Microsoft

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

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 mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

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.
Copyeditor: Kerin Forsyth
Indexer: Lucie Haskins
Cover: Twist Creative • Seattle

[2013-08-09]

Contents at a glance

Contents

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:

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 <*span*> 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 <*span*> elements, you need to apply special formatting to the content.

You might be wondering what kind of meaning the <*div*> and <*span*> 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.nvdaproject.org/.*

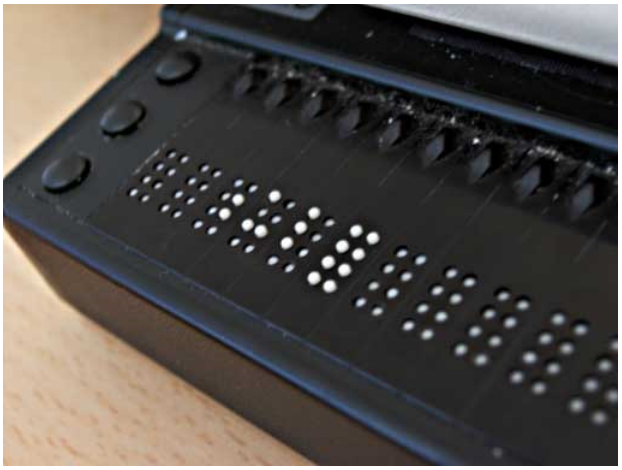


FIGURE 5-1 The refreshable Braille display

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 <*span*> 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 <*span*> 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 <*span*> 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.

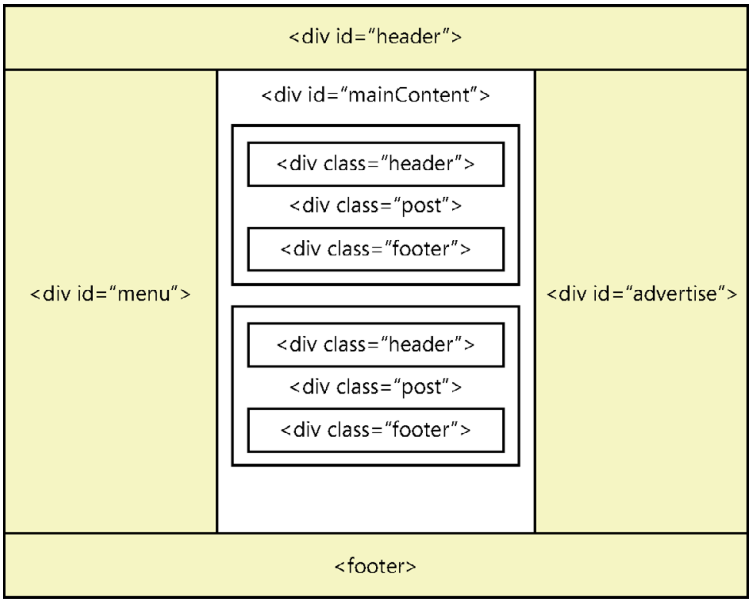


FIGURE 5-2 A blog site layout container using * * elements

A *layout container* lays out its children in a way that is flexible and reusable. For the developer, the purpose of the * * 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.

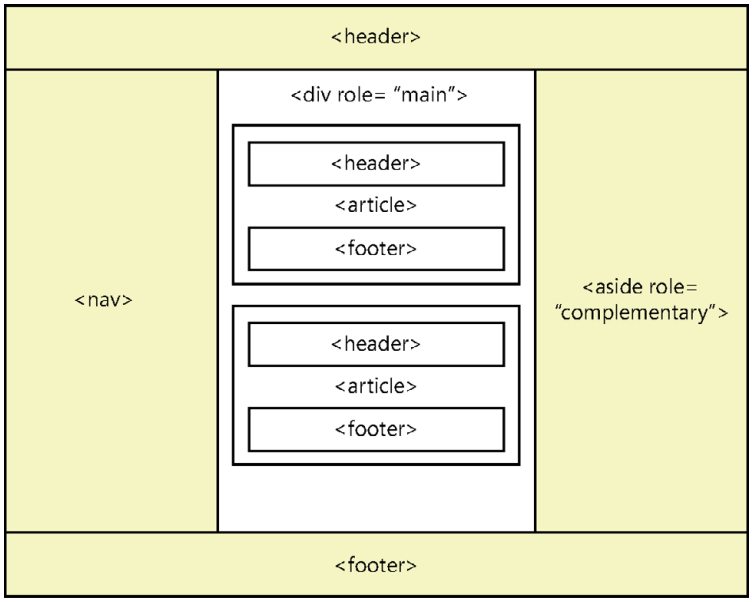


FIGURE 5-3 Layout container example, using the new HTML5 elements

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>
   <img src="logo.jpg" alt="Company Logo" />
   <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 <*img*> 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 <*b*> 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 <*b*> and <*i*> elements and many other elements that can be used to annotate content.

Using the <*b*> element

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

According to the W3C, the <*b*> 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 <*b*> element, although keeping the bold style makes the most sense.

The <*b*> element is the element of last resort because headings should be denoted with the <*hn*> element, emphasized text should be denoted with the <*em*> element, important text should be denoted with the <*strong*> element, and marked or highlighted text should use the <*mark*> element. Refrain from using the <*b*> 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 <*strong*> element

Closely related to the <*b*> element is the <*strong*> element, which represents strong importance for its contents. You can show relative importance by nesting <*strong*> elements within <*strong*> 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 <*strong*> and <*b*> elements look the same.

Using the <*i*> element

The <*i*> element was used to produce italic text, but like the <*b*> 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 that you can apply any style to the <*i*> element, although, like the <*b*> element, you probably should keep the default style.

The <*em*> element

Use the <*em*> 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 <*em*> 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 <*strong*> 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="email: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 <*br /*> and <*wbr /*> elements

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

The <*br /*> 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.

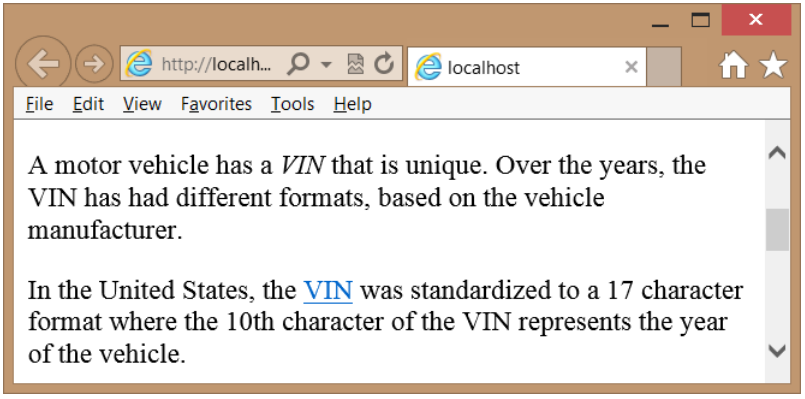


FIGURE 5-4 The <*dfn*> element italicizing its text by default

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">
        <img src="/pbj.jpg" alt="peanut butter and jelly requirements" />
        <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>
```

```
            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">
            <img src="/pbj.jpg" alt="peanut butter and jelly requirements" />
            <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.
- ***<del>*** 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 *<li>* 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 *<ol>* 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.

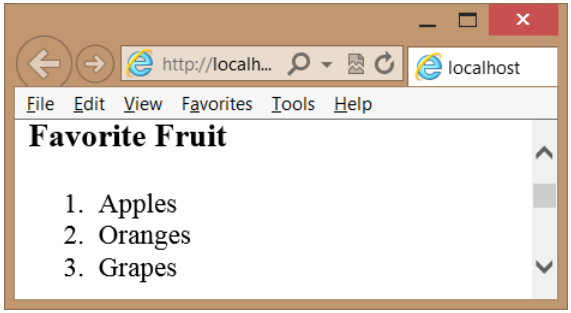


FIGURE 5-5 The ordered list numbering its items automatically

The *<ol>* 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.

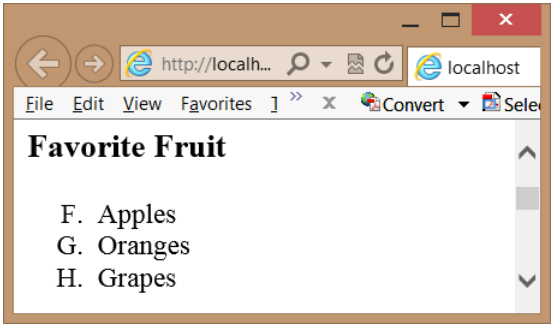


FIGURE 5-6 The ordered list with its type and start attributes set

Unordered lists

An *unordered list* is not auto-numbered. Use the <*ul*> 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.

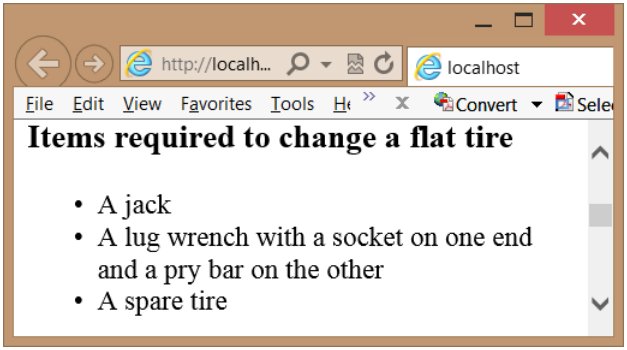


FIGURE 5-7 The unordered list rendering each list item as a bullet

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.

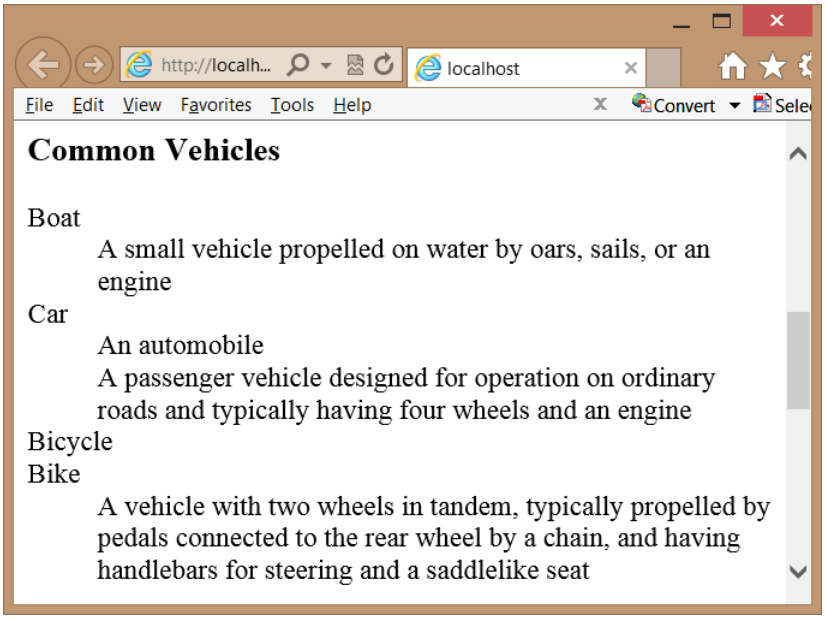


FIGURE 5-8 The definition list with its terms and definitions

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>
    <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.

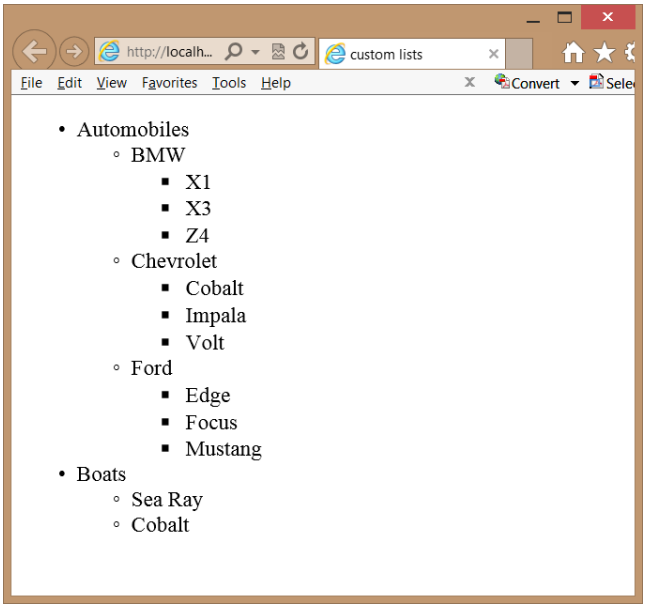


FIGURE 5-9 The rendered output with default styles

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.

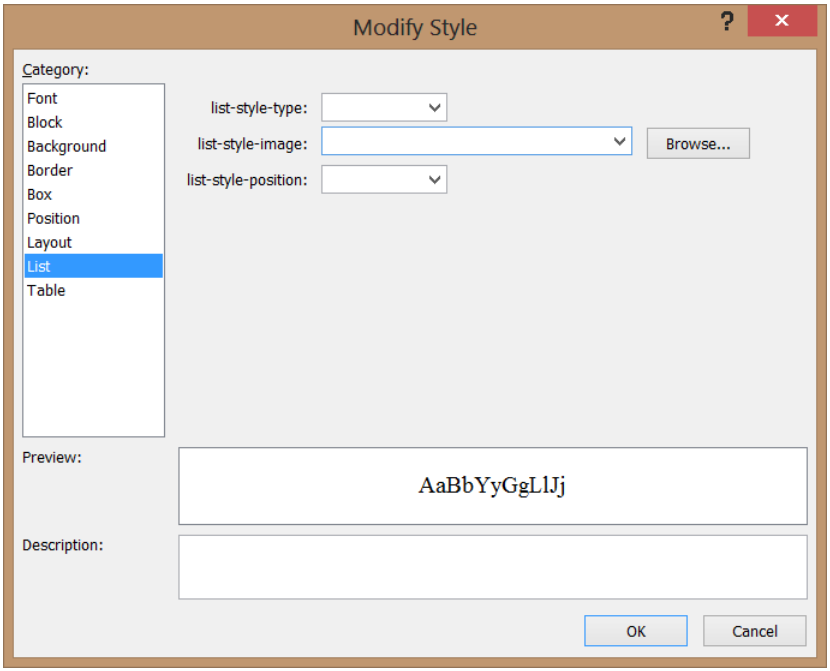


FIGURE 5-10 The Modify Style window showing menu-based style settings

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 <*li*> element that is a child of a <*ul*> element with a CSS class of level1 is rendered. It increments the section counter by one. It then inserts the content property before the <*li*> 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 <*li*> element that is a child of a <*ul*> element with a CSS class of level2 is rendered. It increments the subsection counter by one. It then inserts the content property before the <*li*> 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-rule 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.

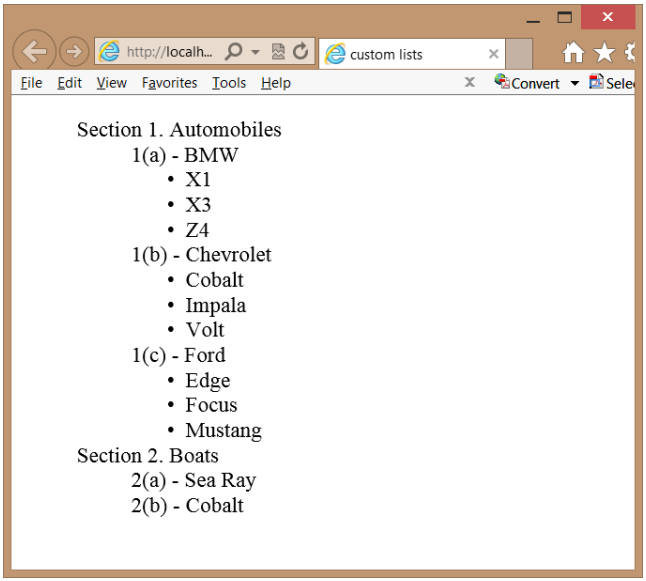


FIGURE 5-11 The rendered custom list

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 <*span*> 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 <*b*> element, use the <*strong*> element. In lieu of the <*i*> element, use the <*em*> 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>*
   - **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 * * element should be as close to the top of the HTML document as possible. The * * 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 * * 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?

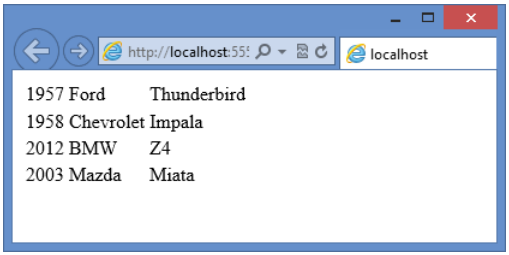


FIGURE 5-12 The rendered table with rows and columns

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.

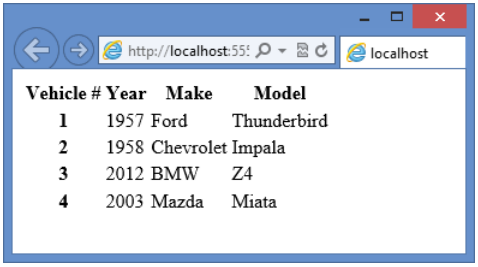


FIGURE 5-13 The revised table with horizontal and vertical headers

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>
```

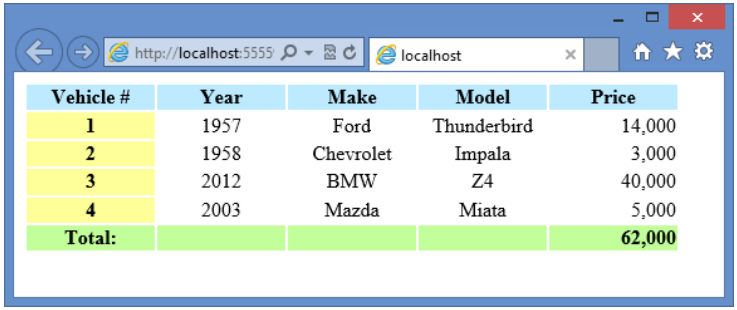
```
        <tr>
            <th>Total:</th>
            <th></th>
            <th></th>
            <th></th>
            <th>62,000</th>
        </tr>
    </tfoot>
</table>
```

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>
                    <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>
        </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.

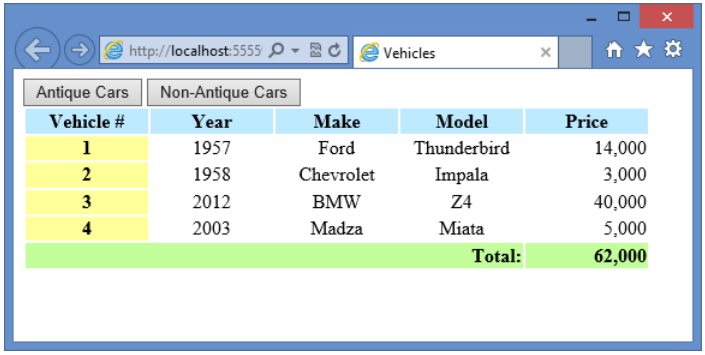


FIGURE 5-15 The rendered page with the footer containing only two cells

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.

```
<table>
    <thead>
        <tr>
            <th>Vehicle #</th>
            <th>Category</th>
            <th>Year</th>
            <th>Make</th>
            <th>Model</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody id="antiqueCars">
        <tr>
            <th>1</th>
            <td rowspan="2">Antique</td>
            <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>
    </tbody>
    <tbody id="nonAntiqueCars">
        <tr>
            <th>3</th>
            <td rowspan="2">Non-Antique</td>
```

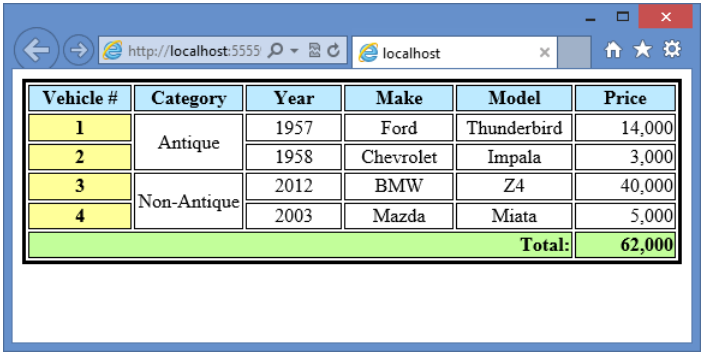
```
            <td>2012</td>
            <td>BMW</td>
            <td>Z4</td>
            <td>40,000</td>
        </tr>
        <tr>
            <th>4</th>
            <td>2003</td>
            <td>Mazda</td>
            <td>Miata</td>
            <td>5,000</td>
        </tr>
    </tbody>
    <tfoot>
        <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 <*col-group*> 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.

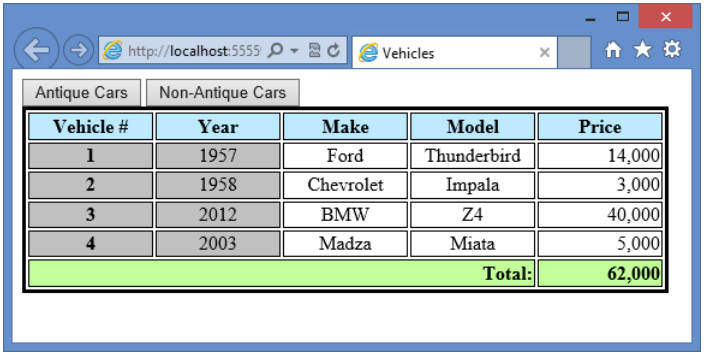


FIGURE 5-17 Using the <*colgroup*> and <*col*> elements to apply a style to multiple columns

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 &copy; 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 &copy; 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; }
   ```

2. 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;
}
```

3. 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**.
4. To add the image from the Resource folder, in the Solution Explorer window, right-click the Images folder that you just added.
5. Click Add, choose Existing Item, and select the ContosoLogo.png file that is located in the Chapter05 Resources folder.
6. 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.
7. 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;
}
```

8. 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.
9. 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>
```

```
                <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>
```

7. 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%;
}
```

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

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

9. 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;
}
```

10. 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;
}
```

11. 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 <*b*> element.
   - **B. Incorrect:** The <*em*> element indicates emphatic stress but not necessarily importance.
   - **C. Correct:** The <*strong*> 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:

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.

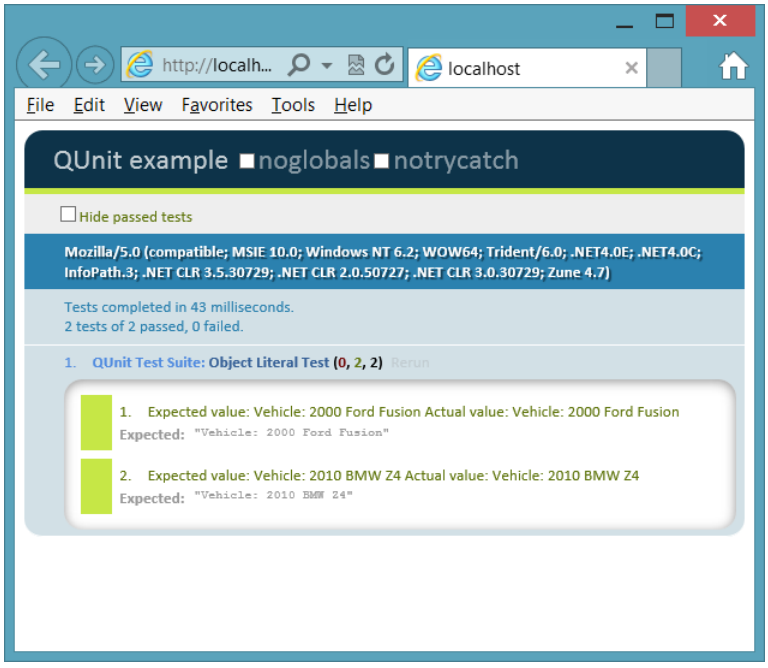


FIGURE 6-1 The JavaScript object literal test

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
- **toLocalString** 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.

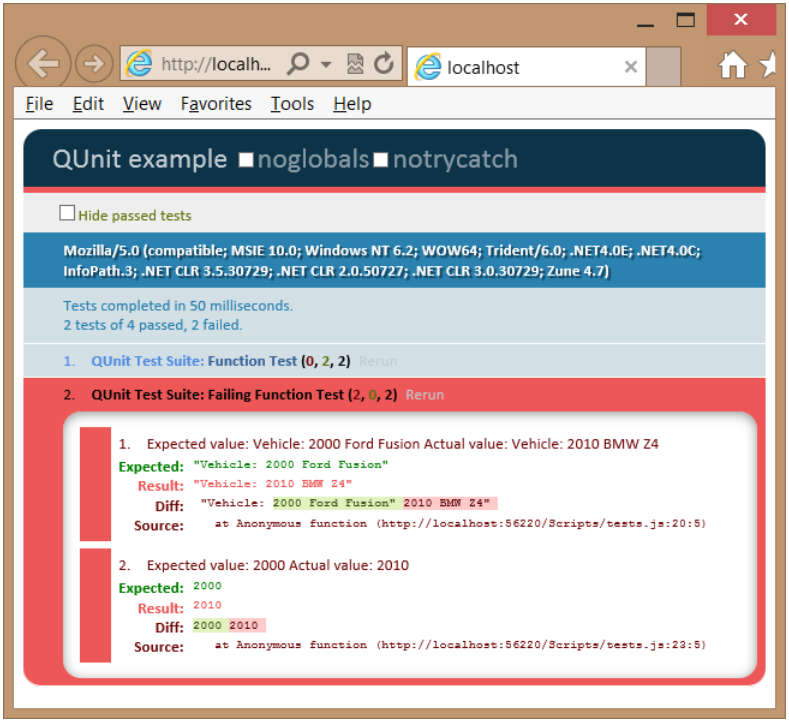


FIGURE 6-2 The failing test assertions after a second vehicle is used

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.

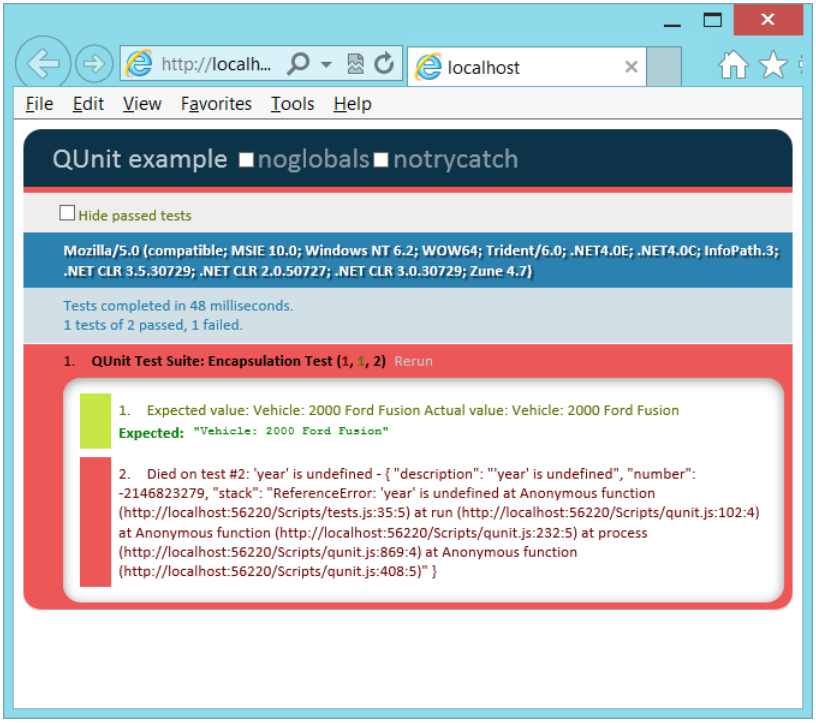


FIGURE 6-3 Successful first assertion and failed second assertion

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.**

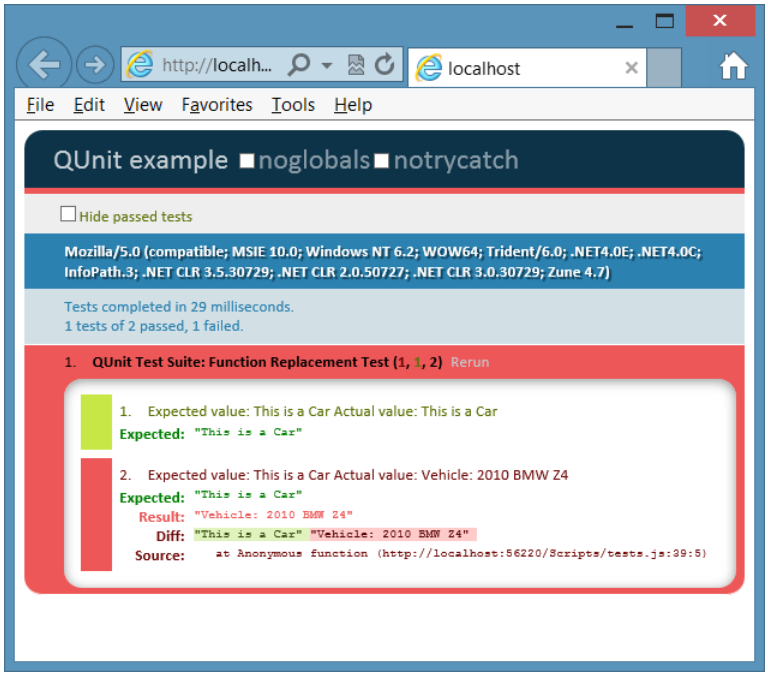


FIGURE 6-4 Successful first assertion, proving that the function was replaced; failed second assertion, proving that the second Vehicle's function was not replaced

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.

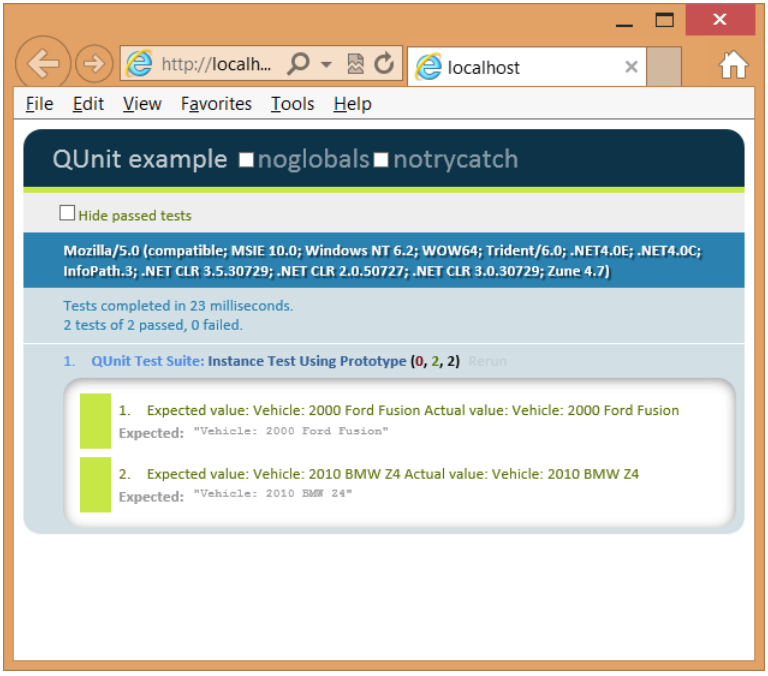


FIGURE 6-5 The modified class using the prototype property to create the getInfo function

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.

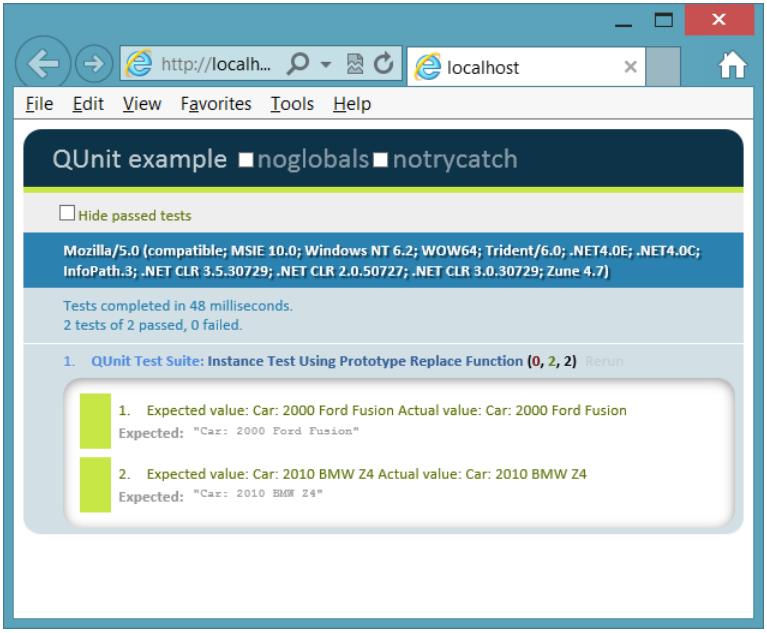


FIGURE 6-6 The modification of the getInfo prototype affected all instances

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.

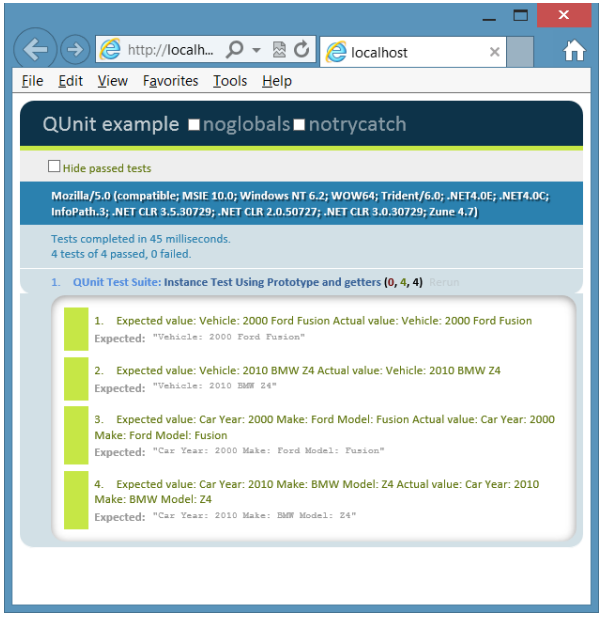


FIGURE 6-7 The use of getters to expose read-only data as a good compromise

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.

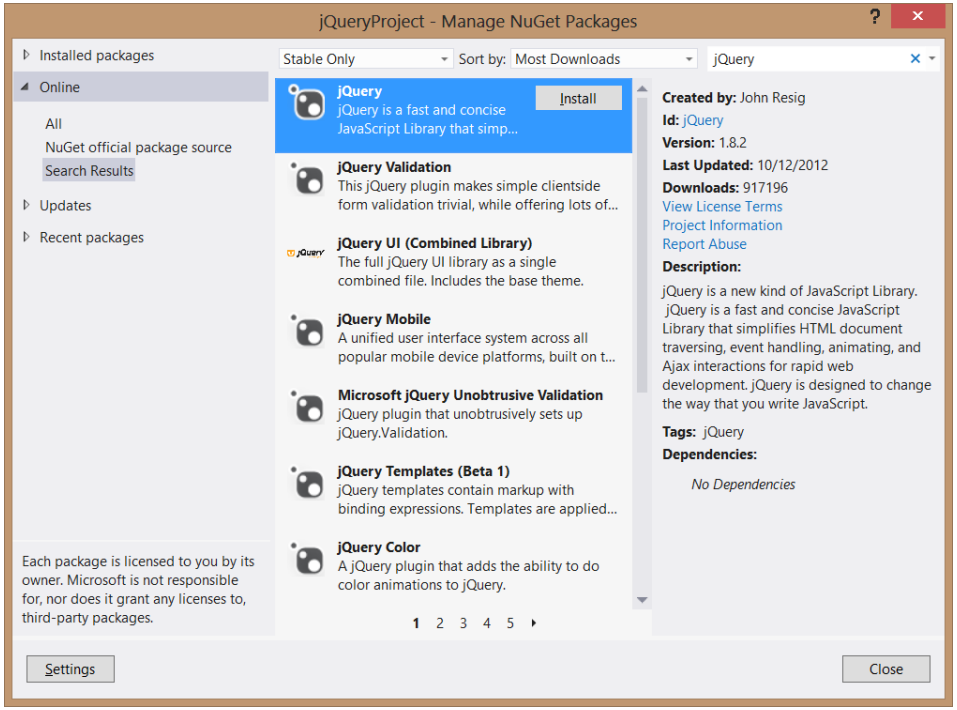


FIGURE 6-9 The NuGet package manager

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.

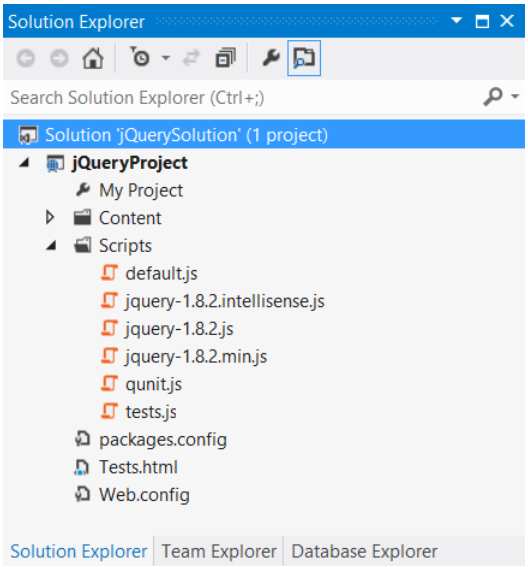


FIGURE 6-10 The completed installation of jQuery

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.

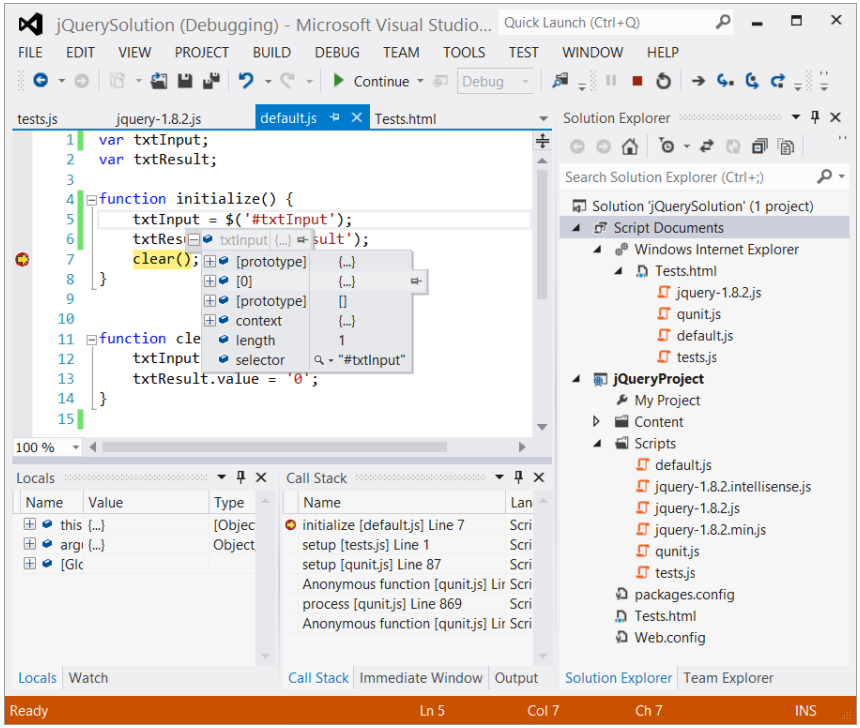


FIGURE 6-11 The jQuery wrapper object for *txtInput* with one element

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.

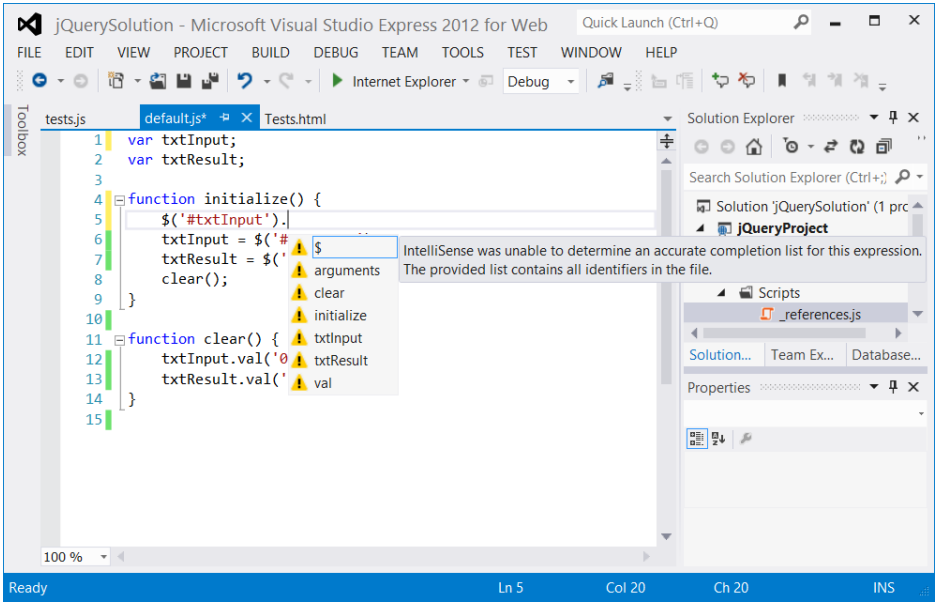


FIGURE 6-12 The IntelliSense window when not properly set up for jQuery

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.

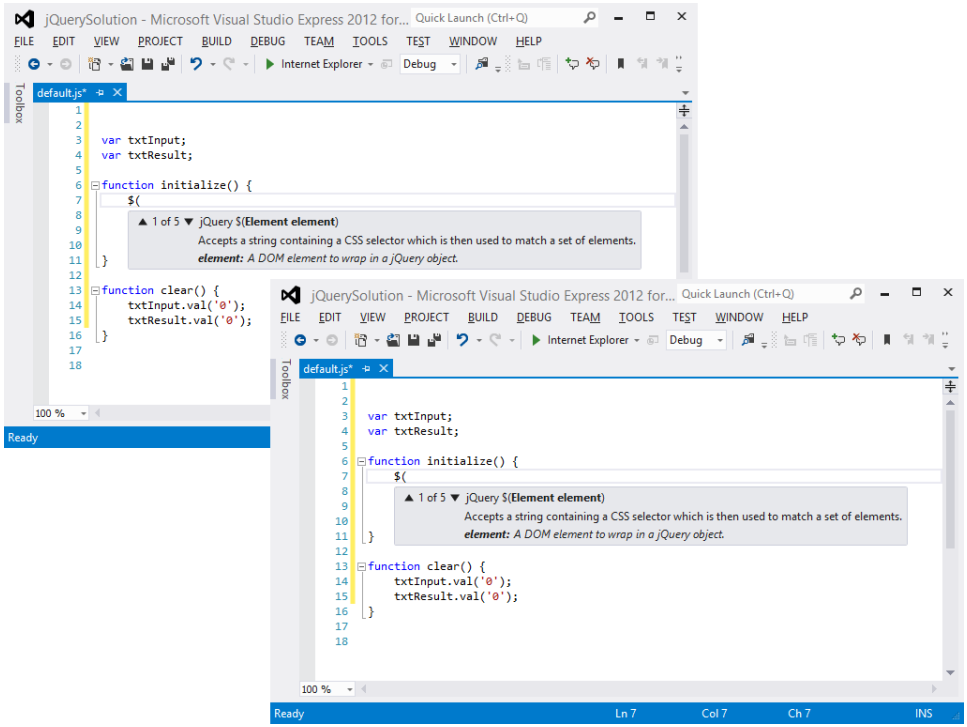


FIGURE 6-13 The jQuery IntelliSense providing help as you type

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 btnNumber*X* where *X* is the number on the button.

14. Open the CalculatorTests.html file and replace the number button ids with btnNumber*X* 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.minusClick);
        $('#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>
                    <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 &copy; 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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

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