4');
    var expected = 'Vehicle: 2000 Ford Fusion';
    var actual = car1.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var expected = 'Vehicle: 2010 BMW Z4';
    var actual = car2.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    Vehicle.prototype.getInfo = function () {
        return 'Car Year: ' + this.getYear()
            + ' Make: ' + this.getMake()
            + ' Model: ' + this.getModel();
    };
    var expected = 'Car Year: 2000 Make: Ford Model: Fusion';
    var actual = car1.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var expected = 'Car Year: 2010 Make: BMW Model: Z4';
    var actual = car2.getInfo();
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});
```

The test result is shown in Figure 6-7. You can replace the getInfo method and, because the data is exposed as read-only, it's available to be used in the new method. In addition, the privileged getters are small, which minimizes the amount of memory consumed when each instance has a copy of the method. Remember to create only getter methods as needed and to keep them small and concise.



Quick check

- How can you expose private data as read-only?

Quick check answer

- Add a getter method that retrieves the data but cannot change the data.

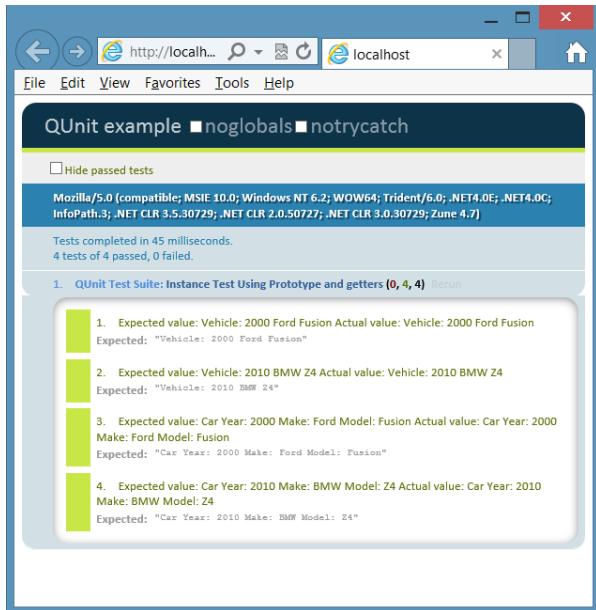


FIGURE 6-7 The use of getters to expose read-only data as a good compromise

Implementing namespaces

One problem to watch for is the pollution of the global namespace. As your program gets larger and libraries are added, more entries are added to the global object. How can you minimize this global namespace pollution?

JavaScript doesn't have a namespace keyword, but you can implement the equivalent of a namespace by using techniques that are similar to those used to create objects. Consider the following code sample.

```
var vehicleCount = 5;

var vehicles = new Array();

function Car() { }
function Truck() { }

var repair = {
    description: 'changed spark plugs',
    cost: 100
};
```

This code sample places five entries in the global namespace, and as the application grows, this global namespace pollution also grows. You can implement the namespace pattern to solve the problem. The following example shows the creation of an object that contains the five items from the previous example.

```

var myApp = {};

myApp.vehicleCount = 5;

myApp.vehicles = new Array();

myApp.Car = function () { }
myApp.Truck = function () { }

myApp.repair = {
    description: 'changed spark plugs',
    cost: 100
};

```

In this sample, `myApp` is the only entry in the global namespace. It represents the name of the application and its root namespace. Notice that object literal syntax is used to create an empty object and assign it to `myApp`. Everything else is added to the object. Sub-namespaces can also be created and assigned to `myApp`.

You can see that a namespace was created by creating an object. Although only one entry is made in the global namespace, all the members of `myApp` are globally accessible. In addition, if you create a namespace for your application, and your application has many JavaScript files, you might want to have logic to create the namespace object only if it hasn't been created. In the following example, the code for `myApp` is modified to create the namespace object if it doesn't already exist. This code uses the OR operator to create a new object if `myApp` does not have a value.

```
var myApp = myApp || {};
```

You can use the object techniques already defined in this lesson to make some members of the namespace private and some public. The difference is that the namespace is a singleton object, so you create a single instance for the namespace. You don't need to worry about functions defined in the constructor function consuming additional memory for each instance because there is only one instance. Here is an example of the use of an *immediately invoked function expression* (IIFE) to create the `myApp` namespace in which `Car` and `Truck` are public, but `vehicleCount`, `vehicles`, and `repair` are private.

```

(function () {
    this.myApp = this.myApp || {};
    var ns = this.myApp;

    var vehicleCount = 5;
    var vehicles = new Array();

    ns.Car = function () { }
    ns.Truck = function () { }

    var repair = {
        description: 'changed spark plugs',
        cost: 100
    };
}());

```



An *IIFE* (pronounced *iffy*) is an anonymous function expression that has a set of parentheses at the end of it, which indicates that you want to execute the function. The anonymous function expression is wrapped in parentheses to tell the JavaScript interpreter that the function isn't only being defined; it's also being executed when the file is loaded.

In this IIFE, the first line creates the `myApp` namespace if it doesn't already exist, which represents the singleton object that is used as the namespace. Next, an `ns` variable (for namespace) is created as an alias to the namespace to save typing within the IIFE, so `ns` can be used in place of `this.myApp`. After that, the private members of the namespace are defined by using the `var` keyword. `Car` and `Truck` are public, so they are prefixed with `ns`.

If you're wondering how you would create a sub-namespace under `myApp`, the following example shows how you can add a `billing` namespace under the `myApp` namespace.

```
(function () {
    this.myApp = this.myApp || {};
    var rootNs = this.myApp;
    rootNs.billing = rootNs.billing || {};
    var ns = rootNs.billing;

    var taxRate = .05;
    ns.Invoice = function () {};
})();
```

This example also implements an IIFE to create the namespace. First, the `myApp` namespace is created if it doesn't already exist and is assigned to a local `rootNs` variable to save typing inside the namespace. Next, the `billing` namespace is created and assigned to the local `ns` variable to save typing inside the namespace. Finally, the private `taxRate` property is defined while the public `Invoice` is defined.

Implementing inheritance

JavaScript provides the ability to implement inheritance, which is useful when you can define the relationship between two objects as an "is a" relationship. For example, an apple is a fruit, an employee is a person, and a piano is an instrument. You look for "is a" relationships because they provide an opportunity to implement code reuse. If you have several types of vehicles, you can create `Vehicle` with the common vehicle traits defined in it. After `Vehicle` is created, you can create each vehicle type and inherit from `Vehicle` so you don't need duplicate code in each vehicle type.

As an example of inheritance, start by defining the base class. Using the `Vehicle` example, the following is an example of a `Vehicle` base class.

```
var Vehicle = (function () {
    function Vehicle(year, make, model) {
        this.year = year;
        this.make = make;
        this.model = model;
    }
});
```

```

Vehicle.prototype.getInfo = function () {
    return this.year + ' ' + this.make + ' ' + this.model;
};

Vehicle.prototype.startEngine = function () {
    return 'Vroom';
};

return Vehicle;
})();

```

This class is wrapped in an IIFE. The wrapper encapsulates the function and the `Vehicle` prototype. There is no attempt to make the data private. The code works as follows.

- When the code is loaded into the browser, the IIFE is immediately invoked.
- A nested function called `Vehicle` is defined in the IIFE.
- The `Vehicle` function's prototype defines `getInfo` and `startEngine` functions that are on every instance of `Vehicle`.
- A reference to the `Vehicle` function is returned, which is assigned to the `Vehicle` variable.

This is a great way to create a class, and all future class examples use this pattern. To create `Vehicle` objects, you use the `new` keyword with the `Vehicle` variable. The following test creates an instance of `Vehicle` and tests the `getInfo` and `startEngine` methods.

```

test('Vehicle Inheritance Test', function () {
    expect(2);
    var v = new Vehicle(2012, 'Toyota', 'Rav4');
    var actual = v.getInfo();
    var expected = '2012 Toyota Rav4';
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var actual = v.startEngine();
    var expected = 'Vroom';
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});

```

Now that you have a `Vehicle` parent class with three properties and two methods, you can create child classes for `Car` and `Boat` that inherit from `Vehicle`. Start by writing an IIFE but, this time, pass `Vehicle` into the IIFE as follows.

```

var Car = (function (parent) {
    })(Vehicle);

```

Because `Vehicle` in this example is the `Vehicle` variable, not the `Vehicle` function, `Car` needs to be defined after `Vehicle`. `Vehicle` is passed into the IIFE and is available inside the IIFE as `parent`. Next, the function for `Car` can be added inside the IIFE. Inside the function, add any additional properties, such as `wheelQuantity`, and initialize to four. In the function, call the parent class's constructor for `Car` to allocate memory slots for the `year`, `make`, and `model`. To call the parent constructor function, use a `call` method that exists on the `Function` object,

which accepts a parameter for the *this* object, and parameters for the parameters on the function being called, as follows.

```
var Car = (function (parent) {
    function Car(year, make, model) {
        parent.call(this, year, make, model);
        this.wheelQuantity = 4;
    }
    return Car;
})(Vehicle);
```

Notice how this example used the *call* method to modify the *this* object; the *this* object is the Car object, so the call to the parent constructor function creates year, make, and model on the Car object. The Function object has another method, *apply*, that does the same thing, but the extra parameters are passed as an array instead of as a comma-delimited list.

Next, the inheritance must be set up. You might think that you've already set up inheritance because the previous example calls the parent class's constructor, and the year, make, and model are created on Car, but *getInfo* and *startEngine* were not inherited. The inheritance is accomplished by changing the Car prototype object to be a new Vehicle object. Remember that the prototype is the object that is cloned to create the new object. By default, the prototype is of type Object. After the new Vehicle is assigned to the prototype, the constructor of that Vehicle is changed to be the Car constructor as follows.

```
var Car = (function (parent) {
    Car.prototype = new Vehicle();
    Car.prototype.constructor = Car;
    function Car(year, make, model) {
        parent.call(this, year, make, model);
        this.wheelQuantity = 4;
    }
    return Car;
})(Vehicle);
```

Finally, you can add more methods into Car. In this example, the *getInfo* method is added, which replaces the Vehicle *getInfo* method. The new *getInfo* gets some code reuse by calling the existing *getInfo* method on the parent Vehicle object's prototype. However, you must use the *call* method and pass the *this* object as follows.

```
var Car = (function (parent) {
    Car.prototype = new Vehicle();
    Car.prototype.constructor = Car;
    function Car(year, make, model) {
        parent.call(this, year, make, model);
        this.wheelQuantity = 4;
    }
    Car.prototype.getInfo = function () {
        return 'Vehicle Type: Car ' + parent.prototype.getInfo.call(this);
    };
    return Car;
})(Vehicle);
```

This completes Car, and Boat is similar except that Boat has a propellerBladeQuantity, which is initialized to three, instead of the wheelQuantity property. In addition, getInfo returns the vehicle type of Boat and calls the Vehicle getInfo method as follows.

```
var Boat = (function (parent) {
    Boat.prototype = new Vehicle();
    Boat.prototype.constructor = Boat;
    function Boat(year, make, model) {
        parent.call(this, year, make, model);
        this.propellerBladeQuantity = 3;
    }
    Boat.prototype.getInfo = function () {
        return 'Vehicle Type: Boat ' + parent.prototype.getInfo.call(this);
    };
    return Boat;
})(Vehicle);
```

In addition to the Vehicle tests already presented, you need to verify the following for the child classes.

- Car and Boat have the inherited year, make, and model properties.
- Car has its wheelQuantity property and it's set.
- Boat has its propellerBladeQuantity and it's set.
- Car and Boat return the proper value from the replaced getInfo method.
- Car and Boat return the proper value from the inherited startEngine method.

The following are the Car and Boat tests.

```
test('Car Inheritance Test', function () {
    expect(6);
    var c = new Car(2012, 'Toyota', 'Rav4');
    var actual = c.year;
    var expected = 2012;
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var actual = c.make;
    var expected = 'Toyota';
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var actual = c.model;
    var expected = 'Rav4';
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var actual = c.wheelQuantity;
    var expected = 4;
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
    var actual = c.getInfo();
    var expected = 'Vehicle Type: Car 2012 Toyota Rav4';
    equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
```

```
var actual = c.startEngine();
var expected = 'Vroom';
equal(actual, expected, 'Expected value: ' + expected +
      ' Actual value: ' + actual);
});

test('Boat Inheritance Test', function () {
  expect(6);
  var b = new Boat(1994, 'Sea Ray', 'Signature 200');
  var actual = b.year;
  var expected = 1994;
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
  var actual = b.make;
  var expected = 'Sea Ray';
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
  var actual = b.model;
  var expected = 'Signature 200';
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
  var actual = b.propellerBladeQuantity;
  var expected = 3;
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
  var actual = b.getInfo();
  var expected = 'Vehicle Type: Boat 1994 Sea Ray Signature 200';
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
  var actual = b.startEngine();
  var expected = 'Vroom';
  equal(actual, expected, 'Expected value: ' + expected +
        ' Actual value: ' + actual);
});
```

Figure 6-8 shows the test output. All tests have passed.



FIGURE 6-8 The passing inheritance tests

Lesson summary

- A class is a blueprint for an object in which an object is an instance of a class.
- The three pillars of object-oriented programming are encapsulation, inheritance, and polymorphism.
- The class from which you inherit is called the parent, base, super, or generalized class. The class that is derived from the parent is called the child, derived, sub, or specialized class. You can implement inheritance by replacing the Child class prototype with a new instance of the parent and replacing its constructor with the Child class constructor function.

- JavaScript is a prototype-based, object-oriented programming language. A prototype is the object used to create a new instance.
- The literal pattern can be used to create an object by using curly braces to create the object. The factory pattern can be used to create a dynamic object.
- JavaScript does not have a class keyword, but you can simulate a class by defining a function.
- Creating private members is possible but usually involves creating privileged getter methods that can be memory consuming.
- The *new* keyword constructs an object instance.
- The *function* is an object. The function that simulates a class is called the *constructor function*.
- Namespaces can be created by using an immediately invoked function expression (IIFE).

Lesson review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. What is the blueprint for an object called?
 - A. property
 - B. method
 - C. class
 - D. event
2. What does JavaScript use as a starting object when constructing a new object?
 - A. prototype
 - B. property
 - C. class
 - D. event
3. How is inheritance supported in JavaScript?
 - A. You replace the prototype of the child object with a new instance of the parent object and then replace the prototype constructor with the child constructor.
 - B. You call the `createChild` method on the parent object.
 - C. You call the `setParent` method on the child object.
 - D. JavaScript does not support inheritance.

Lesson 2: Working with jQuery

This lesson introduces jQuery, which is very well documented at <http://jquery.com>. Subsequent chapters take advantage of jQuery whenever possible to minimize typing and benefit from jQuery's cross browser–compatible helper functions.

After this lesson, you will be able to:

- Explain the benefits of using jQuery.
- Use jQuery to select DOM elements.
- Use jQuery to modify the DOM.
- Use jQuery to set styles.

Estimated lesson time: 30 minutes

Introducing jQuery

jQuery is a library of helper functions that are cross browser–compatible. If you feel comfortable working with JavaScript, you might think that you don't need jQuery, but you do. You can minimize the amount of browser-specific code you must write by using jQuery, an open-source add-in that provides an easy, browser-agnostic means for writing JavaScript.

jQuery is written in JavaScript, so it is JavaScript. You can read the jQuery source code to understand how jQuery works. Probably millions of developers use jQuery. It's easy to use, it's stable, it's fully documented, and it works well with other frameworks. The following is a list of the categories of functionality jQuery provides.

- **Ajax** Methods that provide synchronous and asynchronous calls to the server
- **Attributes** Methods that get and set attributes of document object model (DOM) elements
- **Callbacks object** An object that provides many methods for managing callbacks
- **Core** Methods that provide core jQuery functionality
- **CSS** Methods that get and set CSS-related properties
- **Data** Methods that assist with associating arbitrary data with DOM elements
- **Deferred object** A chainable object that can register multiple callbacks into callback queues and relay the success or failure state of any synchronous or asynchronous functions
- **Dimensions** Helper methods for retrieving and setting DOM element dimensions
- **Effects** Animation techniques that can be added to your webpage
- **Events** Methods that provide the ability to register code to execute when the user interacts with the browser

- **Forms** Methods that provide functionality when working with form controls
- **Offset** Methods for positioning DOM elements
- **Selectors** Methods that provide the ability to access DOM elements by using CSS selectors
- **Traversing** Methods that provide the ability to traverse the DOM
- **Utilities** Utility methods

This lesson only scratches the surface of jQuery's capabilities, but subsequent lessons use jQuery whenever possible.

Getting started with jQuery

To get started with jQuery, add the jQuery library to your project. In this example, the QUnit testing framework has already been added to an empty web project, and it will demonstrate jQuery capabilities. You can add jQuery by either downloading the library from <http://jQuery.com> or adding the library from NuGet. To add it from NuGet, open your project and, in the Project menu, click Manage NuGet Packages. In the Search Online text box, type **jQuery** and press Enter. You should see a screen that is similar to that shown in Figure 6-9.

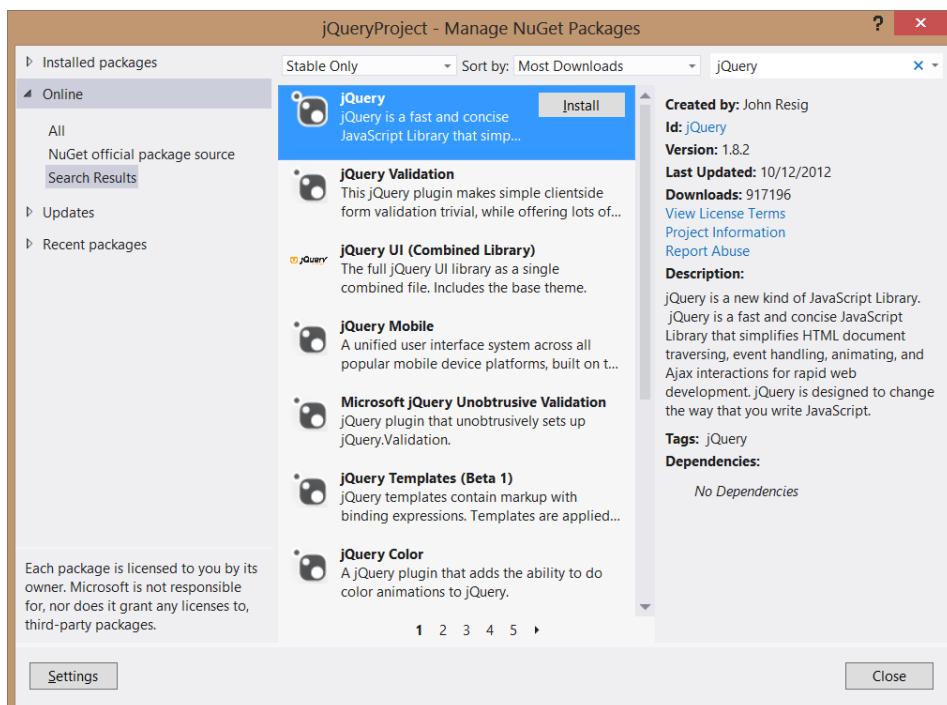


FIGURE 6-9 The NuGet package manager

After locating jQuery, click the Install button. The installation will start and, in a moment, you'll see a green check box on jQuery, indicating that the installation has completed successfully. Click the Close button and look at the Solution Explorer window, as shown in Figure 6-10. If your project didn't have a Scripts folder, a Scripts folder was added. Inside the Scripts folder, you'll find the latest release of jQuery. There is a file for IntelliSense and a complete jQuery library file. Finally, there is a minimized version of jQuery, which is the file you use at production time to minimize bandwidth usage.

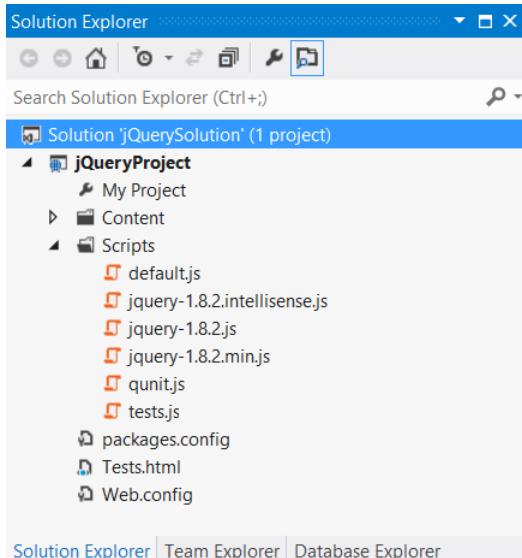


FIGURE 6-10 The completed installation of jQuery

Using jQuery

You're probably still trying to understand what jQuery is and how you benefit from using it, so the first feature to learn is how to use jQuery to locate an element or a group of elements. First, the jQuery library must be referenced on the page on which you will be using it. In this first example, the basic QUnit Test.html file is used, and the jQuery library is added so that the file contains the following HTML.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <link rel="stylesheet" type="text/css" href="Content/qunit.css" />
    <script type="text/javascript" src="Scripts/qunit.js"></script>
    <script src="Scripts/jquery-1.8.2.js"></script>
    <script type="text/javascript" src="Scripts/default.js"></script>
    <script type="text/javascript" src="Scripts/tests.js"></script>
</head>
<body>
```

```

<h1 id="qunit-header">QUnit example</h1>
<h2 id="qunit-banner"></h2>
<div id="qunit-testrunner-toolbar"></div>
<h2 id="qunit-userAgent"></h2>
<ol id="qunit-tests"></ol>
<div id="qunit-fixture">
    test markup, will be hidden
    <input id="txtInput" type="text" /><br />
    <input id="txtResult" type="text" /><br />
</div>
</body>
</html>

```

In the Solution Explorer window, the Test.html file has been set as the startup page by right-clicking the file and choosing Set As Start Page.

In the default.js file, the following code sets a reference to the *txtInput* and *txtResult* text boxes and then calls the clear function to initialize the two text boxes to '0'.

```

var txtInput;
var txtResult;

function initialize() {
    txtInput = document.getElementById('txtInput');
    txtResult = document.getElementById('txtResult');
    clear();
}

function clear() {
    txtInput.value = '0';
    txtResult.value = '0';
}

```

The tests.js file contains a simple test of the initialize method. When the test is run, the two assertions pass. The following is the tests.js file contents.

```

module('QUnit Test Suite', { setup: function () { initialize(); } });

test("Initialize Test", function () {
    expect(2);
    var expected = '0';
    equal(txtInput.value, expected, 'Expected value: ' + expected +
        ' Actual value: ' + txtInput.value);
    equal(txtResult.value, expected, 'Expected value: ' + expected +
        ' Actual value: ' + txtResult.value);
});

```

Now that the test is passing, change some code to use jQuery. The jQuery library code is in the jQuery namespace, but this namespace also has an alias of \$ (dollar sign) and can be used as follows.

```

jQuery.someFeature
$.someFeature

```

You can use either of these names to access the library features, so in the interest of minimizing keystrokes, use the dollar sign. First, change the code inside the initialize function of the default.js file. The code to locate elements can be rewritten to use jQuery and CSS selectors as follows.

```
function initialize() {
    txtInput = $('#txtInput');
    txtResult = $('#txtResult');
    clear();
}
```

This code uses the CSS selector to retrieve the elements that match. In this example, there is only one match for each of the jQuery selectors. The hash (#) symbol indicates that you want to search for the id of the element. When the statement is executed, the *txtInput* variable will contain a jQuery object, which is a wrapper object that contains the results. This is different from the original code, in which the *txtInput* variable contained a direct reference to the DOM element. The wrapper object has an array of elements that match the search criteria or has no elements if there is no match. Even if the query doesn't match any elements, *txtInput* still contains the wrapper object, but no elements would be in the results.

When a breakpoint is added to the code after the two statements are executed, you can debug the code and explore the jQuery wrapper, as shown in Figure 6-11.

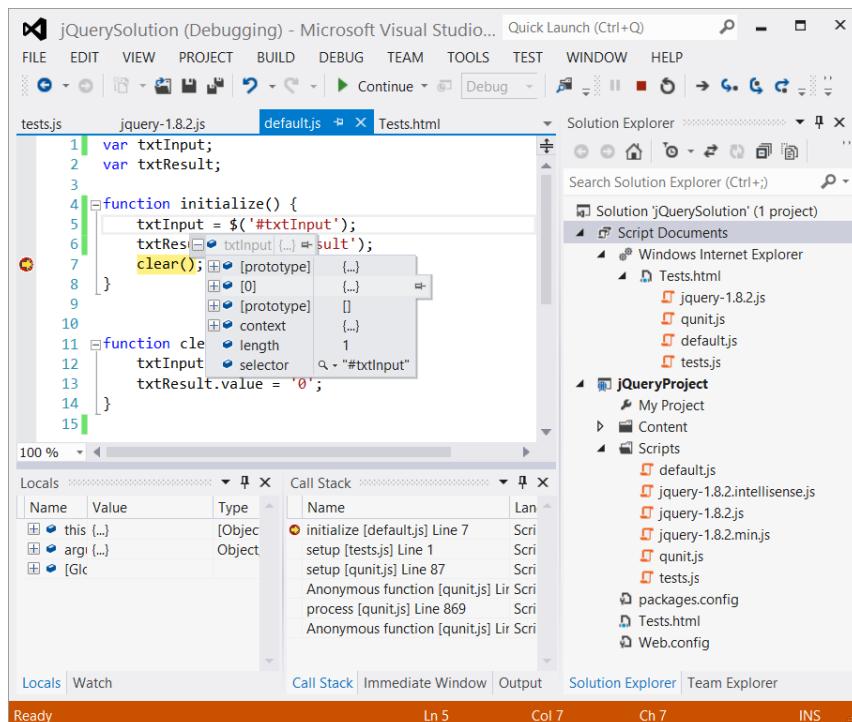


FIGURE 6-11 The jQuery wrapper object for *txtInput* with one element

In Figure 6-11, notice there is an array element (shown as [0]), and the length property is set to 1. This is how you can verify the result of the query. Element 0 is a direct reference to the `txtInput` DOM element.

When you run the test, it will pass but not for the correct reason; `txtInput` and `txtResult` reference the jQuery wrapper, not the actual DOM element. When the `value` property is set to '0', a new property is dynamically created on the jQuery object and set to '0'. However, the intent of this query is to set the text box value to '0'. To correct this problem, you can use the `val` method on the jQuery object. The `val` method gets or sets the `value` property of a form control that has a `value` property. The following is the modified test code.

```
module('QUnit Test Suite', { setup: function () { initialize(); } });

test("Initialize Test", function () {
    expect(2);
    var expected = '0';
    equal(txtInput.val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + txtInput.val());
    equal(txtResult.val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + txtResult.val());
});
```

After the four changes are made to the test, running the test shows that test assertions fail because `value` properties on the DOM elements are not being set. To fix the problem, modify the code in the `clear` function to set the value by using jQuery's `val` method. The following is the completed code.

```
var txtInput;
var txtResult;

function initialize() {
    txtInput = $('#txtInput');
    txtResult = $('#txtResult');
    clear();
}

function clear() {
    txtInput.val('0');
    txtResult.val('0');
}
```

IMPORTANT REFRESH YOUR SCREEN

HTML documents and JavaScript files are normally cached by the browser, so you might not see changes you made by just running the webpage. To refresh, press **Ctrl+F5** after the screen is displayed.

This code is complete, the tests pass, and the text boxes are populated with '0'. It's important for you to use the jQuery object whenever possible so you can benefit from the cross-

browser-compatible features that jQuery has. If you need to reference the DOM object from the jQuery wrapper, you can do it as follows.

```
var domElement = $('#txtInput')[0];
```

Don't forget that you can put this code inside a conditional statement that checks the length property to see whether an element exists before attempting to access element 0 of the result.

```
var domElement;
if($('#txtInput').length > 0){
    domElement = $('#txtInput')[0];
}
```

Enabling JavaScript and jQuery IntelliSense

When learning a new language or library, it's always good to have some help to keep you from getting stuck on every statement you write. When you installed jQuery, an IntelliSense file was added, but it is not yet being used. For example, in the default.js file, if you type a jQuery expression that includes a selector and then press the Period key, you would like to see a valid list of available methods and properties. Before setting up IntelliSense, Figure 6-12 shows an example of what you see in the IntelliSense window when you type in a jQuery expression with a selector and press Period.

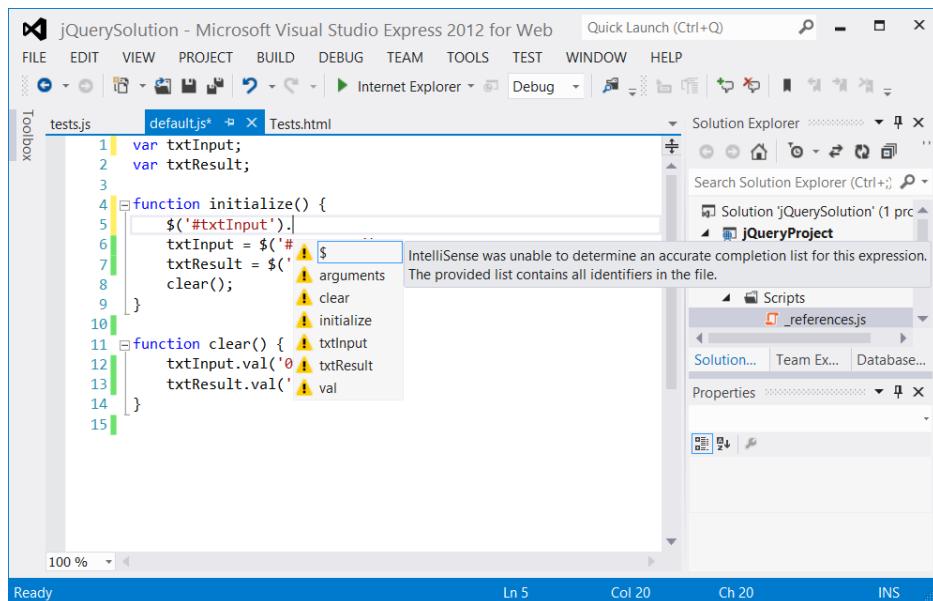


FIGURE 6-12 The IntelliSense window when not properly set up for jQuery

All the IntelliSense suggestions have a yellow warning triangle, and a message is displayed that states, “IntelliSense was unable to determine an accurate completion list for this expression. The provided list contains all identifiers in the file.”

To activate IntelliSense, you must set a reference to the jQuery file (not the IntelliSense file) in every JavaScript file that requires IntelliSense. The following is an example of the default.js file with the reference set.

```
/// <reference path="jquery-1.8.2.js" />
var txtInput;
var txtResult;

function initialize() {
    txtInput = $('#txtInput');
    txtResult = $('#txtResult');
    clear();
}

function clear() {
    txtInput.val('0');
    txtResult.val('0');
}
```

This reference was added by just dragging and dropping the jquery-1.8.2.js file to the top of the file. You can imagine that this can become a problem because you add many libraries and have hundreds of JavaScript files in a project. You might also want to benefit from IntelliSense in HTML files. To solve the problem, Microsoft has provided the ability to create a reference list and then just add the reference list to the top of the JavaScript files. You do so by adding a _references.js JavaScript file to your Scripts folder and then referencing that file in your JavaScript files. Even though you need to add the reference to the _references.js file to all your JavaScript files, when you add another library, you need to add it only to the _references.js file.

Why do you need the special name and why does it need to be in the Scripts folder when you need to reference the file explicitly? If you use a file called _references.js that is located in the Scripts folder, you automatically have a reference to this file in your HTML pages, although you still need to add the reference to your JavaScript files. The following is the contents of the _references.js file.

```
/// <reference path="jquery-1.8.2.js" />
/// <reference path="qunit.js" />
```

Visual Studio automatically locates the associated IntelliSense file, if one exists with the same name as the library, in the `libraryName.intellisense.js` format. In addition to using IntelliSense files if they exist, Visual Studio looks at all referenced libraries and provides default IntelliSense.

```
var txtResult;

function initialize() {
    txtInput = $('#txtInput');
```

```

txtResult = $('#txtResult');
clear();
}

function clear() {
    txtInput.val('0');
    txtResult.val('0');
}

```

After adding the reference, if you type a jQuery expression, you activate IntelliSense as soon as you enter the dollar sign and the opening parenthesis, as shown in Figure 6-13.

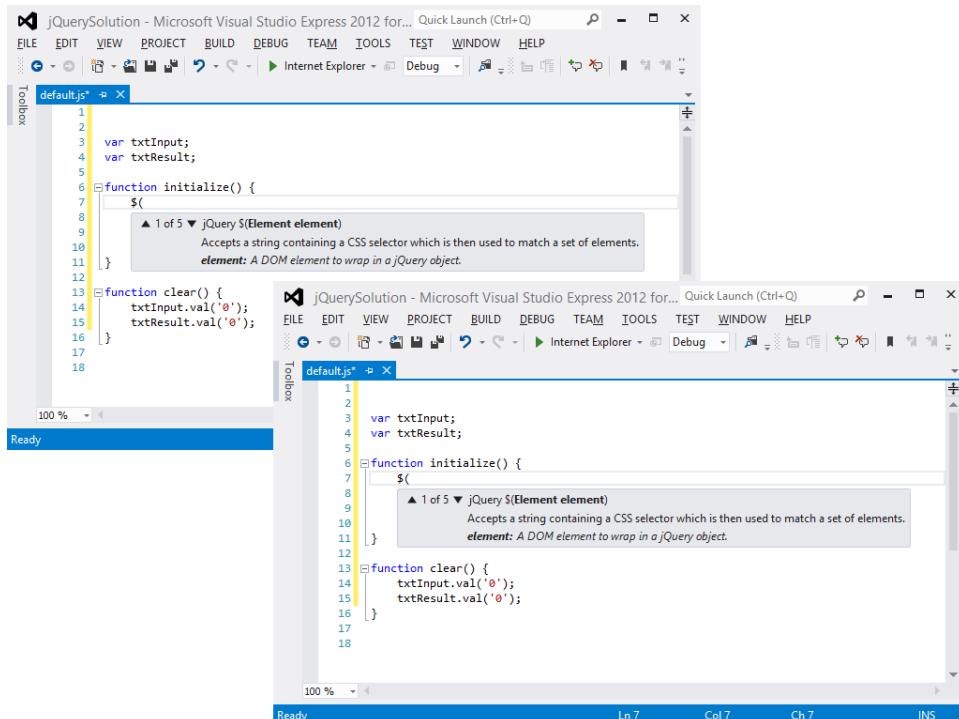


FIGURE 6-13 The jQuery IntelliSense providing help as you type

Notice in Figure 6-13 that after you finish typing the selector and you press Period, you are provided with a valid list of properties and methods for the jQuery wrapper object.

What happens if you are in the clear function and type `txtInput` and press period? Did IntelliSense make sense? You get an IntelliSense response that is similar to that in Figure 6-12. Simply put, don't activate IntelliSense; `txtInput` and `txtResult` are global variables that can be set to anything anywhere in your application, so Visual Studio can't possibly provide accurate IntelliSense. However, if you try typing `txtInput` and press period at the bottom of the initialize function, you get proper IntelliSense that's similar to that in Figure 6-13. The difference is that Visual Studio is examining your code and knows that you just assigned a jQuery object to

`txtInput`, so proper IntelliSense can be provided. To take advantage of IntelliSense, the global variables are eliminated, as shown in the following, modified `default.js` file.

```
function initialize() {
    clear();
}

function clear() {
    $('#txtInput').val('0');
    $('#txtResult').val('0');
}
```

This code is much smaller without the global variables, but the test is now failing because the test still references the global variables. To fix the test, replace the global variable references as follows.

```
module('QUnit Test Suite', { setup: function () { initialize(); } });

test("Initialize Test", function () {
    expect(2);
    var expected = '0';
    equal($('#txtInput').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtInput').val());
    equal($('#txtResult').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtResult').val());
});
```

When the test is run, it passes.

Quick check

- You want to save time when writing JavaScript code. Which library can you use to accomplish this goal?

Quick check answer

- Use the jQuery library.

Creating a jQuery wrapper for a DOM element reference

You've seen how the use of CSS selectors create a jQuery result, which is a wrapper around zero to many DOM elements that match the selector. You can also create a jQuery wrapper from a DOM element reference, as shown in the following examples.

```
var doc = $(document);
var innerText = $(this).text();
```

The first expression wraps the `document` object and assigns the result to a `doc` variable so you can use jQuery's methods with the `document` object. The second expression wraps the `this` object, which is a DOM element being passed to an event listener. After wrapping the

this object, the jQuery `text` method retrieves the inner text of the element and assigns it to an `innerText` variable.

Adding event listeners

In HTML5, you can use the `addEventListener` function to subscribe to an event. If you want a browser-independent way to add an event listener, you can use the jQuery `.on` method. There is also a corresponding `.off` method to remove an event listener. The `.on` method can be used as follows.

```
$('#btnSubmit').on('click', myFunction);
```

In this example, a button whose id is `btnSubmit` is located, using jQuery and the `.on` method to add a call to the user-defined `myFunction` function to the click event of the button. To remove the event listener, use the same code but replace the `.on` with `.off` as follows.

```
$('#btnSubmit').off('click', myFunction);
```

Triggering event handlers

When you need to trigger the event handlers by using code, you'll find that jQuery can help. Probably the most common reason to trigger event handlers by using code is to test your code. Using jQuery's `trigger` or the `triggerHandler` method causes the handler code to execute.

The `trigger` method causes the default behavior of the control to execute, whereas the `triggerHandler` method does not. For example, executing the `trigger` method on a submit button causes the submit action to take place in addition to executing your event handler code. Another difference is that the `trigger` method executes for all elements matched in the jQuery selector, whereas the `triggerHandler` method executes for only the first element. The following is an example of triggering the event handler code for the click event on a submit button.

```
$('#btnSubmit').triggerHandler('click');
```

Initializing code when the browser is ready

You will often need to execute initialization code after the HTML document is loaded and ready, and jQuery executes with a browser-independent way to execute code when the document is loaded as follows.

```
<script>
  $(document).ready(function () {
    initialize();
  });
</script>
```

It's best to place this at the bottom of your HTML document and call an initialize function that contains all initialization code.

Lesson summary

- Download jQuery from <http://jQuery.com> or install it from the NuGet package manager.
- The jQuery library is in a jQuery namespace and is aliased as a dollar sign (\$).
- Use the \$(selector) syntax to locate document object model (DOM) elements. The result of \$(selector) is a jQuery wrapper object containing zero to many DOM elements that match the selector. You can use the length property to find out whether there are any matches to the selector.
- Use jQuery's val method to get or set the value of a DOM element that has a value property.
- To enable IntelliSense, create a _references.js file in the Scripts folder and add library references to this file. In your JavaScript files, add a reference to the _references.js file.
- Use jQuery's .on and .off methods to add and remove event listeners.
- Use the \$(document).ready(function(){ initialize(); }); expression to add initialization code.

Lesson review

Answer the following questions to test your knowledge of the information in this lesson. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You want to locate all the elements on your webpage that are assigned the CSS class name Hidden. Which jQuery statement can you use?
 - A. var hidden = \$('#Hidden');
 - B. var hidden = \$('.Hidden');
 - C. var hidden = \$ ('Hidden');
 - D. var hidden = \$('class=Hidden');
2. You are interested in writing event-driven JavaScript code that will work on most browsers without writing browser-specific code. How can you accomplish this?
 - A. Use the jQuery library to help.
 - B. Use only JavaScript statements that are the same across all browsers.
 - C. Do not use any JavaScript.
 - D. It's impossible to write event-driven JavaScript code that is not browser-specific.

3. You are interested in locating all `<p>` elements on your webpage, so your statement is `var paragraphs = $('p');`. Which line of code would confirm whether at least one element is found?
- A. `if(paragraphs.exists)`
 - B. `if(paragraphs==null)`
 - C. `if(paragraphs.length)`
 - D. `if(paragraphs.count > 0)`

Practice exercises

If you encounter a problem completing any of these exercises, the completed projects can be installed from the Practice Exercises folder that is provided with the companion content.

Exercise 1: Create a calculator object

In this exercise, you apply your JavaScript object-oriented programming knowledge by modifying the calculator you've been using to have a calculator object in the `calculatorLibrary` namespace and changing JavaScript code to use jQuery when necessary.

1. Start Visual Studio Express 2012 for Web. Click File and choose Open Project. Navigate to the solution you created in Chapter 5, "More HTML," and select the `webCalculator.sln` file. Click Open.
If you didn't complete the exercises in Chapter 5, you can use the solution in the Chapter 6 Exercise 1 Start folder.
2. In the Solution Explorer window, right-click the `CalculatorTests.html` file and choose Set As Start Page. Press F5 to verify that your test runs and passes.
3. In the Solution Explorer window, add jQuery to the project by right-clicking the project node. Choose Manage NuGet Packages. Type **jQuery** in the search online text box and click the search button. Click the Install button on the jQuery result.
4. Add a file to the Scripts folder called `_references.js` and, in the file, add a reference to jQuery, QUnit, and the `calculatorLibrary`.

Your file should look like the following.

```
/// <reference path="jquery-1.8.2.js" />
/// <reference path="qunit.js" />
/// <reference path="calculatorLibrary.js" />
```

5. Open the `calculatorLibrary.js` file and add a reference to the `_references.js` file.
6. Create the `calculatorLibrary` namespace by surrounding the existing code in the `calculatorLibrary.js` file with an immediately invoked function expression (IIFE). In the IIFE, create an alias to `calculatorNamespace` called `ns`, which will save you from typing the complete namespace while you're in the IIFE.

Your code should look like the following.

```
/// <reference path="_references.js" />

(function () {
    this.calculatorNamespace = this.calculatorNamespace || {};
    var ns = this.calculatorNamespace;

    //existing code here.....

}());
```

7. Remove the variables that reference *txtInput* and *txtResult* because jQuery will be used to access these DOM elements as needed.

The initialize function will remain in the namespace.

8. Surround the *numberClick*, *plusClick*, *minusClick*, *clearEntry*, and *clear* functions with an IIFE that is assigned to a *Calculator* property in *calculatorNamespace*.

Your code should look like the following.

```
ns.Calculator = (function () {

    function numberClick() {
        txtInput.value = txtInput.value == '0' ?
            this.innerText : txtInput.value + this.innerText;
    }

    function plusClick() {
        txtResult.value = Number(txtResult.value) + Number(txtInput.value);
        clearEntry();
    }

    function minusClick() {
        txtResult.value = Number(txtResult.value) - Number(txtInput.value);
        clearEntry();
    }

    function clearEntry() {
        txtInput.value = '0';
    }

    function clear() {
        txtInput.value = '0';
        txtResult.value = '0';
    }
}());
```

9. Add a *Calculator* function inside the IIFE, which will be the constructor function. There is no code for the constructor at this time. At the bottom of the IIFE, add code to return this constructor function. Try this on your own, but if you have a problem, the sample code is shown in step 10.

- 10.** Modify the numberClick, plusClick, minusClick, clearEntry, and clear functions to define these functions on the Calculator prototype.

The CalculatorLibrary.js should look like the following.

```
///<reference path="_references.js" />

(function () {
    this.calculatorNamespace = this.calculatorNamespace || {};
    var ns = this.calculatorNamespace;

    function initialize() {
        for (var i = 0; i < 10; i++) {
            document.getElementById('btn' + i)
                .addEventListener('click', numberClick, false);
        }
        txtInput = document.getElementById('txtInput');
        txtResult = document.getElementById('txtResult');

        document.getElementById('btnPlus')
            .addEventListener('click', plusClick, false);
        document.getElementById('btnMinus')
            .addEventListener('click', minusClick, false);
        document.getElementById('btnClearEntry')
            .addEventListener('click', clearEntry, false);
        document.getElementById('btnClear')
            .addEventListener('click', clear, false);
        clear();
    }

    ns.Calculator = (function () {

        function Calculator() {}

        Calculator.prototype.numberClick = function () {
            txtInput.value = txtInput.value == '0' ?
                this.innerText : txtInput.value + this.innerText;
        };

        Calculator.prototype.plusClick = function () {
            txtResult.value = Number(txtResult.value) + Number(txtInput.value);
            clearEntry();
        };

        Calculator.prototype.minusClick = function () {
            txtResult.value = Number(txtResult.value) - Number(txtInput.value);
            clearEntry();
        };

        Calculator.prototype.clearEntry = function () {
            txtInput.value = '0';
        };

        Calculator.prototype.clear = function () {
```

```

        txtInput.value = '0';
        txtResult.value = '0';
    };

    return Calculator;
}());

})();

```

- 11.** In the initialize function, create a *calculator* variable and assign a new Calculator object to it. Be sure to use the namespace when creating the new Calculator object.

The state should look like the following.

```
var calculator = new ns.Calculator();
```

- 12.** Convert the loop that adds event listeners to each of the number buttons to a single jQuery statement based on finding all button elements that have an id that starts with *btnNumber*.

The statement should look like the following.

```
$('.button[id^="btnNumber"]').on('click', calculator.numberClick);
```

To make the code work in this step, change the ids on the number buttons.

- 13.** Open the default.html file and replace the number button ids with *btnNumberX* where *X* is the number on the button.

- 14.** Open the CalculatorTests.html file and replace the number button ids with *btnNumberX* where *X* is the number on the button.

- 15.** In the CalculatorLibrary.js file, locate the initialize function and delete the statements that set *txtInput* and *txtResult*.

- 16.** Convert the code that adds event listeners to *btnPlus*, *btnMinus*, *btnClearEntry*, and *btnClear* to use jQuery.

The completed initialize function should look like the following.

```

function initialize() {
    var calculator = new ns.Calculator();
    $('.button[id^="btnNumber"]').on('click', calculator.numberClick);
    $('#btnPlus').on('click', calculator.plusClick);
    $('#btnMinus').on('click', calculator_MINUS.click);
    $('#btnClearEntry').on('click', calculator.clearEntry);
    $('#btnClear').on('click', calculator.clear);
    clear();
}

```

- 17.** Convert the *numberClick* method to use jQuery.

You can use the jQuery *text* method to retrieve the inner text. The completed method should look like the following.

```
Calculator.prototype.numberClick = function () {
    $('#txtInput').val($('#txtInput').val() == '0' ?
```

```
        $(this).text() : $('#txtInput').val() + $(this).text());
    };

```

18. Convert the plusClick method to use jQuery.

You must call the clearEntry method, but you can't use the *this* keyword to call clearEntry because the clicked button is referenced by *this*. Because there is only one copy of the clearEntry method, and it's on the prototype, call the clearEntry method from the Calculator prototype. Your code should look like the following.

```
Calculator.prototype.plusClick = function () {
    $('#txtResult').val(Number($('#txtResult').val()) +
        Number($('#txtInput').val()));
    Calculator.prototype.clearEntry();
};


```

19. Convert the minusClick method to use jQuery.

Your code should look like the following.

```
Calculator.prototype_MINUSClick = function () {
    $('#txtResult').val(Number($('#txtResult').val()) -
        Number($('#txtInput').val()));
    Calculator.prototype.clearEntry();
};


```

20. Convert the clearEntry method and the clear method to use jQuery.

The completed CalculatorLibrary.js file should look like the following.

```
/// <reference path="_references.js" />

(function () {
    this.calculatorNamespace = this.calculatorNamespace || {};
    var ns = this.calculatorNamespace;

    ns.initialize = function () {
        var calculator = new ns.Calculator();
        $('button[id^="btnNumber"]').on('click', calculator.numberClick);
        $('#btnPlus').on('click', calculator.plusClick);
        $('#btnMinus').on('click', calculator_MINUSClick);
        $('#btnClearEntry').on('click', calculator.clearEntry);
        $('#btnClear').on('click', calculator.clear);
        calculator.clear();
    }

    ns.Calculator = (function () {

        function Calculator() {}

        Calculator.prototype.numberClick = function () {
            $('#txtInput').val($('#txtInput').val() == '0' ?
                $(this).text() : $('#txtInput').val() + $(this).text());
        };
    });
}


```

```

Calculator.prototype.plusClick = function () {
    $('#txtResult').val(Number($('#txtResult').val()) +
        Number($('#txtInput').val()));
    Calculator.prototype.clearEntry();
};

Calculator.prototype_MINUSClick = function () {
    $('#txtResult').val(Number($('#txtResult').val()) -
        Number($('#txtInput').val()));
    Calculator.prototype.clearEntry();
};

Calculator.prototype.clearEntry = function () {
    $('#txtInput').val('0');
};

Calculator.prototype.clear = function () {
    $('#txtInput').val('0');
    $('#txtResult').val('0');
};

return Calculator;
}());
}
);

```

- 21.** Open the default.html file and add a reference to the jQuery library.

Be sure to add the reference before the reference to the CalculatorLibrary.js file because that file uses jQuery. Don't forget that you can drag and drop the file to create the reference. The `<head>` element should look like the following.

```

<head>
    <title>web Calculator</title>
    <link href="Content/default.css" rel="stylesheet" />
    <script src="Scripts/jquery-1.8.2.js"></script>
    <script type="text/javascript" src="Scripts/CalculatorLibrary.js"></script>
</head>

```

- 22.** At the bottom of the default.html file, change the code so that the initialize function in calculatorNamespace is executed when the document is ready.

The completed default.html file should look like the following.

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>web Calculator</title>
    <link href="Content/default.css" rel="stylesheet" />
    <script src="Scripts/jquery-1.8.2.js"></script>
    <script type="text/javascript" src="Scripts/CalculatorLibrary.js"></script>
</head>
<body>
    <div id="container">
        <header>
            <hgroup id="headerText">

```

```
<h1>Contoso Ltd.</h1>
    <h2>Your success equals our success</h2>
</hgroup>
</header>
<nav>
    <a href="default.html">Home</a>
</nav>
<div role="main">
    <div id="calculator">
        <table>
            <tr>
                <td colspan="4">
                    <input id="txtResult" type="text"
                           readonly="readonly" />
                </td>
            </tr>
            <tr>
                <td colspan="4">
                    <input id="txtInput" type="text" />
                </td>
            </tr>
            <tr>
                <td></td>
                <td></td>
                <td>
                    <button id="btnClearEntry">CE</button>
                </td>
                <td>
                    <button id="btnClear">C</button>
                </td>
            </tr>
            <tr>
                <td>
                    <button id="btnNumber7">7</button></td>
                <td>
                    <button id="btnNumber8">8</button></td>
                <td>
                    <button id="btnNumber9">9</button></td>
                <td>
                    <button id="btnPlus">+</button>
                </td>
            </tr>
            <tr>
                <td>
                    <button id="btnNumber4">4</button>
                </td>
                <td>
                    <button id="btnNumber5">5</button>
                </td>
                <td>
                    <button id="btnNumber6">6</button>
                </td>
                <td>
                    <button id="btnMinus">-</button>
                </td>
            </tr>
        </table>
    </div>
</div>
```

```

        </tr>
        <tr>
            <td>
                <button id="btnNumber1">1</button>
            </td>
            <td>
                <button id="btnNumber2">2</button>
            </td>
            <td>
                <button id="btnNumber3">3</button>
            </td>
            <td></td>
        </tr>
        <tr>
            <td></td>
            <td>
                <button id="btnNumber0">0</button>
            </td>
            <td></td>
            <td></td>
        </tr>
    </table>
</div>
</div>
<aside>
    <p>Advertisements</p>
</aside>
<footer>
    <p>
        Copyright © 2012, Contoso Ltd., All rights reserved
    </p>
</footer>
</div>
<script type="text/javascript">
    $(document).ready(function () {
        calculatorNamespace.initialize();
    });
</script>
</body>
</html>

```

You must modify the tests to use jQuery.

23. Open the CalculatorTests.html file and add a reference to the jQuery library.

The completed CalculatorTests.html file should look like the following.

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <link rel="stylesheet" type="text/css" href="Content/qunit.css" />
    <script type="text/javascript" src="Scripts/qunit.js"></script>
    <script src="Scripts/jquery-1.8.2.js"></script>
    <script type="text/javascript" src="Scripts/CalculatorLibrary.js"></script>
    <script type="text/javascript" src="Scripts/tests.js"></script>

```

```

</head>
<body>
    <h1 id="qunit-header">QUnit example</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
    <div id="qunit-fixture">
        test markup, will be hidden
        <input id="txtResult" type="text" readonly="readonly" /><br />
        <input id="txtInput" type="text" /><br />
        <button id="btnNumber7">7</button>
        <button id="btnNumber8">8</button>
        <button id="btnNumber9">9</button><br />
        <button id="btnNumber4">4</button>
        <button id="btnNumber5">5</button>
        <button id="btnNumber6">6</button><br />
        <button id="btnNumber1">1</button>
        <button id="btnNumber2">2</button>
        <button id="btnNumber3">3</button><br />
        <button id="btnClear">C</button>
        <button id="btnNumber0">0</button>
        <button id="btnClearEntry">CE</button><br />
        <button id="btnPlus">+</button>
        <button id="btnMinus">-</button>
    </div>
</body>
</html>

```

You must modify the tests.js file to use jQuery, calculatorNamespace, and the Calculator object.

- 24.** Open the tests.js file.
- 25.** In the tests.js file, add a reference to the _references.js file and modify the module function to call calculatorLibrary.initialize() as follows.

```

/// <reference path="_references.js" />
module('Calculator Test Suite', {
    setup: function () {
        calculatorNamespace.initialize();
    }
});

```

- 26.** Modify the Initialize Test.

You don't need to set *txtInput* and *txtResult* because the initialize method calls the clear method to set these text boxes.

- 27.** Modify the rest of the method to use jQuery and run the test to see it pass.

The completed Initialize Test should look like the following.

```

test("Initialize Test", function () {
    expect(2);
    var expected = '0';
    equal($('#txtInput').val(), expected, 'Expected value: ' + expected +

```

```
        ' Actual value: ' + $('#txtInput').val());
    equal($('#txtResult').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtResult').val());
});
```

- 28.** Modify the Button Click Test to use jQuery. Run the test to see it pass. Use jQuery's triggerHandler method to test each button.

Your code should look like the following.

```
test("Button Click Test", function () {
    var buttonQuantity = 10;
    expect(buttonQuantity * 2);
    for (var i = 0; i < buttonQuantity; i++) {
        $('#btnNumber' + i).triggerHandler('click');
        var result = $('#txtInput').val()[$('#txtInput').val().length - 1];
        var expected = String(i);
        equal(result, expected, 'Expected value: ' + expected +
            ' Actual value: ' + result);
        var expectedLength = i < 2 ? 1 : i;
        equal($('#txtInput').val().length, expectedLength,
            'Expected string length: ' + expectedLength +
            ' Actual value: ' + $('#txtInput').val().length);
    }
});
```

- 29.** Modify the Add Test to use jQuery. Run the test to see it pass.

Your code should look like the following.

```
test("Add Test", function () {
    expect(2);
    $('#txtInput').val('10');
    $('#txtResult').val('20');
    $('#btnPlus').triggerHandler('click');
    var expected = '30';
    equal($('#txtResult').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtResult').val());
    expected = '0';
    equal($('#txtInput').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtInput').val());
});
```

- 30.** Modify the Subtract Test to use jQuery. Run the test to see it pass.

Your code should look like the following.

```
test("Subtract Test", function () {
    expect(2);
    $('#txtInput').val('10');
    $('#txtResult').val('20');
    $('#btnMinus').triggerHandler('click');
    var expected = '10';
    equal($('#txtResult').val(), expected, 'Expected value: ' + expected +
        ' Actual value: ' + $('#txtResult').val());
    expected = '0';
```

```
        equal($('#txtInput').val(), expected, 'Expected value: ' + expected +  
              ' Actual value: ' + $('#txtInput').val());  
    });
```

- 31.** Modify the Clear Entry Test to use jQuery. Run the test to see it pass.

Your code should look like the following.

```
test("Clear Entry Test", function () {  
    expect(1);  
    $('#txtInput').val('10');  
    $('#btnClearEntry').triggerHandler('click');  
    var expected = '0';  
    equal($('#txtInput').val(), expected, 'Expected value: ' + expected +  
          ' Actual value: ' + $('#txtInput').val());  
});
```

- 32.** Modify the Clear Test to use jQuery. Run the test to see it pass.

Your code should look like the following.

```
test("Clear Test", function () {  
    expect(2);  
    $('#txtInput').val('10');  
    $('#txtResult').val('20');  
    $('#btnClear').triggerHandler('click');  
    var expected = '0';  
    equal($('#txtInput').val(), expected, 'Expected value: ' + expected +  
          ' Actual value: ' + $('#txtInput').val());  
    equal($('#txtResult').val(), expected, 'Expected value: ' + expected +  
          ' Actual value: ' + $('#txtResult').val());  
});
```

At this point, you should be able to run all the tests, and they should all pass.

- 33.** Right-click the default.html file and choose Set As Start Page. To see that your calculator still works, press F5 to start debugging the application.
- 34.** Try entering data and clicking the plus and minus signs.

You might need to refresh your screen, but the calculator should be working.

Suggested practice exercises

The following additional exercises are designed to give you more opportunities to practice what you've learned and to help you successfully master the lessons presented in this chapter.

- **Exercise 1** Learn more about JavaScript objects by adding more features to the calculator that you created in the practice exercise.
- **Exercise 2** Learn more about jQuery by exploring the jQuery site at <http://jQuery.com>.

Answers

This section contains the answers to the lesson review questions in this chapter.

Lesson 1

- 1. Correct answer: C**
 - A. Incorrect:** A property is a variable that's defined on an object.
 - B. Incorrect:** A method is a function that's defined on an object.
 - C. Correct:** A class is a blueprint for an object.
 - D. Incorrect:** An event takes place from external input, usually from user input.
- 2. Correct answer: A**
 - A. Correct:** The prototype is the starting object that is cloned when creating a new object.
 - B. Incorrect:** A property is a variable that's defined on an object.
 - C. Incorrect:** A class is a blueprint for an object.
 - D. Incorrect:** An event takes place from external input, usually from user input.
- 3. Correct answer: A**
 - A. Correct:** You replace the prototype of the child object with a new instance of the parent object and then replace the prototype constructor with the child constructor.
 - B. Incorrect:** The `createChild` method is not a valid method.
 - C. Incorrect:** The `setParent` method is not a valid method.
 - D. Incorrect:** JavaScript does support inheritance by replacing the prototype of the child object with a new instance of the parent object and then replacing the prototype constructor with the child constructor.

Lesson 2

- 1. Correct answer: B**
 - A. Incorrect:** The use of the hash (#) symbol in the CSS selector indicates that you want to locate an element based on its id.
 - B. Correct:** The use of the period (.) in the CSS selector indicates that you want to locate the elements that match the CSS class name.
 - C. Incorrect:** Supplying a name for a CSS selector indicates that you want to locate the elements that have that tag name.
 - D. Incorrect:** The `var hidden = $('#class=Hidden');` syntax is invalid.

2. Correct answer: A

- A. Correct:** Using jQuery will help you create event-driven, browser-independent code.
- B. Incorrect:** The code for creating and subscribing to events is browser-specific.
- C. Incorrect:** You need to use JavaScript to write event-driven code.
- D. Incorrect:** Use the jQuery library to write browser-independent code.

3. Correct answer: C

- A. Incorrect:** jQuery does not have an exists property.
- B. Incorrect:** Even if no elements are found, jQuery will return a wrapper object.
- C. Correct:** If no elements are found, the length property will be 0, which converts to a Boolean false.
- D. Incorrect:** jQuery does not have a count property.

Index

Symbols

&& (and) operator, 70–71, 84
+ (addition) operator, 67–68, 70
& (ampersand), 41, 322
* (asterisk) symbol, 147, 155
\ (backslash) character, 69
^ (caret) symbol, 156
: (colon), 150, 327
© (copyright), 41
/ (division) operator, 67–68, 70
\$ (dollar sign), 72–73, 157, 288
" (double quotes), 41, 69
/ (forward slash) character, 337
> (greater-than sign), 41, 148
(hash) symbol
 id selectors and, 146
 internal hyperlinks and, 46
 jQuery support, 289
 in manifest file, 609
 in RGB values, 166
< (less-than sign), 41
* (multiplication) operator, 67–68, 70
! (not) logical operator, 70–71
|| (or) logical operator, 70–71
!= operator, 84
== operator, 84
() (parentheses), 68–69
. (period) symbol, 146, 327
+ (plus sign), 67–69, 151, 327
? (question mark), 322, 327
® (registered trademark), 41

; (semicolon), 71, 139
' (single quotes), 69
- (subtraction) operator, 67–69
~ (tilde) character, 152
_ (underscore), 45, 72–73

A

<*a*> element
 data-linktype attribute, 157–158
 described, 32
 <*dfn*> element and, 218
 href attribute, 46, 153–157

- ::after pseudo element, 150
- :after selector, 224
- afterprint event, 117
- aggregate modules, 348
- aggregating functions, 588
- AJAX calls to web services
 - described, 368–369
 - handling errors, 372–373
 - handling progress, 371–372
 - jQuery promises, 377–380, 394–395
 - jQuery XMLHttpRequest wrappers, 373–377
 - XMLHttpRequest object, 369–371
- alert() function, 76, 79, 101
- alt attribute
 - <*area*> element, 49
 - <*img*> element, 47–48
- ampersand (&), 41, 322
- and (&&) operator, 70–71, 84
- annotating content, 213
- APIs (application programming interfaces), 31
- App Fabric Caching Service (Microsoft), 422
- application class, 212
- application programming interfaces (APIs), 31
- applicationCache object
 - swapCache() method, 610
 - update() method, 610
- applications. *See* offline web applications; packages (Node.js)
- arbitrary web services, 364, 366
- arcs
 - drawing using arc() method, 485–487
 - drawing using arcTo() method, 481–485
- <*area*> element
 - alt attribute, 49
 - coords attribute, 49
 - creating image maps, 49–50
 - described, 32
 - href attribute, 49
 - shape attribute, 49
 - as void element, 38
- arguments to functions, 73–76
- ARIA (Accessible Rich Internet Applications), 212
- arithmetic operators, 68
- array items, 109
- array methods, 110–112
- Array object
 - as built-in object, 67
 - concat() method, 110
 - creating instance, 108–109
 - indexOf() method, 110
 - join() method, 110
 - lastIndexOf() method, 110
 - length property, 109–110
 - pop() method, 110
 - push() method, 110
 - reverse() method, 111
 - shift() method, 111
 - slice() method, 111
 - sort() method, 111
 - splice() method, 111
 - toString() method, 111–112
 - unshift() method, 112
 - valueOf() method, 112
- array properties, 109–110
- arrays
 - condensed, 109
 - creating, 108–109
 - described, 108
 - literal, 109
 - populating, 108–109
- artboard (Blend), 15
- <*article*> element, 32, 209
- <*aside*> element
 - described, 32, 208
 - <*figure*> element and, 219
 - in layout containers, 208, 211–212
 - role attribute, 212
- ASP.NET
 - Empty Web Application template, 10–11
 - QUnit tool and, 90–95

SignalR library support, 423
 asterisk (*) symbol, 147, 155
 async attribute (`<script>` element), 101
 asynchronous operations
 conditional calls, 401–402
 described, 393
 lesson summary and review, 402–403, 405–406, 413–414
 practice exercises, 406–412
 promise objects and, 394–402
 working with web workers, 404–405, 563
`attr()` function, 153
 attribute contains value in list selector, 157–158
 attribute contains value selector, 155–156
 attribute selector, 153–154
 attribute value ends with selector, 157
 attribute value selector, 154–155
 attributes
 adding to elements, 35–36
 author-defined, 39
 Boolean, 35–36, 321, 325
 custom, 39
 described, 35
 expando, 39
 global, 37
 inherit value, 163
 retrieving values of, 153
`<audio>` element
 autoplay attribute, 445
 configuring, 445–446
 controls attribute, 445
 described, 32, 443–445
 `HTMLMediaElement` object and, 444, 447
 loop attribute, 445–446
 preload attribute, 446
 `<source>` element and, 445
 src attribute, 446
 audio and sounds. *See also* `<audio>` element
 audio formats, 444
 described, 443–444

lesson summary and review, 446–447, 456–457
 author-defined attributes, 39
 autofocus Boolean attribute, 321
 autoIncrement key generator, 592
 autoplay attribute
 `<audio>` element, 445
 `<video>` element, 441

B

`` element
 annotating content, 213–214
 described, 31, 33, 213
 background-color property (CSS), 139
 backslash (\) character, 69
 backward compatibility
 of browsers, 30
 of CSS3, 138
 banner class, 212
`<base>` element, 33, 38
 base classes, 262
`<bb>` element, 33
`<bdo>` element, 33
`::before` pseudo element, 150
`:before` selector, 224
 beforeonload event, 117
 beforeprint event, 117
 Berkeley Software Distribution (BSD) license, 438
 big web services, 364, 366
 binary operators, 70
 Blank App template, 6–9, 14, 96
 Blend for Visual Studio 2012
 artboard, 15
 Assets panel, 14–15, 17
 CSS Properties tab, 15–16
 described, 13–16
 Device panel, 15, 17–18
 HTML Attributes tab, 15–16
 lesson summary and review, 19–20, 27

<blockquote> element

Live DOM panel, 14–15, 19
practice exercises, 24–25
Projects panel, 14–17
Style Rules panel, 14–15, 18–19
Tools panel, 14–15
<blockquote> element
 annotating content, 215
 cite attribute, 215
 described, 33, 215
blur event, 117–118
<body> element
 basic document structure, 40–41
 described, 9, 33
 <object> tag and, 51
 working with events, 117
Boolean attributes
 described, 35–36, 321, 325
 minimized form, 36
 quoted form, 36
Boolean object, 67
Boolean primitive type
 described, 67, 70
 logical operators, 70–71
 short-circuit evaluation, 71
border properties (CSS), 176–178
box model (CSS)
 border properties, 176–178
 described, 175–176
 margin properties, 176–178
 padding properties, 176–178
box-sizing property (CSS), 190–192

 element
 annotating content, 217
 described, 33, 217
 self-closing tags and, 37–38
 as void element, 38, 217
breakpoints
 setting in JavaScript, 103–104
 setting in jQuery, 289

browsers
 audio formats supported, 444
 backward compatibility, 30
 built-in functions, 76–77
 centering content in window, 193
 comments and, 39–40
 drag and drop operations, 507, 518
 forward compatibility, 30
 Geolocation API support, 540
 HTML5 support, 206–207
 JavaScript considerations, 101–102
 nested context, 34
 no-quirks mode, 40
 special characters and, 41–42
 storage mechanisms and, 559–561, 566–567, 590, 601, 608
 triggering form submission, 319
 User Data API, 557
 validation errors, 325
 video formats supported, 438
 web communications, 312–314
 WebSocket support, 416
 working with elements, 32
 working with styles, 139, 159
BSD (Berkeley Software Distribution) license, 438
built-in browser styles, 159
built-in functions, 76–77
<button> element
 described, 33, 317
 disabling, 36
 triggering form submission, 320
 type attribute, 317

C

cache (HTTP). *See* HTTP cache
CACHE MANIFEST statement, 609
cached event, 610
calculating distances, 548–549

call methods, 279–280
calling functions, 74
canplay event, 119
canplaythrough event, 119
<canvas> element
described, 33, 460
drawing with images, 490
getContext() method, 460
height attribute, 460
implementing, 462–463
toDataUrl() method, 460
width attribute, 460
Canvas Pattern object, 468–469
CanvasGradient object, 466–468
CanvasRenderingContext2D object
addColorStop() method, 461, 466–467
arc() method, 461, 481–485
arcTo() method, 461, 485–487
beginPath() method, 461, 475
bezierCurveTo() method, 461
clearRect() method, 461, 463–464
clip() method, 461
closePath() method, 461, 477–478
configuring drawing state, 465–474
createImageData() method, 461
createLinearGradient() method, 461, 466–467
createPattern() method, 461, 468–469
createRadialGradient() method, 461, 466–467
data property, 461
described, 460
drawImage() method, 461, 490–493
drawing arcs, 481–487
drawing rectangles, 463–475, 478–479
drawing text, 488–490
drawing using paths, 475–487
drawing with images, 490–494
endPath() method, 475
fill() method, 461, 475, 477–481
fillRect() method, 461, 463–464, 478
fillStyle property, 461, 465–470
fillText() method, 461, 488–489
font property, 461, 488
getImageData() method, 461
globalAlpha property, 461
globalCompositeOperation property, 461
implementing, 462–463
isPointInPath() method, 461
lineCap property, 461
lineJoin property, 461, 471–472, 474–475
lineTo() method, 461, 475–477
lineWidth property, 461, 470–471, 474–475, 479
measureText() method, 461
miterLimit property, 461
moveTo() method, 462, 475, 478
putImageData() method, 462
quadraticCurveTo() method, 462
rect() method, 462, 475, 478–479
restore() method, 462, 474–475
rotate() method, 462
save() method, 462, 474–475
scale() method, 462
setTransform() method, 462
shadowBlur property, 462
shadowColor property, 462
shadowOffsetX property, 462
shadowOffsetY property, 462
stroke() method, 462, 475, 478–481
strokeRect() method, 462–464, 478
strokeStyle property, 462, 472–475
strokeText() method, 462, 488–489
textAlign property, 462, 488–489
textBaseline property, 462, 488–489
transform() method, 462
translate() method, 462
<caption> element, 33, 241
captions, adding to tables, 241
caret (^) symbol, 156
cascading if statements, 81

Cascading Style Sheets. *See* CSS (Cascading Style Sheets)

cascading styles, 160–161

case sensitivity

- for tags, 32
- for variable names, 72

catch block, 87

chained if statements, 81

change event, 118

character encoding, style sheets and, 143

@charset rule, 41, 143

checked Boolean attribute, 36

:checked pseudo class, 149

checking event, 610

child classes, 262

child selectors, 148

Chinese characters, 220–221

<*circle*> element

- cx attribute, 498
- cy attribute, 498
- described, 498
- fill attribute, 498
- id attribute, 498
- r attribute, 498

circles, drawing, 485–487, 498–499

citations and quotations, 215–216

<*cite*> element, 33, 215–216

cite attribute

- <*blockquote*> element, 215
- <*q*> element, 215

CJK languages, 220–221

class global attribute, 37

class selectors, 146–147

classes

- creating, 266–271
- described, 262

<*clear*> element, 189–190

clear property (CSS), 189–190

click event, 119

cm measurement unit, 175

<*code*> element, 33, 216–217

code blocks, 74

code loops

- breaking out of, 86–87
- described, 84
- do loop, 85
- for loop, 85–86
- while loop, 84–85

<*col*> element

- described, 33
- styling columns, 241–242
- as void element, 38

<*colgroup*> element, 33, 241–242

colon (:), 150, 327

<*color*> value type, 138

color names table, 167–171

color selector, 4

colors. *See* CSS colors

cols attribute (<*textarea*> element), 316

colspan attribute

- <*td*> element, 238–240
- <*th*> element, 238–240

<*command*> element, 33, 38

comments

- adding, 39
- adding conditional, 40
- adding within style sheets, 139
- in manifest file, 609
- TODO, 7

CommonJS, 394

communications

- asynchronous, 393–414
- web, 312–316
- WebSocket, 415–436

complementary class, 212

complex objects, storing, 562

condensed arrays, 109

conditional comments, 40

conditional programming

- if/else keywords, 80–81

- implementing code loops, 84–87
- switch keyword, 82–83
- confirm() function, 76
- CONNECT method (HTTP), 316
- constructor functions, 266–271
- constructs, 262
- content annotation, 213
- content embedding
 - adding images to HTML documents, 47–50
 - described, 44
 - embedding plug-in content, 50–52
 - using inline frames, 44–46
- lesson summary and review, 52–53, 63
- practice exercises, 58–61
- sandboxing, 45
- seamless, 45
 - working with hyperlinks, 46–47
- contenteditable global attribute, 37
- contentinfo class, 212
- context object. *See* CanvasRenderingContext2D object
- contextmenu event, 118
- contextmenu global attribute, 37
- controls attribute
 - <audio> element, 445
 - <video> element, 441, 450
- cookie plug-in (jQuery), 556–557
- cookies
 - alternatives to, 557–558
 - described, 556–557
 - limitations of, 556–557
- Coordinates object
 - accuracy property, 541
 - altitude property, 541
 - altitudeAccuracy property, 541
 - described, 540–541
 - heading property, 541
 - latitude property, 541
 - longitude property, 541
 - speed property, 541
- coords attribute (<area> element), 49
- copyright (©), 41
- CORS (cross-origin resource sharing), 380–381
- counter variables, 86
- cross-origin resource sharing (CORS), 380–381
- CRUD operations, 316, 365
- CSS box model
 - border properties, 176–178
 - described, 175–176
 - margin properties, 176–178
 - padding properties, 176–178
- CSS (Cascading Style Sheets)
 - adding comments within style sheets, 139
 - browser built-in styles, 159
 - cascading styles, 159–160
 - creating embedded styles, 140–141
 - creating external style sheets, 141–144
 - creating inline styles, 140
 - defining and applying styles, 139
 - defining selectors, 146–159
 - described, 31, 137–139
 - extending styles, 159
 - inheritance and, 162–163
 - lesson summary and review, 144–145, 163–165, 193–194, 202–204
 - practice exercises, 194–201
 - specificity in selectors, 161–162
 - working with important styles, 159–160
- CSS colors
 - described, 166
 - hsl() function, 172–173
 - rgb() function, 171
 - rgba() function, 172
 - setting fillStyle property, 465
 - setting value of, 166
 - table of color names and values, 167–171
 - transparency, 172
- CSS properties
 - box-sizing property, 190–192
 - centering content in browser window, 193

- clear property, 189–190
 - declaration block and, 139
 - described, 165
 - float property, 186–189
 - positioning div elements, 178–186
 - transparency, 172
 - working with colors, 166–173
 - working with CSS box model, 175–178
 - working with text, 173–175
 - CSS3**
 - adding comments within style sheets, 139
 - backward compatibility, 138
 - browser built-in styles, 159
 - cascading styles, 159–160
 - creating embedded styles, 140–141
 - creating external style sheets, 141–144
 - creating inline styles, 140
 - defining and applying styles, 139
 - defining selectors, 146–159
 - described, 137–139
 - extending styles, 159
 - inheritance and, 162–163
 - specificity in selectors, 161–162
 - Visual Studio 2012 support, 4
 - working with important styles, 159–160
 - CSS3 Color module, 138
 - CSS3 Media Queries module, 138
 - CSS3 Namespaces module, 138
 - CSS3 Selectors Level 3 module, 138
 - curly braces, 77, 81, 263–264
 - cursive font families, 173
 - cursors
 - applying key range limits, 597–599
 - described, 596–597
 - indexing, 597
 - opening, 596
 - curved lines, drawing, 481–487
 - custom attributes, 39
 - custom lists, 224–228
 - cx attribute (*<circle>* element), 498
 - cy attribute (*<circle>* element), 498
- D**
- d attribute (*<path>* element), 496
 - data
 - described, 67
 - expressions producing, 67
 - number types, 67–69
 - private, 268–271, 274–276
 - data attribute (*<object>* element), 51
 - data-linktype attribute (*<a>* element), 157–158
 - Database object
 - changeVersion() method, 583–584
 - described, 583
 - openDatabase() method, 583
 - readTransaction() method, 585, 587
 - transaction() method, 584–585, 587
 - version property, 583–584
 - <datagrid>* element, 33
 - <datalist>* element, 33
 - DataTransfer object
 - clearData() method, 514
 - described, 513–515, 521
 - dragging and dropping files, 519
 - dropeffect property, 514
 - effectAllowed property, 514–515
 - files property, 514
 - getData() method, 514
 - setData() method, 514
 - types property, 514
 - Date object, 67
 - dblclick event, 119
 - <dd>* element, 33, 223
 - DEBUG method (HTTP), 316
 - debugging JavaScript code
 - about, 103
 - examining variables, 104–105

setting breakpoints, 103–104
stepping through code, 105–107

declaration block, 139–140

declare attribute (*<object>* element), 322

default.css file, 15, 141

default.html file

- Blend for Visual Studio 2012, 15
- QUnit setup, 91
- Visual Studio 2012, 8–9, 11

default.js file

- Hello World example, 92
- jQuery example, 288, 291
- location of, 7
- media control, 450–451

defer attribute (*<script>* element), 101

deferred object. *See* promise object (jQuery)

** element, 33, 220

DELETE method (HTTP), 315–316, 365

deleting

- directories, 605–606
- files, 604
- records in object stores, 595

derived classes, 262

descendant selectors, 147–148

description lists, 223–224

<details> element, 33, 219–220

<dfn> element, 33, 218

<dialog> element, 33

dir global attribute, 37

directories

- creating, 604–605
- deleting, 605–606
- opening, 604–605
- writing files to, 605

DirectoryEntry object

- described, 602
- getDirectory()* method, 604–605
- getFile()* method, 602, 604
- remove()* method, 605
- removeRecursively()* method, 606

disabled Boolean attribute, 36

distances, calculating, 548–549

<div> element

- box-sizing property, 190–192
- centering content in browser window, 190–192
- clear property, 189–190
- control format using, 213
- described, 32–33, 178
- draggable attribute, 509
- <figure>* element and, 219
- float property, 186–189
- historical usage, 206
- in layout containers, 208
- positioning, 178–186
- role attribute, 212
- <table>* element and, 230

division (/) operator, 67–68, 70

<dl> element, 33, 37, 223–224

do loop, 85

<!DOCTYPE html> instruction, 8–9, 40

document object

- about, 112
- getElementById()* method, 113
- getElementsByClassName()* method, 113
- getElementsByName()* method, 113
- getElementsByTagName()* method, 113
- querySelector()* method, 113
- querySelectorAll()* method, 113–114

document object model (DOM)

- creating jQuery wrappers, 294–295
- described, 112
- navigating, 112–114
- preventing default operation, 116
- <script>* elements and, 103
- this* keyword, 117
- working with events, 114–120

dollar sign (\$), 72–73, 157, 288

DOM (document object model)

- creating jQuery wrappers, 294–295
- described, 112

- navigating, 112–114
- preventing default operation, 116
- <script> elements and, 103
- this keyword, 117
- working with events, 114–120
- dot notation, 586
- double quotes ("), 41, 69
- downloading event, 610
- drag and drop operation
 - DataTransfer object, 513–515
 - described, 507
 - dragging process, 509–510
 - File API, 517–521
 - lesson summary and review, 516–517, 521–522, 536–537
 - practice exercises, 522–535
 - technique illustrated, 508–509
- drag event, 119, 510
- dragend event, 119, 510–511
- dragenter event, 119, 511–513
- draggable global attribute, 37, 509–510
- draggingEnded() function, 511
- dragleave event, 119, 511
- dragover event, 119, 511–513
- dragstart event, 119, 510–511, 513
- drawing arcs
 - using arc() method, 485–487
 - using arcTo() method, 481–485
- drawing circles, 485–487, 498–499
- drawing curved lines, 481–487
- drawing state
 - configuring, 465–474
 - saving and restoring, 474–475
- drawing text, 488–490
- drawing triangles, 478–481
- drawing using paths
 - creating shapes, 475
 - described, 475
 - drawing arcs with arc method, 485–487
 - drawing arcs with arcTo method, 481–485
- drawing lines, 476–478
- drawing rectangles, 478–479
- ordering fill and stroke method calls, 479–481
- drawing with HTML5
 - configuring drawing state, 465–474
 - described, 459
 - drawing arcs, 481–487
 - drawing rectangles, 463–464
 - drawing text, 488–490
 - drawing using paths, 475–487
 - drawing with images, 490–494
 - implementing the canvas, 462–463
 - lesson summary and review, 494–495, 501–502, 506
 - practice exercises, 502–505
 - using <canvas> element, 460, 462–463
 - using CanvasRenderingContext2D object, 460–494
 - using scalable vector graphics, 495–501
- drawing with images, 490–494
- drop event, 119, 511–515
- dropItem() function, 512, 515, 519
- <dt> element, 33, 223
- durationchange event, 119

E

- ECMA-262 specification, 66
- ECMAScript, 66
- element type selectors, 146
- elements
 - adding attributes to, 35–36
 - adding comments, 39–40
 - described, 32–35
 - expando attributes, 39
 - floating, 186–189
 - HTML5 global attribute reference, 37
 - inherit value, 163
 - inheriting styles, 162–163

language, 220–221
nesting, 179, 210
reference list of, 32–35
self-closing tags, 37
tags and, 32
void, 38–39
working with styles, 138–140

else keyword, 80–81

**** element, 33, 213–214

em measurement unit, 174

email messages, hyperlinks in, 47

<embed> element

- described, 33
- embedding plug-in content, 50
- height attribute, 50
- src attribute, 50
- type attribute, 50
- width attribute, 50

Embedded Open Type (.eot) files, 144

embedded styles, 140–141

embedding content. *See content embedding*

emptied event, 119

encapsulation

- described, 262–263
- implementing, 268–269

encrypting web communications, 366

ended event, 119

&entity_name;, 41

&#entity_number;, 41

Entry object, 604–605

environment, creating for variables, 73

.eot (Embedded Open Type) files, 144

error event, 117, 119, 610

error handling

- AJAX calling web service example, 372–373
- for Geolocation object, 543–544
- for JavaScript code, 87–88
- for promise objects, 397

QuotaExceededError exception, 562

validation errors, 325

escape sequences, 69

event bubbling, 114–115, 567

event capturing, 114–115

event handlers, 295

event listeners, 295

Event object

- described, 114
- preventDefault() method, 116
- stopPropagation() function, 116

events

- bubbling, 114–115, 567
- canceling, 567
- canceling propagation, 116
- described, 114
- sessionStorage and, 568
- subscribing to, 115–116, 567
- triggered by drag and drop operation, 510–513
- triggered by form actions, 118
- triggered by keyboard, 118
- triggered by media, 119–120
- triggered by mouse actions, 118–119
- triggered by Window object, 117
- unsubscribing from, 116
- W3C recommendations, 566

exception handling, 87–88

expando attributes, 39

express framework (Node.js)

- adding webpage to applications, 357–360
- creating Hello web application, 356–357
- described, 354
- getting started, 354–355
- parsing posted form data, 360–362
- Visual Studio 2012 Express for Web, 356

expressions

- described, 67
- function, 75

eXtensible Markup Language (XML), 30–31

external JavaScript files, 102

- external style sheets
 - described, 141
 - specifying character encoding of, 143
 - specifying target devices using media, 141–142
 - using @font-face rule to import fonts, 144

F

- F5 function key, 104
- F9 function key, 103
- F10 function key, 105
- F11 function key, 105
- F12 function key, 314
- factory pattern, 265–266
- fantasy font families, 173
- <*fieldset*> element, 33
- <*figcaption*> element, 219
- <*figure*> element
 - annotating content, 219
 - <*aside*> element and, 219
 - described, 33, 219
 - <*div*> element and, 219
- File API, 517–521
- File object
 - described, 517–521
 - name property, 517
 - size property, 517
 - type property, 517
- FileEntry object
 - deleting files, 604
 - file() method, 603
 - reading files, 603
 - remove() method, 604
 - writing to files, 602
- FileList object, 517–521
- FileReader object
 - described, 603
 - readArrayBuffer() method, 604
 - readAsDataURL() method, 604

- readAsText() method, 603–604
- files
 - creating, 602
 - deleting, 604
 - dragging and dropping, 517–521
 - opening, 602
 - reading, 603–604
 - writing to, 602–603
 - writing to directories, 605
- Filesystem API
 - browser support, 559, 601
 - creating and opening directories, 604–605
 - creating and opening files, 602
 - deleting directories, 605–606
 - deleting files, 604
 - described, 600
 - lesson summary and review, 606–607, 618
 - opening file system, 601
 - reading files, 603–604
 - as storage mechanism, 558
 - writing files to directories, 605
 - writing to files, 602–603
- FileWriter object
 - described, 603–604
 - write() method, 602
- fill attribute
 - <*circle*> element, 498
 - <*path*> element, 496
- finally block, 87
- Firefogg website, 439–440
- ::first-letter pseudo element, 150
- ::first-line pseudo element, 150
- ::first-of-type pseudo class, 149
- Fixed Layout App template, 7
- fixed position (<*div*> element), 183–184
- Flash Player (Adobe), 557
- float property (CSS), 186–189
- focus event, 117–118
- ::focus pseudo class, 149
- @font-face rule, 144

- font families, 173–174
- font-family property (CSS), 174
- font licensing, 144
- font-size property (CSS), 174–175
- font typeface, 173–174
- <footer> element
 - <cite> element and, 215
 - described, 33, 208
 - in layout containers, 208, 210–211
- for attribute (<label> element), 318
- for loop, 85–86
- <form> element
 - described, 33
 - required validation, 325–327
 - submitting data to web servers, 316
 - triggering form submission, 319
 - validating numbers and ranges, 329
 - validating URL input, 327–328
- form attribute
 - form submission elements, 319
 - <object> element, 51
- form class, 212
- form submission elements
 - data submission constraints, 322
 - described, 316–318
 - form attribute, 319
 - placeholder attribute, 326
 - required attribute, 325
- formatting text
 - controlling with <div> element, 213
 - CSS properties, 173–175
- formchange event, 118
- formidable package, 360
- forminput event, 118
- forms
 - autofocus attribute, 321
 - data submission constraints, 322
 - described, 311–312
 - events triggered by, 118
- GET method, 322–323
- lesson summary and review, 323–324, 330–331, 338–339
- parent, 319
- POST method, 322–323
- practice exercises, 331–337
- sending data when submitting, 316–318
- serializing, 321
- submitting form data to web servers, 316
- triggering submission, 319–321
- using <label> element, 318–319
- validating, 324–330
 - web communication and, 312–316
- forward compatibility of browsers, 30
- forward slash (/) character, 337
- fs package, 348
- function declarations, 74, 78
- function expressions, 75
- Function object, 67
- functions. *See also* specific functions
 - aggregating, 588
 - arguments to, 73–76
 - built-in, 76–77
 - constructor, 266–271
 - declaring, 74, 78
 - described, 67, 73
 - nesting, 78
 - parameters in, 73–74
 - return values, 73
- furigana phonetic characters, 220–221
- future object. *See* promise object (jQuery)

G

- generalized classes, 262
- Geolocation API
 - basic positioning, 540–545
 - described, 539
 - lesson summary and review, 545–546, 549, 554

monitored positioning, 546–549
practice exercises, 550–553

Geolocation object
addressing privacy, 544
basic positioning, 540–545
calculating distance between samples, 548–549
clearWatch() method, 540, 546
getCurrentPosition() method, 540–545
handling errors, 543–544
lesson summary and review, 545–546, 549, 554
monitored positioning, 546–549
practice exercises, 550–553
retrieving current position, 541–542
specifying options, 544–545
watchPosition() method, 540, 546–547, 549

Geolocation Working Group, 539

GET method (HTTP)
CRUD operations and, 316, 365
described, 315, 322–323
REST support, 365–366, 368

Get Windows Azure SDK For .NET template, 10

getDistance() function, 548–549

getter methods, 274–275

GIF file type, 48–49

git source control manager, 350

global attributes, 37

global objects, 67

Global Positioning System (GPS), 539, 541

global scope, 77–78

global variables, 77–78

Go To Definition feature, 4

Google Gears, 557

GPS (Global Positioning System), 539, 541

gradient fill, 466–468

greater-than sign (>), 41, 148

Grid App template, 6

grouping selectors, 150–151

H

<h1> element, 33
<h2> element, 33
<h3> element, 33
<h4> element, 33
<h5> element, 34
<h6> element, 34
H.264 (MPEG-4) format, 438
handling errors. *See* error handling
haschange event, 117
hash (#) symbol
id selectors and, 146
internal hyperlinks and, 46
jQuery support, 289
in manifest file, 609
in RGB values, 166
haversine formula to calculate distances, 548–549

<head> element
described, 9, 34, 40
<script> element and, 102–103
<style> element and, 140

HEAD method (HTTP), 315

<header> element, 34, 208

height attribute
<canvas> element, 460
<embed> element, 50
<object> element, 51
<video> element, 441

Hello World program
creating from JavaScript, 90–99
creating from Node.js, 342–344
creating with express, 356–357

hexadecimal values, color names table, 167–171

<hgroup> element, 210

hidden global attribute, 37

.hidden selector, 237

<hn> element, 209–210, 213

hole variables, 512

:hover pseudo class, 149, 154–155

<hr> element
 adding thematic breaks, 213
 described, 34
 as void element, 38

href attribute
 <a> element, 46, 153–157
 <area> element, 49
 <link> element, 141

hsl() function, 172–173

HSL (hue-saturation-lightness), 172–173

HTML (Hypertext Markup Language)
 browser support, 314
 CSS support, 138
 described, 30–32

<html> element
 described, 9, 34, 40
 manifest attribute, 608

HTML documents
 adding images to, 47–50
 basic document structure, 40–41
 documenting code in, 216–217
 embedded styles and, 140–141
 embedding webpages in, 51
 no-quirks mode, 40
 normalizing, 217
 special characters, 41–42

HTML entities, 41–42

HTML forms. *See* forms

HTML5
 adding thematic breaks, 213
 annotating content, 213–221
 browser support, 206–207
 controlling format with <div> element, 213
 creating layout containers, 207–212
 described, 30–31
 drawing with, 459–506
 embedding content, 44–52
 lesson summary and review, 42–43, 52–53, 62–63, 228–229, 242–243, 258–259
 named attributes supported, 37

practice exercises, 53–61, 243–257

semantic markup, 31–32, 206

Visual Studio 2012 support, 3–4

working with elements, 32–40

working with lists, 221–228

HTML5 documents
 creating semantic, 207
 documenting code in, 216–217
 normalizing, 217

HTML5 layout containers. *See* layout containers

HTMLMediaElement object
 addTextTrack() method, 447
 audioTracks property, 447
 autoplay property, 447
 buffered property, 448
 canPlayType() method, 447
 controller property, 448
 controls property, 448
 crossOrigin property, 448
 currentSrc property, 448
 currentTime property, 448
 defaultMuted property, 448
 defaultPlaybackRate property, 448
 described, 447
 duration property, 448
 ended property, 448
 error property, 448
 inheritance and, 444, 447
 lesson summary and review, 451–452
 load() method, 447
 loop property, 448
 media control, 450–451
 mediaGroup property, 448
 muted property, 448
 networkState property, 448
 onabort event, 449
 oncanplay event, 449
 oncanplaythrough event, 449
 ondurationchange event, 449
 onemptied event, 449

- onended event, 449
- onerror event, 449
- onloadeddata event, 449
- onloadedmetadata event, 449
- onloadstart event, 449
- onpause event, 449
- onplay event, 449
- onplaying event, 449
- onprogress event, 449
- onratechange event, 449
- onreadystatechange event, 449
- onseeked event, 449
- onseeking event, 449
- onstalled event, 449
- onsuspend event, 449
- ontimeupdate event, 449
- onvolumechange event, 449
- onwaiting event, 449
- pause() method, 447
- paused property, 448
- play() method, 447
- playbackRate property, 448
- played property, 448
- preload property, 448
- readyState property, 448
- seekable property, 448
- seeking property, 448
- src property, 448
- startDate property, 448
- textTracks property, 449
- videoTracks property, 449
- volume property, 449
- HTTP cache**
 - browser support, 608
 - cache manifest file, 608–609
 - described, 607
 - updating, 609
- HTTP cookies, 556–557
- HTTP (Hypertext Transfer Protocol)**
 - described, 314–315
- HTTP() method, 315–316, 322–323
- REST web services and, 365–366
- return status codes, 315
- web servers, 312
- WebSocket protocol and, 416
- HTTP methods, 315–316, 322–323
- http module, 342
- HTTP verbs, 315–316, 322–323
- HTTPS protocol, 366
- hue-saturation-lightness (HSL), 172–173
- hyperlinks
 - creating image links, 49
 - described, 46
 - <i> element and, 44
 - sending email with, 47
 - specifying target, 46–47
 - styles working with, 138
 - working with events, 114–115
 - working with selectors, 153–158
- Hypertext Markup Language (HTML)
 - browser support, 314
 - CSS support, 138
 - described, 30–32
- Hypertext Transfer Protocol. *See* HTTP (Hypertext Transfer Protocol)

I

- <i> element, 31–32, 34, 213–214
- id attribute
 - <circle> element, 498
 - as global attribute, 37
 - <path> element, 496
 - <video> element, 450
- id selectors, 146
- IDBCursor object, 596
- IDBDatabase object
 - createObjectStore() method, 591
 - deleteObjectStore() method, 593

described, 590
mode property, 593
objectStoreNames property, 593
setVersion() method, 591
transaction() method, 593
IDBFactory object
 deleteDatabase() method, 599
 open() method, 590–591
IDBKeyRange object
 bound() method, 597–598
 described, 597
 lower property, 597–598
 lowerBound() method, 598–599
 lowerOpen property, 598
 only() method, 599
 upper property, 597–598
 upperBound() method, 598
 upperOpen property, 598
IDBOObjectStore object
 add() method, 594
 createIndex() method, 592
 delete() method, 595
 deleteIndex() method, 593
 get() method, 595
 keypath property, 592
 name property, 592
 openCursor() method, 596–597
 optionalParameters property, 592
 put() method, 594
IDBRequest object, 590
IDBTransaction object, 593
identifiers, 74
IETF (Internet Engineering Task Force), 415
if keyword, 80–81, 83
<iframe> element
 creating nested browser context, 34
 described, 34, 44–46
 name attribute, 44–45
<object> tag and, 51
sandbox attribute, 45
seamless attribute, 45–46
src attribute, 44
working with hyperlinks, 46–47
IIFE (immediately invoked function expression), 277–279
IIS Express, 442
IIS (Internet Information Server), 323, 608
image maps, 49–50
images
 adding to HTML documents, 47–50
 drawing with, 490–494
 styles working with, 138
** element
 alt attribute, 47–48
 creating image links, 49
 creating pattern fill, 468–469
 described, 30, 34
 drawing with images, 490
 embedding content, 47–48, 51
 file types supported, 48
 src attribute, 47–48
 usemap attribute, 49
 as void element, 38
immediately invoked function expression (IIFE), 277–279
@import rules, 143–144
importing
 fonts, 144
 style sheets, 143–144
in layout containers, 208–210
in measurement unit, 175
index numbers (arrays), 108–109
IndexedDB tool
 browser support, 559, 590
 creating and opening database, 590–591
 described, 589–590
 lesson summary and review, 599–600, 617–618
 object stores, 591–593
 practice exercises, 611–616
 as storage mechanism, 558

indexing cursors, 597
inherit value, 163
inheritance
 described, 162–163
 HTMLMediaElement object and, 444, 447
 JavaScript objects and, 262, 278–283
inline frames. *See <iframe> element*
inline JavaScript code, 102
inline styles, 140
INNER JOIN statement, 588
<input> element
 described, 34
 form validation and, 326
 max attribute, 329
 min attribute, 329
 pattern attribute, 327–328
 step attribute, 329
 title attribute, 328
 type attribute, 317–318
 as void element, 38
input event, 118
<ins> element, 34, 220
installing
 Node.js packages, 351–353
 Node.js platform, 342
instances, 262
integer numbers, 67
IntelliSense feature
 custom lists and, 226
 enabling for JavaScript and jQuery, 291–294
 HTML5 support, 3
 JavaScript support, 4
 untyped languages and, 66
Internet Engineering Task Force (IETF), 415
Internet Information Server (IIS), 323, 608
invalid event, 118
:invalid pseudo class, 330
irregular tables, 238–241
isNaN() function, 80–81
ISO/IEC 16262 standard, 66

J

Japanese characters, 220–221
Java Applets, 557
JavaScript. *See also* jQuery
 AJAX and, 368
 browser considerations, 101–102
 CommonJS support, 394
 conditional programming, 80–84
 converting to different types, 78–80
 creating Hello World program, 90–99
 creating objects, 262–283
 debugging, 103–107
 described, 66, 263
 enabling IntelliSense, 291–294
 handling errors, 87–88
 implementing code loops, 84–87
 inline versus external files, 102
 lesson summary and review, 88–89, 107–108, 120–121, 134–135
 naming variables, 72
 nested local variable scoping, 78
 nesting functions, 78
 practice exercises, 121–133
 role of data, 66–71
 scoping variables, 77–78
 <script> element and, 100–103
 Socket.IO library and, 424
 testing code, 89–103
 as untyped language, 66
 using statements, 71–73
 Visual Studio 2012 support, 4–5
 working with functions, 73–77
 writing code, 89–103
JavaScript Console, 67
JavaScript Object Notation (JSON), 349, 366, 374
JavaScript objects
 built-in, 67
 creating, 263–264
 creating classes, 266–271

- described, 262–263
- factory pattern, 265–266
- implementing inheritance, 278–283
- implementing namespaces, 276–278
- lesson summary and review, 283–284, 308
- object literal syntax, 263–265
- practice exercises, 297–307
- prototype pattern, 271
- prototype property, 271–274
- JetBrains Resharper tool, 67
- JOIN commands (SQL), 588
- joining lines, 471–472
- JPEG file type, 48
- jQuery
 - adding event listeners, 295
 - autofocus attribute, 321
 - binding to storage events, 567–568
 - categories of functionality, 285–286
 - cookie plug-in, 556–557
 - creating wrappers, 294–295
 - DataTransfer object and, 513, 515, 519
 - drag and drop operations, 507
 - enabling IntelliSense, 291–294
 - getting started with, 286–287
 - initializing code when browser is ready, 295–296
 - lesson summary and review, 296–297, 308–309
 - promise objects, 377–380, 394–402
 - serialize() method, 321
 - submit() method, 319–320
 - triggering event handlers, 295
 - usage considerations, 287–291
 - verifying versions, 369
 - XMLHttpRequest wrappers, 373–377
- jQuery object
 - ajax() method, 373–375
 - ajaxCompleted() method, 373
 - ajaxError() method, 373
 - ajaxPrefilter() method, 373
 - ajaxSend() method, 373
 - ajaxSetup() method, 373
 - ajaxStart() method, 373
 - ajaxStop() method, 373
 - ajaxSuccess() method, 373
 - get() method, 373
 - getJSON() method, 373, 375
 - getScript() method, 373
 - load() method, 373
 - param() method, 373
 - post() method, 373, 375
 - serialize() method, 374
 - serializeArray() method, 374
 - text() method, 295
 - trigger() method, 295
 - triggerHandler() method, 295
- JSON (JavaScript Object Notation), 349, 366, 374
- JSON object, 67, 368
- junction, 352

K

- kanji characters, 220–221
- <*kbd*> element, 34, 220
- keepAlive() function, 422
- keyboard actions, events triggered by, 118
- keydown event, 118
- <*keygen*> element, 38
- keypress event, 118
- keyup event, 118
- Korean characters, 220–221

L

- <*label*> element
 - for attribute, 318
 - described, 34, 318
 - forms and, 318–319
- landmark role class, 212
- lang global attribute, 37

:lang(language) pseudo class

- :lang(language) pseudo class, 149
- language elements, 220–221
- last-in, first-out (LIFO), 474
- layout containers
 - <article> element in, 209
 - <aside> element in, 208, 211–212
 - creating, 207–209
 - described, 208
 - <div> element in, 208
 - <footer> element in, 208, 210–211
 - <header> element in, 208–210
 - <nav> element in, 208, 211
 - <section> element in, 208
 - using roles, 212
- LEFT JOIN statement, 588
- <legend> element, 34
- less-than sign (<), 41
- element, 34
- licensing fonts, 144
- LIFO (last-in, first-out), 474
- lightness (HSL), 172–173
- lines
 - drawing, 476–478
 - drawing curved, 481–487
 - joining, 471–472
 - setting width for, 470–471
- <link> element
 - described, 34
 - href attribute, 141
 - media attribute, 141–142
 - rel attribute, 141
 - type attribute, 141
 - as void element, 38
- :link pseudo class, 149
- list-item-style property (CSS), 226
- list-style-image property (CSS), 226
- list-style-position property (CSS), 226
- list-style-type property (CSS), 225–226
- lists
 - custom, 224–228
 - described, 221
 - description, 223–224
 - ordered, 221–222
 - unordered, 222–223
- literal arrays, 109
- live NodeList, 112–113
- load event, 117
- loadeddata event, 120
- loadedmetadata event, 120
- loadstart event, 120
- local functions, 77–78
- local scope, 77–78
- local variables, 77–78
- Local window (debugger), 104–105
- localStorage global variable, 560–564
- location awareness. *See* Geolocation API
- logical operators, 70–72
- long polling concept, 415–416
- loop attribute
 - <audio> element, 445–446
 - <video> element, 441
- loops. *See* code loops

M

- mailto protocol, 47
- main class, 212
- Manage NuGet Packages window, 90–91, 95–96
- manifest attribute (<html> element), 608
- manifest file
 - CACHE section, 609
 - described, 608–609
 - FALLBACK section, 609
 - NETWORK section, 609
 - Node.js package and, 346
 - updating cache, 609
- <map> element
 - creating image maps, 49–50
 - described, 34
 - name attribute, 49

margin properties (CSS), 176–178, 193
`<mark>` element, 34, 213, 220
Markdown files, 348
MarkdownPad editor, 348
mashups, 364
Math object, 67
mathematical operators, 68, 72
max attribute (`<input>` element), 329
maxlength attribute (`<textarea>` element), 316
measurement units, font-size, 174–175
media. *See* multimedia
media attribute (`<link>` element), 141–142
`<menu>` element, 34
message event, 117
`<meta>` element
 @charset setting, 41, 143
 described, 34
 as void element, 38
`<meter>` element, 34
methods
 array, 110–112
 described, 67, 110, 263
 getter, 274–275
 privileged, 269
 setter, 274
Microsoft App Fabric Caching Service, 422
Microsoft Visual Studio 2012. *See* Visual Studio 2012
Microsoft Web Embedding Fonts Tool (WEFT), 144
min attribute (`<input>` element), 329
Miro Video Converter, 439–440
mm measurement unit, 175
modernizr.js library, 221
Modify Style window, 226
modulo (%) operator, 69
monospace font families, 173
mouse action, events triggered by, 118
mousedown event, 119

mousemove event, 119
mouseout event, 119
mouseover event, 119
mouseup event, 119
mousewheel event, 119
movies. *See* video and movies
.mp3 file extension, 444–445
MP3 format, 444
.mp4 file extension, 438–439, 444
MP4 format, 444
.mp4a file extension, 444
MPEG-4/H.264 format, 438
multimedia
 events triggered by, 119–120
 HTMLMediaElement object, 447–452
 lesson summary and review, 442–443, 446–447, 451–452, 456–457
 playing audio, 443–447
 playing video, 437–443
 practice exercises, 452–455
 specifying target devices using, 141–142
multiple attribute (`<select>` element), 316–317
multiplication (*) operator, 67–68, 70
music. *See* audio and sounds
muted attribute (`<video>` element), 441
MVC technologies, 341

N

\n escape sequence, 69
name attribute
 `<iframe>` element, 44–45
 `<map>` element, 49
 `<object>` element, 51
named styles, 146–147
namespaces
 CSS3 support, 138
 JavaScript objects and, 276–278
 jQuery, 288

naming variables, 72–73
<nav> element, 34, 208, 211
navigating DOM, 112–114
Navigation App template, 7
navigation class, 212
navigator.geolocation global variable, 540
nesting
 elements, 179, 210
 functions, 78
 operations, 398–399
new keyword, 108, 269
no-quirks mode, 40
no value coalescing operators, 83–84
node package manager (npm), 342
Node.js platform
 creating Hello World program, 342–344
 creating Node.js module, 344–345
 creating Node.js package, 345–354
 creating RESTful web service, 366–368
 described, 341
 express framework, 354–363
 installing, 342
 lesson summary and review, 363, 391
 practice exercises, 382–386, 390
 Socket.IO library and, 424
nonbreaking space, 41–42
Nonvisual Desktop Access (NVDA) devices, 206–207
normalizing HTML documents, 217
<noscript> element, 34
not (!) logical operator, 70–71
:not pseudo class, 149
noupdate event, 610
npm (node package manager), 342
:nth-child(formula) pseudo class, 149
NuGet package-management system, 90, 95, 286
null primitive type, 67
Number() function, 78–79
Number object, 67

number primitive type
 arithmetic operations, 68
 described, 67
 operator precedence, 68–69
 special values supported, 68
number types, 67–69
numbers, validating in forms, 329
NVDA (Nonvisual Desktop Access) devices, 206–207

O

<object> element
 creating nested browser context, 52
 data attribute, 51
 declare attribute, 322
 described, 34, 51
 embedding plug-in content, 50–52
 form attribute, 51
 height attribute, 51
 name attribute, 51
 <param> tag and, 52
 passing parameters to objects, 52
 type attribute, 51
 usemap attribute, 51
 width attribute, 51
object literals, 263–265
Object object
 constructor() method, 265
 described, 67
 hasOwnProperty() method, 265
 isPrototypeOf() method, 265
 propertyIsEnumerable() method, 265
 prototype property, 271–274
 toLocaleString() method, 265
 toString() method, 265
 valueOf() method, 265
object-oriented programming
 JavaScript caveat, 263

- private data and, 268
- terminology used, 262–263
- object stores
 - adding indexes, 592–593
 - deleting records, 595
 - described, 591
 - inserting new records, 594
 - removing, 593
 - removing indexes, 593
 - retrieving records, 595–596
 - understanding cursors, 596–599
 - updating existing records, 594–595
 - using transactions, 593–594
 - versioning and, 591
- objects. *See also* JavaScript objects
 - accessing DOM objects, 112–120
 - described, 262
 - lesson summary and review, 120–121, 135
 - passing parameters to, 52
 - types of, 262
 - working with arrays, 108–112
- obsolete event, 610
- offline event, 117
- offline web applications
 - described, 581
 - FileSystem API and, 600–606
 - HTTP cache and, 607–610
 - IndexedDB tool and, 589–599
 - lesson summary and review, 588–589, 599–600, 606–607, 610–611, 617–619
 - practice exercises, 611–616
 - Web SQL database and, 582–588
- .oga file extension, 444–445
- .ogg file extension, 444
- Ogg/Theora format, 438
- Ogg/Vorbis format, 444
- .ogv file extension, 438–439
- element, 34, 221–222
- onabort event, 449
- oncanplay event, 449
- oncanplaythrough event, 449
- onclose event, 417, 419
- ondurationchange event, 449
- onemptied event, 449
- onended event, 449
- onerror event
 - FileEntry object and, 604
 - FileReader object and, 603
 - FileWriter object and, 603
 - HTMLMediaEvent object and, 449
 - IDBObjectStore object and, 594
 - IDBRequest object and, 591
 - WebSocket object and, 417, 419
- online event, 117
- onload event, 469
- onloadeddata event, 449
- onloadedmetadata event, 449
- onloadend event, 603
- onloadstart event, 449
- :only-child pseudo class, 149
- :only-of-type pseudo class, 149
- onmessage event, 417, 419
- onopen event, 417, 419
- onpause event, 449
- onplay event, 449
- onplaying event, 449
- onprogress event, 449
- onratechange event, 449
- onreadystatechange event, 371, 449
- onseeked event, 449
- onseeking event, 449
- onstalled event, 449
- onsuccess event, 591, 594
- onsuspend event, 449
- ontimeupdate event, 449
- onupgradeneeded event, 591
- onvolumechange event, 449
- onwaiting event, 449
- onwriteend event, 603
- opacity property (CSS), 172

operands
binary operators and, 70
described, 67–68
operator precedence, 68–69
<optgroup> element, 34
<option> element
described, 34, 317
selected attribute, 317
triggering form submission, 320
value attribute, 320
`:optional` pseudo class, 330
OPTIONS method (HTTP), 315
or (||) logical operator, 70–71, 83–84
ordered lists, 221–222
<output> element, 34

P

<p> element, 34
package.json file, 349–350
packages (Node.js)
creating aggregate modules, 348
creating package.json file, 349–350
creating README.md file, 348–349
described, 345–348
installing and using, 351–353
publishing, 350–351
uninstalling, 354
packages.config file, 96
padding properties (CSS), 176–178
Page Inspector feature, 3
pagehide event, 117
pageshow event, 117
Parallel Watch window (debugger), 106
<param> element
described, 34, 52
<object> element and, 52
as void element, 39

parameters
arguments versus, 74
described, 73–74
handling errors via, 543
passing to objects, 52
parent classes, 262
parent forms, 319
parentheses (), 68–69, 74
<path> element
commands supported in, 496–497
`d` attribute, 496
described, 496
`fill` attribute, 496
`id` attribute, 496
path package, 348
paths
commands supported in, 496–497
creating, 496–498
described, 475, 496
drawing using, 475–487
pattern attribute (*<input>* element), 327–328
pattern fill, 468–469
pause event, 120
pc measurement unit, 175
performance considerations
POST method, 323
storage mechanisms, 563–564
universal selector, 147
period (.) symbol, 146, 327
.pfx file extension, 9
Pin To Source option, 106
placeholder attribute
form submission elements, 326
<textarea> element, 316
play event, 120
playing event, 120
playStop() function, 451
plug-ins
cookie alternatives, 557
described, 31

- embedding content from, 50–52
- limitations of, 558
- plus sign (+), 67–69, 151, 327
- PNG file type, 48
- polymorphism, 262
- popstate event, 117
- Position object**
 - coords property, 540–541
 - timestamp property, 541
- PositionOptions object**
 - described, 544
 - enableHighAccuracy property, 544
 - maximumAge property, 545
 - timeout property, 544
- POST method (HTTP), 315–316, 322–323, 365–366
- poster attribute (<*video*> element), 441
- <*pre*> element, 34, 217
- precedence order
 - for element styles, 160–161
 - for operators, 68–69
- preformatted content, displaying, 217
- preload attribute
 - <*audio*> element, 446
 - <*video*> element, 441
- preventDefault() function, 512
- primitive values, 67
- private data
 - JavaScript objects and, 268–271
 - prototype pattern and, 274–276
- privileged methods, 269
- <*progress*> element, 34
- progress event, 120, 371, 610
- Promise/A specification, 394
- promise object (jQuery)
 - always() method, 378, 397–399
 - asynchronous operations and, 394–395
 - chaining promises, 398–400
 - conditional calls, 401–402
 - creating, 395–397
 - Deferred() method, 395–397
- described, 377–380, 394–395
- done() method, 378, 398–399
- fail() method, 378, 397, 399
- handling completion cleanup, 397–398
- handling failure, 397
- parallel execution and, 400
- pipe() method, 398–400
- progress() method, 378, 400–401
- subscribing to completed, 398
- then() method, 400, 402
- timeouts, 396–397
- updating progress, 400–401
- when() method, 400, 402
- prompt() function**, 76–77
- properties**. *See also* CSS properties
 - array, 109–110
 - described, 109, 262–263
- Properties window**, 7, 11
- prototype pattern, 271, 274–276
- prototype property, 271–274
- prototypes, 263
- pseudo-class selectors, 148–149
- pseudo classes, 148–149, 330
- pseudo-element selectors, 148–150
- pseudo elements, 149–150
- pt measurement unit, 175
- publisher-subscriber design pattern, 114
- publishing packages, 350–351
- PUT method (HTTP), 315–316
- px measurement unit, 174

Q

- <*q*> element
 - annotating content, 215–216
 - cite attribute, 215
 - described, 34
- QueryString**
 - form submissions and, 320–323
 - Node.js and, 343, 359–360

REST web services and, 365, 368
question mark (?), 322, 327
QUnit-Metro tool, 95–100
QUnit tool, 90–95, 286
qunitmetro.js file, 96
QuotaExceededError exception, 562
quotations and citations, 215–216

R

r attribute (*<circle>* element), 498
radio frequency identification (RFID), 539
ranges, validating in forms, 329
ratechange event, 120
Rauch, Guillermo, 423
reading files, 563, 603–604
README.md file, 348–349
readonly Boolean attribute, 36
readystatechange event, 120
rectangles
 configuring drawing state, 465–474
 drawing, 463–464, 478–479
 saving and restoring drawing state, 474–475
 setting fillStyle property, 465–470
 setting lineJoin property, 471–472
 setting lineWidth property, 470–471
 setting strokeStyle property, 472–474
Redis (remote dictionary service), 422
redo event, 117
Reference Groups feature, 5
refreshing screens, 290
RegExp object, 67
registered trademark (®), 41
rel attribute (*<link>* element), 141
relative position (*<div>* element), 181–182
removeEventListener() function, 116
Representational State Transfer (REST) web services
 creating, 366–368

 described, 364–366
request/response model
 long polling concept, 415
 Node.js and, 357
 in stateless model, 312–313
require() function, 345
required Boolean attribute, 325
:required pseudo class, 330
Resharper tool, 67
resize event, 117
resource sharing, cross-origin, 380–381
REST (Representational State Transfer) web services
 creating, 366–368
 described, 364–366
retries variable, 85–86
return status codes (HTTP), 315
return values (functions), 73
RFC 6455, 415
RFID (radio frequency identification), 539
rgb() function, 171
RGB value, 166–171
rgba() function, 172
role attribute
 <aside> element, 212
 <div> element, 212
rowspan attribute
 <td> element, 238–240
 <th> element, 238–240
<rp> element, 34, 221
<rt> element, 34, 221
<ruby> element, 35, 221
ruby phonetic characters, 220–221

S

<s> element, 220
<samp> element, 35, 216–217
sans serif font families, 173–174

- sandbox attribute (*<iframe>* element), 45
- sandboxing, 45, 51
- saturation (HSL), 172–173
- saveData() function, 116
- scalable vector graphics (SVG)
 - described, 459, 495–496
 - using ** element, 499–501
 - using *<svg>* element, 496–499
- schemas
 - tables and, 591
 - updating, 583
- scoping variables, 77–78
- screens, refreshing, 290
- <script>* element
 - async attribute, 101
 - defer attribute, 101
 - described, 35, 236
 - placing, 102–103
 - type attribute, 100
 - usage considerations, 38, 100–101
- scroll event, 119
- seamless attribute (*<iframe>* element), 45–46
- search capabilities, 563–564
- search class, 212
- <section>* element, 35, 208
- sectioning root, 215
- security
 - Geolocation object and, 544–545
 - HTTPS protocol and, 366
 - storage mechanisms and, 558–559
- seeked event, 120
- seeking event, 120
- <select>* element
 - described, 35, 316–317
 - multiple attribute, 316–317
 - size attribute, 317
- select event, 118
- SELECT statement (SQL)
 - aggregating functions and, 588
 - JOIN commands and, 588
- reading values from tables, 586–587
- WHERE clause, 587
- selected attribute (*<option>* element), 317
- selected Boolean attribute, 36
- selector chains, 147
- selectors
 - adjacent, 151–152
 - attribute, 153–154
 - attribute contains value, 155–156
 - attribute contains value in list, 157–158
 - attribute value, 154–155
 - attribute value ends with, 157
 - child, 148
 - class, 146–147
 - CSS3 support, 138
 - custom lists and, 224–226
 - defining, 146
 - descendant, 147–148
 - described, 138–139
 - element type, 146
 - grouping, 150–151
 - id, 146
 - jQuery-supported, 289
 - pseudo-class, 148–149
 - pseudo-element, 148–150
 - sibling, 151–153
 - specificity in, 161–162
 - subsequent, 151–153
 - universal, 147
- self-closing tags, 37
- semantic markup, 31–32
- semicolon (;), 71, 139
- serializing forms, 321
- serif font families, 173–174
- sessionStorage global variable, 562–564, 568
- setter methods, 274
- SGML (Standard Generalized Markup Language), 30, 137
- shape attribute (*<area>* element), 49
- short-circuit evaluation, 71

- sibling selectors**, 151–153
- SignalR library**, 423
- Simple Object Access Protocol (SOAP)**, 366
- single quotes ('), 69**
- size attribute (<select> element)**, 317
- .sln file extension**, 356
- <small> element**, 35, 220
- SOAP (Simple Object Access Protocol)**, 366
- Socket.IO library**, 423–424
- sounds. See audio and sounds**
- <source> element**
 - described**, 35
 - setting audio source**, 445
 - setting video source**, 439–440
 - src attribute**, 439, 445
 - type attribute**, 439, 445
 - as void element**, 39
- element**
 - described**, 35
 - expando attributes and**, 39
 - historical usage**, 206
- special characters (HTML entities)**, 41–42
- specialized classes**, 262
- specificity (selectors)**
 - calculating**, 161–162
 - in cascading styles**, 160
- spellcheck global attribute**, 37
- Split App template**, 6
- square brackets**, 109, 264
- src attribute**
 - <audio> element**, 446
 - <embed> element**, 50
 - <iframe> element**, 44
 - element**, 47–48
 - <source> element**, 439, 445
 - <video> element**, 441
- SRT (SubRip Text)**, 441
- stalled event**, 120
- Standard Generalized Markup Language (SGML)**, 30, 137
- start() function**, 345
- stateless model**, 312–313
- statements**, 71
- statements, variables and**, 71–73
- static NodeList**, 112–113
- static position (<div> element)**, 181
- step attribute (<input> element)**, 329
- stopPropagation() function**, 116
- storage event**, 117
- storage mechanisms**
 - browser support**, 561
 - capacity considerations**, 561–562
 - cookie considerations**, 556
 - described**, 555
 - handling storage events**, 565–568
 - HTML5-supported**, 558–560
 - jQuery cookie plug-in**, 556–557
 - lesson summary and review**, 564–565, 568–569, 579–580
 - localStorage global variable**, 560–562
 - potential performance pitfalls**, 563–564
 - practice exercises**, 569–578
 - sessionStorage global variable**, 562–563
 - storing complex objects**, 562
 - web storage**, 555–564
- Storage object**
 - clear() method**, 560
 - described**, 560, 563
 - getItem() method**, 560
 - key() method**, 561
 - length property**, 560
 - removeItem() method**, 560
 - setItem() method**, 560
- StorageEvent object**
 - described**, 566
 - key property**, 566
 - newValue property**, 566
 - oldValue property**, 566
 - storageArea property**, 566
 - url property**, 566

- String() function, 78, 80
- String object, 67
- string primitive type
 - described, 67, 69
 - unary operators, 70
- <*strong*> element, 35, 213–214
- <*style*> element, 35, 140
- style global attribute, 37, 140
- style sheets
 - adding comments within, 139
 - browser built-in styles, 159
 - described, 138
 - external, 141
 - imported, 143–144
 - user-defined, 159
- styles
 - applying, 139
 - in browsers, 139, 159
 - cascading, 160–161
 - creating, 138
 - defining, 139
 - described, 138
 - embedded, 140–141
 - extending, 159
 - inheriting, 162–163
 - inline, 140
 - named, 146–147
 - validating input, 330
 - working with important, 159–160
- <*sub*> element, 35, 220
- sub-paths, 475
- subclasses, 262
- submit event, 118
- SubRip Text (SRT), 441
- subscribing to events, 115–116, 567
- subsequent adjacent sibling selectors, 151–152
- subsequent sibling selectors, 152–153
- subtraction (-) operator, 67–68, 70
- <*summary*> element, 219–220
- <*sup*> element, 35, 220
- super classes, 262
- suspend event, 120
- <*svg*> element
 - creating a path, 496–498
 - described, 496
 - drawing circles, 498–499
 - viewBox attribute, 500–501
- svg-edit editor, 499
- SVG file type, 48, 499–501
- SVG (scalable vector graphics)
 - described, 459, 495–496
 - using <*img*> element, 499–501
 - using <*svg*> element, 496–499
- switch keyword, 82–83
- synchronous read/writes, 563

T

- \t escape sequence, 69
- tabindex global attribute, 37
- <*table*> element
 - creating tables, 230
 - described, 35, 205, 229
 - misuse of, 230
 - styling columns, 241–242
- table headers
 - creating, 231–232
 - styling, 232
- tables
 - adding captions, 241
 - creating, 230–231
 - creating header cells, 231–232
 - creating irregular, 238–241
 - declaring footers, 233–237
 - declaring headers, 233–237
 - declaring table body, 233–237
 - described, 229
 - lesson summary and review, 242–243, 258–259
 - misuse of, 230
 - schemas and, 591

tags

- styling columns, 241–242
- styling rows, 241
- styling table headers, 232
- in Web SQL databases, 584–588
- tags
 - case sensitivity, 32
 - described, 30–31
 - elements and, 32–35
 - self-closing, 37
 - semantic markup, 31–32
- target attribute (*<a>* element), 46–47
- <tbody>* element, 35, 233–237
- TCP
 - arbitrary web services and, 366
 - WebSocket support, 415–417
- <td>* element
 - colspan attribute, 238–240
 - creating tables, 230
 - described, 35, 229, 241
 - rowspan attribute, 238–240
- TDD (test-driven development), 90, 93, 98
- templates. *See also* specific templates
 - described, 5–6
 - included with Visual Studio Express for Web, 10–11
 - included with Visual Studio Express for Windows 8, 6–7
- test-driven development (TDD), 90, 93, 98
- testing JavaScript code, 89–103
- test.js file, 92, 97, 288
- text
 - drawing, 488–490
 - formatting, 173–175, 213
- <textarea>* element
 - cols attribute, 316
 - described, 35, 316
 - maxlength attribute, 316
 - placeholder attribute, 316
 - wrap attribute, 316
- <tfoot>* element, 35, 233
- <th>* element
 - colspan attribute, 238–240
 - creating header cells, 231–232
 - described, 35, 231
 - rowspan attribute, 238–240
- <thead>* element, 35, 233
- thematic breaks, 213
- this keyword, 117, 264, 268–269
- threads, 393
- tilde (~) character, 152
- <time>* element, 35, 220
- Timed Text Markup Language (TTML), 441
- timeouts
 - PositionOptions object, 544
 - promise object, 396–397
 - WebSocket object, 420–422
- timeupdate event, 120
- <title>* element, 35, 41
- title attribute
 - as global attribute, 37
 - <input>* element, 328
- TODO comments, 7
- <tr>* element
 - creating tables, 230
 - described, 35, 229
 - styling rows, 241
 - <tbody>* element and, 233
- TRACE method (HTTP), 315
- <track>* element, 441
- trademarks, 41
- transactions
 - IndexedDB and, 558
 - Web SQL databases and, 584–586
 - web storage and, 564
- transparency (color), 172
- triangles, drawing, 478–481
- TrueType (.ttf) files, 144
- try block, 87
- .ttf (TrueType) files, 144
- TTML (Timed Text Markup Language), 441

type attribute
 <button> element, 317
 <embed> element, 50
 <input> element, 317–318
 <link> element, 141
 <object> element, 51
 <script> element, 100
 <source> element, 439, 445

U

<u> element, 220
 \u escape sequence, 69
 ui-dark.css file, 9
 ui-light.css file, 9
 element, 35, 222–223
 unary operators, 70
 undefined primitive type, 67
 underscore (_), 45, 72–73
 undo event, 117
 uninstalling packages, 354
 universal selectors, 147
 unload event, 117
 unordered lists, 222–223
 unsubscribing from events, 116
 updateReady event, 610
 URL input, validating, 327–328
 url module, 343
 usemap attribute
 element, 49
 <object> element, 51
 User Data API, 557
 user-defined style sheets, 159
 utf-8 character set, 41

V

:valid pseudo class, 330
 validating forms
 described, 324–325

required validation, 325–327
 styling validations, 330
 validating numbers and ranges, 329
 validating URL input, 327–328
 value attribute (<option> element), 320
 value types, 67, 138
 values
 described, 67
 determining for variables, 83
 determining if same type and equal, 84
 retrieving for attributes, 153
 return, 73
 setting for colors, 166–171
 <var> element, 35, 217
 variables
 assigning function expressions to, 75
 case sensitivity, 72
 converting to different types, 78–80
 counter, 86
 creating environment for, 73
 described, 71
 determining values of, 83
 examining in debugger, 104–105
 global, 77–78
 JavaScript support, 66
 local, 77–78
 naming, 72–73
 retries, 85–86
 scoping, 77–78
 statements and, 71–73
 working with functions, 73–77
 Vehicle Identification Number (VIN), 365
 versioning, IndexedDB, 591
 versions
 including information in manifest files, 609
 verifying for jQuery, 369
 <video> element
 autoplay attribute, 441
 configuring, 441
 controls attribute, 441, 450

described, 35, 437
drawing with images, 490
height attribute, 441
HTMLMediaElement object and, 444, 447
id attribute, 450
implementing, 438–439
loop attribute, 441
muted attribute, 441
poster attribute, 441
preload attribute, 441
<source> element and, 439–440
src attribute, 441
width attribute, 441

video and movies. *See also* <video> element
accessing tracks, 441–442
described, 437
lesson summary and review, 442–443, 456
practice exercises, 452–455
setting <source> element, 439–440
video formats, 438

viewBox attribute (<svg> element), 500–501

VIN (Vehicle Identification Number), 365

.visible selector, 237

:visited pseudo class, 149

Visual Studio 2012
CSS3 support, 4
editions supported, 2–3
HTML5 support, 3–4
JavaScript support, 4–5
lesson summary and review, 11–12, 26
practice exercises, 20–25

Visual Studio 2012 Express for Web
described, 3, 9–11
New Project screen, 10
Node.js support, 356
Start Page screen, 9

Visual Studio 2012 Express for Windows 8
described, 3, 5–7
New Project link, 5
New Project screen, 6

Start Page screen, 5
templates included, 6–9

Visual Studio Premium 2012, 2

Visual Studio Professional 2012, 2

Visual Studio Team Foundation Server Express 2012, 2–3

Visual Studio Test Professional 2012, 2

Visual Studio Ultimate 2012, 2

void elements, 38–39

volumechange event, 120

W

W3C (World Wide Web Consortium)
CSS recommendations, 137–138
event recommendations, 566
multimedia standards, 437–438, 443–444
open standards for web, 30–31
storage capacity, 561
Web SQL support, 582
WebSocket API, 415–417
WebVTT standard, 441–442

WAI-ARIA, 212

WAI (Web Accessible Initiative), 212

waiting event, 120

.wav file extension, 444–445

WAV format, 444

<wbr> element, 35, 39, 217

WCF (Windows Communication Foundation), 366

Web Accessible Initiative (WAI), 212

web browsers. *See* browsers

web communications
described, 312
encrypting, 366
HTTP() method, 315–316, 322–323
HTTP protocol basics, 314–315
web browsers, 314
web servers, 312–313

Web Embedding Fonts Tool (WEFT), 144

web farms, 422–423
web servers
described, 312–313
submitting form data to, 316
WebSocket protocol, 416
web services. *See also* Node.js platform
AJAX calling, 368–380
cross-origin resource sharing, 380–381
described, 364–366
lesson summary and review, 381–382, 391–392
practice exercises, 386–390
RESTful, 366–368
Web Services Description Language (WSDL), 366
Web SQL databases
adding tables, 584
aggregating functions, 588
browser support, 559
creating and opening, 582–583
deleting records, 586
executeSql() method, 585–586
filtering results, 587–588
inserting new records, 585–586
JOIN commands, 588
lesson summary and review, 588–589, 617
longevity considerations of, 582
performing schema updates, 583
reading values from, 586–587
as storage mechanism, 558
transactions in, 584–585
updating existing records, 586
web storage. *See* storage mechanisms
web workers, 404–405, 563
Web.config file, 10
.webm file extension, 438–439
WebM/VP8 format, 438
WebSocket API
dealing with web farms, 422–423
described, 415–417
handling connection disconnects, 422
handling timeouts, 420–422
lesson summary and review, 424–425, 436
practice exercises, 425–435
WebSocket libraries, 423–424
WebSocket object
binaryType property, 417
bufferedAmount property, 417
close() method, 416
extensions property, 417
implementing, 417–420
onclose event property, 417
onerror event property, 417
onmessage event property, 417
onopen event property, 417
protocol property, 417
readyState property, 417, 419
send() method, 416–417, 419
url property, 417
WebSocket constructor, 416
WebSocket protocol
dealing with web farms, 422–423
described, 415–416
handling connection disconnects, 422
handling timeouts, 420–422
lesson summary and review, 424–425, 436
practice exercises, 425–435
WebSocket libraries, 423–424
WebVTT (Web Video Text Tracks) standard, 441–442
WEFT (Web Embedding Fonts Tool), 144
WHERE clause (SQL), 587
while loop, 84–85
width attribute
 `<canvas>` element, 460
 `<embed>` element, 50
 `<object>` element, 51
 `<video>` element, 441
window object
 events triggered by, 117
 frameElement property, 44
 parent property, 44

- requestFileSystem() method, 601
- top property, 44
 - webkitRequestFileSystem() method, 601
- Windows 8 applications, QUnit-Metro tool and, 95–100
- Windows Communication Foundation (WCF), 366
- WinJS, 394
- worker object
 - close() method, 405
 - described, 405
 - postMessage() method, 563
 - terminate() method, 405
- World Wide Web Consortium. *See* W3C (World Wide Web Consortium)
- wrap attribute (`<textarea>` element), 316
- writing
 - to directories, 605
 - to files, 563, 602–603
 - JavaScript code, 89–103
- WSDL (Web Services Description Language), 366

X

- XHTML, 30–32
- XML (eXtensible Markup Language), 30–31
- XML Schema Definition (XSD), 30
- XMLHttpRequest object
 - described, 369–371
 - error handling, 372–373
 - jQuery wrappers, 373–377
 - open() method, 370
 - response property, 370
 - sample code, 394–395
 - send() method, 370
- XSD (XML Schema Definition), 30
- XSS attacks, 381

About the author



GLENN JOHNSON is a professional trainer, consultant, and developer whose experience spans the past 25 years. As a consultant and developer, he has worked on many large projects, mostly in the insurance industry. Glenn's strengths are with Microsoft products such as ASP.NET, Model-View-Controller (MVC), Silverlight, Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), and Microsoft SQL Server using C#, Visual Basic, and T-SQL. This is yet another of many .NET books that Glenn has authored. He also develops courseware for and teaches classes in many countries on HTML5, JavaScript, Microsoft MVC, Microsoft ASP.NET, Visual Basic .NET, C#, and the .NET Framework.

Glenn holds the following Microsoft certifications: MCT, MCPD, MCTS, MCAD, MCSD, MCDBA, MCP + Site Building, MCSE + Internet, MCP + Internet, and MCSE. You can find Glenn's website at <http://GJTT.com>.

Training Guide: Programming in HTML5 with JavaScript and CSS3 and Exam 70-480

This book is designed to help build and advance your job-role expertise. In addition, it covers some of the topics and skills related to Microsoft Certification Exam 70-480 and might be useful as a complementary study resource.

Note: This book is not designed to cover all exam topics; see the following chart. If you are preparing for the exam, use additional materials to help bolster your readiness, in conjunction with real-world experience.

EXAM OBJECTIVES/SKILLS	SEE TOPIC-RELATED COVERAGE HERE
IMPLEMENT AND MANIPULATE DOCUMENT STRUCTURES AND OBJECTS	
Create the document structure.	Chapters 2 and 5
Write code that interacts with UI controls.	Chapters 3, 7, 11, 12, and 13
Apply styling to HTML elements programmatically.	Chapter 4
Implement HTML5 APIs.	Chapters 10, 14, 15, and 16
Establish the scope of objects and variables.	Chapters 3 and 6
Create and implement objects and methods.	Chapter 6
IMPLEMENT PROGRAM FLOW	
Implement program flow.	Chapter 3 and 6
Raise and handle an event.	Chapter 3 and 6
Implement exception handling.	Chapter 3
Implement a callback.	Chapter 3, 6, 8, and 9
Create a web worker process.	Chapter 9
ACCESS AND SECURE DATA	
Validate user input by using HTML5 elements.	Chapter 7
Validate user input by using JavaScript.	Chapter 7
Consume data.	Chapter 8
Serialize, deserialize, and transmit data.	Chapter 7

USE CSS3 IN APPLICATIONS

Style HTML text properties.	Chapter 4
Style HTML box properties.	Chapter 4
Create a flexible content layout.	Chapter 4
Create an animated and adaptive UI.	Chapter 4
Find elements by using CSS selectors and jQuery.	Chapter 4 and 6
Structure a CSS file by using CSS selectors.	Chapter 4

For complete information about Exam 70-480, visit <http://www.microsoft.com/learning/en/us/exam.aspx?ID=70-480>. In addition, for more information about Microsoft certifications, visit <http://www.microsoft.com/learning>.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft®
Press