

Microsoft®

MCT USE ONLY. STUDENT USE PROHIBITED

OFFICIAL MICROSOFT LEARNING PRODUCT

20480B

Programming in HTML5 with JavaScript and
CSS3

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2012 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at

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

Product Number: 20480B

Part Number (if applicable):

Released: xx/20xx

MICROSOFT LICENSE TERMS

OFFICIAL MICROSOFT LEARNING PRODUCTS

MICROSOFT OFFICIAL COURSE Pre-Release and Final Release Versions

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the Licensed Content named above, which includes the media on which you received it, if any. These license terms also apply to any updates, supplements, internet based services and support services for the Licensed Content, unless other terms accompany those items. If so, those terms apply.

BY DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT DOWNLOAD OR USE THE LICENSED CONTENT.

If you comply with these license terms, you have the rights below.

1. DEFINITIONS.

- a. “Authorized Learning Center” means a Microsoft Learning Competency Member, Microsoft IT Academy Program Member, or such other entity as Microsoft may designate from time to time.
- b. “Authorized Training Session” means the Microsoft-authorized instructor-led training class using only MOC Courses that are conducted by a MCT at or through an Authorized Learning Center.
- c. “Classroom Device” means one (1) dedicated, secure computer that you own or control that meets or exceeds the hardware level specified for the particular MOC Course located at your training facilities or primary business location.
- d. “End User” means an individual who is (i) duly enrolled for an Authorized Training Session or Private Training Session, (ii) an employee of a MPN Member, or (iii) a Microsoft full-time employee.
- e. “Licensed Content” means the MOC Course and any other content accompanying this agreement. Licensed Content may include (i) Trainer Content, (ii) software, and (iii) associated media.
- f. “Microsoft Certified Trainer” or “MCT” means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program, and (iii) holds a Microsoft Certification in the technology that is the subject of the training session.
- g. “Microsoft IT Academy Member” means a current, active member of the Microsoft IT Academy Program.
- h. “Microsoft Learning Competency Member” means a Microsoft Partner Network Program Member in good standing that currently holds the Learning Competency status.
- i. “Microsoft Official Course” or “MOC Course” means the Official Microsoft Learning Product instructor-led courseware that educates IT professionals or developers on Microsoft technologies.

- j. "Microsoft Partner Network Member" or "MPN Member" means a silver or gold-level Microsoft Partner Network program member in good standing.
 - k. "Personal Device" means one (1) device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular MOC Course.
 - l. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
 - m. "Trainer Content" means the trainer version of the MOC Course and additional content designated solely for trainers to use to teach a training session using a MOC Course. Trainer Content may include Microsoft PowerPoint presentations, instructor notes, lab setup guide, demonstration guides, beta feedback form and trainer preparation guide for the MOC Course. To clarify, Trainer Content does not include virtual hard disks or virtual machines.
2. **INSTALLATION AND USE RIGHTS.** The Licensed Content is licensed not sold. The Licensed Content is licensed on a one copy per user basis, such that you must acquire a license for each individual that accesses or uses the Licensed Content.

2.1 Below are four separate sets of installation and use rights. Only one set of rights apply to you.

a. **If you are a Authorized Learning Center:**

- i. If the Licensed Content is in digital format for each license you acquire you may either:
 - 1. install one (1) copy of the Licensed Content in the form provided to you on a dedicated, secure server located on your premises where the Authorized Training Session is held for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session, **or**
 - 2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session.
- ii. You agree that:
 - 1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
 - 2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
 - 3. for all Authorized Training Sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
 - 4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,

5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of the Authorized Training Session,
 6. you will only provide access to the Licensed Content to End Users and MCTs,
 7. you will only provide access to the Trainer Content to MCTs, and
 8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.
- b. **If you are a MPN Member.**
- i. If the Licensed Content is in digital format for each license you acquire you may either:
 1. install one (1) copy of the Licensed Content in the form provided to you on (A) one (1) Classroom Device, or (B) one (1) dedicated, secure server located at your premises where the training session is held for use by one (1) of your employees attending a training session provided by you, or by one (1) MCT that is teaching the training session, **or**
 2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for use by one (1) End User attending a Private Training Session, or one (1) MCT that is teaching the Private Training Session.
 - ii. You agree that:
 1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
 2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
 3. for all training sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
 4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,
 5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of each training session,
 6. you will only provide access to the Licensed Content to End Users and MCTs,
 7. you will only provide access to the Trainer Content to MCTs, and
 8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.

c. **If you are an End User:**

You may use the Licensed Content solely for your personal training use. If the Licensed Content is in digital format, for each license you acquire you may (i) install one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install another copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content; or (ii) print one (1) copy of the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.

d. If you are a MCT.

- i. For each license you acquire, you may use the Licensed Content solely to prepare and deliver an Authorized Training Session or Private Training Session. For each license you acquire, you may install and use one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.
- ii. **Use of Instructional Components in Trainer Content.** You may customize, in accordance with the most recent version of the MCT Agreement, those portions of the Trainer Content that are logically associated with instruction of a training session. If you elect to exercise the foregoing rights, you agree: (a) that any of these customizations will only be used for providing a training session, (b) any customizations will comply with the terms and conditions for Modified Training Sessions and Supplemental Materials in the most recent version of the MCT agreement and with this agreement. For clarity, any use of “*customize*” refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

2.2 Separation of Components. The Licensed Content components are licensed as a single unit and you may not separate the components and install them on different devices.

2.3 Reproduction/Redistribution Licensed Content. Except as expressly provided in the applicable installation and use rights above, you may not reproduce or distribute the Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.

2.4 Third Party Programs. The Licensed Content may contain third party programs or services. These license terms will apply to your use of those third party programs or services, unless other terms accompany those programs and services.

2.5 Additional Terms. Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to that respective component and supplements the terms described in this Agreement.

3. PRE-RELEASE VERSIONS. If the Licensed Content is a pre-release (“**beta**”) version, in addition to the other provisions in this agreement, then these terms also apply:

- a. **Pre-Release Licensed Content.** This Licensed Content is a pre-release version. It may not contain the same information and/or work the way a final version of the Licensed Content will. We may change it for the final version. We also may not release a final version. Microsoft is under no obligation to provide you with any further content, including the final release version of the Licensed Content.
- b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software, technologies, or products to third parties because we include your feedback in them. These rights

survive this agreement.

- c. **Term.** If you are an Authorized Training Center, MCT or MPN, you agree to cease using all copies of the beta version of the Licensed Content upon (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) sixty (60) days after the commercial release of the Licensed Content, whichever is earliest (“**beta term**”). Upon expiration or termination of the beta term, you will irretrievably delete and destroy all copies of same in the possession or under your control.
4. **INTERNET-BASED SERVICES.** Microsoft may provide Internet-based services with the Licensed Content, which may change or be canceled at any time.
 - a. **Consent for Internet-Based Services.** The Licensed Content may connect to computer systems over an Internet-based wireless network. In some cases, you will not receive a separate notice when they connect. Using the Licensed Content operates as your consent to the transmission of standard device information (including but not limited to technical information about your device, system and application software, and peripherals) for internet-based services.
 - b. **Misuse of Internet-based Services.** You may not use any Internet-based service in any way that could harm it or impair anyone else's use of it. You may not use the service to try to gain unauthorized access to any service, data, account or network by any means.
5. **SCOPE OF LICENSE.** The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
 - install more copies of the Licensed Content on devices than the number of licenses you acquired;
 - allow more individuals to access the Licensed Content than the number of licenses you acquired;
 - publicly display, or make the Licensed Content available for others to access or use;
 - install, sell, publish, transmit, encumber, pledge, lend, copy, adapt, link to, post, rent, lease or lend, make available or distribute the Licensed Content to any third party, except as expressly permitted by this Agreement.
 - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation;
 - access or use any Licensed Content for which you are not providing a training session to End Users using the Licensed Content;
 - access or use any Licensed Content that you have not been authorized by Microsoft to access and use; or
 - transfer the Licensed Content, in whole or in part, or assign this agreement to any third party.
6. **RESERVATION OF RIGHTS AND OWNERSHIP.** Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content. You may not remove or obscure any copyright, trademark or patent notices that appear on the Licensed Content or any components thereof, as delivered to you.

7. **EXPORT RESTRICTIONS.** The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, End Users and end use. For additional information, see www.microsoft.com/exporting.
8. **LIMITATIONS ON SALE, RENTAL, ETC. AND CERTAIN ASSIGNMENTS.** You may not sell, rent, lease, lend or sublicense the Licensed Content or any portion thereof, or transfer or assign this agreement.
9. **SUPPORT SERVICES.** Because the Licensed Content is "as is", we may not provide support services for it.
10. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon any termination of this agreement, you agree to immediately stop all use of and to irretrievable delete and destroy all copies of the Licensed Content in your possession or under your control.
11. **LINKS TO THIRD PARTY SITES.** You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.
12. **ENTIRE AGREEMENT.** This agreement, and the terms for supplements, updates and support services are the entire agreement for the Licensed Content.
13. **APPLICABLE LAW.**
 - a. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
 - b. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.
14. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.
15. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS," "WITH ALL FAULTS," AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES GIVE NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS UNDER OR IN RELATION TO THE LICENSED CONTENT. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES EXCLUDE ANY IMPLIED WARRANTIES OR CONDITIONS, INCLUDING THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**

16. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. TO THE EXTENT NOT PROHIBITED BY LAW, YOU CAN RECOVER FROM MICROSOFT CORPORATION AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO USD\$5.00. YOU AGREE NOT TO SEEK TO RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES FROM MICROSOFT CORPORATION AND ITS RESPECTIVE SUPPLIERS.

This limitation applies to

- anything related to the Licensed Content, services made available through the Licensed Content, or content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection dues consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 \$ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence , aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

MCT USE ONLY. STUDENT USE PROHIBITED

Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.
- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance¹. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.
- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning
www.microsoft.com/learning



¹ IDC, Value of Certification: Team Certification and Organizational Performance, November 2006

Acknowledgments

Microsoft Learning wants to acknowledge and thank the following for their contribution toward developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

John Sharp – Content Developer

John Sharp gained an honors degree in Computing from Imperial College, London. He has been developing software and writing training courses, guides, and books for over 25 years. John has experience in a wide range of technologies, from database systems and UNIX through to C, C++ and C# applications for the .NET Framework, together with Java and JavaScript development. He has authored several books for Microsoft Press, including six editions of C# Step By Step, Windows Communication Foundation Step By Step, and the J# Core Reference.

Andrew Davey – Subject Matter Expert

Andrew Davey has a Bachelor degree in Computer Science and has been working in the software industry for 10 years, primarily with Microsoft technologies.

His current focus is on building rich interactive web applications by using ASP.NET, HTML5 and JavaScript. He is an active member of the developer community and the creator of a number of popular open source projects.

Andy Olsen – Subject Matter Expert

Andy Olsen studied Physics in Southampton University in the UK, and has been working in IT for more than 25 years. He has been working with the .NET Framework since Beta 1, and has written and designed numerous Microsoft courses covering a wide range of subjects, including ASP.NET, WPF, WCF, ADO.NET, XML, and HTML5.

Andy is a regular speaker at conferences and spends most of his spare time developing .NET solutions for large companies based in the UK, Europe, and Asia.

Dan Maharry - Subject Matter Expert

Dan Maharry is an experienced technical author with over a dozen books to his name and many more as technical reviewer and editor for a variety of publishers including Wrox, Apress, Microsoft Press, and O'Reilly. He's also a .NET developer with past stints for the dotCoop TLD registry and various web development houses.

David Day - Subject Matter Expert

David Day is a senior web developer for a local authority in the United Kingdom. He builds web applications that combine MVC3, the .NET Framework 4, NHibernate, SQL Server, HTML5, CSS, JavaScript, and jQuery.

Carsten Thomsen – Technical Reviewer

Carsten Thomsen is currently doing SharePoint 2010 development, but his interests are varied when it comes to IT and include development of ASP.NET, Windows Forms, Windows Store, Windows Phone, and other types of applications and components.

MCT USE ONLY STUDENT USE PROHIBITED

He has authored a number of development books, as well as more than 20 Microsoft Learning courses.

MCT USE ONLY. STUDENT USE PROHIBITED

Contents

Module 1: Overview of HTML and CSS

Lesson 1: Overview of HTML	page 2
Lesson 2: Overview of CSS	page 15
Lesson 3: Creating a Web Application by Using Visual Studio 2012	page 22
Lab: Exploring the Contoso Conference Application	page 28

Module 2: Creating and Styling HTML Pages

Lesson 1: Creating an HTML5 Page	page 2
Lesson 2: Styling an HTML5 Page	page 9
Lab: Creating and Styling HTML5 Pages	page 19

Module 3: Introduction to JavaScript

Lesson 1: Overview of JavaScript	page 2
Lesson 2: Introduction to the Document Object Model	page 13
Lesson 3: Introduction to jQuery	page 19
Lab: Displaying Data and Handling Events by Using JavaScript	page 27

Module 4: Creating Forms to Collect and Validate User Input

Lesson 1: Creating HTML5 Forms	page 2
Lesson 2: Validating User Input by Using HTML5 Attributes	page 6
Lesson 3: Validating User Input by Using JavaScript	page 10
Lab: Creating a Form and Validating User Input	page 14

Module 5: Communicating with a Remote Server

Lesson 1: Sending and Receiving Data by Using the XMLHttpRequest Object	page 2
Lesson 2: Sending and Receiving Data by Using the jQuery Library	page 8
Lab: Communicating with a Remote Server	page 12

Module 6: Styling HTML5 by Using CSS3

Lesson 1: Styling Text by Using CSS3	page 2
Lesson 2: Styling Block Elements	page 6
Lesson 3: Pseudo-classes and Pseudo-elements	page 14
Lesson 4: Enhancing Graphical Effects by Using CSS3	page 17
Lab: Styling Text and Block Elements by Using CSS3	page 24

Module 7: Creating Objects and Methods by Using JavaScript

Lesson 1: Writing Well-Structured JavaScript Code	page 2
Lesson 2: Creating Custom Objects	page 6
Lesson 3: Extending Objects	page 12

Lab: Refining Code for Maintainability and Extensibility page 17

Module 8: Creating Interactive Pages by Using HTML5 APIs

Lesson 1: Interacting with Files page 2
Lesson 2: Incorporating Multimedia page 7
Lesson 3: Reacting to Browser Location and Content page 12
Lesson 4: Debugging and Profiling a Web Application page 17
Lab: Creating Interactive Pages by Using HTML5 APIs page 22

Module 9: Adding Offline Support to Web Applications

Lesson 1: Caching Offline Data by Using the Application Cache API page 2
Lesson 2: Persisting User Data by Using the Local Storage API page 10
Lab: Adding Offline Support to Web Applications page 15

Module 10: Implementing an Adaptive User Interface

Lesson 1: Supporting Multiple Form Factors page 2
Lesson 2: Creating an Adaptive User Interface page 6
Lab: Implementing an Adaptive User Interface page 13

Module 11: Creating Advanced Graphics

Lesson 1: Creating Interactive Graphics by Using SVG page 2
Lesson 2: Programmatically Drawing Graphics by Using the Canvas API page 19
Lab: Creating Advanced Graphics page 26

Module 12: Animating the User Interface

Lesson 1: Applying CSS Transitions page 2
Lesson 2: Transforming Elements page 7
Lesson 3: Applying CSS Key-frame Animations page 16
Lab: Animating the User Interface page 22

Module 13: Implementing Real-time Communication by Using Web Sockets

Lesson 1: Introduction to Web Sockets page 2
Lesson 2: Using the Web Socket API page 4
Lab: Performing Real-time Communication by Using Web Sockets page 10

Module 14: Performing Background Processing by Using Web Workers

Lesson 1: Understanding Web Workers page 2
Lesson 2: Performing Asynchronous Processing by Using Web Workers page 5
Lab: Creating a Web Worker Process page 11

Lab Answer Keys

Module 1 Lab: Exploring the Contoso Conference Application	page 1
Module 2 Lab: Creating and Styling HTML5 Pages	page 1
Module 3 Lab: Displaying Data and Handling Events by Using JavaScript	page 1
Module 4 Lab: Creating a Form and Validating User Input	page 1
Module 5 Lab: Communicating with a Remote Server	page 1
Module 6 Lab: Styling Text and Block Elements by Using CSS3	page 1
Module 7 Lab: Refining Code for Maintainability and Extensibility	page 1
Module 8 Lab: Creating Interactive Pages by Using HTML5 APIs	page 1
Module 9 Lab: Adding Offline Support to Web Applications	page 1
Module 10 Lab: Implementing an Adaptive User Interface	page 1
Module 11 Lab: Creating Advanced Graphics	page 1
Module 12 Lab: Animating the User Interface	page 1
Module 13 Lab: Performing Real-time Communication by Using Web Sockets	page 1
Module 14 Lab: Creating a Web Worker Process	page 1

MCT USE ONLY. STUDENT USE PROHIBITED

About This Course

This section provides a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This course provides an introduction to HTML5, CSS3, and JavaScript. It helps students gain basic HTML5, CSS3, and JavaScript programming skills. Students will learn how to use HTML5, CSS3, and JavaScript to build responsive and scalable web applications that can dynamically detect and adapt to different form factors and device capabilities.

Audience

This course is intended for professional developers who have six to 12 months of programming experience and who are interested in developing applications by using HTML5 with JavaScript and CSS3 (either Windows Store apps for Windows 8 or web applications).

This course is intended for students who have the following experience:

- One to three months of experience creating web applications, including writing simple JavaScript code.
- One month of experience creating Windows client applications.
- One month of experience using Visual Studio 2010 or 2012.

This course is not intended for developers with three or more months of HTML5 coding experience.

Student Prerequisites

Before attending this course, students must have at least three months of professional development experience.

In addition to their professional experience, students who attend this training should have a combination of practical and conceptual knowledge related to HTML5 programming. This includes the following prerequisites:

- Understand the basic HTML document structure:
 - How to use HTML tags to display text content.
 - How to use HTML tags to display graphics.
 - How to use HTML APIs.
- Understand how to style common HTML elements by using CSS, including:
 - How to separate presentation from content.
 - How to manage content flow.
 - How to control the position of individual elements.
 - How to implement basic CSS styling.
- Understand how to write JavaScript code to add functionality to a web page:
 - How to create and use variables.
 - How to use:

- arithmetic operators to perform arithmetic calculations involving one or more variables;
- relational operators to test the relationship between two variables or expressions;
- logical operators to combine expressions that contain relational operators.
- How to control the program flow by using if ... else statements.
- How to implement iterations by using loops.
- How to write simple functions.

Course Objectives

After completing this course, students will be able to:

- Explain how to use Visual Studio 2012 to create and run a web application.
- Describe the new features of HTML5, and create and style HTML5 pages.
- Add interactivity to an HTML5 page by using JavaScript.
- Create HTML5 forms by using different input types, and validate user input by using HTML5 attributes and JavaScript code.
- Send and receive data to and from a remote data source by using XMLHttpRequest objects and jQuery AJAX operations.
- Style HTML5 pages by using CSS3.
- Create well-structured and easily maintainable JavaScript code.
- Use common HTML5 APIs in interactive web applications.
- Create web applications that support offline operations.
- Create HTML5 web pages that can adapt to different devices and form factors.
- Add advanced graphics to an HTML5 page by using Canvas elements, and Scalable Vector Graphics.
- Enhance the user experience by adding animations to an HTML5 page.
- Use Web Sockets to send and receive data between a web application and a server.
- Improve the responsiveness of a web application that performs long-running operations by using Web Worker processes.

Course Outline

The course outline is as follows:

Module 1, "Overview of HTML and CSS"

Module 2, "Creating and Styling HTML Pages"

Module 3, "Introduction to JavaScript"

Module 4, "Creating Forms to Collect and Validate User Input"

Module 5, "Communicating with a Remote Server"

Module 6, "Styling HTML5 by Using CSS3"

- Module 7, "Creating Objects and Methods by Using JavaScript"**
- Module 8, "Creating Interactive Pages by Using HTML5 APIs"**
- Module 9, "Adding Offline Support to Web Applications"**
- Module 10, "Implementing an Adaptive User Interface"**
- Module 11, "Creating Advanced Graphics"**
- Module 12, "Animating the User Interface"**
- Module 13, "Implementing Real-time Communication by Using Web Sockets"**
- Module 14, "Performing Background Processing by Using Web Workers"**

Course Materials

The following materials are included with the course:

- **Course Handbook:** a succinct classroom learning guide that provides the critical technical information in a crisp, tightly focused format, which is essential for an effective in-class learning experience.
 - **Lessons:** guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
 - **Labs:** provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
 - **Module Reviews and Takeaways:** provide on-the-job reference material to boost knowledge and skills retention.
 - **Lab Answer Keys:** provide step-by-step lab solution guidance.



Course Companion Content: searchable, easy-to-browse digital content with integrated premium online resources that supplement the Course Handbook.

- **Modules:** include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.

Resources: include well-categorized additional resources that give you immediate access to the most current premium content on TechNet, MSDN®, or Microsoft® Press.



Student Course files on the <http://www.microsoft.com/learning/companionmoc> Site: includes the Allfiles.exe, a self-extracting executable file that contains all required files for the labs and demonstrations.

- **Course evaluation:** at the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
 - To provide additional comments or feedback on the course, send an email to support@mscourseware.com. To inquire about the Microsoft Certification Program, send an email to mcphelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Microsoft Hyper-V™ to perform the labs.

Important: At the end of each lab, you must close the virtual machine and must not save any changes. To close a virtual machine (VM) without saving the changes, perform the following steps:

1. On the VM, on the **Action** menu, click **Close**.
2. In the **Close** dialog box, in the **What do you want the virtual machine to do?** list, click **Turn off** and delete changes, and then click **OK**.

The following table shows the role of each VM that is used in this course:

Virtual machine	Role
MSL-TMG1	Gateway computer for Internet access
20480B-SEA-DEV11	Development computer used for building web applications. The demonstration and lab files are located on the E:\Drive, in folders named Mod01, Mod02, and so on up to Mod14.

Software Configuration

The following software is installed on each VM:

- Microsoft Windows 8 Enterprise
- Microsoft Visual Studio 2012 Ultimate
- Microsoft Office Professional Plus 2010

Course Files

The files associated with the labs in this course are located in the E:\Labfiles folder on each VM.

Classroom Setup

Each classroom computer will have the same VMs configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware is taught.

Hardware Level 6+

- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120 GB hard disks 7200 RM SATA or better*
- 8GB or higher
- DVD drive
- Network adapter with Internet connectivity
- Super VGA (SVGA) 17-inch monitor
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

*Striped

In addition, the instructor computer must be connected to a projection display device that supports SVGA 1024 x 768 pixels, 16 bit color.

Module 1

Overview of HTML and CSS

Contents:

Module Overview	1-1
Lesson 1: Overview of HTML	1-2
Lesson 2: Overview of CSS	1-15
Lesson 3: Creating a Web Application by Using Visual Studio 2012	1-22
Lab: Exploring the Contoso Conference Application	1-28
Module Review and Takeaways	1-38

Module Overview

Most modern web applications are built upon a foundation of HTML pages that describe the content that users read and interact with, style sheets to make that content visually pleasing, and JavaScript code to provide a level of interactivity between user and page, and page and server. The web browser uses the HTML markup and the style sheets to render this content, and runs the JavaScript code to implement the behavior of the application. This module reviews the basics of HTML and CSS, and introduces the tools that this course uses to create HTML pages and style sheets.

Objectives

After completing this module, you will be able to:

- Explain how to use HTML elements and attributes to lay out a web page.
- Explain how to use CSS to apply basic styling to a web page.
- Describe the tools that Microsoft® Visual Studio® provides for building web applications.

Lesson 1

Overview of HTML

HTML has been the publishing language of the web since 1992. In this lesson, you will learn the fundamentals of HTML, how HTML pages are structured, and some of the basic features that can be added to an HTML page.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the structure of an HTML page.
- Explain basic HTML elements and attributes.
- Create and correctly markup an HTML page containing text elements.
- Display graphics in an HTML page by using image elements, and link pages together by using anchor elements.
- Create an HTML form page.
- Integrate JavaScript code into an HTML page.

The Structure of an HTML Page

HTML is an acronym for **Hyper Text Markup Language**. It is a static language that determines the structure and semantic meaning of a web page. You use HTML to create content and metadata that browsers use to render and display information. HTML content can include text, images, audio, video, forms, lists, tables, and many other items. An HTML page can also contain hyperlinks, which connect pages to each other and to websites and resources elsewhere on the Internet.

Every HTML page has the same basic structure:

- A DOCTYPE declaration stating which version of HTML the page uses.
- An html section that contains the following elements:
 - A header that contains information about the page for the browser. This may include its primary language (English, Chinese, French, and so on), character set, associated style sheets and script files, author information, and keywords for search engines.
 - A body that contains all the viewable content of the page.

This is true for all versions of HTML up to and including HTML5.

An HTML5 web page should include a DOCTYPE declaration, and a `<html>` element that in turn contains a `<head>` element containing the title and character set of the page and a `<body>` element for the content.

The minimum maintainable page

```
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <title>The Smallest Page</title>
  </head>
  <body>
  </body>
</html>
```

The code example above uses the DOCTYPE declaration for HTML5.

```
<!DOCTYPE html>
```

You should write all your new web pages by using HTML5, but you are likely to see many web pages written by using HTML 4.01 or earlier. Pages that are not based on HTML5 commonly use one of the following classes of DOCTYPE:

- **Transitional** DOCTYPES, which allow the use of deprecated, presentation-related elements from previous versions of HTML.

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

- **Frameset** DOCTYPES, which allow the use of frames in addition to the elements allowed by the transitional DOCTYPE.

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

- **Strict** DOCTYPES, which do not permit the use of frames or deprecated elements from previous versions of HTML.

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

At all times, if you cannot use the HTML5 DOCTYPE you should use the strict HTML 4.01 DOCTYPE. If an HTML file has no DOCTYPE, browsers may use their own value and might render your web page inconsistently, so **it is important to include the DOCTYPE**.

Tags, Elements, Attributes, and Content

The head and body of a web page both use HTML elements to define its structure and contents. For example, a paragraph element, representing a paragraph of text on the page, consists of:

- An opening tag, **<p>**, to denote the start of the paragraph.
- Text content.
- A closing tag, **</p>**, to denote the end of the paragraph.

Tags and elements are sometimes referred to interchangeably, although this is incorrect. An element consists of tags and content.

- HTML elements define the structure and semantics of content on a web page

- Elements identify their content by surrounding it with a start and an end tag

- Elements can be nested:

```
<p>
  <strong>Elements</strong> consist of
  <strong>content</strong> bookended by a
  <em>start</em> tag and an <em>end</em> tag.
</p>
```

- Use attributes to provide additional information about the content of an element

Nest elements within each other to elicit more semantic information about the content. If it is not obvious from the context, indent nested elements to help keep track of which are parent and which are child elements.

The body of a simple document

```
<body>
  <h1 class="blue">An introduction to elements, tags and contents</h1>
  <p>
    <strong>Elements</strong> consist of <strong>content</strong> bookended by a
    <em>start</em> tag and an <em>end</em> tag.
  </p>
  <hr />
  <p>
    Certain elements, such as the horizontal rule
    and consist of a
      single, self-closing element. These are known
  </p>
</body>
```

An introduction to elements, tags and contents

Elements consist of content bookended by a *start* tag and an *end* tag.

Certain elements, such as the horizontal rule element, do not need content however and consist of a single, self-closing element. These are known as **empty elements**.

Each **HTML element** tells the browser something about the information that sits between its opening and closing tags. For example, the **strong** and **em** elements represent "strong importance" and "emphasis" for their contents, which browsers tend to render respectively as text in bold and text in italics. **h1** elements represent a top-level heading in your document, which browsers tend to render as text in a large, bold font.

Attributes provide additional information, presentational or semantic, about the contents of an element. They appear within the start tag of an element and have a **name** and a **value**. The name should be in lowercase characters. Most values are pre-defined based on the attribute they are for, and should be contained within quotes. In the previous example, the h1 start tag contains the *class* attribute set to the value *blue*.

Most attributes can qualify only certain elements. However, HTML defines a group of *global attributes* for use with any element.



Additional Reading: You can view a complete list HTML global attributes on the W3C website, at <http://go.microsoft.com/fwlink/?LinkId=267708>.

Displaying Text in HTML

Every web page requires content: text and images. HTML defines many elements that enable you to structure that content and to give it some semantic context.

Headings and Paragraphs

HTML has included elements to identify paragraphs and headings in a document since v1 in 1992.

- **<p>** identifies paragraphs of text.
- **<h1>, <h2>, <h3>, ..., <h6>** identify six levels of heading text. Use **<h1>** to identify the main heading of the entire page, **<h2>** to identify the headings of each section in the page, **<h3>** to identify the sub-sections within those secondary headings, and so on.

Text in HTML can be marked up:

- As headings and paragraphs

```
<h1>An Introduction to HTML</h1>
<p>In this module, we look at the history of HTML and CSS.</p>
<h2>In the Beginning</h2>
<p>WorldWideWeb was created by Sir Tim Berners-Lee at CERN. </p>
```

- With emphasis

To **emphasize** is to give extra weight to (a communication), **"Her gesture emphasized her words"**

- In lists

```
<ul>
  <li>Notepad</li>
  <li>Textmate</li>
  <li>Visual Studio</li>
</ul>
```

It is important to use the heading and paragraph tags to identify sections, sub-sections, and text content in a web page. Headings and tags make the content more understandable to readers and indexers, as well as easier to read on screen.

Marking up text

```
<body>
  <h1>An Introduction to HTML</h1>
  <p>In this module, we look at the history of HTML and CSS.</p>
  <h2>In the Beginning</h2>
  <p>
    WorldWideWeb was a piece of software written
    replacement for
    Gopher. It and HTML v1 were made open source
    we know it
    started with this piece of software.
  </p>
  <h3>Browser Wars</h3>
  <p>The openness of WorldWideWeb meant many diff
on, including Netscape Navigator and NCSA Mosaic,
Explorer.</p>
</body>
```

An Introduction to HTML

In this module, we look at the history of HTML and CSS.

In the Beginning

WorldWideWeb was a piece of software written by Sir Tim Berners-Lee at CERN as a replacement for Gopher. It and HTML v1 were made open source software in 1993. The World Wide Web as we know it started with this piece of software.

Browser Wars

The openness of WorldWideWeb meant many different web browsers were created early on, including Netscape Navigator and NCSA Mosaic, which later became Internet Explorer.

When writing HTML markup, remember that any sequence of whitespace characters—spaces, tabs, and carriage returns—inside text are treated as a single space. The only exception to this is when that sequence is inside a `<pre>` element, in which case the browser displays all the spaces.

Emphasis

HTML defines four elements that denote a change in emphasis in the text they surround from the text in which they are nested:

- `` identifies text that is more important than its surrounding text. The browser usually renders this content in **bold**.
- `` identifies text that needs to be stressed. The browser usually renders this content in *italics*.
- `` and `<i>` identify text to be rendered in bold or in italics, respectively.

You can combine and nest the ``, ``, ``, and `<i>` elements to indicate different types of emphasis.

Browsers can render emphasized text in many different ways.

Adding stress to text

```
<p>
  To <strong>emphasize</strong> is to give extra weight to (a communication); <em>"Her
  gesture emphasized her words"</em>.
</p>
```



Note: The `` and `<i>` elements in HTML4 are simply instructions for displaying the text, rather than specifying some semantic meaning. In HTML5, it is better to use `` and `` rather than `` and `<i>`.

Lists

Lists organize sets of information in a clear and easily understood format. HTML defines three types of list:

- `Unordered lists` group sets of items in no particular order.
- `Ordered lists` group sets of items in a particular order.

- Definition lists group sets of name-value pairs, such as terms and their definitions.

All three list types use a tag to define the start and end of the list - ``, ``, and `<dl>` respectively. Individual list entries are identified with the `` tag for unordered and ordered lists, while definition lists use two tags per list item; `<dt>` for the name, or term, and `<dd>` for its value, or definition.

HTML provides for listing sets of things, steps, and name-value pairs.

Unordered, ordered, and definition lists

```
<body>
  <p>Here's a small list of HTML editors</p>
  <ul>
    <li>Notepad</li>
    <li>Textmate</li>
    <li>Visual Studio</li>
  </ul>
  <p>Here's how to write a web page</p>
  <ol>
    <li>Create a new text file</li>
    <li>Add some HTML</li>
    <li>Save the file to a website</li>
  </ol>
  <p>Here's a small list of people in the</p>
  <dl>
    <dt>Sir Tim Berners Lee</dt>
    <dd>Invented HTML and wrote WorldWideWeb</dd>
    <dt>Linus Torvalds</dt>
    <dd>Originator of Linux</dd>
    <dt>Charles Herzfeld</dt>
    <dd>Authorized the creation of ARPANET, the predecessor of the Internet</dd>
  </dl>
</body>
```

Here's a small list of HTML editors

- Notepad
- Textmate
- Visual Studio

Here's how to write a web page

1. Create a new text file
2. Add some HTML
3. Save the file to a website

Here's a small list of people in the Internet Hall of Fame and what they did

Sir Tim Berners Lee	Invented HTML and wrote WorldWideWeb
Linus Torvalds	Originator of Linux
Charles Herzfeld	Authorized the creation of ARPANET, the predecessor of the Internet

You can also include another list within a list item, as long as the nested list relates to that one specific item. The nested list does not have to be the same type of list as its parent, although context dictates that this is usually the case.

You may write a table of contents as an ordered list of chapter names. Each list item may then include a nested list of headings within that chapter.

Writing nested lists

```
<body>
  <ol>
    <li>Lesson One: Introduction to HTML
      <ol>
        <li>The structure of an HTML page</li>
        <li>Tags, Elements, Attributes and Content</li>
        <li>Text and Images</li>
        <li>Forms</li>
      </ol>
    </li>
    <li>Lesson Two: Introduction to CSS</li>
      <li>Lesson Three: Using Visual Studio 2012</li>
    </ol>
  </body>
```

1. Lesson One: Introduction to HTML
 1. The structure of an HTML page
 2. Tags, Elements, Attributes and Content
 3. Text and Images
 4. Forms
2. Lesson Two: Introduction to CSS
3. Lesson Three: Using Visual Studio 2012

Browsers usually render nested lists by indenting them further into the page, and additionally changing the bullet point style for unordered lists or restarting the list numbering for ordered lists.

Displaying Images and Linking Documents in HTML

You use the HTML **** tag to insert an image into your web page. It does not require an end tag as it does not contain any content. In addition to the global attributes, the **** tag has a number of attributes to define it:

- The **src** attribute specifies a URL that identifies the location of the image to be displayed.
- The **alt** attribute identifies a text alternative for display in place of the image if the browser is still downloading it or cannot display it for some reason; for example, if the image file is missing. It typically describes the content of the image.
- The **title** attribute identifies some text to be used in a **tool tip** when a user's cursor hovers over the image.
- The **longdesc** attribute identifies another web page that describes the image in more detail.
- The **height** and **width** attributes set the dimensions in pixels of the box on the web page that will contain the image; if the dimensions are different from those of the image, browsers will **resize** the image on the fly to fit the box.

Only the **src** attribute is mandatory.

One of the more common types of image to include in a web page is a logo of some kind, like this:

Adding an image to a web page

```
<body>
  <p>
    
  </p>
  <h1>Welcome to my site!</h1>
</body>
```

 **Additional Reading:** With so many hardware devices—phones, tablets, televisions, and monitors—offering the user a chance to browse a web page in many different resolutions, it has become very important to offer the same image at different sizes rather than always getting the browser to scale the picture. The problem now is how to identify which version of the image should be displayed at which resolution. The W3C Responsive Images Community Group <http://www.w3.org/community/respimg/> is hoping to figure out this problem soon.

Hypertext Links

The main reason for the invention of HTML was to link documents together. The **<a>** tag, also known as the anchor tag, allows you to identify a section of content in your web page to link to another resource on the web. Typically the target of this hypertext link is another web page, but it could equally be a text file, image, archive file, email, or web service. When you view your web page in a browser, you click the content surrounded by the anchor tags to have the linked document downloaded by the browser.

Anchor tags have the following non-global attributes:

- The **href** attribute identifies the web page or resource to link to

MCT USE ONLY STUDENT USE PROHIBITED

- The **target** attribute identifies where the browser will display the linked page; valid values are **_blank**, **_parent**, **_self**, and **_top**.
- The **rel** attribute identifies what kind of link is being created.
- The **hreflang** attribute identifies the language of the linked resource.
- The **type** attribute identifies the MIME type of the linked resource.

One common use for hypertext links is to build a navigation menu on a page so the user can visit other pages in the site.

Adding hypertext links to your web page

```
<body>
  <ul>
    <li><a href="default.html" alt="Home Page">Home</a></li>
    <li><a href="about.html" alt="About this Web site">About</a></li>
    <li><a href="essays.html" alt="A list of my essays">Essays</a></li>
  </ul>
</body>
```

The **href** attribute is the most important part of linking one online resource to another. You can use several different value types:

- A URL in the same folder (for example: about.html).
- A URL relative to the current folder (for example: ./about.html).
- A URL absolute to the server root (for example: /pages/about.html).
- A URL on a different server (for example: http://www.microsoft.com/default.html).
- A fragment identifier or id name preceded by a hash (for example: #section2).
- A combination of URL and fragment identifier (for example: about.html#section2).

Gathering User Input by Using Forms in HTML

Many web sites require the user to input information, such as a user name, password, or address. Text and images define content that a user can read, but a form provides a user with a basic level of interaction with a site, giving the user an opportunity to provide data that will be sent to the server for collation and processing.

Use the HTML **<form>** element to identify an area of your web page that will act as an input form. This element has the following attributes:

- The **action** attribute, which identifies the URL of the page to which the form data submitted by the user will be sent for processing.
- The **method** attribute, which defines how the data is sent to the server. Valid values are:
 - GET** for HTTP GET. This is the default value, but is not secure.
 - POST** for HTTP POST. This is the preferred value.

- The **<form>** element provides a mechanism for obtaining user input
 - The **action** attribute specifies where the data will be sent
 - The **method** attribute specifies how the data will be sent
 - Many different input types are available

First name:	<input type="text" value="Paul"/>
Last name:	<input type="text" value="West"/>
Email address:	<input type="text" value="paul.west@contoso.com"/>
Choose a password:	<input type="password" value="*****"/>
Confirm your password:	<input type="password" value="*****"/>
Website/blog:	<input type="text" value="http://www.contoso.com"/>
<input type="button" value="Register"/>	

- The **accept-charset** attribute, which identifies the character encoding of the data submitted in the form by the user.
- The **enctype** attribute, which identifies the MIME-type used when encoding the form data for submission when the method is **POST**.
- The **target** attribute, which identifies where the browser will display the action page; valid values are **_blank**, **_parent**, **_self**, and **_top**.

You can add controls and text elements to the form element's content to define its layout.

Form Controls

An **<input>** element represents the main HTML data entry control and has many different forms based on its **type** attribute, as shown in the following table. In addition, the **value** attribute sets a default value for numeric or text-based controls, and the **name** attribute to identify the **name** of the control.

Value	Result
text (default)	A single-line text box
password	A single-line text box where the text entered into the box is replaced by asterisks.
hidden	A field not visible to the user
checkbox	A checkbox. Provides a yes/no or true/false choice. Use the selected attribute to indicate if it is checked by default.
radio	A radio button control. Use the name attribute to group several radio button controls together. The form will allow either none or one of the grouped radio buttons to be selected.
reset	A reset button. Clicking this resets all the fields to their initial values.
submit	A submit button. Clicking this will submit the current form values to the action page for processing.
image	An image for use as a submit button. Use the src attribute to identify the image to be used.
button	A button. This has no default behavior and may be used to run a script when clicked, for example.
file	A file control. Provides a way to submit a file to the server when the submit button is clicked.

There are four other HTML elements that you can use in a form:

- <textarea>**, which generates a free-form, multiline, plain text edit box; use the **rows** and **cols** attributes to set its size.
- <select>**, which defines a list box or drop-down list. Use the **multiple** attribute to indicate if the user can select more than one item from the list and **<option>** elements nested within **<select>** to identify the items. Use the **<option>**'s **selected** attribute to indicate that it is selected by default and its **value** attribute to indicate a value other than its text content to be sent to the server when the form is submitted.
- <button>**, which defines a button. Use the **type** attribute to indicate whether it is a **submit**, **reset**, or **button** (does nothing) button. The default is **submit**.

You should use the **<button>** element rather than its **<input>** equivalent if you need the content displayed by the button to be more complex than a simple piece of text or a single picture.

Form Layout Elements

You can use **<p>** and **<div>** tags to apply a basic layout to a form. HTML also defines two further tags that can help to improve a form's presentation:

- **<fieldset>**, which identifies a group of controls in a form. The browser reflects this by drawing a box around the contents of the **<fieldset>** and labeling the box with a name. This name is set by using the **<legend>** element, which must be the first child of the **<fieldset>** element.
- **<label>**, which identifies a text label associated with a form control. It does so either by surrounding both the text and the control, or by surrounding the text and setting its **for** attribute to the **id** of the form control.

You can use a form to gather many types of input from the user.

Using a form to obtain the details of a user

```
<form method="post" action="/registration/new" id="registration-form">
  <label for="first-name">First name:</label><br />
  <input type="text" id="first-name" name="FirstName"/><br />
  <label for="last-name">Last name:</label><br />
  <input type="text" id="last-name" name="LastName"/><br />
  <label for="email-address">Email address:</label><br />
  <input type="email" id="email-address" name="EmailAddress"/><br />
  <label for="password">Choose a password:</label><br />
  <input type="password" id="password" name="Password"/><br />
  <label for="confirm-password">Confirm your password:</label><br />
  <input type="password" id="confirm-password" name="ConfirmPassword"/><br />
  <label for="website">Website/blog:</label><br />
  <input type="url" id="website" name="WebsiteUrl" /><br />
  <button type="submit">Register</button>
</form>
```

First name:	<input type="text"/>
Last name:	<input type="text"/>
Email address:	<input type="text"/>
Choose a password:	<input type="text"/>
Confirm your password:	<input type="text"/>
Website/blog:	<input type="text"/>
	<input type="button" value="Register"/>

Demonstration: Creating a Simple Contact Form

In this demonstration, you will see how to build a simple contact form that enables a user to send a message to the organization running a web site.

Demonstration Steps

Create an HTML Page

1. On the Windows 8 **Start** screen, right-click outside of any tile, and in the task bar click **All apps**.
2. On the Windows 8 **Start** screen, in the **Windows Accessories** section, click **Notepad**.
3. Add the following basic HTML structure to the blank text file.

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Contact Us</title>
  </head>
  <body>
  </body>
</html>
```

Save the file as **E:\Mod01\Democode\ ContactUs.html**.

Add Content to the Page

1. Add a level 1 page heading to the **<body>** element.

```
<h1>Contact Contoso Conferencing</h1>
```

2. Add the company address by using two paragraphs and the line break element as follows:

```
<p>Contoso Conferencing Ltd.</p>
<p>123 South Street<br />
Somewhere<br />
Over There<br />
<em>USA</em></p>
```

3. Add a hyperlink for the company's contact email address. Note the use of the **mailto:** protocol rather than the more common **http://**.

```
<p>
<a href="mailto:contact@contoso.com">
    contact@contoso.com</a>
</p>
```

4. Add the following text to invite users to use the contact form.

```
<p>
If you would like to contact Contoso Conferencing, whether you're interested in our services or in a conference we're currently organizing, don't hesitate to contact us using our enquiry form (<strong>Bold fields</strong> are required).
</p>
```

Save the file.

Add a Form with Input Controls

1. Add the following **<form>** element to the page beneath the text created previously.

```
<form method="Post" action="support.aspx">
</form>
```

2. Add the following **<fieldset>** element and submit button to the form (between the **<form ...>** and **</form>** tags).

```
<fieldset>
<legend>
    Your Details and Enquiry
</legend>
</fieldset>
<input type="submit" value="Send" />
```

3. Add the unordered list shown in the following code sample to the **<fieldset>** element below the **</legend>** tag. This list contains input elements for the user's name, telephone number, email address, and a message.

```
<ol>
<li>
    <label>
        <strong>Name</strong><br />
        <input type="text"
            name="UserName" />
    </label>
</li>
<li>
```

```
<label>
    Telephone<br />
    <input type="text"
           name="Phone" />
</label>
</li>
<li>
    <label>
        Email Address<br />
        <input type="text"
               name="Email" />
    </label>
</li>
<li>
    <label>
        <strong>Message</strong><br />
        <textarea name="Message"
                  cols="30" rows="10">Add your message here
        </textarea>
    </label>
</li>
</ol>
```

4. Save the file and close Notepad.

View the Page

1. In the Windows taskbar, click the File Explorer icon.
2. Browse to the folder **E:\Mod01\Democode**.
3. Double-click **ContactUs.html** to display the page in Internet Explorer.
4. In the **How do you want to open this type of file (.html)?** dialog box, click **Internet Explorer**.

The page should look like this:

FIGURE 1.1:THE CONTACTUS PAGE

- Enter some sample data, but do not click **Send**.

 **Note:** If you do click Send, Internet Explorer will display the message **This page can't be displayed**. This occurs because the URL that is specified as the action attribute for the form (support.aspx) does not exist.

Attaching Scripts to an HTML Page

HTML enables you to define the layout for your web pages, but apart from the **<form>** element it does not provide for any interaction with the user. Additionally, the layout defined by using HTML markup tends to be fairly static. You can add dynamic behavior to a page by writing JavaScript code.

There are several ways that you can include JavaScript in your web page, all involving the **<script>** element:

- Write the JavaScript on the page as the content part of a **<script>** element.

- HTML is static, but pages can use JavaScript to add dynamic behavior

- Use the **<script>** element to specify the location of the JavaScript code:

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

- The order of **<script>** elements is important
- Make sure objects and functions are in scope before they are used

- Use the **<noscript>** element to alert users with browsers that have scripting disabled.

```
<script type="text/javascript">
    alert('I am a line of JavaScript');
</script>
```

- Save the JavaScript in a separate file on your web site and then reference it by using the **src** attribute of the **<script>** element.

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

- Reference a third-party JavaScript file on a different web site.

```
<script type="text/javascript"
    src="http://ajax.contoso.com/ajax/jQuery/jquery-1.7.2.js">
</script>
```

The **<script>** element has three attributes:

- The **type** attribute, which identifies which script language is used; the default is **text/javascript**.
- The **src** attribute, which identifies a script file for download; do not use **src** if you are writing script into the content part of the **<script>** element.
- The **charset** attribute, which identifies the character encoding (for example, utf-8, Shift-JIS) of the external script file; if you are not using the **src** attribute, do not set the **charset** attribute.

Always specify both start and end **<script>** tags, even if you are linking to an external script file and you have no content between the tags.

It is common for a web application to divide JavaScript functionality into several scripts. Additionally, many web applications use third party JavaScript files (such as those that implement jQuery). The order in which you add links to JavaScript files is important, and to ensure that they are in scope you must add links to scripts that define objects and functions before the scripts that use these objects and functions.

Older browsers do not always support JavaScript, and sometimes users running more modern browsers may disable JavaScript functionality for security reasons. In these cases, any features on your web pages that use JavaScript may not run correctly. You can alert the user that this is the case by using the **<noscript>** element. This element enables a browser to display a message, warning the user that the page may not operate correctly unless JavaScript is enabled.

Use the **<noscript>** element to alert users that your page uses JavaScript, and so the user should enable JavaScript in the browser in order to display your page correctly.

The **<noscript>** element

```
<body>
    <noscript>This page uses JavaScript. Please enable it in your browser</noscript>
    ...
    Rest of page
    ...
    <script src="MyScripts.js"></script>
</body>
```

 **Best Practice:** In general, it is good practice to add links to scripts as the last elements nested inside the **<body>** element. Also, remember that the **order of the scripts is important**, so add links for dependent scripts after those on which they depend.

Lesson 2

Overview of CSS

Where HTML defines the structure and context of a web page, CSS defines how it should appear in a browser. In this lesson, you will learn the fundamentals of CSS, how to create some basic styles, and how to attach these styles to elements of an HTML page.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain basic CSS syntax.
- Describe how CSS selectors work.
- Describe how CSS styles inherit and cascade.
- Attach CSS to an HTML page.

Overview of CSS Syntax

CSS is an acronym for Cascading Style Sheets. CSS provides the standard way of defining how a browser should display the contents of a web page. CSS enables you to attach presentation rules to fragments of HTML based on selectors that target HTML elements by name, id, or class. It also enables you to vary how a page is presented according to the form factor of the device on which it is displayed, from a large monitor to a smartphone, and even on an audio reader or printer.

Every CSS rule has the same basic structure:

```
selector {
    property1:value;
    property2:value;
    ..
    propertyN:value;
}
```

This example shows the four parts of every CSS rule:

- A **selector** defines the element or set of elements to target. The styling specified by the CSS rules associated with this selector is applied to all elements on the web page that match this selector. A CSS selector can specify the type of element such as `div` to select all `<div>` elements, or the name attribute of a specific instance of an element. You can also select multiple element types such as `p,div` to select all `<p>` and all `<div>` elements. You can even `*` to select all elements. Other selector expressions are also possible, and these are described later in this course.
- A pair of curly braces encloses the rules for the selected elements. A rule defines how to render the selected element; it contains a property-value pair suffixed by a semi-colon.
- A **property** identifies the visual aspect of the selected element to change.

• All CSS rules have the same syntax:

```
selector {
    property1:value;
    property2:value;
    ..
    propertyN:value;
}
```

• Comments are enclosed in `/* ... */` delimiters

```
/* Targets level 1 headings */
h1 {
    font-size: 42px;
    color: pink;
    font-family: 'Segoe UI';
}
```

- A **values** variable specifies the styling to apply to the property. Values can vary depending on the property. They might be color names, size values in percentages, pixels, ems, or points, or the name of a font, to name three possibilities.

You can also add **comments** to your style sheets by using `/* */` delimiters. The browser will ignore comments. Comments can span one or more lines. You can write them outside of or within a CSS rule.

All CSS rules have the same basic syntax. Beyond that, the first key to CSS is to know the properties to apply to sets of elements. This example demonstrates the use of some text-specific properties.

Some simple CSS rules

```
/* Targets level 1 headings and renders them as large pink text using the Segoe UI font */
h1 {
    font-size: 42px;
    color: pink;
    font-family: 'Segoe UI';
}
/* Targets emphasized text, rendering it as italicized on a yellow background */
em {
    background-color: yellow; /* Yellow is a good highlight color */
    font-style: italic;
}
```

In the example above, the two rules translate as follows:

- Every `<h1>` element should have text that is 42px high, pink, and in Segoe UI font.
- Every `` element should have a yellow background color and its text in italics.

When writing CSS, note that any sequence of whitespace characters is treated as a single space character.

How CSS Selectors Work

CSS selectors specify the content to be styled by using the associated set of rules, and understanding how CSS selectors work is the key to defining reusable and extensible style sheets.

The CSS specification provides many different ways to select the element or set of elements in a web page to which presentation rules will apply. The following list summarizes the basic selectors and the set of elements that they identify.

- **The element selector** identifies the group of all elements in the page with that name. For example, `h2 {}` returns the set of all level two headings in the page.
- **The class selector**, identified by a period, returns the set of all elements in the page where the **class** attribute is set to the specified value. For example, `.myClass {}` returns the set of elements where the **class** attribute is set to "myClass".
- **The id selector**, identified by a hash, returns the set of all elements in the page where the **id** attribute is set to the specified value. For example, `#thisId {}` returns the set of any elements where the **id** attribute is set to "thisId".

- There are three basic CSS selectors

- The element selector: `h2{}`
- The class selector: `.myClass {}`
- The id selector: `#thisId {}`

- CSS selectors can be combined to create more specific rules

- The wildcard `*` selector returns the set of all elements

- Use [...] to refine selectors based on attribute values

The class selector may return a set containing different types of HTML elements—for example, `<p>`, `<section>`, and `<h3>`—if they all have the same class attribute value. The same is true of the id selector, although this selector should only return a single element because the id attribute in a page should be unique (this is not enforced).

Style sheets are often written with the least specific selectors first and the most specific selectors last. Element selectors are the least specific, followed by class and id selectors, and then combinations of the three.

Introducing the element, class, and id selectors

```
h2 {
    font-size: 24px;
}
.red {
    color: red;
}
#alert {
    background-color: red;
    color: white;
}
```

You can combine selectors by using concatenation. In the following example, the two rules combine selectors to identify a more specific set of elements than either does on its own.

The selector, `h2.blue` returns the set of `<h2>` elements with the class "blue", and `h2#toc` returns the set of `<h2>` elements with id "toc".

Combining selectors

```
h2.blue {
    color: blue;
}
h2#toc {
    font-weight: bold;
}
```

Note that these two sets may intersect, in which case the CSS properties and values for both rules will apply—in this case to the set of `<h2>` elements with id "toc" and class "blue".

The following table shows examples of the various ways you can concatenate selectors and the set of elements that the browser returns:

<code>h2.blue</code>	Returns any <code><h2></code> elements of class "blue"
<code>h2#blue</code>	Returns any <code><h2></code> elements with id "blue"
<code>section, h2</code>	Returns any <code><h2></code> and any <code><section></code> elements
<code>section h2</code>	Returns any <code><h2></code> elements nested within a <code><section></code> element at any level.
<code>section > h2</code>	Returns any <code><h2></code> elements nested immediately under a <code><section></code> element
<code>section + h2</code>	Returns any <code><h2></code> elements immediately following and sharing the same parent element as a <code><section></code> element

h2.blue	Returns any <code><h2></code> elements of class "blue"
<code>section ~ h2</code>	Returns any <code><h2></code> elements following and sharing the same parent element as a <code><section></code> element

The Wildcard Selector

The wildcard selector `*` returns the set of all the elements in a document. This selector is rarely used by itself, but it can be useful in combination with other elements. For example, the following rule returns a set of all elements within an `<aside>` element and makes them slightly fainter.

```
aside * { opacity : 0.6; }
```

The Attribute Selector

You can further refine any of the selectors already described by inspecting an element for the declaration of an attribute and its value. The attribute selector is contained in a pair of square brackets appended to a selector, and can have any of the following forms:

<code>input[type]</code>	Returns any <code><input></code> elements that use the <code>type</code> attribute, whatever its value.
<code>input[type="text"]</code>	Returns any <code><input></code> elements where the <code>type</code> attribute value is exactly equal to the string "text".
<code>input[foo~="red"]</code>	Returns any <code><input></code> elements where the <code>foo</code> attribute (for instance, the <code>class</code> attribute) contains a space-separated list of values, one of which is exactly equal to "red".
<code>input[type^="sub"]</code>	Returns any <code><input></code> elements where the <code>type</code> attribute value begins exactly with the string "sub".
<code>input[type\$="mit"]</code>	Returns any <code><input></code> elements where the <code>type</code> attribute value ends exactly with the string "mit".
<code>input[type*="ubmi"]</code>	Returns any <code><input></code> elements where the <code>type</code> attribute value contains the substring "ubmi".
<code>input[foo = "en"]</code>	Returns any <code><input></code> elements where the <code>foo</code> attribute value is either exactly "en" or begins exactly with "en-", i.e. the value plus a hyphen.

You can combine attribute selectors by concatenating them. For example, to return a set of all checkboxes that are checked by default, you would use the following selector:

```
input[type="checkbox"][selected] {}
```

How HTML Inheritance and Cascading Styles Affect Styling

As the phrase Cascading Style Sheets suggests, elements on a page can be subjected to several cascading transformations depending on relationships among the elements and the style sheets associated with a page. To create successful style sheets, it is important to understand two concepts: inheritance between HTML elements, and how multiple CSS rules cascade as they are applied to HTML elements.

- HTML inheritance and the CSS cascade mechanism govern how browsers apply style rules
- HTML inheritance determines which style properties an element inherits from its parent
- The cascade mechanism determines how style properties are applied when conflicting rules apply to the same element

HTML Inheritance

In a web page, HTML elements inherit some properties from their parent elements unless specified otherwise. This is of great importance; without inheritance, you would have to declare several rules for every single element on a page.

Consider the scenario in which you want all text on a web page to use the Candara font. You can set the `<body>` element of your page to use this font, and inheritance means that text in every other element nested inside the body will also use Candara unless another, more specific rule supplants it.

```
body {
    font-family: Candara;
}
```

If inheritance didn't exist, you would have to set this property on every single type of element containing text content. You might end up writing many repeating styles as shown in the following code example, which can be difficult to maintain. You would probably also be looking for an alternative to CSS.

```
h1, h2, h3, h4, h5, h6 {
    font-family: Candara;
}
p {
    font-family: Candara;
}
...
```

 **Note:** Not all CSS properties are inherited from parent to child, because it would make no sense to do so. For example, if you set a background image for an `<article>` element, it would probably not be useful for all the child sections and paragraphs to display the same background image.

Cascading Rules

A single element in an HTML page may be matched against more than one selector in a style sheet and be subjected to several different styling rules. The order in which these rules are applied could cause the element to be rendered in different ways. The cascade mechanism is the way in which style rules are derived and applied when multiple, conflicting rules apply to the same element; it ensures that all browsers display the element in the same way.

There are three factors that browsers must take into consideration when they apply styling rules:

1. **Importance.** You can ensure a certain property is always applied to a set of elements by appending the rule with `!important`.

```
h2 { font-weight : bold !important; }
```

2. **Specificity.** Style rules with the least specific selector are applied first, then those for the next specific, and so on until the style rules for the most specific selector are applied.
3. **Source order.** If styles rules exist for selectors of equal specificity, they are applied in the order given in the style sheet.



Reference Links: For more information on inheritance and the cascade, including details on how the specificity of a selector is calculated, go to <http://go.microsoft.com/fwlink/?LinkId=267710>

Adding Styles to An HTML Page

HTML enables you to attach your CSS rules to your web page in three ways. You can:

- Write rules specific to an element within its style attribute.

```
<p style="font-family : Candara; font-size: 12px; "> ... </p>
```

- Write a set of rules specific to a page within its `<head>` element by using `<style>` tags.

```
<style type="text/css">
  p {
    font-family : Candara; font-size: 12px;
  }
</style>
```

- Write all your rules in a separate style sheet file with the .css extension, and then reference it in the markup of the page by using a `<link>` tag. The most common place to add a `<link>` tag is within the `<head>` element.

```
<link rel="stylesheet" type="text/css" href="mystyles.css" media="screen">
```

The `<link>` element has four CSS-relevant attributes:

- The **href** attribute specifies a URL that identifies the location of the style sheet file.
- The **rel** attribute indicates the type of document the `<link>` element is referencing; set this to **style sheet** when linking to style sheets.
- The **media** attribute indicates the type of device targeted by the style sheet; possible values include **speech** for speech synthesizers, **print** for printers, **handheld** for mobile devices and **all** (the default), indicating the style sheet is all purpose.
- The **type** attribute indicates the MIME type of the document being referenced; the correct type for style sheets is **text/css** which is also the default value for this attribute.

The **type** and **media** attributes have the same function for the `<style>` element as their namesakes for the `<link>` element.

Note that styles are applied in order, from top to bottom, as the page is parsed and processed. Later styles override earlier styles that are applied to the same element. For example, if you define styles in a `<head>`

element and then subsequently add a <link> that references a stylesheet with different styling for the same elements, then the styles in the stylesheet will override those defined directly in the <head> element. However, if you define the styles in a <head> element after a <link> that references a stylesheet, then the styles defined directly will take precedence over those in the stylesheet. If you define styles inline as part of an element, they always override any other styling for that element.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3

Creating a Web Application by Using Visual Studio 2012

Visual Studio 2012 offers a comprehensive set of tools for developing web applications. From HTML formatting to JavaScript debugging, it is a must-use tool for web application developers using the Windows platform. Internet Explorer 10 also contains a number of useful tools for inspecting a web page and the style sheets and scripts it references.

This lesson introduces Visual Studio 2012 and the support it includes for building web applications. This lesson also examines the F12 Developer Tools for Internet Explorer.

Lesson Objectives

After completing this lesson, you will be able to:

- Open, inspect and run a web application by using Visual Studio 2012.
- Explain how to use the F12 Developer Tools for Internet Explorer.

Developing Web Applications by Using Visual Studio 2012

Visual Studio 2012 is the current version of Microsoft's primary suite of tools for developers. With Visual Studio 2012, you can perform the following tasks:

- Create web applications using HTML5, CSS3, and JavaScript.

Visual Studio 2012 provides a collection of project templates to get you started building web applications. If you are building applications by using HTML5 and JavaScript only, the most appropriate templates are the ASP.NET Empty Web Site and ASP.NET Empty Web Application templates. You can structure your web application to organize the content by creating additional folders inside the project.

- Debug web applications.

- Visual Studio 2012 provides tools for:
 - Creating a web application project, and adding folders to structure the content
 - Debugging JavaScript code, examining and modifying variables, and viewing the call stack
 - Deploying a web application to a web server or to the cloud
- Visual Studio 2012 features include:
 - Full support for HTML5
 - IntelliSense for JavaScript code
 - Support for CSS3 properties and values
 - CSS color picker

Visual Studio 2012 provides extensive debugging features for JavaScript code. You can set breakpoints, single step through code, examine and modify the contents of variables, view the call stack, and perform many other common debugging tasks.

- Deploy web applications to a web server, or to the cloud with Windows Azure.

Visual Studio 2012 provides wizards that enable you to quickly deploy a web application to a web server. If you have the Windows Azure SDK installed, you can deploy a web application directly to the cloud.

There are several editions of Visual Studio. At one end of the scale, Visual Studio Express 2012 for Web is free to download and install. It targets hobbyists. At the other end, Visual Studio Ultimate 2012 is aimed at large teams of developers building and maintaining enterprise-level applications.

You can use Visual Studio in conjunction with Team Foundation Server to provide source code control and tracking for team development.

All versions of Visual Studio 2012 have the same core level of support for developing web applications. This includes:

- An HTML editor updated to recognize HTML5 tags and attributes.
- A new JavaScript editor with IntelliSense support.
- A CSS editor updated to support CSS3 properties and value types.
- A new CSS color picker.

Visual Studio also includes a new development web server called IIS Express. This web server provides many of the same features as IIS, except that it is optimized for the development environment.

 **Note:** Previous editions of Visual Studio provided the ASP.NET Development Server. This server is still available, but IIS Express has fewer limitations. For example, IIS Express includes the integrated pipeline mode of the full IIS that was not available in the ASP.NET Development Server.

Demonstration: Creating a Web Site by Using Visual Studio 2012

Visual Studio 2012 is the current version of the Microsoft integrated development environment (IDE). This demonstration shows the primary features that Visual Studio provides for building a web site.

Demonstration Steps

Create a Web Site Project

1. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
2. In Visual Studio, on the **File** menu, point to **New**, and then click **Web Site**.
3. In the **New Web Site** dialog box, click **ASP.NET Empty Web Site**.

 **Note:** It does not matter whether you select the **Visual Basic** or **Visual C#** templates in the left pane; the templates for both languages enable you to create HTML5 web pages and implement functionality by using JavaScript.

4. From the web location drop-down list, click **File System**, set the file path to **E:\Mod01\Democode\DemoWebSite**, and then click **OK**.
5. On the **File** menu, click **Save DemoWebSite**.

Add and Edit Files in the Project

1. In Visual Studio, click **Solution Explorer**.
2. In the Solution Explorer window, right-click the **DemoWebSite** project.
3. In the context menu, point to **Add** and then click **Existing Item**.
4. In the **Add Existing Item** dialog box, browse to **E:\Mod01\Democode**, click **ContactUs.html**, and then click **Add**.
5. In the Solution Explorer window, right-click **ContactUs.html**, and then click **Set As Start Page**.
6. In the Solution Explorer window, right-click the **DemoWebSite** project, point to **Add**, and then click **New Folder**.

7. Change the name of the folder to **styles**.
8. Right-click the **styles** folder, point to **Add**, and then click **Add New Item**.
9. In the **Add New Item - DemoWebSite** dialog box, in the middle pane, click **Style Sheet**, in the **Name** box, type **ContactUsStyles.css**, and then click **Add**.

 **Note:** You can also use the **Add New Item** dialog box to create new JavaScript and HTML files and add them to a project.

10. In the **ContactUsStyles.css** file, add the following style shown in bold:

```
body {  
    font-family: 'Times New Roman';  
    color: blue;  
}
```

 **Note:** Notice that the new CSS color picker appears when you specify the **color** property. You can use the color picker to select a color and to generate the corresponding code for the color.

11. In the Solution Explorer window, double-click **ContactUs.html**.
12. Add the following markup shown in bold to the **<head>** element of the page.

```
<head>  
    <meta charset="UTF-8" />  
    <title>Contact Us</title>  
    <link href="styles/ContactUsStyles.css" rel="stylesheet" type="text/css" />  
</head>
```

 **Note:** HTML IntelliSense provides hints to help ensure that you enter valid HTML. The **Pick URL** wizard enables you to quickly select a style sheet.

13. On the **File** menu, click **Save All**.

Run the Web Application

1. On the **Debug** menu, click **Start Debugging**.
2. In the **Debugging Not Enabled** dialog box, click **Modify the Web.config file to enable debugging**, and then click **OK**.
3. Verify that Internet Explorer starts running and displays ContactUs.html. The text for the page should appear in blue.
4. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.

 **Note:** You can enter some sample data, but do not click **Send** because the URL that is the target of the form is not available.

5. Leave Internet Explorer running and return to Visual Studio 2012.

Modify the Live Application

1. In the **ContactUs.html** file, make the following modifications:
 - a. Change **Name** to **Full name**.
 - b. Change **Telephone** to **Telephone number**.

2. In the ContactUsStyles.css file, add the following style:

```
h1 {
  font-family: 'Copperplate Gothic';
  color: red;
}
```

3. On the **File** menu, click **Save All**.
4. Return to Internet Explorer and press F5 to refresh the display.
5. Verify that the **Name** field changes to **Full name**, the **Telephone** field changes to **Telephone number**, and that the style of the heading has changed.
6. Return to Visual Studio 2012.
7. On the **Debug** menu, click **Stop Debugging**.

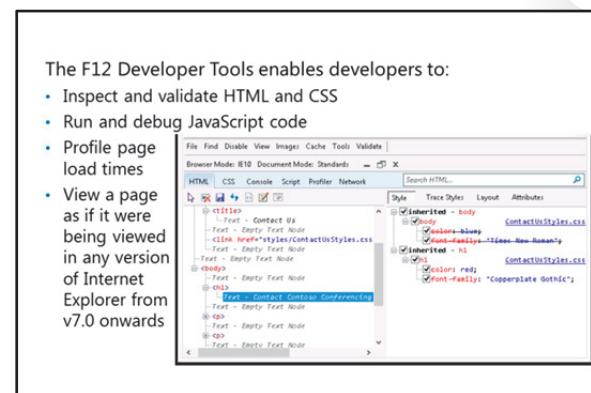
Using the Internet Explorer F12 Developer Tools

As an aid to developers wishing to inspect and debug how their web pages are delivered to their browser, and also in response to similar capabilities in other browsers, Microsoft has provided the F12 Developer Tools as a feature of Internet Explorer. These tools enable a developer to quickly perform the following tasks while the application is running in Internet Explorer:

- Inspect and validate HTML markup and CSS style sheets.
- Run and debug JavaScript code.
- Profile page load times to optimize an application.
- View a page as if it were being viewed in Internet Explorer from v7.0 onwards.

The F12 Developer Tools enables developers to:

- Inspect and validate HTML and CSS
- Run and debug JavaScript code
- Profile page load times
- View a page as if it were being viewed in any version of Internet Explorer from v7.0 onwards



In Internet Explorer, press **F12** to display the F12 Developer Tools while a web application is running

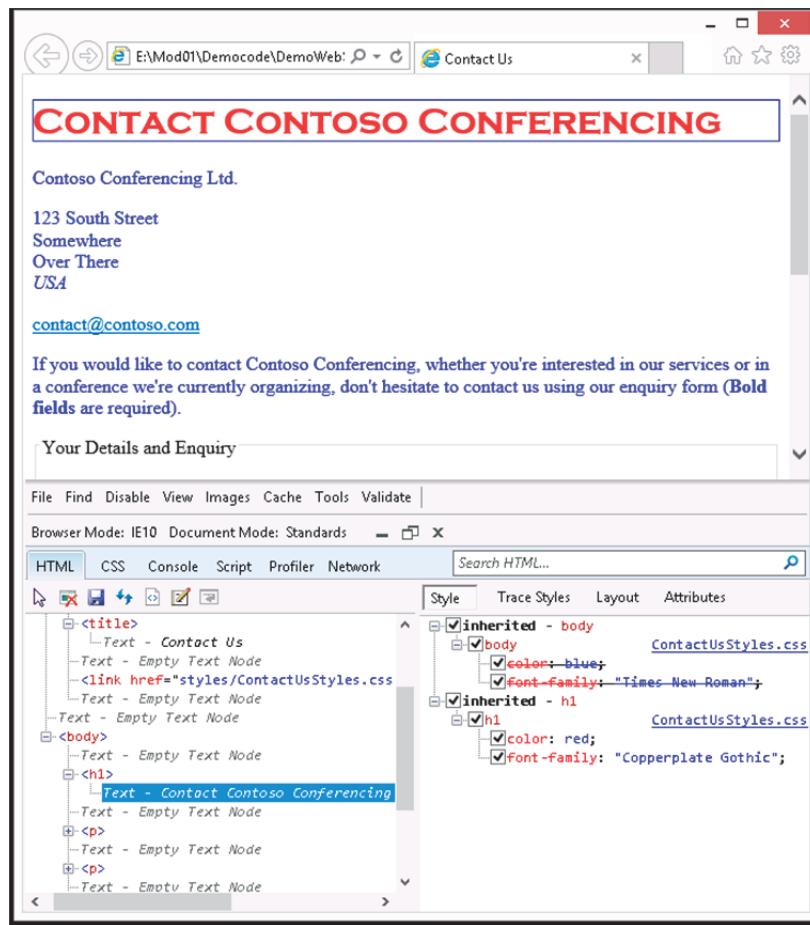


FIGURE 1.1:THE F12 DEVELOPER TOOLS

The menu bar across the top of the **Tools** window gives you quick access to some Internet Explorer features and some online document validation services:

- The **File** menu gives access to help documents and to close the Tools window.
- The **Find** menu enables you to select the HTML markup for an element by clicking on it.
- The **Disable** menu enables you to toggle Internet Explorer's support for blocking popups, script and CSS.
- The **View** menu enables you to toggle the display of information about the elements on the page such as link paths and tab indexes; this information overlays the page.
- The **Images** menu enables you to toggle Internet Explorer's support for displaying images along with the display of information about the images in the page.
- The **Cache** menu enables you to change Internet Explorer's cache and cookie settings.
- The **Tools** menu gives access to several useful tools; for example, you can change the user agent string to emulate the behavior of the page in other browsers.
- The **Validate** menu allows you to validate your HTML and CSS documents against the current standards and accessibility checklists.

The rendering engine that displays HTML in the Internet Explorer window changes with each new version, so the **Browser Mode** and **Document Mode** drop-down lists allow you to select which version of Internet Explorer you wish to mimic to see how it would render your page.

Beneath the menu bar, there are six tabs:

- The **HTML** tab has two panes. The left shows a tree-view of the HTML elements in the current page: its Document Object Model. Clicking on any element in the tree-view populates the right pane with its CSS styles and attribute values. You can also use this pane to edit the HTML content of a page.
- The **CSS** tab enables you to inspect any of the style sheets referenced by the current document and enable\disable any rule or property within a rule to see how that affects the presentation of the page.
- The **Console** tab lets you view any error messages from Internet Explorer during the execution of the page.
- The **Script** tab enables you to step through and debug any JavaScript executing on your page.
- The **Profiler** tab lets you to examine the performance of each call and function in your JavaScript code. This feature is useful for determining the hot spots in your application and for highlighting areas where it may be beneficial to optimize your JavaScript code.
- The **Network** tab enables you to inspect how and when the various images, style sheets, scripts, and other resources are downloaded as part of your page, why they were downloaded, and how long they took to download.

Demonstration: Exploring the Contoso Conference Application

In this demonstration, you will learn how to open the Contoso Conference application in Visual Studio, and how to run the application.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Exploring the Contoso Conference Application

Scenario

ContosoConf is an annual technical conference that describes the latest tools and techniques for building HTML5 web applications. The conference organizers have created a web site to support the conference, using the same technologies that the conference showcases.

You are a developer that creates web sites by using HTML, CSS, and JavaScript, and you have been given access to the code for the web site for the latest conference. You decide to take a look at this web application to see how it works, and how the developer has used Visual Studio 2012 to create it.

Objectives

After completing this lab, you will be able to:

- Describe the structure of the Contoso Conference web application.
- Use Visual Studio 2012 to examine the structure of a web application, run a web application, and modify a web application.
- Estimated Time: 30 minutes.
- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: Student.
- Password: Pa\$\$w0rd.

Exercise 1: Exploring the Contoso Conference Application

Scenario

In this exercise, you will run the Contoso Conference web application and examine each of the functions it provides.

The Contoso Conference web application contains the following pages:

- The Home page, which provides a brief overview of the conference, the speakers, and the sponsors. The Home page also includes a video from the previous conference.
- The About page, which provides more detail about the conference and the technologies that it covers.
- The Schedule page, which lists the conference sessions. The conference has two concurrent tracks, and the sessions are organized by track. Some sessions are common to both tracks.
- The Register page, which enables the user to provide their details and register for the conference.
- The Location page, which provides information about the conference location and a map of the venue.
- The Live page, which enables an attendee to submit technical questions to the speakers running the conference sessions. The page displays the answer from the speaker, together with questions (with answers) posted by other conference attendees.
- The Feedback page, which enables the user to rate conference sessions and speakers.

The main tasks for this exercise are as follows:

1. Start the web application and view the Home page.
2. View the About and Schedule pages.
3. View the Register page and register as a new attendee.

MCT USE ONLY. STUDENT USE PROHIBITED

4. View the Location page.
5. Submit a question and provide conference feedback.

► **Task 1: Start the web application and view the Home page**

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio.
4. Open the **ContosoConf** solution in the **E:\Mod01\Labfiles\Starter** folder
5. Start the application without debugging.

The **Home** page is displayed in Internet Explorer, like this:

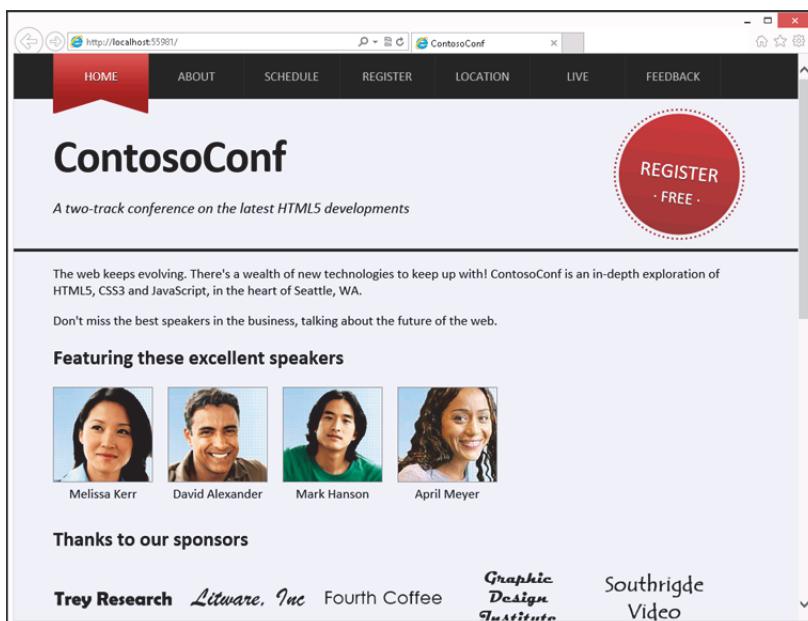


FIGURE 1.2:THE HOME PAGE

The graphics elements for the speakers and sponsors are implemented by using HTML **img** elements. The sources of the images are jpg and png files.

6. Scroll to the bottom of the page and play the video from the previous conference. This functionality is implemented by using the HTML5 **video** element.
7. Pause the video.



Note: You will not hear any sounds because Hyper-V does not provide a virtual audio device.

8. Scroll to the top of the **Home** page and hover the mouse over the **Register Free** icon. Notice that the icon rotates and expands as the mouse enters the icon. This feature is implemented by using CSS.
9. At the very top of the page, move the mouse over the menu bar listing the names of the pages in the application. Do not click any menu items. Notice that each item is highlighted as the mouse traverses it. This feature is implemented by using an HTML **nav** element and CSS.

► Task 2: View the About and Schedule pages

1. Using the menu bar, move to the **About** page.

The **About** page looks like this:

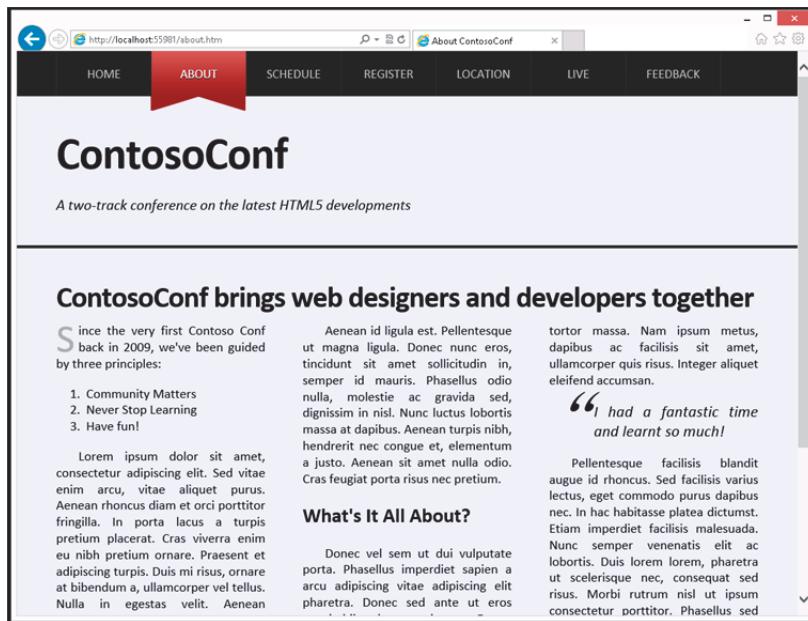


FIGURE 1.3:THE ABOUT PAGE

Notice that when you click an item in the menu bar, the style of the item changes; it is displayed with a ribbon effect. This feature is implemented by using CSS.

The other styling features, including the large drop-capital "S" at the start of the first paragraph, the column layout, and the quotation in the third column, are also implemented by using CSS.

2. Move to the **Schedule** page.

The **Schedule** page looks like this:

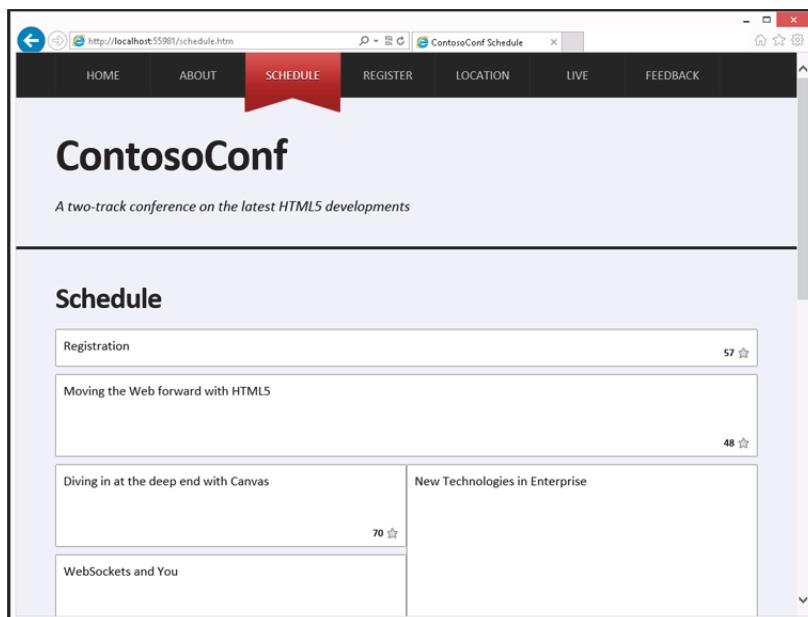


FIGURE 1.4:THE SCHEDULE PAGE

The list of sessions is held in a database that is accessed by using a web service. This page uses JavaScript code to connect to the web service, retrieve the list of sessions, and dynamically populate the body of this page with the session information.

3. Select the session **Moving the Web forward with HTML5** and click the star icon. When this happens, notice that the star changes color and that number next to the star increases. This number indicates how many attendees have expressed an interest in this session; to get a good seat, the user may need to arrive early for popular sessions.
4. Click the star again to deselect the session. The number of interested attendees drops by one.

The functionality is implemented by a combination of CSS and JavaScript code that sends information to another web service about the sessions that a user selects.

► Task 3: View the Register page and register as a new attendee

1. Move to the **Register** page.

The **Register** page looks like this:

A screenshot of a web browser window displaying the 'Register for ContosoConf' page. The page has a dark header with navigation links: HOME, ABOUT, SCHEDULE, REGISTER (which is highlighted in red), LOCATION, LIVE, and FEEDBACK. The main content area has a light gray background. At the top, it says 'ContosoConf' and 'A two-track conference on the latest HTML5 developments'. Below that is a section titled 'Register for the conference' containing several input fields: 'First name:' with a placeholder 'Eric', 'Last name:' with a placeholder 'Gruber', 'Email address:' with a placeholder 'dummy data', 'Choose a password:' with a placeholder 'password123', 'Confirm your password:' with a placeholder 'password123', and 'Website/blog:' with a placeholder 'http://'. The entire form is enclosed in a light gray border.

2. Register the details for a new attendee. Enter the following information and then click **Register**:
 - o First name: **Eric**
 - o Last name: **Gruber**
 - o Email address: **dummy data**

Notice that the page performs the following validations:

- o All fields apart from **Website/blog** are mandatory.
- o The **Email address** must be in the correct format.
- o The **password** must contain at least 5 letters and numbers.
- o The value entered for the **Confirm your password** field must match the **password** field.

This validation is performed by using a combination of HTML5 forms validation controls, and JavaScript code. The styling of the fields when they display an error is controlled by using CSS.

3. Complete the data by providing the following information, and then click **Register**:

- Email address: **grubere@contoso.com**
- Choose a password: **abc1234**
- Confirm your password: **wxyz9999**

Notice that this time a different error message appears because the values specified for the two password fields are not the same.

4. Change the value in the **Confirm your password** field to **abc1234**, and then click **Register** again.

When you have successfully registered, the confirmation page appears.

The confirmation page looks like this:

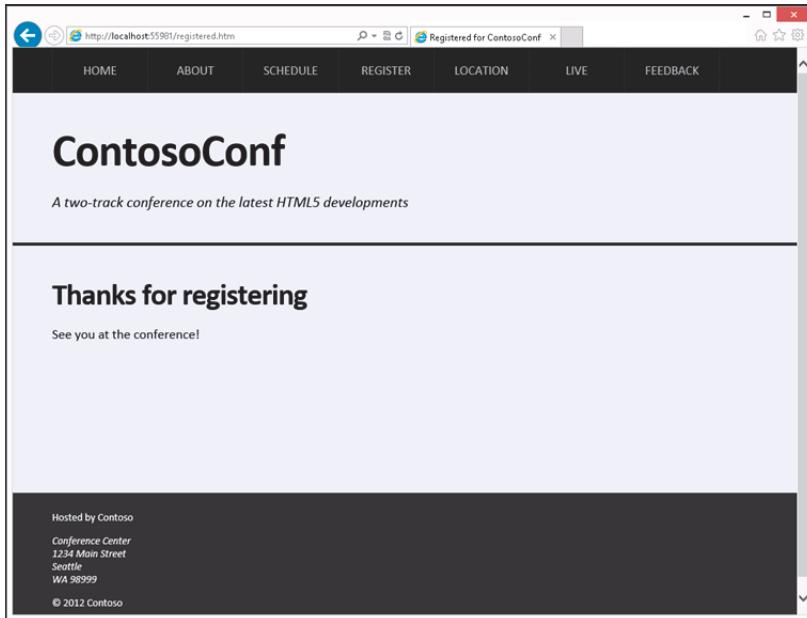


FIGURE 1.6:THE REGISTRATION CONFIRMATION PAGE

► **Task 4: View the Location page**

1. Move to the **Location** page.

If the message **localhost wants to track your physical location** appears in the Internet Explorer message bar, click **Allow once**. In the **Enable Location Services** message box, click **Yes**.

The page displays information about your current location (the distance from the conference venue) by using the Geolocation API in JavaScript.

The **Location** page looks like this:

MCT USE ONLY. STUDENT USE PROHIBITED

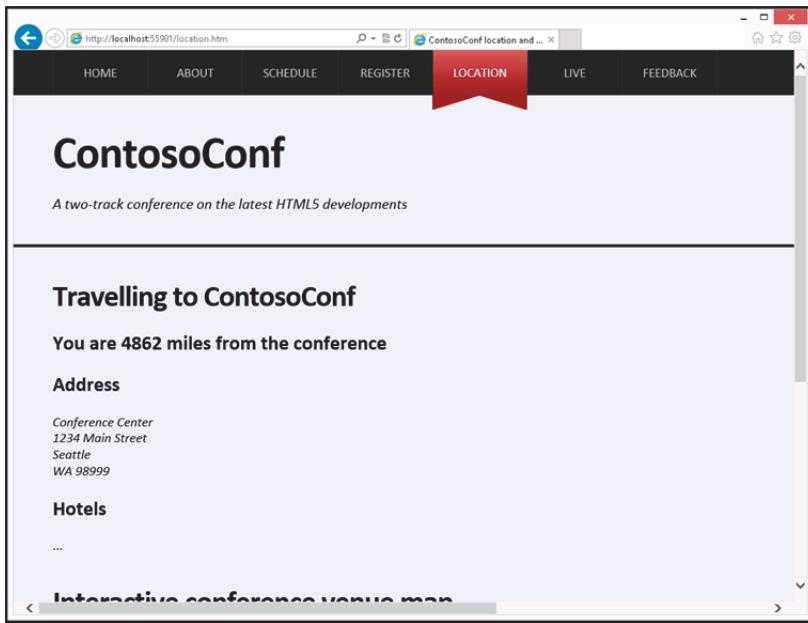


FIGURE 1.7:THE LOCATION PAGE

2. Scroll to the bottom of the page. The venue map that appears is generated by using Scalable Vector Graphics.

► **Task 5: Submit a question and provide conference feedback**

1. Move to the **Live** page.

The **Live** page looks like this:

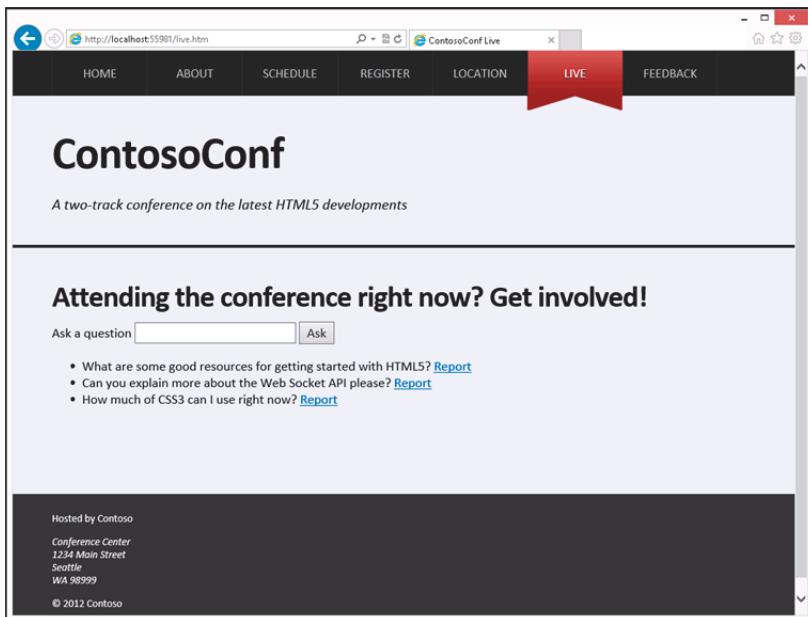


FIGURE 1.8:THE LIVE PAGE

The **Live** page enables an attendee to submit questions to a speaker and to view the response. The page also displays questions asked by other attendees.

2. Type the question **What is the best way to learn HTML5?**, and then click **Ask**.
3. Review the questions that are displayed. This page also enables an attendee to report any questions that they feel are unsuitable or offensive.

4. Select the question that you just asked and report it. The question will be vetted and then disappear.

Questions and reporting are managed by using a web socket server. The application connects to this server by opening a client connection and sending requests asynchronously. As other attendees post questions, the JavaScript code behind this page automatically updates the list of questions that is displayed.

5. Move to the **Feedback** page.

The **Feedback** page looks like this:

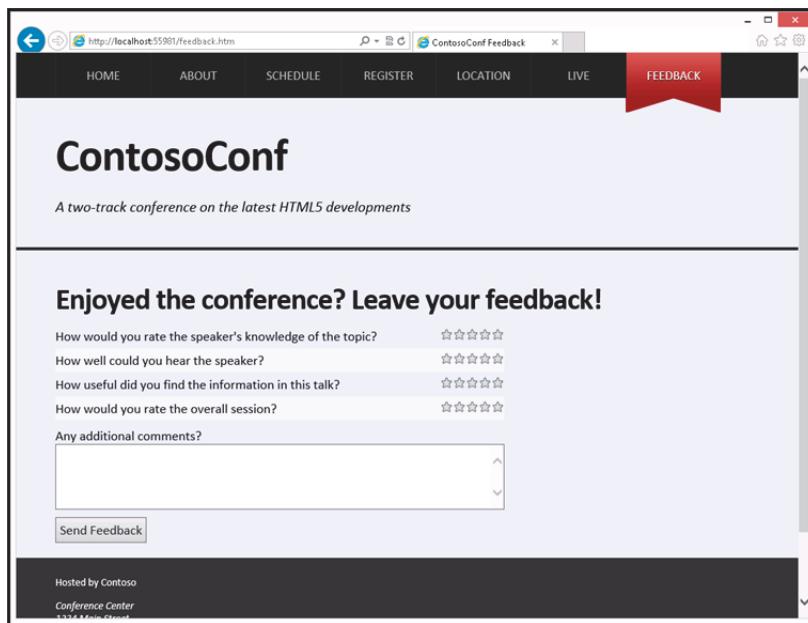


FIGURE 1.2:THE FEEDBACK PAGE.

The **Feedback** page enables an attendee to provide feedback about a session by selecting a star rating and by providing additional comments.

The star rating is implemented by using a combination of JavaScript code and CSS styles behind HTML5 input fields.

6. Provide a rating for a session; click the third star adjacent to the first question, the fifth star adjacent to the second question, and the fourth star adjacent to the two remaining questions.
7. In the **Any additional comments** box, type **Good conference**, and then click **Send Feedback**.

Notice that when you send the feedback, the form flies off the screen to indicate that the feedback has been posted. This animation is performed by using CSS.

8. Close Internet Explorer.

Results: After completing this exercise, you will be able to describe the features of the Contoso Conference web application and list the technologies that are used to implement them.

Exercise 2: Examining and Modifying the Contoso Conference Application

Scenario

In this exercise, you will examine the Visual Studio 2012 project for the Contoso Conference application. You will see how the project is structured, and how the files and scripts for the project are organized into

folders. You will then run the application again, make some modifications to the HTML markup and CSS, and view the results.

The main tasks for this exercise are as follows:

1. Explore the web pages for the application by using Visual Studio 2012.
2. Explore the structure of the project.
3. Run the application and make live modifications.

► **Task 1: Explore the web pages for the application by using Visual Studio 2012**

1. In Visual Studio, open the **index.htm** file. This file contains the HTML markup for the **Home** page as static text. Examine the following items in the file:
 - o The **nav** element at the start of the **body** section. This element defines the menu that appears at the top of the page (the same menu appears on the other HTML pages as well). The item tagged with the **active** class specifies the item that refers to the current page. This item is styled differently when it is rendered.
 - o The section with the **video** class above the page footer. This section implements the video player.
 - o The **link** elements near the top of the file. These elements specify the CSS files that provide the styling for this page. The **index.css** style sheet contains the styles specifically for this page, while the other style sheets contain styles that are used throughout the application.
 - o The **script** elements just before the closing **body** tag. These elements specify the JavaScript files that implement the functionality for this page.
2. Open the **about.htm** file. This file contains the HTML markup for the **About** page as static text. Notice that:
 - o This page implements the same navigation menu as the **Home** page. Notice that the **About** item is tagged with the **active** class; this causes the **About** item to be displayed using the ribbon style when it is rendered by using the **nav.css** stylesheet.
 - o Styling is handled by a set of CSS files. The **about.css** style sheet implements the styling specific to this page.
3. Open the **schedule.htm** file. This file contains the HTML markup for the **Schedule** page.
In this page, notice that the list of sessions in the **<section class="page-section schedule">** element is empty; it is populated when the page is displayed by using the JavaScript code in the **schedule.js** script referenced near the end of the file.
4. Open the **register.htm** file. This file contains an HTML form in the **<section class="page-section register">** element. This form validates the data that an attendee enters.
When the user submits the form, their details are posted to the registration service at the URL **registration/new**.
5. Open the **location.htm** file. This file contains an HTML page that displays the distance of the user from the conference site, together with a venue map.
The distance to the conference site is calculated by using JavaScript code that calls the Geolocation API, in the script **location.js**. The script displays the distance in the **<h2>** element with the **id** of **distance** in the **<section class="travel">** element.
The venue map is drawn by using Scalable Vector Graphics in the **<section class="venue">** element.
6. View the **live.htm** file.

This file contains a form in the `<section class="page-section Live">` element that enables a user to submit questions.

Questions are posted to a server listening on a web socket.

Questions posted by other users are received by using a web socket, and then added to the list on the page. The JavaScript code that implements the web socket code is located in the `live.js` file.

7. View the `feedback.htm` file. This page contains the feedback form in the `<section class="page-section feedback">` element, enabling attendees to provide their feedback on the conference.

The input fields for the first four questions are rendered as stars by using the JavaScript code in the `StartRatingView.js` file and the styles in the `feedback.css` style sheet. Properties of the input fields define the maximum and minimum ratings, and each rating is displayed as a single yellow star.

The input field for the comments feedback is a `<textarea>` element.

When the user submits the feedback, JavaScript code in the `feedback.js` file and styles in the `feedback.css` style sheet animate the form to make it fly off the screen.

► Task 2: Explore the structure of the project

1. The files for the project are organized into the following folders. In Solution Explorer, examine the contents of each folder in turn:
 - o **images**. This folder contains photographs of the conference speakers, and logos of conference sponsors.
 - o **scripts**. This folder contains the JavaScript files used throughout the application. The **pages** subfolder contains the JavaScript files containing the code that is specific to each page. Each file is named after the corresponding HTML file.
 - o **styles**. This folder contains the styles for the application. It is organized in a similar manner to the **scripts** folder. The **images** subfolder contains the graphic image of a star, used by the feedback and schedule pages.



Note: For the purposes of this lab you can ignore the Controllers and Views folders. These folders contain C# and ASP.NET code that implement the web services used by the application. In the real world, they would be implemented separately from the web application. You can also disregard the Properties and References folders, which contain items that support the web services, as does the Global.asax file. You will not use any of these items in this course.

► Task 3: Run the application and make live modifications

1. Build and run the web application without debugging, and display the **Home** page.
2. Leave the application running and return to Visual Studio 2012.
3. Edit the HTML markup for the Home page and change the text for the **Register Free** button to **Register Now**.
4. Open the `nav.css` style sheet in the **styles** folder; this style sheet contains the styles used to render the contents of the `<nav>` element and change the background color to blue.
5. Save the changes, return to Internet Explorer, refresh the view, and verify that you can see the effects of the changes.

Results: After completing this exercise, you will be able to describe how the Contoso Conference application is structured as a Visual Studio 2012 project.

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways

In this module, you have learned how you can use HTML to define the content, structure, and semantics of a web page, to use CSS to define the way in which a web page is displayed, and to use JavaScript code to add dynamic functionality.

You have learned how to write a basic HTML page, and how to use CSS selectors to identify a set of elements to apply presentation rules to, and how to attach both style sheets and script files to a web page.

Finally, you have seen how to use Visual Studio 2012 to create and run a web application, and how to use the F12 Developer Tools in Internet Explorer to examine a live application.

Review Question(s)

Question: What are the four elements that define the basic structure of an HTML page?

Test Your Knowledge

Question	
What is the best way to apply CSS rules to HTML elements that occur in several different pages?	
Select the correct answer.	
	Include all rules for each element in the <style> attribute of the element.
	Include the rules for each page in a <style> element in the <head> element.
	Write the rules for the whole site in one or more style sheets and reference them by using a <style> element in the <head> element of each page.
	Write the rules for the whole site in one or more style sheets and reference them by using a <link> element in the <head> element of each page.
	Write the rules for the whole site in one or more style sheets and reference them by using a <stylesheet> element in the <head> element of each page.

Module 2

Creating and Styling HTML Pages

Contents:

Module Overview	2-1
Lesson 1: Creating an HTML5 Page	2-2
Lesson 2: Styling an HTML5 Page	2-9
Lab: Creating and Styling HTML5 Pages	2-19
Module Review and Takeaways	2-26

Module Overview

The technologies forming the basis of all web applications—HTML, CSS, and JavaScript—have been available for many years, but the purpose and sophistication of web applications have changed significantly. HTML5 is the first major revision of HTML in 10 years, and it provides a highly suitable means of presenting content for traditional web applications, applications running on handheld mobile devices, and also on the Windows 8 platform.

This module introduces HTML5, describes its new features, demonstrates how to present content by using the new features in HTML5, and how to style this content by using CSS.

Objectives

After completing this module, you will be able to:

- Describe the purpose of and new features in HTML5, and explain how to use new HTML5 elements to lay out a web page.
- Explain how to use CSS to style the layout, text, and background of a web page.

Lesson 1

Creating an HTML5 Page

Many developers creating web applications use HTML5 and CSS3 because these technologies are lightweight, feature-rich, and platform independent. In this lesson, you will learn about the new features of HTML5 that are attracting so many new developers to it, and how to create and structure a page by using some of the new elements in HTML5.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the new features of HTML5.
- Explain how to use the new HTML5 elements for describing areas of a document and marking up text.
- Explain how to use the new HTML5 elements for adding hyperlinks and images to a page.

What's New in HTML5?

In 1998, the World Wide Web Consortium (W3C) decided that it had finished developing HTML and stopped work on it once version 4.01 became a web standard. However, a small group of individuals associated with this project disagreed, and continued to develop it themselves. In 2006, W3C reversed its decision based on the work performed by this group, now named the Web Hypertext Application Technology Working Group (WHATWG), and started work on HTML5, based on a subset of the features defined by WHATWG.

HTML5 has several aims:

- Not to "break the web". HTML5 is backwards compatible with previous versions of HTML.
- Add new features that reflect how the web is now used. For example, HTML5 supports form validation and video.
- Specify how every browser should behave when working with HTML. All HTML implementations can be interoperable. This behavior includes defining how to handle errors.
- Be universally accessible. Features should work across all devices, in any language, and for the disabled.

HTML5 provides many extensions over previous versions, including:

- Rules for browser vendors
- New elements that reflect modern web application development
- JavaScript APIs that support desktop and mobile application capabilities



Reference Links: You can see these aims explained in full at <http://go.microsoft.com/fwlink/?LinkID=267713>.

The most obvious new feature in HTML5 is the DOCTYPE declaration, which describes which version of HTML a page uses.



Note: The DOCTYPE declaration was described in module 1.

HTML5 defines many other new features, including:

- New elements that improve the semantic structure of a document.
- New form controls and built-in validation.
- Native audio and video support, so users do not have to rely on browser plug-ins.
- The `<canvas>` element and the associated JavaScript API provide a freeform area in a page to draw on, and the JavaScript commands to do the drawing, importing, and exporting.
- Support for uploading files to a web server.
- Support for dragging and dropping elements on the page.
- Support to enable web applications to continue running when the browser is offline.
- Support for local data storage, over and above that provided by cookies.

There are also a number of HTML5-associated specifications authored by W3C that are outside of the wider WHATWG work, including:

- A formalization of the JavaScript object that underpins AJAX by using the `XmIHttpRequest` object.
- Support for continuous communication between browser and web server by using web sockets.
- Support for using multiple threads to handle processing for a web page by using web workers.
- Support for accessing a device's GPS capabilities by using the Geolocation API.



Reference Links: You can find the current draft of the W3C HTML5 specification at <http://go.microsoft.com/fwlink/?LinkId=267714>.

You can find a version of the specification for web developers (minus the browser interoperability instructions) at <http://go.microsoft.com/fwlink/?LinkId=267715>.

Document Structure in HTML5

HTML5 includes new elements that enable you to mark up your content and present a better structure for your documents, compared to earlier versions of HTML.

One of the most common tasks in a page is to identify different areas of the document: the navigation bar, the header, the footer, and so on. In HTML4, you used the `id` attribute. For example:

```
<ul id="navigation"> ... </ul>
<div id="footer"> ... </div>
```

HTML5 provides new elements to define the structure of a web page:

- `<section>` to divide up main content
- `<header>` and `<footer>` for page headers and footers
- `<nav>` for navigation links
- `<article>` for stand-alone content
- `<aside>` for quotes and sidebar content



While this approach is convenient, it does not convey the semantic meaning of the different areas. HTML5 provides a much richer semantic structure for documents, including:

- The `<section>` element, which identifies component pieces of content on a page. For example, the ingredients and the method in a recipe displayed by a page could be two separate sections.
- The `<header>` element, which identifies the content in the header of the page. For example, a company website might include its logo, name, and motto in the `<header>`.

- The **<footer>** element, which identifies the content in the footer of the page. For example, links to site maps, privacy statements, and terms and conditions are often included in the **<footer>**.
- The **<nav>** element, which identifies the content providing the main navigation sections of the page. Developers often use this element to implement a menu listing the various features of a web application, organized as a series of web pages.
- The **<article>** element, which identifies standalone pieces of content that would make sense outside of the context of the current page. For example, a blog post, a recipe, or a catalog entry.
- The **<aside>** element, which identifies content related to an **<article>** that isn't part of its main flow. For example, you might use **<aside>** to identify quotes or sidebar content.

The following markup example shows one way to mark up an HTML5 document by using the new structural elements.

Content Structure in HTML5

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>My Best Recipes</title>
</head>
<body>
  <nav>
    <a href="/">Home</a>
  </nav>
  <header>
    <h1>My Best Recipes</h1>
    <p>My favorite recipes</p>
  </header>
  <article>
    <h1>Beans On Toast</h1>
    <section>
      <h2>Ingredients</h2>
      <ul>
        <li>Beans</li>
        <li>Bread</li>
      </ul>
    </section>
    <section>
      <h2>Method</h2>
      <ol>
        <li>Toast bread</li>
        <li>Heat beans</li>
        <li>Put beans on the toast</li>
      </ol>
    </section>
  </article>
  <footer>
    <small>Last updated on <time datetime="2012-08-12">August 12, 2012</time></small>
  </footer>
</body>
</html>
```

The screenshot shows a web page with the following structure:

- Header:** A red-bordered button labeled "Home".
- Title:** "My Best Recipes" (dark blue text).
- Header Content:** "My favorite recipes" (black text).
- Article:** "Beans On Toast" (dark blue text).
- Section:** "Ingredients" (black text).
 - Items: Beans, Bread
- Section:** "Method" (black text).
 - Items: Toast bread, Heat beans, Put beans on the toast
- Footer:** "Last updated on August 12, 2012" (black text).

It is important to realize that there is no prescribed order in which to use these elements. For instance, you could decide to include your navigation links in the header, footer, or sidebar. Similarly, you could split your page up into thematic sections (such as Breakfasts, Lunches, and Dinners) and include **<article>** elements within those sections. The important thing is to grasp the purpose of each new element and use it appropriately.



Note: Note how there are three `<h1>` elements on the page, rather than just the one you would expect. Previously, this would have been semantically incorrect, but the `<section>` and `<article>` elements are defined in HTML5 such that you may restart the heading numbering. The *HTML5 outlining algorithm* defines this and the use of `<hgroup>` in the next topic.

Text and Images in HTML5

HTML5 supports the header, paragraph, and emphasis elements used in the previous version of HTML. HTML5 also defines a number of new elements to improve the semantic context of the document, including:

- The `<hgroup>` element, which indicates that its contents should be treated as a single heading. This element can contain header tags `<h1>` to `<h6>`.

```
<hgroup>
  <h1>My Recipes</h1>
  <h2>Great to eat, easy to make</h2>
</hgroup>
```

HTML5 defines new text elements, including:

- `<hgroup>`
- `<time>`
- `<mark>`
- `<small>`
- `<figure>` and `<figcaption>`

```
<hgroup>
  <h1>My Recipes</h1>
  <h2>Great to eat, easy to make</h2>
</hgroup>
```

```
<time datetime="2012-08-08">Today</time>
```

```
<p>This text should be <mark>noted for future use.</mark></p>
```

```
<p>Heat your beans for five minutes. <small>Or until they are hot enough for you.</small></p>
```

```
<figure>
```

```

<figcaption>A plate of beans in five minutes flat</figcaption>
</figure>
```

- The `<time>` element, which enables you to define an unambiguous time, duration, or period that is both human and machine readable. The **datetime** attribute contains the ISO standard representation of the contents of an element.

```
<time datetime="2012-08-08">Today</time>
<time datetime="2012-08-08T09:00:00-0500">9am today in New York</time>
<time>4h</time>
<time>2012</time>
```

- The `<mark>` element, which identifies that its contents should be treated as text to be marked or highlighted for reference purposes.

```
<p>This text should be <mark>noted for future use</mark> rather than
<em>emphasized</em>. </p>
```

- The `<small>` element, which identifies that its contents should be treated as side comments, such as small print or author attributions.

```
<p>Heat your beans for five minutes. <small>Or until they are hot enough for
you.</small></p>
```

- The `<figure>` element, which is typically used to identify an image, video, or code listing and its associated descriptive content and other elements. If the content needs a caption, you can nest the `<figcaption>` element inside the `<figure>` element.

```
<figure>
  
  <figcaption>A wonderful plate of beans in five minutes flat</figcaption>
</figure>
```



Additional Reading: HTML5 also adds to the global attributes defined in HTML4. You can find a full list at <http://go.microsoft.com/fwlink/?LinkId=267716>.

Demonstration: Using HTML5 Features in a Simple Contact Form

In this demonstration, you will see how to add some of the new HTML5 elements to flesh out the contact form created during the demonstrations in Module 1 and add some semantic richness. You will then use the F12 Developer Tools to view the structure of the page. You will also see how to use the F12 Developer Tools to make a temporary change to a page in a browser for testing purposes.

Demonstration Steps

Divide the Content for a Page into an Article with Sections

1. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
2. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, browse to the **E:\Mod02\Democode\Starter** folder, click **DemoWebSite.sln** and then click **Open**.
4. In Solution Explorer, expand the **E:\...\DemoWebSite** web application, and then double-click **ContactUs.html**.
5. In the **ContactUs.html** file, enclose the entire contents of the **<body>** element in an **<article>** element as shown in bold in the following code example:

```
<!DOCTYPE HTML>
<html lang="en">
  ...
  </head>
  <body>
    <article>
      ...
    </article>
  </body>
</html>
```

6. Within the **<article>** element, enclose the first three **<p>** elements containing the company name, address, and contact email in a **<section>** element, as shown in bold in the following code example:

```
...
<h1>Contact Contoso Conferencing</h1>
<section>
  <p>Contoso Conferencing Ltd.</p>
  <p>123 South Street<br />
  Somewhere<br />
  Over There<br />
  <em>USA</em></p>
  <p>
    <a href="mailto:contact@contoso.com">contact@contoso.com</a>
  </p>
</section>
<p>
  If you would like to contact Contoso Conferencing ...
</p>
  ...
```

7. Wrap the HTML form and **<p>** element immediately above it in a second **<section>** element, as shown in bold in the following code example:

```
...
<section>
  <p>
    If you would like to contact Contoso Conferencing ...
  </p>
  <form method="POST" action="support.aspx">
```

```
...  
  </form>  
</section>  
...
```

8. On the **File** menu, click **Save All**.

Add a Header and Footer to the Page

1. Enclose the **<h1>** element near the top of the **ContactUs.html** file in a **<header>** element, as shown in bold in the following code example:

```
...  
<article>  
  <header>  
    <h1>Contact Contoso Conferencing</h1>  
  </header>  
  ...  
</article>  
...
```

2. Add the following **** element shown in bold to the **<header>** element above the **<h1>** element.

```
<header>  
    
  <h1>Contact Contoso Conferencing</h1>  
</header>
```

3. Add the following HTML markup shown in bold immediately after the **</article>** tag near the end of the document.

```
...  
  </article>  
  <footer>  
    <p>  
      <small>  
        Last updated  
        <time datetime="2012-08">  
          August 2012  
        </time>  
      </small>  
    </p>  
  </footer>  
</body>  
</html>
```

4. On the **File** menu, click **Save All**.

View the Structure of the Page by Using the F12 Developer Tools

1. On the **Debug** menu, click **Start Without Debugging**.
2. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
3. Press F12.
4. In the **F12** window, click the **HTML** tab.
5. Expand the **<html>** element.
6. Expand the **<body>** element.

7. Expand the **<article>** element and verify that it contains a **<header>** element and two **<section>** elements.
8. Expand the **<header>** element.
9. Expand the **<h1>** element.
10. Click each element, and verify that Internet Explorer surrounds each element on the page with a box as it is selected in the **F12** window.

Make a Temporary Change to the Page by Using the F12 Developer Tools

1. In the **<h1>** element, click **Contact Contoso Conferencing**.
2. Change this text to **We'd love to hear from you...**, and then press ENTER.
3. Verify that Internet Explorer displays the modified text.
4. Press F12 to close the **F12** window.
5. Close Internet Explorer, and then close Visual Studio 2012.

Lesson 2

Styling an HTML5 Page

After you have defined the structure and set the content of a web page, your next task is to apply some presentation rules to it by using CSS. In this lesson, you will learn about the core CSS properties that you can use to style the text and background of a page. You'll also learn how to use the CSS box model to position block-level elements on a page.

Lesson Objectives

After completing this lesson, you will be able to

- Use CSS text styles to set the fonts used in a page and other text properties.
- Explain how to use the CSS box model to position elements on a page.
- Use CSS to set the background for the elements on a page.

Understanding CSS Text Styles

However good the layout of a website, it can be nullified if you write poor content or present it badly. If content is the foundation of the web, then typography is the foundation of web design, and so the first set of CSS properties that you should learn are those that concern the ways in which text is displayed.

Fonts

The most obvious aspect of text content is the font. The discussion of when to use which type of font for headings, body text, sidebars, and so on is outside the scope of this course. Also, many organizations have style guides that specify the fonts for their web pages. That said, you can use the following CSS properties to set the selected font.

• CSS Text Styling supports:

• Fonts

```
font-family : Arial, Candara, Verdana, sans-serif;
font-size : 16px;
font-style : italic;
font-weight : bold;
```

• Colors

```
color : rgb(128, 128, 0);
opacity: 0.6;
```

• Typography

```
letter-spacing : 2em;
line-height : 16px;
text-align : left;
text-decoration : underline;
text-transform : lowercase;
```

- The **font-family** property, which sets a comma-separated list of preferred font families for the specified text. You should write the list in order of preference and wrap any font-family names containing spaces in double quotes. When the text is rendered, the browser will use the first font that is available on the computer. If none of the specified fonts are available, the browser will use its own algorithm to select another font. The following code shows some examples:

```
font-family : Arial, Candara, Verdana, sans-serif;
font-family : Georgia, Corbel, "Times New Roman", serif;
font-family : Consolas, "Courier New", monospaced;
```

- The **font-size** property, which sets the height of the font. You can set the font-size value to an absolute value in pixels (for on-screen display), in points (for printing), or to a value relative to the parent element's font-size (in percent), or relative to the base font size of the page (in ems). The following code shows some examples:

```
font-size : 16px;
font-size : 150%; /* Font-size of the parent element * 150% */
font-size : 1em; /* 1em = base font-size of the page. Usually 16px */
```

- The **font-style** property, which enables you to select a normal (vertical), italic, or oblique version of the font to be displayed, as shown in the following example:

```
font-style : italic;
```

- The **font-weight** property, which enables you to set the weight of a font. Usually, this means setting the property value to **bold**, but you can also set it to one of nine numerical values (100, 200, ..., 900) reflecting different font weights in that family (black, book, semi-bold, and so on). The following code shows some examples:

```
font-weight : bold;
font-weight : normal;
font-weight : 800;
```

CSS also provides a shortcut property simply called **font**, which enables you to set some or all of these four properties (plus **line-height**) in a single rule rather than having to write out all five rules for every element. You must set the value for these properties in the following order (note that the **font-family** and **font-size** properties are mandatory, but the other properties are optional):

1. font-style.
2. font-weight.
3. font-size/line-height.
4. font-family.

For example:

```
p { font : bold 16px/1.5 "Arial"; }
/* The above is a shorthand for the following rules. The default font-style is used. */
p {
  font-weight: bold;
  font-size: 16px;
  line-height : 1.5em;
  font-family: Arial;
}
```

Colors

There are two color-related properties in CSS: color and opacity.

- The **color** property, which enables you to set the color of a font. You can specify the color as an RGB (red-green-blue) value or as one of the 147 predefined color names in the HTML and CSS specification. The following code shows some examples:

```
/* The following color values are all equivalent. */
color : olive;
color : #808000;
color : rgb(128, 128, 0);
```

- The **opacity** property, which enables you to set the transparency of some text or of an image. This property takes a value between 0.0 (fully transparent) and 1.0 (fully opaque). The following code shows an example:

```
p {
  opacity : 0.6;
  filter:alpha(opacity=60); /* IE8 and earlier */
}
```

Note that Internet Explorer versions prior to Internet Explorer 9 do not support the **opacity** property. Instead, you must use the **filter** property and set the opacity to a value between 0 and 100.

Typographic Properties

CSS defines another five text-related properties that cover typographic properties such as line height and kerning, as well as more obvious features such as alignment and underlining. These properties are:

- The **letter-spacing** property, which enables you to increase or decrease the space between characters in a block of text. You can set the font-size value to an absolute value in pixels or points, or to a relative value in percentages or in ems. The following code shows some examples:

```
letter-spacing : 2em;  
letter-spacing : -3px;
```

- The **line-height** property, which enables you to increase or decrease the space between lines of text in a block of text. You can set this value to an absolute value in pixels or points, or a value relative to the current font-size by using either a percentage or a positive number. The following code shows some examples:

```
line-height : 16px;  
line-height : normal; /* This is the default */  
line-height : 1.2;  
line-height : 120%;
```

- The **text-align** property enables you to set how the text in the selected blocks is aligned. For example, left, right, or justify. The following code shows an example:

```
text-align : left;
```

- The **text-decoration** property, which enables you to set whether text in selected elements will be decorated with a line, and if so, where. Possible values are none (the default), underline, overline, and line-through. The following code shows an example:

```
text-decoration : underline;
```

- The **text-transform** property, which enables you to set the capitalization of text in selected blocks. Possible values are none (the default), capitalize, uppercase, and lowercase. The following code shows an example:

```
text-transform : lowercase;
```



Note: If you are using Visual Studio to define new styles, you can use the **Build Style** wizard to generate these styles and to ensure that the syntax of your CSS code is valid. This wizard is available in the toolbar in the CSS editor.

The CSS Box Model

To determine the layout of an HTML page, browsers treat each element in the page as a set of four nested boxes. The CSS box model enables you to specify the size of each box, and so modify the layout of each element on the page.

The box model places content inside four boxes: Content, Padding, Border, and Margin.

- The CSS box model treats each element as a collection of four concentric boxes:



- CSS defines properties that:
 - Control how a box is laid out on a page
 - Alter the height and width, and the style of the border



FIGURE 2.1:THE CSS BOX MODEL

In the center box is the **content**, your text and images. Use the **height** and **width** properties to set the height and width of the content box in pixels.

Around the content box is the **padding** box. Use the **padding** property to set the width of the padding box.

Around the padding box is the **border** box, which can also act as a visible line around the content and padding. Use the **border** property to set its width, color, and style.

Around the border box is the **margin** box. Use the **margin** property to set the width of the margin box.

The following code example shows how to use the CSS box model to draw a border around some padded heading text and to set a margin around the border so that it does not interfere with other elements on the page.

Using the box model properties

```
h2.highlight {  
    height : 100px;  
    width : 500px;  
    padding : 10px;  
    border : 2px dotted blue;  
    margin : 25px 0 25px 0; /* Could also be written 25px 0 */  
}
```

Margin and **padding** are both shorthand properties. CSS actually defines individual properties for the top, right, bottom, and left of each box, but if they are the same, you can just use **padding** or **margin**. In the previous code example, padding is set to 10px. You could write this out in full as:

```
padding-top: 10px;  
padding-right : 10px;  
padding-bottom : 10px;  
padding-left : 10px;
```



Reader Aid: You can easily recall the order of the sides by thinking of the word **TRouBLE**: Top, Right, Bottom, and Left.

Border is also a shorthand property for the width, style, and color of the border box. In the previous example, border is set to 2px dotted blue. You could write this out in full as:

```
border-width: 2px;  
border-style: dotted;  
border-color: blue;
```

Using the **border-width**, **border-style**, and **border-color** properties assumes that you want the set values to be the same around all four sides of the box. If this is not the case or you only want to set them for one side of the border, you can use the **border-left-style**, **border-left-width**, **border-left-color**, **border-right-style**, **border-right-width** and so on properties. For example:

```
p.example {  
    padding: 10px;  
    border-bottom: solid 1px black;  
    border-left-style: dotted;  
    border-left-width: 2px;  
}
```

The **border-top**, **border-right**, **border-bottom**, and **border-left** properties are also shorthand properties, like border, but for the width, style, and color of the respective sides of the border box.



Note: Note that the F12 Developer Tools offer you a graphic illustrating the dimensions of each box in a page. To see the illustration, with a page loaded in Internet Explorer, select the **HTML** tab, and then in the right pane click **Layout**.

Beyond the core box model properties, CSS defines several more properties to control how content is viewed in the flow of the elements on the page. Specifically:

- The **visibility** property enables you to blank out the selected elements, but leave the space that they would take up on the page empty.
- The **display** property enables you to set how to display selected elements on the page. This includes hiding the selected elements, or completely removing them from the page.
- The **position** property enables you to set positioning method for the selected elements are positioned. The four possible values are **static** (the default), **fixed**, **absolute**, and **relative**.
- The **float** property enables you to take the selected elements out of the flow of content and 'float' them to the left or right of their containing elements.
- The **overflow** property enables you to set what happens when the content of an element is too big for the box that contains it.
- The **box-sizing** property enables you to set how the width and height properties apply to an element's box model. If set to **content-box** (the default), they work as described above. If set to **border-box**, the width and height apply to the total of content, padding, border and margin taken together.

Styling Backgrounds in CSS

Many websites use a background image, color, or pattern to provide more color and character to the pages. CSS enables you to set a background for any block-level element by using the following elements.

- The **background-image** property, which enables you to specify the URL of an image to use as a background for the selected elements. You may use a relative or an absolute URL. Note that if you provide a relative URL, you must specify the path relative to the location of the style sheet or web page that defines the style.

```
background-image:url('../images/pattern.jpg');
```

Set the background for an element by using the CSS background properties:

```
•background-image  
•background-size  
•background-color  
•background-position  
•background-origin  
•background-repeat  
•background-attachment
```

- The **background-size** property, which enables you to set the height and width of the background image. Use values specified in pixels or as percentages of the height and width of the specified element.

```
background-size: 40px 60px; /* 40px wide, 60px high */
```

- The **background-color** property, which enables you to set the color of an element's background. You can specify the color as an RGB (red-green-blue) value or as one of the 147 predefined color names in the HTML and CSS specification.

```
background-color : green;  
background-color : #00FF00;  
background-color : rgb(0, 255, 0);
```

- The **background-position** property, which enables you to set the position of the background image in the element. The property takes two values: the first for the x-axis and the second for the y-axis. Set them both as absolute values (top, left, bottom, right, center), percentages, or pixels.

```
background-position : left top; /* Image locked into top left corner of element */  
background-position : 100% 100%; /* Image locked into bottom right corner of element */  
background-position : 8px 8px; /* Image starts 8px from left and 8px from top of element */
```

- The **background-origin** property, which enables you to set which of the box model boxes the **background-position** should be relative to. Possible values are **content-box** (the default), **padding-box**, and **border-box**.

```
background-origin : border-box;
```

- The **background-repeat** property, which enables you to set how a background image is repeated behind the selected element if it is smaller than the selected element. Possible values are **repeat** (the default), **repeat-x**, **repeat-y** and **no-repeat**.

```
background-repeat : repeat-x; /* Repeat background image only horizontally */  
background-repeat : no-repeat; /* Don't repeat the image */
```

- The **background-attachment** property, which enables you to set whether a background image scrolls up and down with a page or remains fixed in place. Possible values are **scroll** (the default) and **fixed**.

```
background-position : fixed;
```

CSS also provides the **background** shortcut property, which enables you to set some or all of the elements just described. You must set the values for these properties in the following order (only the **background-image** property is mandatory, the others are optional):

1. background-color.
2. background-position.
3. background-size.
4. background-repeat.
5. background-origin.
6. background-clip.
7. background-attachment.
8. background-image.

For example:

```
article { background : transparent repeat-x url('fluffycat.jpg'); }  
/* The above is a shorthand for the following rules */  
article {  
    background-color : transparent;  
    background-repeat : repeat-x;  
    background-image : url('fluffycat.jpg');  
}
```

Demonstration: Adding CSS Styles to an HTML Page

In this demonstration, you will see how to create new styles for the contact form created during the previous demonstrations. You will then view the page in Internet Explorer and use the F12 Developer Tools to inspect the styles of elements and to modify them to see how they are rendered by the browser.

Demonstration Steps

Create New Styles by Using Visual Studio

1. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
2. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, browse to the **E:\Mod02\Democode\Starter** folder, click **DemoWebSite.sln**, and then click **Open**.
4. In Solution Explorer, expand the **E:\...\DemoWebSite** web application, and then expand the **styles** folder.
5. Double-click **ContactUsStyles.css**
6. Review the existing rules for the **body** and **h1** elements.
7. Modify the **body** rule, remove the color rule, and change the font used on the whole page as shown in bold in the following code example.

```
body
{
    font-family: "Segoe UI", Helvetica, Arial, sans-serif;
}
```

8. Remove the following rule from the stylesheet:

```
h1 {
    font-family: 'Copperplate Gothic';
    color: red;
}
```

9. Add the following rules that make the header appear separately from the rest of the content.

```
header {
    padding-bottom: 10px;
    border-bottom: 2px dotted blue;
    margin-bottom: 10px;
}
header h1 {
    margin-left: 20px;
    display: inline-block;
}
```

10. Add the following empty rule:

```
section {
```

11. Click after the opening curly brace for the section rule, and in the toolbar click the **Build Style** button.

 **Note:** If the toolbar is not visible, right-click in the body of the section rule and then click **Build Style**.

12. In the **Modify Style** dialog box, in the **Category** list, click **Box**.
13. Clear the **padding: Same for all** check box, and in the **bottom** box, type **5**.
14. In the **Category** list, click **Border**.
15. Under **border-style**, clear the **Same for all** check box, and in the **bottom** list box, click **dotted**.
16. Under **border-width**, clear the **Same for all** check box, and in the **bottom** box, type **1**.
17. Under **border-color**, clear the **Same for all** check box, and in the **bottom** box, type **grey**.
18. Click **OK**.

19. Verify that the section rule now looks like this:

```
section {
    padding-bottom: 5px;
    border-bottom-style: dotted;
    border-bottom-width: 1px;
    border-bottom-color: grey;
}
```

20. Add the following rules to style the form and its elements.

```
fieldset {
```

MATERIALS ONLY. STUDENT USE PROHIBITED

```
background-color: pink;
margin-bottom: 10px;
}
legend {
font-size: 1.2em;
font-style: italic;
}
fieldset li {
list-style: none;
margin-bottom: 10px;
}
input[type="submit"] {
background-color: pink;
opacity: 0.6;
width: 200px;
}
```

21. On the **File** menu, click **Save All**.

Use the F12 Developer Tools to Inspect Styles

1. In Solution Explorer, double-click **ContactUs.html**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
4. Verify that the new styles have been applied to the page.

The ContactUs page should look like this:

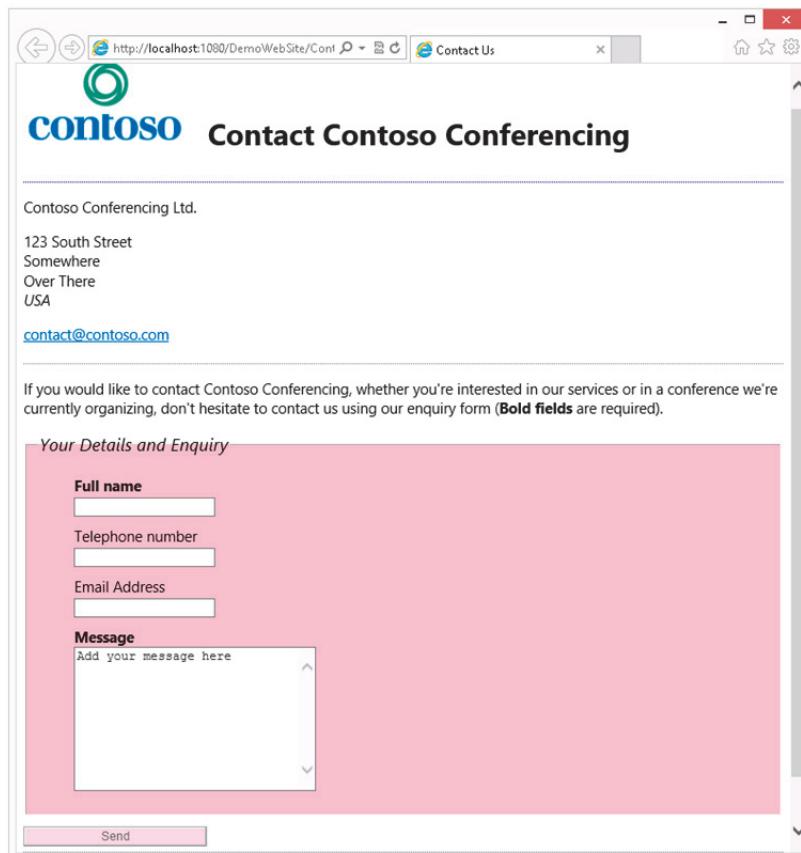


FIGURE 2.2:THE CONTACTUS PAGE WITH STYLING

5. In Internet Explorer, press F12.

6. In the **F12** window, with the **HTML** tab selected, double-click the `<html>` element to expand it.
7. Click the `<body>` element.
8. In the right pane, verify that the following the CSS rule appears:

```
font-family: "Segoe UI", Helvetica, Arial, sans-serif;
```

9. In this rule, select the text "**Segoe UI**".
10. Change the value to read "**Times New Roman**" and press ENTER.
11. Verify that Internet Explorer reflects this change to the font on the page.
12. In the left pane, expand the `<body>` element, expand the `<article>` element, and then click the first `<section>` element.
13. In the right pane, verify that the following styles are specified for this section:

```
inherited - body  
  body  
    font-family: "Times New Roman", Helvetica, Arial, sans-serif;  
  section  
    padding-bottom: 5px;  
    border-bottom-color: grey;  
    border-bottom-width: 1px;  
    border-bottom-style: dotted;
```

14. Press F12 to close the **F12** window.
15. Close Internet Explorer, and then close Visual Studio 2012.

Demonstration: Creating and Styling an HTML5 Page

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Creating and Styling HTML5 Pages

Scenario

You are a web developer working for an organization that builds websites to support conferences. You have been asked to create a website for ContosoConf, a conference that showcases the latest tools and techniques for building HTML5 web applications.

You decide to start by building a prototype website consisting of a Home page that acts as a landing page for conference attendees, and an About page that describes the purpose of the conference. In later labs you will enhance these pages and add further pages that enable attendees to register for the conference, and that provide information about the sessions scheduled to run as part of the conference.

Objectives

After completing this lab, you will be able to:

- Create HTML5 pages.
- Style HTML5 elements.
- Estimated Time: 45 minutes.
- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa\$\$w0rd**

Exercise 1: Creating HTML5 Pages

Scenario

In this exercise, you will begin to create the ContosoConf website.

First you will create a new ASP.NET Web Application. Then you will add two HTML files for the Home and About pages. Next, you will add navigation links to the pages. Finally you will run the web application and verify that the Home page and About page are formatted correctly.

The main tasks for this exercise are as follows:

1. Create a new ASP.NET web application.
2. Add the Home page.
3. Add images to the Home Page.
4. Add the About page.
5. Add navigation links.
6. Run the web application.

► Task 1: Create a new ASP.NET web application

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio.
4. Create a new web application named **ContosoConf** in the **E:\Mod02\Labfiles\Starter\Exercise 1** folder. Use the **Visual C# ASP.NET Empty Web Application** template to create this web application.

► Task 2: Add the Home page

1. Add a new HTML page named **index.htm** to the ContosoConf project. This page is the default page for the website, and will be displayed when a user browses to the URL for the website.
2. Using Notepad, open the **index.txt** file located in the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder.
 - Examine this file, and then add HTML5 elements to the index.htm file in the web application that can display the items specified in the index.txt file.
 - Use HTML5 elements, such as **<header>**, **<section>**, and **<footer>** where appropriate.

For example, the following **<header>** element contains the content from the index.txt file:

```
<header>
  <h1>ContosoConf</h1>
  <p>A two-track conference on the latest HTML5 developments</p>
</header>
```

The following image shows how the unstyled page should look, and how the content should be divided into the elements on the page.

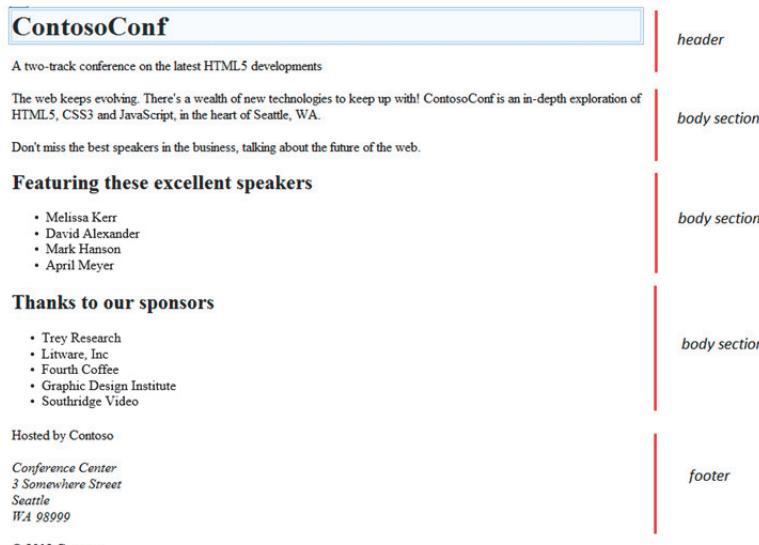


FIGURE 2.3:THE LAYOUT OF THE HOME PAGE.

► Task 3: Add images to the Home Page

1. Add the speaker and sponsor images to the ContosoConf project. These images are located in the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder. They should be added to a new folder called **images** in the project.
2. Update the HTML markup in the **index.htm** file to include the images from the **speakers** and **sponsors** folders in the **images** folder. For example:

```

Melissa Kerr
```

The following image shows how the page should appear:

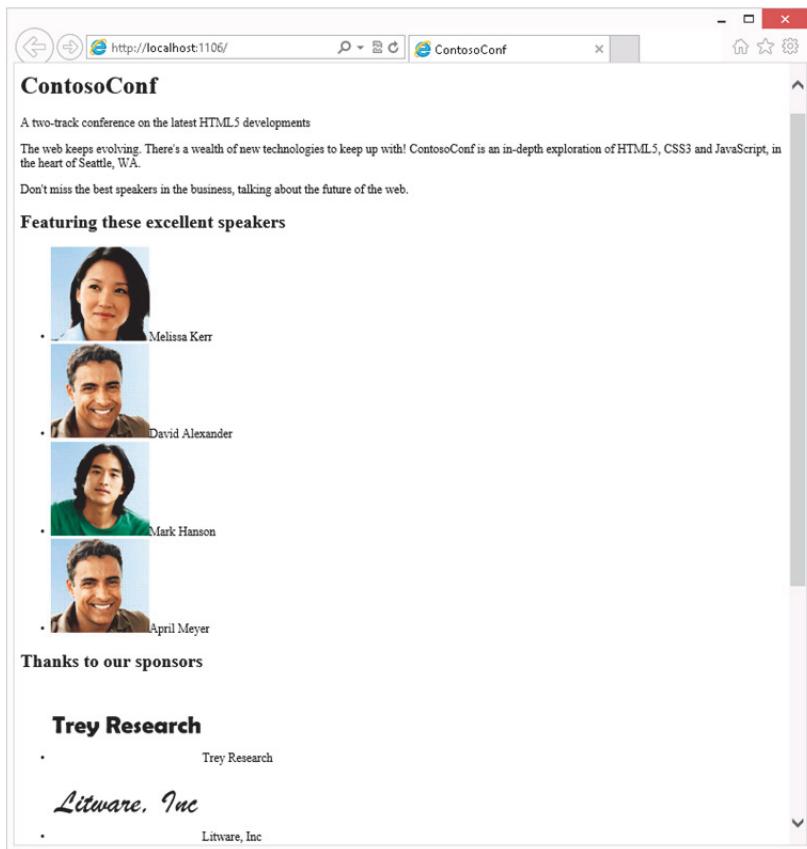


FIGURE 2.4:THE HOME PAGE WITH IMAGES

Note: In an **** element, you can use the **Pick URL** wizard to specify the **src** attribute for an image.

► Task 4: Add the About page

1. Add a new HTML page, named **about.htm** to the ContosoConf project.
 - o Use the **Add New Item** command on the **Project** menu.
 - o Select the **HTML Page** template.
2. Add HTML elements to **about.htm**, using the text provided in the **about.txt** file located in the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder.
 - Add an appropriate title to the page.
 - Copy the **<header>** and **<footer>** elements from **index.htm**.
 - Use the **<article>**, **<blockquote>**, and **** elements where appropriate.

The following image shows how the unstyled page should look.

ContosoConf

A two-track conference on the latest HTML5 developments

ContosoConf brings web designers and developers together

Since the very first Contoso Conf back in 2009, we've been guided by three principles:

1. Community Matters
2. Never Stop Learning
3. Have fun!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae enim arcu, vitae aliquet purus. Aenean rhoncus diam et orci porttitor fringilla. In porta lacus a turpis pretium placerat. Cras viverra enim eu nibh pretium ornare. Praesent et adipiscing turpis. Duis mi risus, ornare at bibendum a, ullamcorper vel tellus. Nulla in egestas velit. Aenean consequat mi sed tellus iaculis laoreet. Donec et odio vel felis commodo porttitor.

Aenean id ligula est. Pellentesque ut magna ligula. Donec nunc eros, tincidunt sit amet sollicitudin in, semper id mauris. Phasellus odio nulla, molestie ac gravida sed, dignissim in nisl. Nunc luctus lobortis massa in dapibus. Aenean turpis nibh, hendrerit nec congue et, elementum a justo. Aenean sit amet nulla odio. Cras feugiat porta risus nec pretium.

What's It All About?

Donec vel sem ut dui vulputate porta. Phasellus imperdiet sapien a arcu adipiscing vitae adipiscing elit pharetra. Donec sed ante ut eros mattis bibendum non erat. Donec sagittis, massa eu accumsan eleifend, eros justo cursus justo, id consequat mauris diam id magna. Vivamus quis tortor massa. Nam ipsum metus, dapibus ac facilisis sit amet, ullamcorper quis risus. Integer aliquet eleifend accumsan.

I had a fantastic time and learnt so much!

Pellentesque facilisis blandit augue id rhoncus. Sed facilisis varius lectus, eget commodo purus dapibus nec. In hac habitasse platea dictumst. Etiam imperdiet facilisis malesuada. Nunc semper venenatis elit ac lobortis. Duis lorem lorem, pharetra ut scelerisque nec, consequat sed risus. Morbi rutrum nisl ut ipsum consectetur porttitor. Phasellus sed nunc id diam tempus congue in a leo.

Proin feugiat, turpis id tempor tempor, lorem libero malesuada.

Hosted by Contoso

Conference Center
3 Somewhere Street
Seattle
WA 98343

FIGURE 2.1:THE LAYOUT OF THE HOME PAGE.

► Task 5: Add navigation links

1. Add a navigation element to **index.htm** and **about.htm**. The navigation element should contain links to both pages.

For example, the **<nav>** element for the index.htm page should look like this:

```
<nav>
    <a href="/index.htm">Home</a>
    <a href="/about.htm">About</a>
</nav>
```

► Task 6: Run the web application

1. Run the web application by using Internet Explorer.
2. Verify that the correct text and images are displayed.
3. Verify that the navigation links reference the correct pages.

Results: After completing this exercise, you will have built a simple HTML5 web application with a Home page and an About page.

Exercise 2: Styling HTML pages

Scenario

In this exercise, you will add styling to the Home and About pages.

You will create a stylesheet in the ContosoConf project. Then you will add CSS rules to style the Home and About pages to match a specified design. Finally, you will run the web application and verify that the pages are styled correctly.

The main tasks for this exercise are as follows:

1. Create a new style sheet.

2. Add CSS rules to style the pages.

3. Run the web application.

► Task 1: Create a new style sheet

1. Add a folder named **styles** to the ContosoConf project.
2. Add a stylesheet named **site.css** to the **styles** folder.
 - o Use the **Add New Item** command on the **Project** menu.
 - o Select the **Style Sheet** template.
3. Add a link to the **site.css** style sheet from the **index.htm** and **about.htm** HTML5 pages.
 - o Use a **<link>** element inside the page header; set the **href** property to "**styles/site.css**", set the **rel** property to "**stylesheet**", and set the **type** property to "**text/css**".

► Task 2: Add CSS rules to style the pages

1. Add a CSS rule to the site.css style sheet to style the **<html>** element of the Home and About pages (the same styling rules should apply to both pages):
 - o Set **background-color** property for the web page to **#EAEFA**, set the list of fonts in the font family to **Calibri, Arial, sans-serif**, and set the font size **62.5%**
2. Add a CSS rule to style the **<body>** element of a web page:
 - o Set the **margin** property to **0**, and the **font-size** property to **1.8rem**.
3. Add a CSS rule to style the **<nav>** element of a web page:
 - o Set the **background-color** property to **#1d1d1d**, set the **line-height** property to **6rem**, and set the **font-size** property to **1.7rem**.
 - o Additionally, add a style for all links (**<a>** elements) that occur inside a **<nav>** element; set the **color** property to **fff** and the **padding** property to **1rem**.



Note: Use the expression **nav a** to specify an **<a>** element inside a **<nav>** element in a CSS selector.

4. Add a CSS rule to style the **<h1>** element of a web page:
 - o Set the **font-size** property to **4rem**, set the **letter-spacing** property to **-1px**, and set the **margin** property to **1em 0 0.25em 0**.
5. Modify the HTML markup for the Home and About pages and wrap the **<a>** elements in the **<nav>** section in **<div class="container">** element, like this:

```
<div class="container">
    <a href="/index.htm">Home</a>
    <a href="/about.htm">About</a>
</div>
```

6. Add rule to the site.css style sheet to achieve the horizontally centered, fixed-width column effect for all sections marked with the container class.
 - o Set the **padding** property to **0.1rem**, the **max-width** property to **94rem**, and the **margin** property to **0 auto**.



Note: Use the expression **.container** as the selector for the CSS rule.

The styled Home page should look similar to this:



FIGURE 2.2:THE STYLED HOME PAGE



Note: Use the Internet Explorer F12 developer tools to experiment with the CSS rules until you achieve the correct styling.

► **Task 3: Run the web application**

1. Run the web application and verify that the About page and Home page are both styled appropriately.

The About page should look like this in Internet Explorer:

MCT USE ONLY. STUDENT USE PROHIBITED

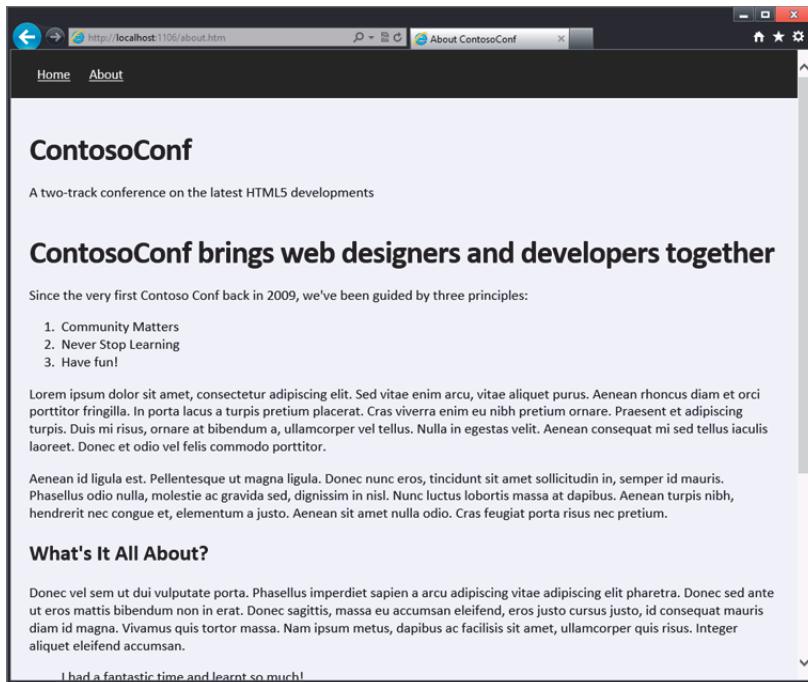


FIGURE 2.3:THE STYLED ABOUT PAGE

Results: After completing this exercise, you will have used CSS rules to style the Home and About pages.

Module Review and Takeaways

In this module, you have learned how to use the new features of HTML5 to organize the content for a web page. You have seen how to use the new tags available in HTML5 to specify the semantics of the text elements in a web page.

You have also learned more about using CSS to style text and background elements in a web page. You learned about the CSS box model, and how to apply styling to elements based on their margin, border, padding, and content.

Review Question(s)

Question: What are the new elements that HTML5 provides for specifying the semantic meaning of content in a web page?

Test Your Knowledge

Question	
Which of the following items is NOT a property of the CSS box model?	
Select the correct answer.	
	Margin
	Content
	Border
	Style
	Padding

Module 3

Introduction to JavaScript

Contents:

Module Overview	3-1
Lesson 1: Overview of JavaScript	3-2
Lesson 2: Introduction to the Document Object Model	3-13
Lesson 3: Introduction to jQuery	3-19
Lab: Displaying Data and Handling Events by Using JavaScript.	3-27
Module Review and Takeaways	3-34

Module Overview

HTML and CSS provide the structural, semantic, and presentation information for a web page. However, these technologies do not describe how the user interacts with a page by using a browser. To implement this functionality, all modern browsers include a JavaScript engine to support the use of scripts in a page. They also implement the Document Object Model (DOM), a W3C standard that defines how a browser should reflect a page in memory to enable scripting engines to access and alter the contents of that page.

This module introduces JavaScript programming and the DOM. It also briefly describes jQuery, a powerful open source JavaScript library that makes web programming in JavaScript easier.

Objectives

After completing this module, you will be able to:

- Describe basic JavaScript syntax.
- Write JavaScript code that uses the DOM to alter and retrieve info from a web page.
- Use jQuery to simplify many common JavaScript programming tasks.

Lesson 1

Overview of JavaScript

There are many programming languages in common use, but JavaScript is by far the most common programming language used for adding functionality to web pages. In this lesson, you will learn about the syntax of JavaScript to enable you to start writing code for your own web pages and to understand how JavaScript code written by other developers works.

Lesson Objectives

After completing this lesson, you will be able to:

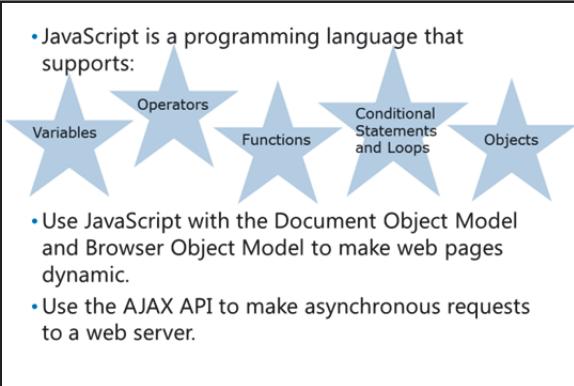
- Explain the purpose of JavaScript.
- Describe the basic syntax of statements and comments in JavaScript.
- Declare variables and write expressions by using JavaScript operators.
- Create and call JavaScript functions.
- Use conditional statements to control execution flow.
- Use loop statements to implement repeated operations.
- Use JavaScript objects in your code.
- Use JavaScript Object Notation (JSON) syntax to define an array of objects.

What is JavaScript?

JavaScript originated as a programming language in the 1990s and has steadily evolved. The standard upon which it is based, ECMA-262, is frequently updated and augmented. Consequently, the JavaScript engines used in modern browsers are now exponentially faster and more functional than their predecessors.

At its heart, JavaScript is a scripting engine with the same basic features as any other programming language. It provides:

- **Variables** for storing information.
- **Operators** for performing calculations and comparisons.
- **Functions** for grouping statements into reusable chunks.
- **Conditional statements** and **loop constructs** to control program flow.
- The ability to create **objects** with properties, methods, and events.



By itself, JavaScript cannot do much more than perform calculations and manipulate text, but in combination with the DOM that all browsers implement you can do far more. For example, you can use JavaScript code to:

- Add or remove items to a list displayed on a page.
- Add, change, or remove text on a page.
- Change the CSS styles applied to a set of elements on the page.

- React to events, such as a mouse clicking a button.
- Validate the contents of a form before they are sent to the web server.
- Obtain information about the browser displaying the web page, such as the manufacturer and version, and even environmental information such as the current window size and local time.
- Display an alert in the user's browser.

Additionally, the combination of JavaScript and the **XMLHttpRequest** API (commonly referred to as AJAX) enables a web page to make asynchronous requests back to the web server. The JavaScript code for a page can use this feature to query a server for more information, without requiring that the entire page be reloaded in the browser.

 **Note:** JavaScript is not Java. It was originally named Mocha and then LiveScript before its creator, Brendan Eich, became a Sun employee and Sun decided to rename it JavaScript. The standards name for the language is ECMA-262 because Sun would not license the JavaScript name to the standards body. Microsoft's implementation of ECMA-262 is called JScript.

JavaScript Syntax

JavaScript has a simple syntax for writing statements, declaring variables, and adding comments. Any code you write must adhere to this syntax.

Statements

A statement is a single line of JavaScript. It represents an operation to be executed. For example, the declaration of a variable, the assignment of a value, or the call to a function. The following code fragments show some examples:

```
var thisVariable = 3;  
counter = counter + 1;  
GoDoThisThing();
```

All statements in JavaScript should be written on a single line and be terminated with a semicolon. The exception to this rule is that you can split a large string over several lines (for readability) by using a backslash. For example:

```
document.write("An incredibly really \  
 very long greeting to the world");
```

- A JavaScript statement represents a line of code to be run
 - Terminate statements with a semicolon
- ```
var thisVariable = 3;
counter = counter + 1;
GoDoThisThing();
document.write("An incredibly really \
 very long greeting to the world");
```
- Use comments to add notes to your scripts
- ```
document.write("I'm learning JavaScript"); // display a message  
  
/* You can use a multi-line comment  
to add more information */
```

 **Note:** The semicolon terminator is actually optional. However, if you do not terminate statements with a semicolon, JavaScript attempts to discern where they should have been placed, sometimes with unintended results.

You can combine statements into blocks, delimited by opening and closing curly braces, { and }. This syntax is used by many common programming constructs such as functions, **if**, **switch**, **while**, and **for** statements, which are discussed later in this lesson.

Comments

Comments enable you to add descriptive notes and documentation to your JavaScript code. The JavaScript interpreter does not treat comments as part of the code and does not attempt to understand their contents. JavaScript supports two different styles of comment: multi-line comments that begin with `/*` and end with `*/`, and single-line comments that begin with `//` and finish at the end of the line.

Comments should describe the purpose or the reasoning of your code in plain English, to enable you or another developer to quickly understand it.

Statements and comments

```
<script type="text/javascript">
    document.write("I'm learning JavaScript"); // display a message to the user
    /* You can use a multi-line comment
       to add more information */
    alert("I'm learning JavaScript too!");
</script>
```

Variables, Data Types, and Operators

Variables

Variables are used to store data. There are three ways to declare a variable:

1. Give it a name and a value.

```
greeting = "Hello";
```

2. Declare it without giving it a value by using the `var` keyword. Until a variable is given a value, JavaScript will return its value as `undefined`.

```
var mystery;
```

3. Combine the above two approaches (this is the recommended style).

```
var code = "Spang";
```

There are two important rules for naming variables in JavaScript:

1. Variable names must begin with a letter or the underscore character.
2. Variable names are case sensitive.

Bearing this in mind, try to avoid giving variables similar names that are differentiated only by letter casing, such as `valueOfMilk` and `ValueOfMilk`. Use descriptive names that will make your code easier to understand and to debug. For example, `selectedTime`, `preferredTrack`, `currentSession`.

Data Types

Unlike C#, Visual Basic, and other common programming languages, you cannot specify the type of a variable in JavaScript. You declare it with the `var` keyword, and then JavaScript attempts to discern its type for you. JavaScript recognizes three simple types:

1. **String:** Any set of characters (alphanumeric and punctuation) enclosed in double quotes. To include special characters such as ',', ;, \, and & in your string, escape them with a backslash. Also use a backslash to split a string over two or more lines.

```
var simple = "Green Eggs and Ham";
var escaped = "\"Green Eggs \& Ham\"";
var verylong = "Cracked, fried, overripe ovoids and \
porcine strips cooked medium well and allowed to cool";
```

2. **Number:** Any integer or decimal number. Do not wrap a number between double quotes when you declare a numeric variable or it will be treated as a string.

```
var answer = 42;
var actuallyAString = "42"; // not treated as a number
```

3. **Boolean:** A Boolean value: true or false.

```
var canYouReadThis = true;
```

JavaScript also converts data between types, which can lead to confusion if you are not careful. For example, the numeric value 0 evaluates to false in Boolean expressions, so it is important to use the correct operator when comparing the values of variables.

 **Note:** Remember that if you declare a variable but do not give it a value, the variable is **undefined**. You can also declare a variable and set it to **null**, like this:

var variableWithValue = null;

Setting a variable to **null** indicates that a value does not exist, rather than that a variable has not been given a value. It is important to understand the difference.

You can determine the current type of data in a variable by using the **typeof** operator:

```
var data = 99;
...
if (typeof data == "number") {
    // data is numeric
}
```

Operators

An operator is a keyword or a symbol indicating how to combine one or more values into an expression. For example, the addition operator + indicates that two numbers should be added together. There are six groups of operators in JavaScript:

4. **Arithmetic operators** indicate a mathematical function to be performed on values/variables:

- + (addition)
- (subtraction)
- * (multiplication)
- / (division)
- % (modulus)
- ++ (increment)
- (decrement)

5. **Assignment operators** assign values to JavaScript variables:

```
x = y  
x += y (x = x + y)  
x -= y (x = x - y)  
x *= y (x = x * y)  
x /= y (x = x / y)  
x %= y (x = x % y)
```

6. **Comparison operators** determine if two values/variables are or are not equal. The first set of comparison operators converts the two values/variables to the same type before comparison.

```
== (is equal to)  
!= (is not equal to)  
> (is greater than)  
< (is less than)  
>= (is greater than or equal to)  
<= (is less than or equal to)
```

The second set of two comparison operators does not convert the two values/variables to the same type before comparison.

```
===(is equal in value and in type)
```

```
!== (is not equal in value or in type)
```

7. **Boolean operators** are used to perform Boolean operations.

`x && y` returns true if `x` and `y` are both true, false otherwise.

`x || y` returns true if either `x` or `y` or both are true, false otherwise.

`!x` returns true if `x` is false, false otherwise.

8. The **ternary conditional operator**:

`:=` assigns one of two values to a variable based on a condition. For example, the expression `x = (condition)?value1:value2;` sets `x` to **value1** if condition is **true**, **value2** otherwise.

9. The **string operator**:

`+` concatenates two strings. For example, `"Bo" + "om"` returns `"Boom"`.

There are a number of issues to be aware of with respect to JavaScript operators and how they convert values/variables between types as the JavaScript interpreter executes expressions.

- If you add a number and a string, the result will be a string!

```
x=10 + 10; // x is set to the number 20;  
y="10"+10; // y is set to the string "1010";  
z="Ten"+10; // x is set to the string "Ten10";
```

- `0`, `""` (the empty string), **undefined**, and **null** all evaluate to **false** in Boolean operations. Always use `==` when comparing to any of these values.

```
var zero = 0;  
var emptyString = "";  
var falseVar = false;  
zero == falseVar; // returns true;  
zero === falseVar; // returns false;
```

```
emptyString == False; // returns true;
emptyString === False; // returns false;
```

Functions

A function is a named sequence of statements that perform a specific task. Functions are useful for defining reusable blocks of code. After it has been defined, a function can be called from elsewhere in the script and the sequence of statements that constitute the function will run before execution returns to the next statement after the one that called the function.

Function definitions in JavaScript all have the same syntax:

```
function aName( argument1, argument2, ...,
argumentN ) {
    statement1;
    statement2;
    ...
    statementN;
}
```

- Functions are named blocks of reusable code:

```
function aName( argument1, argument2, ..., argumentN )
{
    statement1;
    statement2;
    ...
    statementN;
}
```

- Arguments are only accessible inside the function
- A function can return a value
- A function can also declare local variables
- Global variables defined outside of a function are available to all functions in scripts referenced by a page

There are four parts to a function declaration:

- The **function** keyword indicates this is the start of a function definition.
- The **function** name. You use this name to run the function. It is case-sensitive.
- A comma-separated list of values, called **arguments**, which you can pass to the function. This list is enclosed in parentheses. If the function has no arguments, it should still have the pair of parentheses after its name.
- A set of JavaScript statements enclosed in a pair of curly braces. These statements run when the function is invoked.

A function uses the arguments like variables; it can read their values and modify them, but they only exist inside the function.

 **Note:** Function arguments are optional. If you don't specify any arguments, you can still pass parameters into a function. The arguments are available in an array called **arguments**. You can access the first argument by using the expression **arguments[0]**, the second argument by using the expression **arguments[1]**, and so on. This mechanism gives you a way to define methods that can take a variable number of parameters. You can find out how many parameters were passed in by querying the value of **arguments.length**.

A function that calculates a result can use the **return** statement to pass this result back to the statement that called the function. What happens when the value is returned depends on how the function was called.

For example, if you wanted to calculate the total hotel bill for a guest, you might write the following function and call it like so.

Creating and calling a function

```
function CalculateBill(numberOfNightsStay, nightlyRate, extras) {  
    return (numberOfNightsStay * nightlyRate) + extras;  
}  
  
// elsewhere in the script  
var TotalAmountOwed = CalculateBill(10, 100, 50);
```

 **Note:** Not all functions have a name. You can even declare anonymous functions. You typically use anonymous functions when writing code to handle events or implement callbacks. In these cases, the function is invoked by the browser (or whatever environment your code happens to be running in) rather than by your code, and it is referenced by a variable rather than its name. For an example, see the topic "Handling Events in the DOM" in the next lesson.

You can declare a local variable within a function. Only the statements within that function can use it. It will disappear and be removed from memory when the function has finished.

You can also declare global variables. A global variable is a variable declared in your JavaScript code outside of a function. Any function on a web page that references your JavaScript code can use the global variable. A global variable remains in memory until the page is closed.

Conditional Statements

Conditional statements enable you to make decisions in your JavaScript code and execute different statements depending on whether a Boolean condition is true or false. There are two common types of conditional statements:

1. An **if** statement runs a block of code if a condition is true. For syntactic reasons, you must enclose the condition in round brackets. For example:

```
if (TotalAmountOwed > AdvancePaid) {  
    GenerateNewInvoice(); // runs if  
    condition is true  
}
```

• JavaScript provides two conditional constructs

```
• if:  
    if (TotalAmountPaid > AdvancePaid){  
        GenerateNewInvoice();  
    } else {  
        WishGuestAPleasantJourney();  
    }
```

```
• switch:  
    var RoomRate;  
    switch (typeOfRoom){  
        case "Suite":  
            RoomRate = 500;  
            break;  
        case "King":  
            RoomRate = 400;  
            break;  
        default:  
            RoomRate = 300;  
    }
```

 **Note:** A block of code in JavaScript starts with an opening curly brace, **{**, and terminates with a closing curly brace, **}**. The statements inside these braces are executed if the expression specified by the **if** statement evaluates to true. JavaScript uses this same block structure to delimit other syntactic structures, such as the **else** clause of an **if** statement, or the code for looping statement (described later in this lesson).

An **if** statement can have an optional **else** clause that runs a block of code if the Boolean condition is false.

```
if (TotalAmountOwed > AdvancePaid) {  
    GenerateNewInvoice(); // runs if condition is true  
} else {  
    WishGuestAPleasantJourney(); // runs if condition is false
```

```
}
```

2. A **switch** statement performs a series of comparisons against an expression and runs the series of statements code where the comparison matches the value of the expression specified by a **case** clause. As with the **if** statement, the expression must be enclosed in round brackets. The body of the **switch** statement is a single block, and execution runs until it meets the **break** statement, when it jumps to the first statement after the closing **}** that indicates the end of the **switch** statement block. If there is no **break** statement, execution continues into the code for the next case (which might not be what you want to happen, so watch out for bugs caused by missing break statements in your code). The code for the optional **default** case run if no previous cases match.

```
var RoomRate;
switch (typeOfRoom) {
  case "Suite":
    RoomRate = 500;
    break; // Use break to prevent code in next case statement being run.
  case "King":
    RoomRate = 400;
    break;
  default: // code to be executed if typeOfRoom does not match above cases.
    RoomRate = 300;
}
```

 **Best Practice:** Note that in the above **switch** example, if the value of **typeOfRoom** is **suite** or **king**, **RoomRate** will be set to 300 because JavaScript is case sensitive. To solve this, make the text all lowercase using `switch (typeOfRoom.toLowerCase())` and write all the **case** values in lower case.

The **toLowerCase** function is a built-in function that you can use with any JavaScript variable that contains a string value; it returns the string with all the characters in lower case. However, be careful; if you attempt to use this function with a variable that does not contain a string value it will cause an exception

Looping Statements

Looping statements enable you to iterate through a set of statements a number of times or until a Boolean condition is met. There are three types of looping statements in JavaScript:

- A **while** loop evaluates a Boolean condition, and then runs the accompanying block of code if this condition is true. The condition is then evaluated again, and if it is still true the block of code runs again. This process continues until the Boolean condition evaluates to false. If the condition evaluates to false immediately, the block of code might never be run. For example:

```
while (GuestIsStillCheckedIn())
{
  numberOfNightsStay += 1;
}
```

- JavaScript provides three loop constructs

- **while:**

```
while (GuestIsStillCheckedIn())
{
  numberOfNightsStay += 1;
}
```

- **do while:**

```
do {
  eatARoundOfToast();
} while (StillHungry());
```

- **for:**

```
for (var i=0; i<10; i++) {
  plumpUpAPillow();
```

- A **do while** loop runs a set block of code and then evaluates a Boolean condition. If the condition is true then the block of code runs again and the condition is reevaluated. The process repeats until the condition is false. Note that the block of code always runs at least once. For example:

```
do {
    eatARoundOfToast();
} while (StillHungry())
```

- A **for** loop runs a set of statements while a condition is true, and this condition is typically based on the value of a control variable. The execution of a **for** statement is determined by three elements separated by a semicolon: a start condition, an end condition and a step. For example:

```
for (var i=0; i<10; i++) {
    plumpUpAPillow();
}
```

In this example, the start condition sets a counter variable **i** to zero. After **plumpUpAPillow()** is run, the step statement **i++** runs and adds one to the counter. Then, if the end condition (**i < 10**) is still true, the loop continues. In this example, the function **plumpUpAPillow()** runs 10 times.

All three loop types use an end condition to break out of the loop. You can also add a **break** statement to those within the block that defines the body of a loop to perform the same task. When the execution reaches a **break** statement, the loop stops running and execution continues with the first statement after the body of the loop.

Using Object Types

JavaScript is an excellent language in which to write object-oriented web applications. Like many other modern languages, it allows you to define objects that have properties, methods, and events. This subject is described in more detail in module 7, "Creating Objects and Methods by Using JavaScript".

JavaScript provides several built-in object types, including:

- The **String** object type, which enables you to handle strings of text. The **String** type provides properties such as **length** that return the number of characters in a string, and methods such as **concat** which you can use to join strings together, together with other methods that enable you to convert a string to upper or lower case, or search a string to find a substring.

```
var eventWelcome = new String('Welcome to your conference');
var len = eventWelcome.length;
```

- JavaScript has a number of built-in object types:

- String, Date, Array, RegExp

```
var seasonsArray = ["Spring", "Summer", "Autumn", "Winter"];
...
var autumnLocation = seasonsArray.indexOf("Autumn");

var re = new RegExp("[dh]og");
if (re.test("dog")) {...}
```

- JavaScript also provides singleton types providing useful functionality:

- Math, Global

 **Note:** Notice that you use the **new** operator to instantiate variables using the object types.

The **Date** object type, which enables you to work with dates. JavaScript represents data internally as the number of milliseconds elapsed since 01/01/1970 and all date methods use the GMT+0 (UTC) time zone, even though your computer may display a time that is consistent with your time zone.

```
var today = new Date(1346454000); // Number of milliseconds since 01/01/1970
var today = new Date("September 1, 2012");
var today = new Date(2012, 8, 1); // Note January is 0, ..., December is 11.
```

- The **Array** object type, which enables you to create and work with a zero-based, dynamic-length array of values. Arrays also provide methods enabling you to search for matching items in an array, to change the order of elements in the array, or to treat the array as a stack or queue structure.

```
var emptyThreeItemArray = new Array(3);
var seasonsArray = new Array("Spring", "Summer", "Autumn", "Winter");
var thirdSeason = seasonsArray[3]; // Winter
```

-  **Note:** Use square brackets, `[` and `]`, to access an element in an array.

You can also use array-literal notation to create and populate an array:

```
var seasonsArray = ["Spring", "Summer", "Autumn", "Winter"];
```

You can determine whether an object is a member of an array by using the **indexOf** function. You specify an object to look for, and the function returns the index of first matching item that it finds in the array, or -1 if there is no match. For example, the following code sets the variable **autumnLocation** to 2 because "Autumn" is in location 2 in **seasonsArray** (remember that arrays are zero-based, so the first item is in location 0):

```
var autumnLocation = seasonsArray.indexOf("Autumn");
```

- The **RegExp** object type, which enables you to create and work with a regular expression for matching strings. Use the **test** method to determine whether a string matches a regular expression.

```
var re = new RegExp("[dh]og");
if (re.test("dog")) {...}
```

JavaScript also defines some singleton object types. You do not use these types to declare your own variables, rather you use the functionality that these types provide. These singleton objects include:

- The **Math** object gives you access to various mathematical constants (for example, Pi and E) and functions (sine, cosine, square root, and a pseudo-random number generator) as static properties and methods. For example, **Math.E**, **Math.cos()**; **var seed = Math.random();**
- The **Global** object contains global functions and constants and is the parent object for the **undefined**, **NaN**, and **Infinity** constants. It cannot be instantiated.

-  **Additional Reading:** You can find details for all these objects at <http://go.microsoft.com/fwlink/?LinkID=267718>.

Defining Arrays of Objects by Using JSON

JavaScript Object Notation, or JSON as it is more commonly known, is a syntax for representing one or more instances of an object and the values of their properties as a string, in a process known as serialization. The basic syntax is as follows:

```
var myObject = {
    "propertyName1" : "propertyValue1",
    "propertyName2" : "propertyValue2",
    ...
    "propertyNameN" : "propertyValueN"
};
```

For example, you could create an **Attendee** object representing a conference attendee like this:

```
var singleAttendee = {
    "name" : "Eric Gruber",
    "currentTrack" : "1",
};
```

There are a few basic rules:

- Property names and values are separated by a colon.
- Property name-value pairs are separated by a comma.
- All property names and values must be double-quoted strings.
- Trailing commas in objects and arrays are forbidden.
- The property list is enclosed in a pair of curly braces.

A serialized collection of objects is denoted as a comma-separated list of serialized objects enclosed in a pair of square brackets. For example, here are the two **Attendee** objects serialized into JSON.

```
var listOfAttendees = [
    {
        "name": "Eric Gruber",
        "currentTrack": "1"
    },
    {
        "name": "Martin Weber",
        "currentTrack": "2"
    }
];
```

JSON has become increasingly important as it is currently the de facto format for passing data in an AJAX request between a web page and a web server. JavaScript provides APIs for converting data into JSON format (**JSON.stringify**), and for parsing JSON data (**JSON.parse**).



Additional Reading: The full JSON syntax can be found at https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/JSON.

- JSON is a format for serializing objects:

```
var attendees = [
    {
        "name": "Eric Gruber",
        "currentTrack": "1"
    },
    {
        "name": "Martin Weber",
        "currentTrack": "2"
    }
]
```

- JavaScript provides APIs for serializing and parsing JSON data

Lesson 2

Introduction to the Document Object Model

You use HTML to define the structure of a page, but web browsers use another W3C standard, the DOM, to represent that structure internally. In this lesson, you will learn the fundamentals of the DOM, using it to add a level of interactivity to your pages with the help of some JavaScript code.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose and basic structure of the DOM.
- Select elements by using the DOM.
- Add, remove, and modify elements by using the DOM.
- Handle events for controls on a web page.

The Document Object Model

All modern browsers implement an object model called the DOM devised by the World Wide Web Consortium (W3C) to represent the structure of a web page. The DOM provides a programmatic API, enabling you to write JavaScript code that performs common actions such as finding the title of the page, changing the contents of a list, redirecting a page, adding new elements to the page, and much more. The DOM defines the properties that a script can change for an element in the page, and what actions can be made on the document as a whole.

• The DOM provides a programmatic API for controlling a browser and accessing the contents of a web page:

- Finding and setting the values of elements on a page
- Handling events for controls on a page
- Modifying the styles associated with elements
- Serializing and deserializing a page as an XML document
- Validating and updating web pages

Most modern browsers provide access to the following APIs:

- The **DOM Core**, which defines the basic interfaces for accessing a **document**, **elements**, **attributes**, and **text** on a page.
- The **DOM Event Model**, which defines the basic set of UI and page events and how to **add** and **remove** handlers for events in general.
- The **DOM Style Model**, which defines a set of interfaces for accessing any type of style sheet and the CSS rules within them.
- The **DOM Traversal and Range Models**, which define APIs for traversing the elements in a web page and **manipulating** sets of elements at once rather than one at a time.
- The **DOM HTML API**, which defines **objects** and **methods** representing each type of element in the HTML 4.01 and XHTML 1.0 specifications.
- The **DOM Load and Save API**, which enables you to **serialize** and **deserialize** DOM representations of the page into an XML document.
- The **DOM Validation API**, which contains methods to **dynamically update** documents so they become valid against the HTML 4.01 specification.

The DOM specification was completed in 2004 and assumes that a web page is written in HTML 4.01 or XHTML 1, which were the standards available at that time. Work is currently under way to simplify the DOM specification and to update it for HTML5 and CSS3. This work is currently known as **DOM4**.



Note: The DOM was designed to be independent of any one programming language. However, as browsers tend only to have JavaScript interpreters installed in them, programming with the DOM has become synonymous with client-side JavaScript programming.

Finding Elements in the DOM

After a page has loaded, a common action when scripting against the DOM is to find an element or set of elements to query or manipulate. For example, you may need to obtain a reference to a list so that you can populate it with elements retrieved from a web service. The DOM represents the various parts of a document as a **set of arrays**:

- The **forms** array contains details of all the forms in the document.
- The **images** array contains all the images in the document.
- The **links** array contains all the hyperlinks in the document.
- The **anchors** array contains all the **<a>** tags in the document with a name attribute.
- The **applets** array contains all the applets in the document.

Given the following form:

```
<form name="contactForm">
<input type="text" name="nameBox" id="nameBoxId" />
</form>
```

You can reference the form by using:

```
document.forms[0] // forms is a zero-based array
document.forms["contactForm"]
document.forms.contactForm
document.contactForm
```

You can reference the **nameBox** text box by using:

```
document.forms.contactForm.elements[0]
document.forms.contactForm.elements["nameBox"]
document.forms.contactForm.nameBox
document.contactForm.nameBox
document.getElementById("nameBoxId")
```

All of these collections are child properties of the **document** object. This object represents the document as a whole. You can use dot notation to access each collection, and any property, method, or event defined by the DOM. To access individual elements in a collection, you can use an indexer or reference it by the value of its name attribute. For example, if you have the following simple contact form on a page:

```
<form name="contactForm">
<input type="text" name="nameBox" id="nameBoxId" />
</form>
```

You can access the DOM object representing the form in the following ways:

```
document.forms[0] // forms is a zero-based array
document.forms["contactForm"]
document.forms.contactForm
document.contactForm
```

You can also access the DOM object representing the textbox in this form in several ways, including:

```
document.forms.contactForm.elements[0]
document.forms.contactForm.elements["nameBox"]
document.forms.contactForm.nameBox
document.contactForm.nameBox
```

Alternatively, the DOM defines methods on the **document** object that retrieve elements based on the value of their **ID** and **name** attributes. You can use these methods to quickly find a matching element without having to traverse the path to that element. These methods include:

MCT USE ONLY
STUDENT USE PROHIBITED

- **document.getElementById(IdString)**, which returns the single element whose ID attribute has the value specified by **IdString**.
- **document.getElementsByName(NameString)**, which returns an array of elements whose name attribute has the value specified by **NameString**.

The following example shows how to obtain a reference to the **nameBoxId** textbox in the form by using the **getElementById** method:

```
var userNameBox = document.getElementById("nameBoxId");
var username = userNameBox.value;
```

Adding, Removing, and Manipulating Objects in the DOM

After you have found an element or set of elements to work with, the next step is often to change them in some way, adding new child elements, modifying existing elements or removing them.

The DOM Core API defines several methods to create new objects for a document, including:

- **document.createElement(tagname)**
- **document.createTextNode(string)**
- **document.createAttribute(name, value)**
- **document.createDocumentFragment**

All four of these methods actually create a DOM node object, which is the generic representation of an element, text, or attribute in the DOM. After you have created the DOM node object, you add it to the DOM. The simplest way is to call **document.getElementById()** to retrieve the parent element to which you wish to apply this object, and then call one of the following methods on this element:

- **appendChild(newNode)**, which adds the new node as the last child of the selected element.
- **insertBefore(newNode, existingNode)**, which adds the new node into the DOM before but as a sibling to the given **existingNode**.
- **replaceChild(newNode, existingNode)**, which replaces the existing child node with the new node.
- **replaceData(offset, length, string)**, which replaces the text in a text node. The **offset** parameter specifies which character to begin with, **length** specifies how many characters to replace, and **string** specifies the text to insert.

To use these methods effectively and target accurately where to add new nodes, you also need to move around the document tree. You can use the following properties to navigate through the DOM:

- **childNodes**, which returns all the child nodes of a node.
- **firstChild**, which returns the first child of a node.
- **lastChild**, which returns the last child of a node.
- **nextSibling**, which returns the node immediately following the current one.
- **parentNode**, which returns the parent node of a node.
- **previousSibling**, which returns the node immediately prior to the current node.

To modify an element on a page:

1. Create a new object containing the new data.
2. Find the parent element that should contain the new data.
3. Append, insert, or replace the data in the element with the new data.

To remove an element or attribute:

1. Find the parent element.
2. Use **removeChild** or **removeAttribute** to remove the data.

The following example shows how to modify a list in a page.

Modifying a list

```
<!-- HTML Markup for VenueList -->
<ul id="VenueList">
    <li>Room A</li>
    <li>Room B</li>
</ul>
// JavaScript code to modify the items in VenueList
var list = document.getElementById("VenueList");
// Create a new venue
var newItem = document.createElement("li");
newItem.textContent = "Room C";
// Add the new venue to the end of VenueList
list.appendChild(newItem);
```



Note: If you reinsert or reappend a node object into the document, it is automatically removed from its current position.

The DOM also defines methods for removing nodes from the document tree, including:

- **removeChild(node)**, which removes the target node.

```
document.removeChild(
    document.getElementById("VenueList").firstChild
);
```

- **removeAttribute(attributeName)**, which removes the named attribute from the element node.

```
var list = document.getElementById("VenueList");
list.removeAttribute("id");
```

- **removeAttributeNode(node)**, which removes the given attribute node from the element.

```
var list = document.getElementById("VenueList");
list.removeAttribute(list.attributes[0]);
```

To clear a text node rather than removing it completely, just set it to an empty string.

Handling Events in the DOM

HTML defines a number of events initiated by the user and the browser to which you can attach a JavaScript action. For example, when a user moves the mouse over a help icon, the **mouseover** event fires. When an event **fires**, if you have set a **listener** for the event, the browser will run it. Many HTML elements provide callbacks that you can use to capture the various events that occur and define code that acts as a listener. These callbacks typically have the name **oneventname** where *eventname* is the name of the event. For example, you can handle the **mouseover** event for an **** element by using the **onmouseover** callback, as follows:

- The DOM defines events that can be triggered by the browser or by the user
- Many HTML elements define callbacks that run when an event occurs:

```
var helplcon = document.getElementById("helplcon");
document.images.helplcon.onmouseover =
    function() { window.alert('Some help text'); };
```
- You can also define event listeners that run when an event fires:
 - This is useful if the same event needs to trigger multiple actions

```
helplcon.addEventListener("mouseover",
    function() { window.alert('Some help text'); }, false);
```
- To remove an event listener:

```
helplcon.removeEventListener("mouseover", ShowHelpText, false);
```

```

```

-  **Note:** A callback is a reference to a function that runs as the result of another action completing. In the case of an event handler for an HTML element, the browser causes the callback to run when it triggers the corresponding event.

You can also set the event handler as a property of the image displaying the help icon in the DOM, like this:

```
document.images.helpIcon.onmouseover =
    function() { window.alert('Some help text'); };
```

-  **Reader Aid:** This example shows an instance of an anonymous function; the **function** keyword is followed immediately by a pair of parentheses and the code that defines the body of the function. The function name is not required because this function will only ever be invoked when the **onmouseover** callback is triggered.

However, this method only allows you to set one listener on an event. The HTML DOM allows you to dynamically add and remove multiple event listeners from elements in an HTML document by using the following methods:

- **addEventListener(eventName, listenerFunction, bubbles)**, which adds the listener function to the element for the given **eventName**. You can pass the **listenerFunction** by name or as an anonymous function.

```
var helpIcon = document.getElementById("helpIcon");
// Add an event listener for the mouseover event
// by using a named function
Function ShowHelpText()
{
    window.alert('Some help text');
}
helpIcon.addEventListener("mouseover", ShowHelpText, false);
// Alternatively, using an anonymous function
helpIcon.addEventListener("mouseover",
    function() { window.alert('Some help text'); }, false);
```

- **removeEventListener(event, listenerFunction, bubbles)**, which removes the listener function from the element for the given **eventname**.

```
helpIcon.removeEventListener("mouseover", ShowHelpText, false);
```

-  **Note:** Keep the code in an event handler short and concise. Long-running event handlers may impact the responsiveness of the browser.

Some events in the HTML DOM 'bubble', meaning that if the event occurs on an element (and it is or isn't handled), the event will then also fire on the element's parent node and then on its grandparent node and so on, until the event reaches an element where it may not bubble up any further or it reaches the root node. Both **addEventListener** and **removeEventListener** have an optional third Boolean parameter indicating whether or not this is the case.



Note: Note that Internet Explorer 6, Internet Explorer 7, and Internet Explorer 8 do not support `addEventListener()` and `removeEventListener()`. Use the similar `attachEvent()` and `detachEvent()` functions instead.

The following example shows how to bind a small script, which in this case copies the contents of a text box to the screen, to the `onClick` event of a form button.

Binding an action to an event with the DOM

Binding an action to an event with the DOM

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Binding events with JavaScript</title>
</head>
<body>
    <form>
        <p>
            <label>Write Your Name:</label>
            <input type="text" id="NameBox" /></p>
            <input type="button" id="submit" value="Click to submit" />
        </form>
        <div id="thankYouArea"></div>
        <script type="text/javascript">
            function sayThankYou() {
                var userName = document.getElementById("NameBox").value;
                var thankYou = document.createElement("p");
                thankYou.textContent = "Thank you " + userName;
                document.getElementById("thankYouArea").appendChild(thankYou);
            }
            document.getElementById("submit").addEventListener("click", sayThankYou);
        </script>
    </body>
</html>
```

Lesson 3

Introduction to jQuery

Writing JavaScript code to control the browser and the contents of a web page has often been a time-consuming task. The different ways that browsers have implemented the DOM over the years have been similar, but almost never the same. Writing JavaScript code has often meant a lot of effort to support all browsers, or compromising your site by fully supporting only one or two browsers.

jQuery is an open source JavaScript library that deals with cross-browser incompatibilities for you, leaving you free to write one piece of code that runs on all browsers. It provides a lightweight but powerful solution for selecting, creating, and manipulating elements on a page, handling user events in the browser, and making AJAX calls to web services. It also has an active development community writing and improving many plug-in scripts to provide amazing effects that you can use in your sites.

Microsoft actively contributes to jQuery and has includes the jQuery library in new ASP.NET web site and web application projects in Visual Studio.

Lesson Objectives

After completing this lesson, you will be able to:

- List the capabilities of jQuery.
- Select and traverse through set of elements by using jQuery.
- Add, remove, and modify elements from a page by using jQuery.
- Handle control events by using jQuery.

The jQuery Library

The jQuery library is a single JavaScript file that, once referenced in your page, allows you to build potentially complex, cross-browser functions and effects for your site, with much less effort than it would take without it.

Structure of the jQuery Library

jQuery provides a great deal of functionality on top of which many people have built extensions and plug-ins. The jQuery library includes the following APIs:

- **Core.** The jQuery core APIs enable you to mark code for execution once the page has finished loading, create DOM elements, and loop through a set of selected elements.
- **Selector Engine.** The jQuery selector engine ("sizzle") is one of the main highlights of jQuery. It enables you to quickly select a group of elements by using CSS selectors.
- **Filtering and Traversing.** jQuery defines many functions for adding criteria to your element selection (for example, **first**, **contains**, and **last**) and also for traversing the DOM tree (for example, **next**, **prev**, **parent**, and **siblings**).
- **Create, Modify and Delete.** jQuery enables you to modify the text content, attributes, and styles on a selected element, as well as to **insert** and **remove** elements from the DOM.

jQuery provides portability for JavaScript code, enabling you to easily build cross-browser web applications:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>jQuery Example</title>
<script type="text/javascript" src="Scripts/jquery-1.8.0.min.js">
</script>
</head>
<body>
<script type="text/javascript">
$(document).ready(function () {
    // some code
});
</script>
</body>
</html>
```

- **Presentation.** jQuery enables you to create new, change existing, and remove CSS styles, as the need arises.
- **Events.** jQuery enables you to add script to and remove script from event handlers in the browser, such as **mouseover**, **toggle**, **click**, and so on.
- **Animation and Effects.** jQuery enables you to add a number of visual effects to elements on a page. For example, **fade** in and out, **slide** up, slide down, and so on.
- **AJAX.** jQuery provides a number of alternative methods for making AJAX requests to both third-party web services and your own web server for more content.
- **Utilities.** jQuery provides a number of extra methods for working with arrays, strings, and other built-in JavaScript objects.

jQuery is a JavaScript file that you reference in your page with a script tag. You can download it and reference a local copy in your page.

```
<script src="Scripts/jquery-1.8.0.min.js" type="text/javascript"></script>
```

Alternatively, you can reference its location on one of several **Content Delivery Networks** (CDNs), which ensures faster downloads and caching in the browser.

```
<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.8.0.min.js"
type="text/javascript"></script>
```

Note the protocol-relative URL that ensures that jQuery is downloaded by using the same protocol as your page (http or https).

Using the jQuery Function

The primary feature of the jQuery library is the **jQuery** function. You use this function to select items from the DOM. You can then write JavaScript code to manipulate these items. For example, you can use the expression **jQuery (document)** to select the entire HTML document for a web page. The **jQuery** function is used so often that the jQuery library provides the **\$** function as a shorthand name.

Most calls to the jQuery library ensure that the entire page has loaded in the browser before executing, but you can use the **ready** function to ensure that a page has completely loaded and all of the elements that it contains are available in the DOM. This example demonstrates the most common way of doing this. The anonymous function contains the code that will run when the ready event is fired and the page has been loaded.

Using the jQuery ready function in a web page.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>jQuery Example</title>
    <script type="text/javascript" src="Scripts/jquery-1.8.0.min.js"></script>
</head>
<body>
    ...
    <script type="text/javascript">
        $(document).ready(function () {
            // some code
        });
    </script>
</body>
</html>
```

 **Additional Reading:** You can find the latest version of jQuery, full documentation and a list of plug-ins available for use at <http://go.microsoft.com/fwlink/?LinkId=267719>.

Demonstration: Adding jQuery to a Web Project

In this demonstration, you will see how to add the jQuery library to a website project by using the Package Manager in Visual Studio 2012, and how to enable jQuery IntelliSense for script files in your project.

Demonstration Steps

Add jQuery to a project by using nuGet

1. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
2. In Visual Studio, on the **File** menu, point to **New** and then click **Web Site**.
3. In the **New Web Site** dialog box, in the middle pane click **ASP.NET Empty Web Site**.

 **Note:** It does not matter whether you select the **Visual Basic** or **Visual C#** templates in the left pane; the templates for both languages enable you to create HTML5 web pages and implement functionality by using JavaScript.

4. From the **Web location** list, click **File System**, set the file path to **E:\Mod03\Democode\jQueryDemoTest**, and then click **OK**.
5. On the **Tools** menu, point to **Library Package Manager**, and then click **Manage NuGet Packages for Solution**.
6. In the **Manage NuGet Packages** dialog box, click **Online**. A list of available packages for download appears in order of popularity.
7. In the **Search Online** box, type **jQuery**.
8. In the list of packages, click **jQuery**, and click **Install**.
9. In the **Select Projects** dialog box, click **OK**.
10. In the **Manage NuGet Packages** dialog box, click **Close**.
11. In Solution Explorer, notice that a **Scripts** folder has been added to the project and that it contains three files.

 **Note:** The three files are the uncompressed jQuery library, a minified version for use on production servers, and an IntelliSense file for Visual Studio to use.

12. On the **File** menu, click **Save jQueryDemoTest**.

Enable jQuery IntelliSense

1. In the Solution Explorer window, right-click the **Scripts** folder, point to **Add**, and then click **JavaScript File**.
2. In the **Specify Name for Item** dialog box, type **test**, and then click **OK**.
3. Type the following code and notice that IntelliSense is unable to offer suggestions for statement completion for jQuery functions such as **ready**.

MCT USE ONLY. STUDENT USE PROHIBITED

```
$(document).ready(function () {  
    // your code here  
});
```

4. In the Solution Explorer window, right-click the **Scripts** folder, point to **Add**, and then click **JavaScript File**.
5. In the **Specify Name for Item** dialog box, type **_references.js**, and then click **OK**.
6. Add the following code to **_references.js** and then save it.

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

-  **Note:** If necessary, replace the jQuery version number (1.8.2) with that of the files downloaded by the package manager.

7. In the **_references.js** file, type the following code and notice that IntelliSense is now able to offer suggestions for statement completion for jQuery functions, including the **ready** function.

```
$(document).ready(function () {  
    // your code here  
});
```

8. On the **File** menu, click **Save All**.
9. Close Visual Studio 2012.

Selecting Elements and Traversing the DOM by Using jQuery

One of the main features of jQuery is its **powerful selection engine**. The base selector function, **jQuery()**, or **\$()**, uses the same selector syntax as CSS. This means that you can search for a set of elements by:

- Tag name. For example, `$(“h2”)`
- Element ID. For example, `$(“#logo”)`
- Element class. For example, `$(“.blue”)`
- An attribute on an element. For example, `$(“input[type=“text”]”)`

• jQuery uses the same selector syntax as CSS

```
<script type="text/javascript">  
$(document).ready(function () {  
    $("h2").each(function () {  
        this.style.color = "red";  
    });  
</script>
```

• jQuery provides additional functions for traversing and filtering elements

-  **Note:** These selectors were described in Module 1, "Overview of HTML and CSS", in the topic "How CSS Selectors Work".

- Using jQuery selector filters. For example, `$(“p:first”)`

-  **Note:** jQuery selector filters extend the filtering capabilities available by using CSS. The example given above selects only the first `<p>` element on a web page. You can only use jQuery selector filters with jQuery and not in CSS stylesheets.

- Concatenating CSS selectors and jQuery selector filters together. For example, `$("section > h2:last")`

After jQuery has returned the set of selected elements, you can iterate through them, applying a filter as required, and manipulate them in some way: Change their style, add an event, or change text, and so on. In jQuery, the functions that iterate through the set of elements are known as **traversal functions**. For example, the **each** function iterates over every selected element for processing.

In this example, the **each** function is used to iterate over all the **h2** elements in a document and change their CSS styles so that their text content is red.

Selection traversal with the each function

```
<script type="text/javascript">
$(document).ready(function () {
    $("h2").each(function () {
        this.style.color = "red";
    });
});
</script>
```

jQuery includes many functions for traversing and filtering a selected set of elements. The following list highlights some of the more commonly used functions:

- eq(index)**, which returns the single element at the given index. Since sets have a zero-based index, `eq(0)` returns the first element of a set and `eq(9)` returns the tenth element, etc. The following example sets the color of the eighth `<p>` element on the web page to red.

```
$("p").eq(7).css("color", "red");
```

 **Note:** The **css** function enables you to set the CSS style of an element without requiring that you write an anonymous method to do so. The function call `css("color", "red")` is equivalent to writing the statement `this.style.color = "red";` in the body of an anonymous function that runs when the `<p>` element is selected.

- each(function)**, which iterates over a set of selected elements and applies the given function to it.

```
 $("p").each(function() {
    $(this).css("color", "red");
});
```

 **Note:** This example also shows how to use the jQuery **css** function inside the body of a JavaScript function. Note that because **css** is part of the jQuery library, you must call the **jQuery** function, `$`, to return the set of elements defined by **this**. The **css** function is then applied to this list of elements (there will be only one element in the list by virtue of the **each** function).

- filter(expression)**, which returns a subset of elements by applying the given filter over the original set. The expression for the filter can be a tag name, selector, or function that returns a Boolean value.

```
 $("p").filter( $(":first"));
```

- find(selectorString)**, which returns a subset of elements from those in your original set.

```
 $("form").find("input[type=text]");
```

- first()** and **last()**, which return the first or last element in the original set.

```
$(“nav > ol > li”).first();
```



Note: This example assumes that the `<nav>` element contains a list (``) with one or more list items (``):

```
<nav>
  <ol>
    <li> First Item </li>
    <li> Second Item </li>
  </ol>
</nav>
```

The example selects the "First Item" list item.

- **next()** and **prev()**, which return the element immediately following or preceding the selected element.

```
$(“nav”).next();
```

- **size()**, which returns the number of elements in the selected set.

```
$(“p”).size();
```

- **slice(int [, int])**, which returns a subset of the original set starting at the given index through to the end of the set. You can also pass a second argument to **slice()**, which denotes an end index.

```
$(“p”).slice(4);
```



Additional Reading: You can find a full list of traversal and filtering functions at <http://go.microsoft.com/fwlink/?LinkId=267720>.

Adding, Removing, and Modifying Elements by Using jQuery

After you have selected a set of elements, the next step is usually to modify them in some way, perhaps by adding some content, changing a style, or adding to or removing them from the web page. jQuery includes many functions for manipulating sets of elements. You typically use these methods with the **selector** function to determine the elements to change. The following list highlights some of the more commonly used functions:

- **addClass(className)**, which adds the given CSS class to each of the elements in the set.

```
$(“p”).addClass(“strike”);
```

- **append(htmlString)**, which adds the given HTML at the end of the content of each element in the set.

• Use the **selector** function to specify the elements to change or remove

• Common methods include:

• addClass	<code>\$(“p”).addClass(“strike”);</code>
• append	<code>\$(“ul”).append(“New item”);</code>
• detach	<code>\$(#Warning”).detach();</code>
• html	<code>\$(“h1”).html(“<hgroup>…</hgroup>”);</code>
• replaceWith	<code>\$(#Warning”).replaceWith(“<p>Panic over!</p>”);</code>
• val	<code>\$(“input[type=text”]).val();</code>

```
 $("ul").append("<li>The new last list item</li>");
```

- **detach()**, which removes all the elements in the set from the DOM.

```
 $("#Warning").detach();
```

 **Note:** This example assumes that the HTML page contains an element with the **id** property set to **Warning**:

```
<p id="Warning">
    ALERT ALERT
</p>
```

The code removes this item from the DOM and it is no longer displayed.

- **html(htmlString)**, which gets and sets the HTML content of the first element in the set.

```
 $("h1").html("<hgroup><h1>New Heading</h1></hgroup>");
```

- **replaceWith(htmlString)**, which replaces each element in the set with the given HTML.

```
 $("#Warning").replaceWith("<p>Panic Over!</p>");
```

- **val()**, which gets and sets the current text value of the first element in the set.

```
 $("input[type=text]").val();
```

Most of these functions apply to each element in the set of elements retrieved by your selector. However, several apply only to the first element in the set, including **attr()**, **css()**, **height()**, **html()**, **val()**, and **width()**.



Additional Reading: You can find a full list of manipulation functions at <http://go.microsoft.com/fwlink/?LinkId=267721>.

Handling Control Events by Using jQuery

jQuery enables you to add and remove JavaScript code for event handlers in the browser. You can write the JavaScript code for these handlers either in the HTML page or in separate script files and then bind them to the event. You bind a handler function to an event by using the jQuery **bind** method. This method takes two parameters:

- The name of the event being handled.
- The name of the function to run when the event occurs, or an anonymous function containing the script to run.

- Use the jQuery **selector** function to find the item that raises the event
- Use the **bind** method (or a jQuery shortcut) to bind the event handler to the event

```
<script type="text/javascript">
    $(document).ready(function () {
        $("#submit").click(
            function () {
                var userName = $("#NameBox").val();
                $("#thankYouArea").replaceWith(
                    "<p>Thank you " + userName + "</p>");
            }
        );
    });
</script>
```

```
 $("#submit").bind("click", function() {
    // code for click event handler for the submit button
});
```

MICHIGAN STATE UNIVERSITY STUDENT USE ONLY



Note: This example binds the handler function to the **click** event of the button with the **id** property set to **submit**.

You can use the **unbind()** function to remove the event handler, if necessary.

jQuery defines a shortcut method for all the common DOM events, such as **click()** shown in the next example.

The following example shows how to bind a small script, which in this case repeats the contents of a text box out to the screen, to the **click** event of a form's submit button.

Binding an action to an event by using jQuery

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Binding events with JQuery</title>
    <script type="text/javascript" src="Scripts/jquery-1.8.0.js"></script>
</head>
<body>
    <form>
        <p><label>Write Your Name: <input type="text" id="NameBox" /></label></p>
        <input type="button" id="submit" value="Click to submit" />
    </form>
    <div id="thankYouArea"></div>
<script type="text/javascript">
    $(document).ready(function () {
        $("#submit").click(
            function () {
                var userName = $("#NameBox").val();
                $("#thankYouArea").replaceWith("<p>Thank you " + userName + "</p>");
            }
        );
    });
</script>
</body>
</html>
```

The following list summarizes some of the **shortcut methods** that you can use to bind events to elements:

- **click(), dblclick()** : Binds to mouse click events.
- **error()** : Binds to occurrence of errors in the document such as broken links and missing images.
- **focus(), focusin(), focusout()** : Binds to element focus events.
- **keydown(), keyup(), keypress()** : Binds to user keyboard input events.
- **hover(), mousedown(), mouseup(), mouseenter(), mouseleave(), mouseout(), mouseover(), mousemove()** : Binds to mouse and cursor-related events.
- **load(), unload()** : Binds to events triggered when a specified element is loaded or unloaded on the page.
- **select()** : Binds to element selection events.

Each method can be passed to either the name of the handler function or to the function itself.

Demonstration: Displaying Data and Handling Events by Using JavaScript

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Displaying Data and Handling Events by Using JavaScript.

Scenario

The conference being organized by ContosoConf consists of a number of sessions that are organized into tracks. A track groups sessions of related technologies, and conference attendees can view the sessions in a track to determine which ones may be of most interest to them.

To assist conference attendees, you have been asked to create a Schedule page for the ContosoConf website listing the tracks and sessions for the conference.

Objectives

After completing this lab, you will be able to:

- Use JavaScript code to programmatically update the data displayed on an HTML5 page.
- Handle the events that can occur when a user interacts with a page by using JavaScript.
- Estimated Time: 60 minutes.
- Virtual Machine: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa\$\$w0rd**

Exercise 1: Displaying Data Programmatically

Scenario

In this exercise, you will create the Schedule page that displays a list of sessions.

First, you will use the HTML5 DOM to obtain a reference to the page's schedule list element. Then you will implement a function that creates list items (one list item for each session). Information about the sessions is stored in a file in JSON format. You will implement a function that reads this data and adds the details of each session to the list element. Finally, you will run the application and view the Schedule page to verify that it correctly displays the list of sessions.

The main tasks for this exercise are as follows:

1. Review the existing code for the Schedule page.
2. Write code to get the schedule list element on the Schedule page.
3. Implement the `createSessionElement` function that creates the list item for a session.
4. Implement the `displaySchedule` function that adds session items to the list for display.
5. Run the web application and view the Schedule page.

► Task 1: Review the existing code for the Schedule page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod03\Labfiles\Starter\Exercise 1** folder.

4. In the **ContosoConf** project, review the content of the page **schedule.htm**. Notice that the **schedule** page section, which will be used to display the list of sessions, currently contains an empty list, also named **schedule**:

```
<section class="page-section schedule">
    <div class="container">
        <h1>Schedule</h1>
        <ul id="schedule"></ul>
    </div>
</section>
```

5. Also notice that the **schedule.htm** page references the JavaScript code in the **scripts\pages\schedule.js** script file:

```
<script src="/scripts/pages/schedule.js" type="text/javascript"></script>
```

6. Review the **scripts\pages\schedule.js** script file. This file contains the details of each session held in JSON format. The data is held in an array named **schedule**, and each object in the array has three properties that specify the session id, the session title, and the tracks to which the session belongs (a session may be part of more than one track):

```
var schedule = [
    {
        "id": "session-1",
        "title": "Registration",
        "tracks": [1, 2]
    },
    {
        "id": "session-2",
        "title": "Moving the Web forward with HTML5",
        "tracks": [1, 2]
    },
    {
        "id": "session-3",
        "title": "Diving in at the deep end with Canvas",
        "tracks": [1]
    },
    {
        "id": "session-4",
        "title": "New Technologies in Enterprise",
        "tracks": [2]
    },
    ...
];
```

► **Task 2: Write code to get the schedule list element on the Schedule page**

1. In the **schedule.js** file, find the comment **TODO: Task 2**
2. Write JavaScript code to get the **schedule** list element from the DOM and assign it to a variable named **list**. You will use this variable to display the details of each session in the list on the Schedule page.
3. Use the **getElementById** method of the **document** object to find the list that has the **id** property set to **schedule**.

► **Task 3: Implement the createSessionElement function that creates the list item for a session**

1. In the **schedule.js** file, find the comment **TODO: Task 3**. This comment is located in the **createSessionElement** function, which looks like this:

```
function createSessionElement(session) {  
    ...  
};
```

The purpose of this function is to create a list element containing the name of the session passed in as the parameter.

2. Add JavaScript code to create a **** element, set its text content to the session title, and then return this element:
 - o Use the **createElement** method of the **document** object to create a new **li** object.
 - o Set the **textContent** property of the **li** object to the **title** property of the **session** parameter passed in to the **createSessionElement** function.
 - o Return the new **li** element.

► **Task 4: Implement the `displaySchedule` function that adds session items to the list for display**

1. In the **schedule.js** file, find the **TODO: Task 4** comment. This comment is located in the **displaySchedule** function, which looks like this:

```
function displaySchedule () {  
    clearList();  
    ...  
};
```

The purpose of this function is to display the title of each session in the list on the Schedule page.

2. Add JavaScript code to iterate over the **schedule** array containing the JSON data by using a **for** loop. Create a **session** object for each item in the array, and add the title of the session to the **list** element on the Schedule page.
 - Use the **createSessionElement** function that you implemented in Task 3 to create a list item for each session.
 - Use the **list** variable that you created in Task 2 to access the list element on the Schedule page.

► **Task 5: Run the web application and view the Schedule page**

1. Run the application and view the **schedule.htm** page to verify that the list of sessions is displayed.

The Schedule page should look like this:

MCT USE ONLY. STUDENT USE PROHIBITED

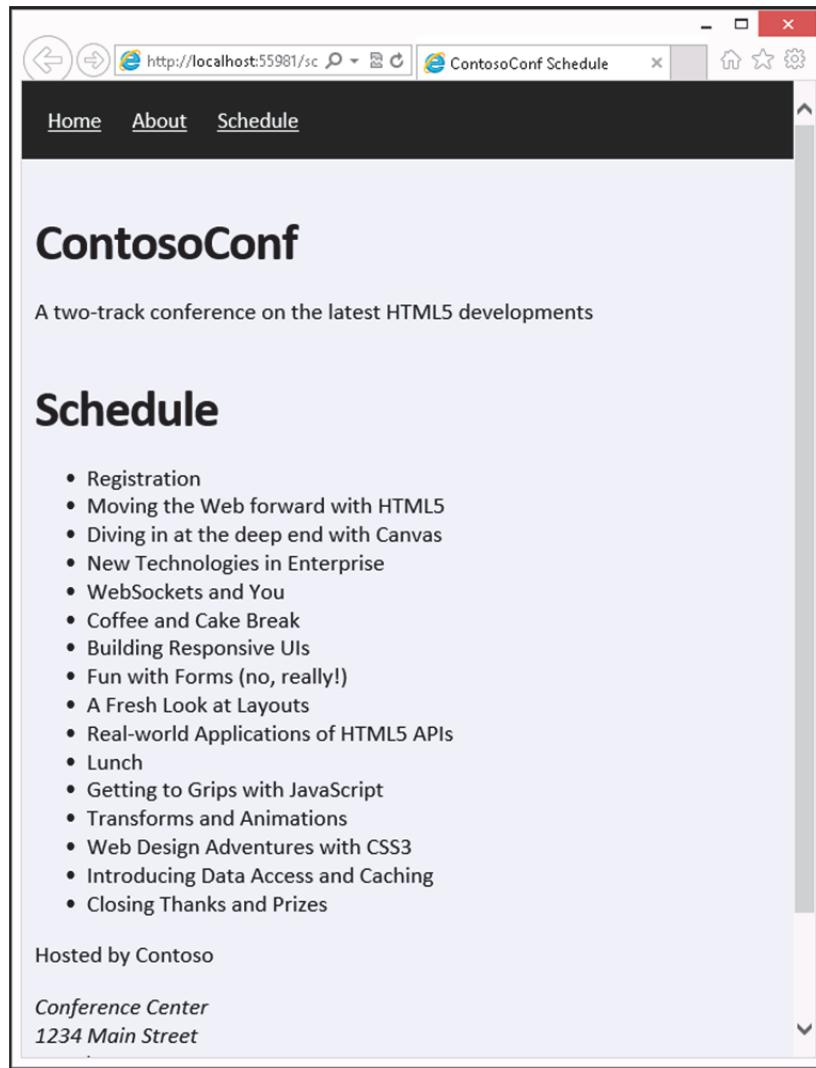


FIGURE 3.1:THE SCHEDULE PAGE WITH SESSIONS DISPLAYED

Note: Remember that you can use the F12 Developer Tools in Internet Explorer to debug your application. Also, if you make any changes to your code, make sure to clear the browser cache before running the application again (press Ctrl+R in the F12 Developer Tools window). Otherwise, Internet Explorer may attempt to run the previous version of your JavaScript code.

Results: After completing this exercise, you will have added a Schedule page to the ContosoConf application that displays the details of conference sessions.

Exercise 2: Handling Events

Scenario

In this exercise, you will add check boxes to the Schedule page to enable the user to specify which sessions should be displayed, according to the tracks that they are in.

First, you will add two checkbox HTML elements to the Schedule page; the first will enable the user to specify that the sessions for track 1 should be listed, and the second will enable the user to specify that the sessions for track 2 should be listed (if both checkboxes are checked, then the sessions for track 1 and

track 2 will both be listed). Then you will add JavaScript code to handle the click events of these checkboxes; you will update the **displaySchedule** function to show only sessions that are in the tracks currently selected by the checkboxes. Finally, you will run the application and view the Schedule page to verify that selecting and deselecting the checkboxes correctly updates the session list.

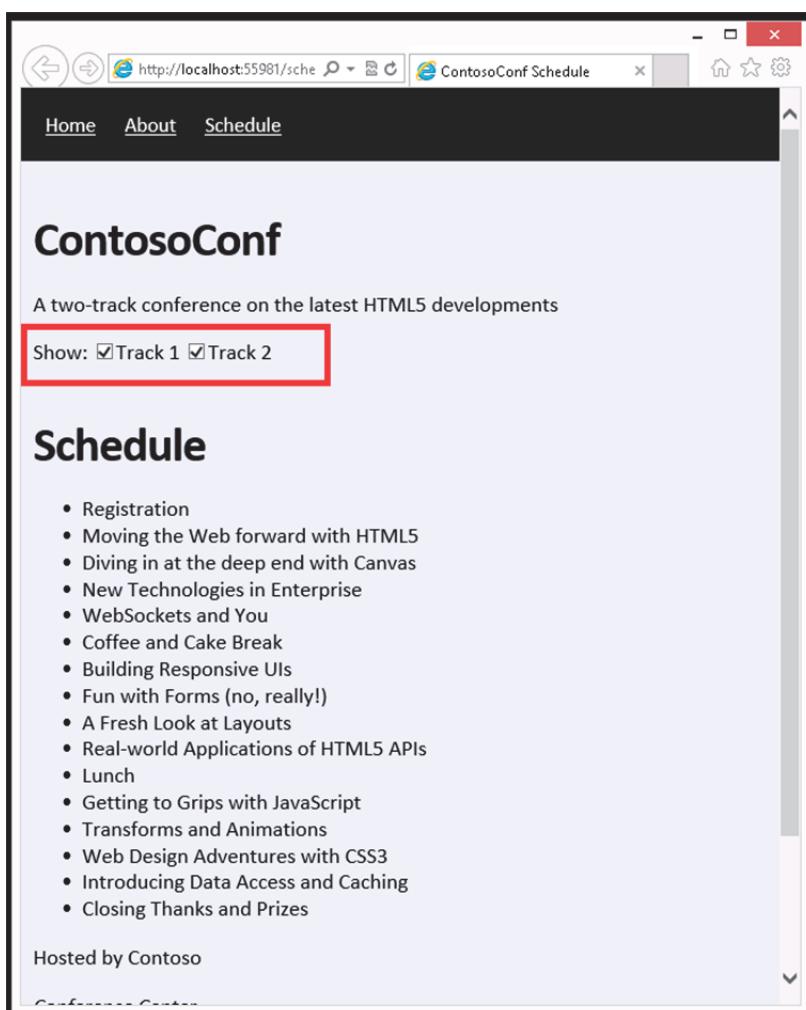
The main tasks for this exercise are as follows:

1. Add checkbox HTML elements.
2. Write code to get the checkbox elements from the Schedule page.
3. Add click event listeners for each checkbox.
4. Update the `displaySchedule` function to display the sessions for selected tracks.
5. Run the web application and view the Schedule page.

► Task 1: Add checkbox HTML elements

1. In Visual Studio, open the **ContosoConf.sln** solution in the **E:\Mod03\LabFiles\Starter\Exercise 2** folder. This project contains a working version of the application as it should appear at the end of exercise 1.
2. In the **schedule.htm** file, before the **schedule** list, add two checkboxes that enable the user to specify for which tracks the page should display session information:

The checkboxes should look like these, highlighted in the following image:



- Label the checkboxes with the text **Track 1** and **Track 2**.
 - Set the **id** attributes of the checkboxes to **show-track-1** and **show-track-2**.
 - Mark the checkboxes as checked by default.
- **Task 2: Write code to get the checkbox elements from the Schedule page**
1. In the **scripts\pages** folder, open the **schedule.js** file.
 2. After the **list** variable is defined, create two variables named **track1Checkbox** and **track2Checkbox**.
 3. Add JavaScript code to get the checkbox elements **show-track-1** and **show-track-2** from the DOM and reference them in these variables.
 - Use the **getElementsByld** method of the **document** object, and get elements with the ids **show-track-1** and **show-track-2**.
- **Task 3: Add click event listeners for each checkbox**
1. At the end of the **schedule.js** file, add an event listener for the click event of each checkbox. The event handler for each checkbox should call the **displaySchedule** function.
 - Use the **addEventListener** method to add the event handler for each check box.
- **Task 4: Update the displaySchedule function to display the sessions for selected tracks**
1. Modify the **displaySchedule** function to add sessions to the list only when they are in the currently selected tracks (one track, both tracks, or neither track might be selected).
 - Examine the **checked** property of the **track1Checkbox** and **track2Checkbox** elements to determine which track the user has selected.
 - The **session** parameter passed in to the **createSessionElement** method has a **tracks** property. This property is an array that specifies which track or tracks a session belongs to.
 - Use the **indexOf** function to determine whether the tracks property specifies that a session is in track 1, track 2, or both.
- **Task 5: Run the web application and view the Schedule page**
1. Run the application and view the **schedule.htm** page.
 2. Verify that if both checkboxes are selected all tracks are listed.
 3. Verify that if only Track 1 or Track 2 is selected, only the sessions for that track appear.
 4. Verify that if neither track is selected, no sessions are listed.

The sessions for Track 1 are:

- Registration.
- Moving the Web forward with HTML5.
- Diving in at the deep end with Canvas.
- WebSockets and You.
- Coffee and Cake Break.
- Building Responsive UIs.

- A Fresh Look at Layout.
- Lunch.
- Getting to Grips with JavaScript.
- Web Design Adventures with CSS3.
- Closing Thanks and Prizes.

The sessions for Track 2 are:

- Registration.
- Moving the Web forward with HTML5.
- New Technologies in Enterprise.
- Coffee and Cake Break.
- Fun with Forms (no, really!).
- Real-world Applications of HTML5 APIs.
- Lunch.
- Transformations and Animations.
- Introducing Data Access and Caching.
- Closing Thanks and Prizes.

Results: After completing this exercise, you will have updated the **Schedule** page to filter sessions based on which tracks have been selected.

Module Review and Takeaways

In this module, you have learned how to use JavaScript code to add dynamic functionality to a website. You have seen how to write JavaScript code to perform common operations such as selecting and modifying the data displayed on a page, and responding to events triggered by a user viewing the page. You have also seen how to use the jQuery library to simplify the code required to process the contents of a page, and to make this code portable so that it will function correctly in different web browsers.

Review Question(s)

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
All variables in JavaScript are strongly typed, and you must specify the type of a variable when you create it. True or False?	

Question: What is the purpose of the DOM?

Test Your Knowledge

Question	
Which jQuery function indicates that the contents for a page have been loaded into the browser?	
Select the correct answer.	
	loaded
	available
	\$
	ready
	bind

Module 4

Creating Forms to Collect and Validate User Input

Contents:

Module Overview	4-1
Lesson 1: Creating HTML5 Forms	4-2
Lesson 2: Validating User Input by Using HTML5 Attributes	4-6
Lesson 3: Validating User Input by Using JavaScript	4-10
Lab: Creating a Form and Validating User Input	4-14
Module Review and Takeaways	4-22

Module Overview

Web applications frequently need to gather user input in order to perform their tasks. A web page needs to be clear and concise about the input expected from a user in order to minimize frustrating misunderstandings about the information that the user should provide. Additionally, all input must be validated to ensure that it conforms to the requirements of the application.

In this module, you will learn how to define input forms by using the new input types available in HTML5. You will also see how to validate data by using HTML5 attributes. Finally, you will learn how to perform extended input validation by using JavaScript code, and how to provide feedback to users when their input is not valid or does not match the application's expectations.

Objectives

After completing this module, you will be able to:

- Create input forms by using HTML5.
- Use HTML5 form attributes to validate data.
- Write JavaScript code to perform validation tasks that cannot easily be implemented by using HTML5 attributes.

Lesson 1

Creating HTML5 Forms

HTML forms are the primary mechanism used by many web applications to retrieve user input. Previous versions of HTML provided basic forms facilities and depended on the server processing the data submitted by the user in order to validate this data. This process often resulted in poor performance and low user satisfaction due to the number of round trips required between the user's browser and the server. HTML5 provides an improved experience by enabling much more functionality to be performed in the user's browser before the data is submitted to the server.

In this lesson, you will learn about the new input types available in HTML5 that enable a browser to provide more responsive and immediate feedback to a user.

Lesson Objectives

After completing this lesson, you will be able to:

- Use the HTML5 **<form>** element and specify its common attributes.
- Explain how to use the new HTML5 input types.
- Describe the attributes available with the new HTML5 input types to improve the user's experience.

Declaring a Form in HTML5

Use the **<form>** element to define an input form in an HTML5 page.

A typical HTML5 form looks like this:

An HTML5 Form

```
<form name="userLogin" method="post" action="login.aspx">
    <fieldset>
        <legend>Enter your log in details:</legend>
        <div id="usernameField" class="field">
            <input id="uname" name="username" type="text" placeholder="First and Last Name" />
            <label for="uname">User's Name:</label>
        </div>
        <div id="passwordField" class="field">
            <input id="pwd" name="password" type="password" placeholder="Password" />
            <label for="pwd">User's Password:</label>
        </div>
    </fieldset>
    <input type="submit" value="Send" />
</form>
```

- Use an HTML5 form to gather user input:

```
<form name="userLogin" method="post" action="login.aspx">
    <fieldset>
        <legend>Enter your log in details:</legend>
        <div id="usernameField" class="field">
            <input id="uname" name="username" type="text" placeholder="First and Last Name" />
            <label for="uname">User's Name:</label>
        </div>
        <div id="passwordField" class="field">
            <input id="pwd" name="password" type="password" placeholder="Password" />
            <label for="pwd">User's Password:</label>
        </div>
    </fieldset>
    <input type="submit" value="Send" />
</form>
```

The **<form>** element has the following attributes:

- The **name** of the form. This attribute is used by the server to reference the fields on the form during processing.
- The **action** performed when the user submits the form to the server. This is a URL to which the form will be sent. If you omit the URL, the form is sent to the current URL of the web page.

- The **method** to be used to send the data. When submitting data to the server by using a form, this attribute should normally be set to **post**.

A form contains one or more input fields enclosed in a **<fieldset>** element. You use this element to draw a box around the set of fields, and add a label to the set by using the **<legend>** element. The **<fieldset>** element is optional, but it can be very useful on larger forms, where it is helpful to the user to see labeled sections.

Within the **<fieldset>** element, it is good practice to add a **<div>** element for each field in the form, specifying **class="field"** as an attribute. CSS can use the **field** class to style the form. Adding a **div** element also gives you a place to state the intention of the input element by providing a value for the **id** attribute. Additionally, the **div** element serves as a container for the HTML5 elements necessary to receive the input, its validation rules, and its label.

Within the **<div>** element, you add an **<input>** element to collect the data. An input element should contain the following attributes:

- An **id** and **name** attribute. The **id** attribute is typically used by CSS to style the field and JavaScript code that control the field in the browser; the **name** attribute is used by the server to reference fields on the form when it is submitted.
- A **type** attribute. This attribute specifies the type of the input. The browser uses the **type** attribute to create a visual control suitable for the data-type required. This is covered in greater detail in the next topic.
- A **placeholder** attribute. The placeholder appears on the control as a prompt to help the user know what to type.

Most input elements have an associated **<label>** element. This provides a way to label the input on the form. The **for** attribute indicates the input fields associated with the label. Note that the value of the **for** attribute should match the **id** attribute of the input element, not its **name**.

Finally, you need to provide a way to post the data to the server for processing; this is the purpose of the **<input>** element with the data type **submit**. This markup creates a clickable control that sends the data in the form to the server.

HTML5 Input Types and Elements

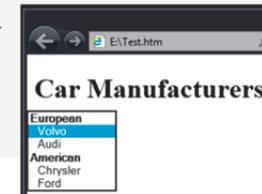
HTML5 provides you with a wide range of input types for forms.

The most generic of all the input types is **text**. Specifying **<input type="text" />** produces a text box in which the user enters data. However, HTML5 provides a range of other types that you can use with the **<input>** element to better define the kind of data that is expected, providing for better client-side validation and formatting.

To change the type of data you wish to collect, specify the **type** attribute of the **<input>** element, providing one of the following values: **button**, **checkbox**, **color**, **date**, **datetime**, **datetime-local**, **email**, **file**, **hidden**, **image**, **month**, **number**, **password**, **radio**, **range**, **reset**, **search**, **submit**, **tel**, **text**, **time**, **url**, or **week**. Note that not all these types are not yet fully adopted or even implemented by most browsers. They are designed to fail back to harmless text fields where they are not implemented.

- HTML5 defines a wide range of new input types and elements, but not all are widely implemented

```
<select id="carManufacturer" name="carManufacturer">
<optgroup label="European">
<option value="volvo">Volvo</option>
<option value="audi">Audi</option>
</optgroup>
<optgroup label="American">
<option value="chrysler">Chrysler</option>
<option value="ford">Ford</option>
</optgroup>
</select>
```



Not all input types create textboxes: **button**, **checkbox**, **radio**, **color**, and **range** create specific controls designed to gather the appropriate type of data. The type **file** triggers an interaction with the operating system to enable the user to upload a file to the server. The types **submit**, **reset**, and **image** all create button controls.

You can use a **<datalist>** element with **text input** to provide autocomplete options. Note that you do not need to specify the input type, you simply indicate that the input is a list and provide the possible options by using the **alist** element, as follows:

```
<input id="ageCategory" name="ageCategory" list="ageRanges" />
<datalist id="ageRanges">
    <option value="Under twos"></option>
    <option value="2 - 7"></option>
    <option value="8 - 12"></option>
    <option value="13-17"></option>
    <option value="Adult"></option>
</datalist>
```

The **<textarea>** element is similar to **<input>** in that it accepts typed input, but the control is rendered as a multi-line area using a fixed-width font. The attributes **cols** and **rows** control how wide it is and how many rows it has. You can set its **maxlength** attribute to set the maximum number of characters it will accept. You use it like this:

```
<textarea id="carDescription" name="carDescription" cols="80" rows="5"
placeholder="Enter a short description of your car" maxlength="399"/>
```

The **<select>** element is a container for a number of fixed **<option>** elements, which predefine inputs the user is allowed to give. This saves the user from typing common data, where the options are well understood and can be defined at design time. To group options into smaller logical lists, you use the **<optgroup>** element. This will help the user make sense of a longer list. You define a **select** element like this:

```
<select id="carManufacturer" name="carManufacturer">
    <optgroup label="European">
        <option value="volvo">Volvo</option>
        <option value="audi">Audi</option>
    </optgroup>
    <optgroup label="American">
        <option value="chrysler">Chrysler</option>
        <option value="ford">Ford</option>
    </optgroup>
</select>
```

HTML5 Input Attributes

HTML5 makes extensive use of attributes to improve the user experience and to guide the user through form completion. Many of these attributes are specific to each of the input types. For example, the **number** input type supports the **max**, **min**, **step**, and **value** attributes to indicate the maximum, minimum, increment, and default values, respectively. As with the input types, not all of these attributes are widely implemented at present.

Some attributes are independent of the input type. For example, a common requirement is to place the cursor in the first field of the form when the page loads. You can achieve this by setting the new **autofocus** attribute on the control.

Also new to HTML5 is the **autocomplete** attribute. If the user has entered the same data before, the previous value is automatically copied into the input by the browser, improving the user experience. This can be applied to either an individual field or the form.

In the following example, the email address field receives the focus when the form loads, and the email address is completed automatically if the user has typed it in before.

```
<form id="loginForm" action="login.aspx" method="post" autocomplete="on">
    Email: <input name="email" type="email" placeholder="Email address"
    autofocus="autofocus"/>
    Password: <input name="password" type="password" autocomplete="off" />
    <input type="submit" />
</form>
```

The **required** attribute indicates that a field is mandatory and that the form should not be submitted if it is left blank.

The **pattern** attribute enables you to apply a regular expression to a text input field to ensure that it conforms to a specific pattern.

The **placeholder** attribute puts temporary content in a text field to prompt the user to enter data.

- Input attributes modify the behavior of input types and forms to provide better feedback and usability:

- **autofocus**
- **autocomplete**
- **required**
- **pattern**
- **placeholder**
- many other input type-specific attributes

MICHIGAN STUDENT USE PROHIBITED

Lesson 2

Validating User Input by Using HTML5 Attributes

When you have completed this lesson you will have gained an understanding of validation, why it is necessary, and how to implement validation by using HTML5 form attributes.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the principles of client-side forms validation.
- Add forms validation to ensure that mandatory fields are not left empty.
- Validate numeric input.
- Validate text input.
- Style fields to highlight input requirements.

Principles of Validation

When you create a form and add it to a website, the data it collects will vary in terms of quality and accuracy. In fact, forms are wide open to collect anything the user types in, whether it is valid or not. User input may even be malicious, designed to probe how your application reacts to unexpected data.

Validation is the process of checking the data for obvious errors, so that when it is eventually handled at the server, there are as few errors as possible.

Validation occurs in two places:

- On the client. When the user completes the form, some data can be validated by HTML markup and JavaScript code running in the browser.
- On the server. When the form is submitted, the back-end process must verify that the data is correct before processing it.

- User input can vary in accuracy, quality, and intent
- Client-side validation improves the user experience
- Server-side validation is still necessary

Although performing validation twice may seem inefficient at first, validating data in the client and on the server achieves different goals. Client-side validation is important because it significantly improves the user's experience. It can take several seconds to post a form to a web server and get the reply. This wait can be avoided if the form can detect simple errors before it is submitted. As a bonus, it also reduces the load on server resources if any obvious errors can be picked up *before* the form is submitted.

Although some data checks may be repeated on the server, this is not considered duplication. The URL that receives the data may be referenced by sources other than the form, and the server can make no assumptions about the data it receives.

Form validation is limited in its scope, and aims to pick up very simple errors. Validation on the server can be far more fine-grained than on the client browser because a server has far more resources at its disposal and has a broader context for processing the data. For example, client-side validation can ensure that a

password entered by a user conforms to a particular pattern or complexity, but only server-side validation can verify that the password is correct.

Ensuring that Fields are Not Empty

To ensure that the user enters data in a mandatory field, add the HTML5 attribute **required** to the **<input>** element. If the user attempts to submit the form before providing a value, it will not be posted to the server. The following example shows how to use the **required** attribute:

```
<input id="contactNo" name="contactNo"
      type="tel" placeholder="Enter your mobile
      number" required="required" />
```

The **required** attribute works with the input types **text, search, url, tel, email, password, number, checkbox, radio, and file**, and with the input types that pick dates where they are implemented. How the browser informs the user that a mandatory field is empty is a function of the browser; Internet Explorer highlights missing fields with a red border and a message.

- Use the **required** attribute to indicate mandatory fields

- The browser checks that they are filled in before submitting the form

```
<input id="contactNo" name="contactNo" type="tel"
      placeholder="Enter your mobile number" required="required" />
```



Validating Numeric Input

With HTML5, you can control the upper and lower limits of numeric input. The following code shows an example:

```
<input id="percentage" type="number"
      min="0" max="100" />
```

The user can type anything into the textbox, but unless it is a number between 0 and 100, the form will not pass validation, and will not be submitted. Note that if the user leaves the field blank, then no validation will occur. If it is important that the user actually enters a value, then the field should also be marked with the **required** attribute.

- Use the **min** and **max** attributes to specify the upper and lower limit for numeric data

```
<input id="percentage" type="number" min="0" max="100" />
```



 **Note:** The **number** input type also supports other attributes such as **step**, which specifies valid increments for numeric values entered in a field. These attributes are not currently implemented in Internet Explorer 10.

Validating Text Input

HTML5 provides input types such as tel and email that expect data in a specific format, but you can also apply your own custom rules to many input fields by using a regular expression. For example, if you require an order reference to comply with the pattern 99XXX, where 9 is any digit and X is any uppercase letter, use the HTML5 **pattern** attribute to specify a regular expression to validate the field. You can provide feedback to the user about the expected format of the data by using the **title** attribute.

```
<input id="orderRef" name="orderReference"
      type="text" pattern="[0-9]{2}[A-Z]{3}" title="2 digits and 3 uppercase letters" />
```

- Use the **pattern** attribute to validate text-based input by using a regular expression

`<input id="orderRef" name="orderReference" type="text" pattern="[0-9]{2}[A-Z]{3}" title="2 digits and 3 uppercase letters" />`



The **pattern** attribute can be used with the input types **text**, **search**, **url**, **tel**, **email**, and **password**.

It is easy to get too complex with regular expressions, so use **pattern** for simple pattern recognition. Expressing a comprehensive password validation rule as a single regular expression will take time and extensive testing to get right. A programmatic solution constructed by using JavaScript may produce a more efficient and maintainable solution.

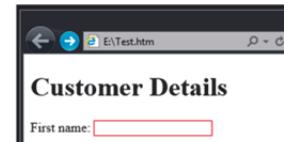
Styling Fields to Provide Feedback

To help the user identify the fields they must complete, you should indicate visually whether they are required or not required. One way to do this is by marking each required field with a text asterisk in the form design. In this example, the asterisk is styled red by using inline CSS:

```
<form id="registerForm" method="post"
      action="registration.aspx">
    <div id="firstNameField" class="field">
      <label for="firstName">First
      name:</label>
      <input id="firstName" name="firstName"
            required="required" placeholder="Your
            first name" />
      <span style="color:red">*</span>
    </div>
  ...
</form>
```

Use CSS to style input fields
Use the **valid** and **invalid** pseudo-classes to detect fields that have passed or failed validation

```
input {
  border: solid 1px;
}
input:invalid {
  border-color: #f00;
}
input.valid {
  border-color: #0f0;
}
```



A better way to indicate that a field is required is to use CSS to style the field according to the **required** attribute. You can create the following styles in a stylesheet. The first rule specifies that all input elements are surrounded by grey border, the second rule specifies that if the **required** attribute is set, the border color is red.

```
input{
  border: solid 1px #888;
}
input:required
```

```
    border-color: #f00;
}
```

This is helpful, but static. These colors will not change as the form is used. Better still would be to use color to indicate that validation has been successful. Remember that although Internet Explorer styles fields that have failed validation, this same styling is not universal and other browsers may behave differently.

To dynamically use validation to change the color of the border, creating instant feedback of successful validation in any browser, you can amend the stylesheet as shown in the following code example, which sets the border for fields containing invalid data to red, while fields with valid data have a green border. This technique works by detecting the validation state of the input control by using the **valid** and **invalid** pseudo-classes, and providing a rule for the border color for each state.

```
input{
  border: solid 1px;
}
input:invalid {
  border-color: #f00;
}
input:valid {
  border-color: #0f0;
}
```

Lesson 3

Validating User Input by Using JavaScript

HTML5 forms input types and attributes are useful for performing simple field-by-field validation of data. However, for more complex types of validation, or where you need to cross-reference fields as part of the validation process, you may need to revert to using JavaScript code. This lesson describes how you can use forms events to validate data and provide feedback by using JavaScript.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to use the **onSubmit** event of a form to perform validation and override the default validation messages implemented by the browser.
- Perform complex data validation by using regular expressions.
- Perform additional checks to verify that mandatory fields are not empty.
- Provide dynamic feedback on validation errors.

Handling Input Events

You can use event handlers to validate forms input data with JavaScript. When the user submits a form, the **submit** event is raised, and you can arrange to catch this event to validate and cross-check the data in every field as a unit prior to the form being sent to the server for processing. If the validation is successful, the event handler can return true and the form will be submitted. If the validation fails, the event handler should return false to prevent the process from continuing.

The following code example shows a form that runs the **validateForm** method when the user submits the data. The **onsubmit** attribute of the form specifies the JavaScript code to run when the **submit** event occurs:

```
<form id="registrationForm" method="post" action="registration.aspx" onsubmit="return validateForm();">
  ...
  <input type="submit" />
</form>
```

The JavaScript function **validateForm** referenced by the code in the **onsubmit** attribute should examine each field in order and validate it. If any field fails validation, the function returns false, otherwise it returns true. This approach enables you to perform more comprehensive checking of each input field, and you can also cross-check and validate fields against each other and to other data sources (for example, if the form enables a user to purchase goods, some products may not be available if the user is under 18 years old).

Each input field in a form also raises an **input** event, and you can catch this event if you only need to perform validation on a character by character basis for selected fields rather than for an entire form. This validation occurs as the user enters the data rather than when the form is submitted, but if the data is not valid you can set the **CustomValidity** flag to indicate that the form should not be submitted until the

data is corrected. For example, if the form contains an input field with an **id** of **confirm-age**, you can validate the data entered in the field like this:

```
function checkAge() {
    // Validate ageInput.value and confirm that the user has specified an age
    // in the range 18 to 120 inclusive
    var ageValid = ...;
    if (!ageValid) {
        ageInput.setCustomValidity("Age is invalid. Please specify a value between 18 and
120");
    } else {
        ageInput.setCustomValidity("");
    }
}

var ageInput = document.getElementById("confirm-age");
ageInput.addEventListener("input", checkAge, false);
```

In this example, the validation logic (*not shown*) verifies that the user enters a value between 18 and 120 in the **confirm-age** field. If this is not the case, the **setCustomValidity** function displays a custom error message and prevents the data from being submitted. If the data is valid, passing the empty string to the **setCustomValidity** function resets the error handling and the data can be submitted.

This type of validation is very fine-grained, and the **input** event runs each time the user enters a character in the input field, so do not use this approach to implement time-consuming validation that may cause the web page to slow down.

 **Note:** You can also use the **oninput** attribute of an **input** field to catch the **input** event, rather than using the **addEventListener** function.

Validating Input

You can use JavaScript code to emulate HTML5 input types and attributes that the user's browser does not support, or to perform validation that is beyond the capabilities of the HTML5 input attributes. For example, the following form defines an input field named **scoreField**:

```
<form id="scoreForm" method="post"
action="..." onsubmit="return
validateForm(); >
...
<div id="scoreField" class="field" >
    <label for="score">Score:</label>
    <input id="score" name="score"
type="number" />
</div>
...
</form>
```

- Use JavaScript code to emulate unsupported HTML5 input types and attributes in a browser:

```
<form id="scoreForm" ... onsubmit="return validateForm(); >
<div id="scoreField" class="field" >
    <input id="score" name="score" type="number" />
</div>
</form>

function isAnInteger( text ){
    var intTestRegex = /\^|\s*(\+|-)?\d+\s*$/;
    return String(text).search(intTestRegex) != -1;
}

function validateForm()
{
    if( !isAnInteger(document.getElementById('score').value))
        return false; /* No, it's not a number! Form validation fails */
    return true;
}
```

You can use the following JavaScript code to check if a value entered by the user in the **scoreField** field is a valid integer:

```
function isAnInteger( text ){

var intTestRegex = /\^|\s*(\+|-)?\d+\s*$/;
```

```
        return String(text).search(intTestRegex) != -1;
    }
    function validateForm()
    {
        if( ! isAnInteger(document.getElementById('score').value))
            return false; /* No, it's not a number! Form validation fails */
        return true;
    }
```

Ensuring that Fields are Not Empty

The HTML5 **required** attribute verifies that a user enters something into a field, but this data can be spaces, tabs, and other whitespace characters. To ensure that a field actually contains non-whitespace data, you can add JavaScript code that uses a regular expression to check the value that the user has entered. In the following example, the regular expression matches one or more groups of non-whitespace characters:

```
<form id="scoreForm" method="post"
action="..." onsubmit="return
validateForm(); >
...
<div id="penaltiesField" class="field" >
    <label for="penalties">Penalties:</label>
    <input id="penalties" name="penalties" type="text" />
</div>
...
</form>
```

Use JavaScript code to ensure that a required field does not contain only whitespace:

```
<form id="scoreForm" ... onsubmit="return validateForm(); >
<div id="penaltiesField" class="field" >
    <input id="penalties" name="penalties" type="text" />
</div>
</form>

function isSignificant( text ){
    var notWhitespaceTestRegex = /^[^\s]{1}/;
    return String(text).search(notWhitespaceTestRegex) != -1;
}

function validateForm() {
    if( ! isSignificant(document.getElementById('penalties').value))
        return false; /* No! Form validation fails */
    return true;
}
```

The following JavaScript code performs the validation:

```
function isSignificant( text ){

    var notWhitespaceTestRegex = /^[^\s]{1}/;

    return String(text).search(notWhitespaceTestRegex) != -1;
}
function validateForm() {
    if( ! isSignificant(document.getElementById('penalties').value))
        return false; /* No! Form validation fails */
    return true;
}
```



Note: The pattern "\s" in a regular expression matches any whitespace character, so the pattern "[^\s]" matches any characters that are not whitespace. The expression "{1, }" applies the preceding pattern one or more times.

Providing Feedback to the User

As when using HTML5 input types and attributes, if a form is not submitted, it is helpful to provide the user with a visual indication of the error. You can achieve this type of feedback programmatically, by using a combination of CSS and JavaScript code that dynamically changes the class of a field depending on whether the contents are valid or not valid.

The following CSS displays a different border color around an input field depending on whether it is tagged with the **validatedFine** or **validationError** class:

```
.validatedFine {  
    border-color: #0f0;  
}  
.validationError {  
    border-color: #f00;  
}
```

- Provide visual feedback to the user by defining styles and dynamically setting the class of an element:

```
.validatedFine {  
    border-color: #0f0;  
}  
.validationError {  
    border-color: #f00;  
}  
  
function validateForm() {  
    var textbox = document.getElementById("penalties");  
  
    if( ! isSignificant(textBox.value)) {  
        textbox.className = "validationError";  
        return false; /* No! Form validation fails */  
    }  
    textbox.className = "validatedFine";  
    return true;  
}
```

The JavaScript code shown below validates the **penalties** field and sets the **className** property to **validationError** or to **validatedFine** according to the outcome:

```
function validateForm() {  
    var textbox = document.getElementById("penalties");  
    if( ! isSignificant(textBox.value)) {  
        textbox.className = "validationError";  
        return false; /* No! Form validation fails */  
    }  
    textbox.className = "validatedFine";  
    return true;  
}
```



Note: Internet Explorer 10 styles fields that fail validation in a similar manner, but the technique shown makes the styling consistent for users running other browsers.

Demonstration: Creating a Form and Validating User Input

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

METHOD ONLY STUDENT USE PROHIBITED

Lab: Creating a Form and Validating User Input

Scenario

Delegates who want to attend ContosoConf will be required to register and provide their details. You have been asked to add a page to the ContosoConf website that implements an attendee registration form.

The server-side code already exists to process the attendee data. However, the registration page performs very minimal validation that is not user friendly. You have decided to add client-side validation to the form to improve the accuracy of the registration data entered by attendees and to provide a better user experience.

Objectives

After completing this lab, you will be able to:

- Create a form by using HTML5 input elements and validate form data by using HTML5 attributes.
- Implement extended data validation by using JavaScript.
- Estimated Time: 60 minutes.
- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa\$\$w0rd**

Exercise 1: Creating a Form and Validating User Input by Using HTML5 Attributes

Scenario

In this exercise, you will create an HTML form that collects conference attendee registration information.

You will select the correct input types for each piece of data collected by the form. Then you will enhance the input with additional attributes to improve the user experience and to add validation. For example, the first input item should automatically receive the focus when a page loads. Also, most of the input items are mandatory, the password must be sufficiently complex to improve security, and the form must prevent incomplete data or data that is not valid from being submitted. Finally, you will run the application, view the Register page, and verify that form validation performs correctly.

The main tasks for this exercise are as follows:

1. Modify the Register page.
2. Add form inputs to the Register page.
3. Make the form more user friendly.
4. Check for missing mandatory data.
5. Add password complexity validation.

► Task 1: Modify the Register page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod04\Labfiles\Starter\Exercise 1** folder.

4. Open the **register.htm** file.
5. Find the **<form>** element and read the TODO comment that describes the form input requirements together with the HTML snippet to use as a template:

```
<form method="post" action="/registration/new" id="registration-form">
    <!--
        TODO: Add form inputs
        FirstName - required string
        LastName - required string
        EmailAddress - required email address
        Password - required password string, at least 5 letters and numbers
        ConfirmPassword
        WebsiteUrl - optional url string
    -->
    <!-- Use the following template for the inputs -->
    <div class="field">
        <label for="{input-id}">label:</label>
        <input type="{type}" id="{input-id}" name="{input-name}" />
    </div>
    <div>
        <button type="submit">Register</button>
    </div>
</form>
```

6. Notice that **register.htm** has a reference to the style sheet **/styles/pages/register.css** in the **<head>** element **/scripts/pages/register.js** just before the **</body>** tag:

```
<link href="/styles/pages/register.css" rel="stylesheet" type="text/css" />
```

This CSS file contains the styles for the registration page.

7. Also notice that the **register.htm** file contains a reference to the **/scripts/pages/register.js** JavaScript file that contains the code used by the registration page:

```
<script src="/scripts/pages/register.js" type="text/javascript"></script>
```

► Task 2: Add form inputs to the Register page

1. Add the input elements specified by the TODO comment to the registration form:
 - Do not include any validation yet.
 - Use the most appropriate HTML5 input types.
 - Use the **<div class="field">** template when creating the form inputs; provide a label for each input, and remove this template when you have added all of the inputs.
2. Run the application, view the **register.htm** page, and test the form with some valid data. The following image shows some suggested valid data values that you can use (use the text **Passw0rd** for the password).

The Register page should look like this:

The screenshot shows a web browser window with the URL <http://localhost:55981/register.htm>. The page title is "ContosoConf" and the subtitle is "A two-track conference on the latest HTML5 developments". The main content is titled "Register for the conference". It contains several input fields: "First name" (Josh), "Last name" (Bailey), "Email address" (josh.bailey@adatum.com), "Choose a password" (*****), "Confirm your password" (*****), and "Website/blog" (http://adatum.com). Below the form is a "Register" button and footer text "Hosted by Contoso" and "Conference Center".

FIGURE 4.1:THE REGISTER PAGE

3. After you have entered a complete set of valid data, click **Register**. Verify that the **Thanks for registering page** appears.
4. Return to the **Register** page. Notice that you have to explicitly click to put the cursor in the First name field to start registering another attendee.

Also, notice that you can leave fields blank, or enter mismatching passwords.

5. Click **Register** again.

The data is validated by the registration server, which generates the following page:

The screenshot shows a web browser window with the URL <http://localhost:55981/registration/new>. The page title is "Contoso Conf". The main content is titled "Invalid registration information" and lists validation errors:

- The FirstName field is required.
- The LastName field is required.
- The EmailAddress field is required.
- The Password field is required.
- The ConfirmPassword field is required.

FIGURE 4.2:THE VALIDATION ERRORS PAGE GENERATED BY THE SERVER

It would be more efficient to trap these issues before the data is transmitted to the server.

6. Close Internet Explorer.

► **Task 3: Make the form more user friendly**

1. In the **register.htm** file, modify the **FirstName** input so that it automatically gets the focus when the browser loads the register page.
 - o Use the **autofocus** attribute.
2. Modify the **Website** input to display the placeholder text **http://** when the page is displayed.
 - o Use the **placeholder** attribute.
3. Run the application, view the **register.htm** page, verify that the **First name** box has focus and that the **Website** box contains the placeholder text.
4. Close Internet Explorer.

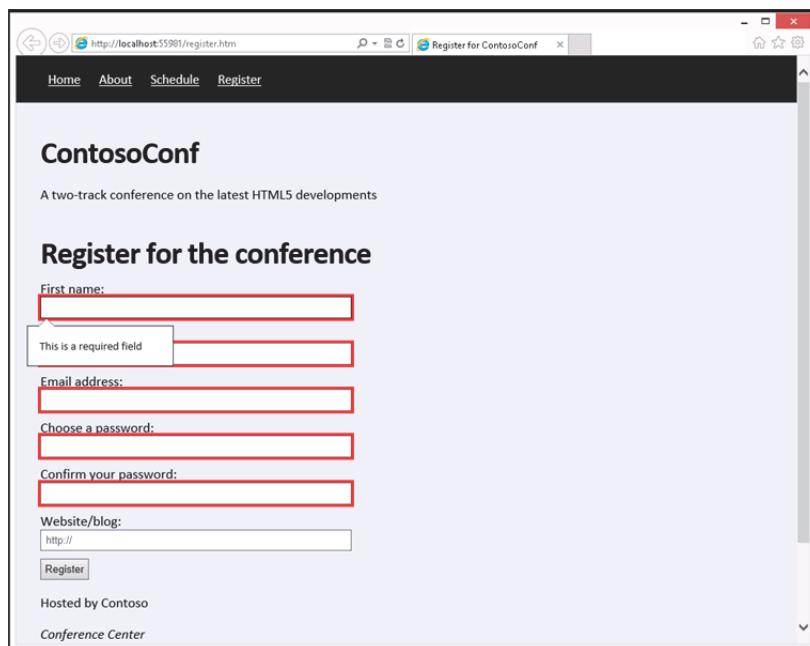
► **Task 4: Check for missing mandatory data**

1. In the **register.htm** file, add **required** attributes to the mandatory form inputs (**FirstName**, **LastName**, **EmailAddress**, **Password**, **ConfirmPassword**).
 - o Use the **required** attribute.

 **Note:** The password validation performed by the registration server requires that password contain letters and numbers only. Punctuation and other characters are not allowed.

2. Run the application, view the **register.htm** page, and click **Register** without entering any data.
3. Try entering different combinations of fields, and verify that the form cannot be submitted if any mandatory field is empty.

The Register page should highlight each missing data item, like this:



The screenshot shows a web browser window for 'ContosoConf' with the URL 'http://localhost:55981/register.htm'. The page title is 'Register for ContosoConf'. At the top, there's a navigation bar with links for Home, About, Schedule, and Register. Below the navigation, the page title 'ContosoConf' and subtitle 'A two-track conference on the latest HTML5 developments' are displayed. The main content is titled 'Register for the conference'. It contains several input fields, each with a red border indicating they are required and have not been filled. The fields are: 'First name:' (empty), 'Email address:' (empty), 'Choose a password:' (empty), 'Confirm your password:' (empty), and 'Website/blog:' (containing 'http://'). Below these fields is a 'Register' button. At the bottom of the page, it says 'Hosted by Contoso' and 'Conference Center'.

FIGURE 4.3:THE REGISTER PAGE HIGHLIGHTING MISSING FIELDS

4. Enter the complete details for an attendee and then click **Register**. Verify that the form still enables the user to submit valid data.

► **Task 5: Add password complexity validation**

1. In the **register.htm** file, modify the **Password** input to ensure that the entered value is at least five characters long, and consists of only letters and numbers.

- Use **pattern** attribute with the following regular expression:

```
[a-zA-Z0-9]{5,}
```

- If the **Password** input doesn't match the regular expression, display the following message by using the **title** attribute:

```
At least 5 letters and numbers
```

2. Run the application, view the **register.htm** page, register as an attendee, and verify that **Password** input displays the error message for password entries that are not valid.
 - Try a short password, such as **abc**.
3. Verify that valid passwords containing at least five letters and numbers are still accepted.
 - Try a longer password such as **password** (this should work).
 - Also, try a password with at least one numeric character, such as **Passw0rd** (this should also work).

Results: After completing this exercise, you will have modified the attendee registration page to validate the data entered by attendees.

Exercise 2: Validating User Input by Using JavaScript

Scenario

The conference registration form requires that the Password and Confirm Password inputs match. You cannot implement this type of validation by using HTML5 attributes. In this exercise, you will extend the registration form validation by using JavaScript. In addition, you will write code to style any input that is not valid to attract the user's attention.

You will implement a function to compare the two passwords and display an error message when the passwords do not match. Then you will add input event listeners for the password inputs, which call the password comparison function. You will test this feature to ensure that a user cannot submit a form with passwords that do not match.

Next, you will add a CSS style to highlight input elements that are not valid (some browsers such as Internet Explorer already highlight them with a red border, but other browsers might not implement this feature by default). You will run the application, view the Register page, and verify that elements that are not valid are highlighted.

The main tasks for this exercise are as follows:

1. Write code to get the contents of the password input elements.
2. Write code to compare the password and confirm-password inputs.
3. Write code to display a custom error message if the passwords differ.
4. Add input event listeners to the inputs to call the checkPasswords method.

5. Style elements that are not valid.

► **Task 1: Write code to get the contents of the password input elements**

1. In Visual Studio, open the **ContosoConf.sln** solution in the **E:\Mod04\Labfiles\Starter\Exercise 2** folder. This project contains a working version of the application as it should appear at the end of exercise 1, together with additional comments and code fragments that are used by this exercise.
2. In the **ContosoConf** project, open the **scripts\pages\register.js** file and find the following comment:

```
// TODO: Task 1 - Get the password <input> elements from the DOM by ID
```

3. Create variables named **passwordInput** and **confirmPasswordInput** that contain references to the password fields on the form.
 - o Use the **getElementById** function.
 - o The **id** attributes of the password **<input>** elements are **password** and **confirm-password**.

► **Task 2: Write code to compare the password and confirm-password inputs**

1. The **register.js** file contains a function called **checkPasswords** that will examine whether the password and confirm-password inputs contain the same text. This function is currently empty (apart from some comments).

In the **scripts\pages\register.js** file find the following comment:

```
// TODO: Task 2 - Compare passwordInput value to confirmPasswordInput value
```

2. Add a statement that tests whether the two password inputs have the same value and store the Boolean result in a variable named **passwordsMatch**.
 - o Use the **value** property of the **passwordInput** and **confirmPasswordInput** variables to read the data that the user has entered into the password fields on the form.
 - o Compare the value by using the **==** operator.

► **Task 3: Write code to display a custom error message if the passwords differ**

1. In the **comparePasswords** function, find the comment that starts with the following text:

```
// TODO: Task 3
```

2. Add code that uses the **setCustomValidity** method of the **confirmPasswordInput** variable to display an error message when the **passwordsMatch** variable indicates that the passwords do not match.

If the passwords do match, clear the error message.

 **Note:** Setting a non-empty custom validity message will prevent the form from being submitted.

► **Task 4: Add input event listeners to the inputs to call the checkPasswords method**

1. The password input elements raise an event named **input** when text is entered.

In the **register.js** file, find the comment that starts with the following text:

```
// TODO: Task 4
```

2. In the **addPasswordInputEventListeners** function, add event listeners for this event to call the **checkPasswords** function.
 - Use the **addEventListener** function.
3. Run the application, view the **register.htm** page, enter valid data for the **First name**, **Last name**, and **Email address** fields, and verify that an error message is displayed when the data in the **Confirm Password** box does not match the data in the **Password** box.
4. Verify that the message does not appear if the passwords are the same.
5. Close Internet Explorer.

► Task 5: Style elements that are not valid

1. In the **ContosoConf** project, open the **styles\pages\register.css** file.
2. At end of the file, find the comment that starts with the following text:

```
/* TODO: Task 5
```

3. Add a CSS rule that changes the background color of input elements that are not valid to **#f9b2b2**.
 - The form has the class **register**, so only apply this styling to inputs that occur in this form (use the **.register** selector).
 - Additionally, ensure that the CSS rule only applies when the **<form>** element has the **submission-attempted** class (concatenate **form.submission-attempted** to the selector).



Note: The extra CSS class is added by the **formSubmissionAttempted** function, which is called when the **Register** button is clicked, as shown in the following code from the **register.js** file:

```
var formSubmissionAttempted = function() {  
    form.classList.add("submission-attempted");  
};  
var addSubmitClickEventListener = function() {  
    submitButton.addEventListener("click", formSubmissionAttempted, false);  
};
```

Initially, the required form inputs are empty and therefore not valid. However, the application should allow the user to complete the form before showing error messages.

- Finally, the styling should only be applied for the input is invalid, so concatenate the pseudo-class **input:invalid** to the end of the selector.
 - Note that Internet Explorer automatically adds a red outline to inputs that are not valid. Remove this default styling by setting the **outline** CSS property to **none**.
4. Run the application, view the **register.htm** page, click the **Register** button without entering any registration details, and verify that inputs that are not valid are highlighted with colored backgrounds.

The Register page should look like this if you attempt to register without providing any details:

MCT USE ONLY. STUDENT USE PROHIBITED

The screenshot shows a registration form for 'ContosoConf' with several fields highlighted in red, indicating they are invalid. The fields are:

- First name: A red input field with the placeholder 'First name:'.
- Email address: A red input field with the placeholder 'Email address:'.
- Choose a password: A red input field with the placeholder 'Choose a password:'.
- Confirm your password: A red input field with the placeholder 'Confirm your password:'.
- Website/blog: A red input field with the placeholder 'Website/blog:' containing 'http://'.
Below this field is a small text area with the placeholder 'Conference Center'.

At the bottom of the form is a 'Register' button.

FIGURE 4.1:REGISTRATION FORM WITH INPUTS THAT ARE NOT VALID HIGHLIGHTED

5. Close Internet Explorer.

Results: After completing this exercise, you will have modified the registration page to validate password inputs.

Module Review and Takeaways

In this module, you have learned how to create HTML5 forms that enable a user to enter data and to send it to a server for processing. You have seen how to use the HTML5 input types and attributes to specify the type of each input field and to validate the data entered by the user in the browser before it is submitted. You have also seen how to provide feedback to the user by styling input that is not valid and displaying meaningful error messages.

In addition, you have learned how to use JavaScript code to implement more complex client-side forms validation and styling that runs in the user's browser.

Review Question(s)

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
If you define a field with an HTML5 input type that is not supported by the user's browser, the field does not appear on the form when the browser displays it. True or False?	

Question: If you perform validation in the browser, is it necessary to perform the same validation checks in the server code that processes the data, or is this additional processing redundant?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
You should always use the input event to validate data that a user enters into a field, in preference to the submit event of the form. True or False?	

Module 5

Communicating with a Remote Server

Contents:

Module Overview	5-1
Lesson 1: Sending and Receiving Data by Using the XMLHttpRequest Object	5-2
Lesson 2: Sending and Receiving Data by Using the jQuery Library	5-8
Lab: Communicating with a Remote Data Source	5-12
Module Review and Takeaways	5-18

Module Overview

Many web applications require the use of data held by a remote site. In some cases, you can access this data simply by downloading it from a specified URL, but in other cases the data is encapsulated by the remote site and made accessible through a web service. In this module, you will learn how to access a web service by using JavaScript code and to incorporate remote data into your web applications. You will look at two technologies for achieving this: the **XMLHttpRequest** object, which acts as a programmatic wrapper around HTTP requests to remote web sites, and the **jQuery** library, which simplifies many of the tasks involved in sending requests and receiving data.

Objectives

After completing this module, you will be able to:

- Send data to a web service and receive data from a web service by using an **XMLHttpRequest** object.
- Send data to a web service and receive data from a web service by using the **jQuery** library.

Lesson 1

Sending and Receiving Data by Using the XMLHttpRequest Object

A web browser interacts with a web application by sending and receiving HTTP requests. Similarly, a web page can make requests to access additional remote resources, and this is typically performed by using HTTP requests. The HTTP network protocol is relatively simple, but HTML5 incorporates the **XMLHttpRequest** object that provides a programmatic interface that a JavaScript application can use to send and receive HTTP requests.

In this lesson, you will learn how **HTTP web methods** are used by the web browser to access external resources from a web page. Then you will learn how to use the **XMLHttpRequest** object to load data from a remote website. Finally, you will learn how to use the **XMLHttpRequest** object to send requests to a web service and to process any response that is received.

Lesson Objectives

After completing this lesson, you will be able to:

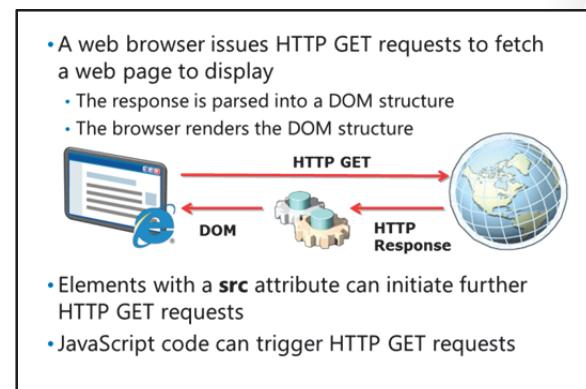
- Explain how a browser uses **HTTP GET** requests to retrieve remote data.
- Explain how to use the **XMLHttpRequest** object to send a request to a remote server.
- Describe how to **handle errors** that can occur when sending requests and receiving responses.
- Handle the **data returned** by a server in response to a request.
- Process the received data **asynchronously**.
- **Send messages** that transmit data to a server.

How a Browser Retrieves Web Pages

When you type a URL in the address bar of your web browser, it initiates an HTTP GET operation to retrieve the HTML structure for the web page from the web server. The web browser receives this structure and parses it to create the document object model (DOM), which it then renders. The appropriate CSS rules for the web page will be applied, and any JavaScript code linked to the page will be run. At this point, the page is ready for the user to view.

While the browser is processing the DOM structure for a page, if it encounters elements with a **src** attribute that specify the URL of an external resource (such as a file), it initiates further HTTP GET operations to download each of these resources in turn, and then loads them as child objects in the DOM. This process can take some time, especially if the resources are references to large files. Many developers use JavaScript code to set the **src** attribute of an element dynamically, after the page has loaded, enabling individual resources to be loaded on demand while the page is running. This operation can also trigger an HTTP GET operation.

Although this mechanism could be used to retrieve data on demand, there are relatively few HTML elements that provide a **src** attribute, such as **<image>**, **<iframe>**, **<script>**, **<video>**, and **<audio>**,



and they are intended to import, respectively, graphics, HTML fragments, JavaScript code, video media, and audio content. None of these elements is suited to retrieving general-purpose data from a web service.

Using the XMLHttpRequest Object to Access Remote Data

To handle more generic requests, HTML5 provides the **XMLHttpRequest** object. This object is specifically designed to load data into the DOM on demand. Not only can it trigger HTTP **GET** operations, it can also invoke **POST** and **HEAD** requests. The **XMLHttpRequest** object can return text, JSON, and XML data, and tracks the status of HTTP operations so that you can perform any appropriate error checking. Additionally, it can be used synchronously or asynchronously. It is supported in all major browsers, so the **XMLHttpRequest** object is ideally suited for retrieving data by using HTTP methods from within a web page.

- To send an HTTP request:
 1. Create a new **XMLHttpRequest** object
 2. Specify the HTTP method and URL
 3. Set the request header
 4. Send the request

```
var request = new XMLHttpRequest();
var url = "http://contoso.com/resources/...";
request.open("GET", url);
request.send();
```

- Requests are asynchronous by default
 - To block and wait for a response:
`request.open("GET", url, false);`

The **XMLHttpRequest** object provides a means to access any type of remote data; you are not restricted to data for elements that provide a **src** attribute. A common use is to call a web service, passing it data entered by the user on a form, and then processing the response that is returned.

In browsers that support HTML5, you use the following JavaScript code to create an **XMLHttpRequest** object:

```
var request = new XMLHttpRequest();
```

You can then use this object to initiate a remote request. To do this, you set properties that specify the URL of the remote data, and that specify the type of data in the request; this is to enable the server receiving the request to correctly parse the data that it receives. The following code shows an example:

```
var url = "http://contoso.com/resources/...";
request.open( "GET", url );
```

 **Note:** Most modern browsers implement a "same origin" policy for JavaScript code that attempts to connect to a web service. This policy only allows the code in a JavaScript web page to retrieve data from a web service that is hosted at the same site as the web page. This is a security measure that is intended to prevent JavaScript code that has been injected into a web page from sending or receiving information from a potentially malicious third-party web site.

The **open()** method defines the HTTP method and the url for this request. The HTTP method can be set to either **GET**, **POST**, or **HEAD**. You use **GET** to request a known resource without parameters, while you use **POST** to send parameterized data to the server, most commonly form data, to be parsed as part of the request. **HEAD** specifies that the response should only be header information, and can be useful, for example, to get a summary of a large resource ahead of actually downloading it.

To transmit the request, call the **send()** method. Note that the browser automatically handles the sending of any cookies associated with the domain, based on the value of the URL.

Calls to the **send()** method are **asynchronous** by default, meaning that your JavaScript code will not block to wait for a response. If you require a response before continuing, you can make the current thread of

execution stop and wait for a response. To do this, use the optional third argument of the **open()** method as follows:

```
request.open(method, url, false);
```

Handling HTTP Errors

You should check the outcome of the **send()** function to verify that the request has actually been sent. To do this, examine the **status** property of the request object. This property contains the HTTP status code of the send operation. A status code of 200 indicates success; other codes indicate some kind of error or warning. For example, a status code of 404 indicates that the resource being requested could not be found. The **statusText** property provides more information in the form of a text message.

```
function tryMyLuck() {
    var request = new XMLHttpRequest();
    request.open("GET", "/luckydip/enter");
    request.send();

    // wait for the request status to be returned
    ...

    if (request.status != 200) {
        alert("Error " + request.status + " - " + request.statusText);
    }
    ...
}
```

- Check the status code of the **XMLHttpRequest** object to verify that the request has been sent:

```
var request = new XMLHttpRequest();
request.open("GET", "/luckydip/enter");
request.send();

...
if (request.status != 200) {
    alert("Error " + request.status + " - " + request.statusText);
}
```

- Wrap your code in a **try...catch** block to handle any unexpected network errors



Note: You should ensure that the request status has been returned before you check it in your code. If you are sending a request asynchronously, you should check the request status by using a **readystatechange** event handler, as described later in this lesson in the topic "Handling an Asynchronous Response".



Additional Reading: For more information about the different HTTP status codes that can occur, and their meaning, see <http://go.microsoft.com/fwlink/?LinkId=267722>.

You should wrap your code in a JavaScript **try...catch** block to handle any other unexpected exceptions caused by network or communications failure in the browser; networks are notoriously unreliable and failures can occur at any time.



Note: The **try...catch** construct in JavaScript is a general-purpose mechanism that enables your code to catch and handle unexpected exceptions that occur when your JavaScript code runs. You can wrap a block of JavaScript code in a **try** statement and catch any errors that might occur while this block of code is running, like this:

```
try {
```

```

/* JavaScript code that might trigger an exception */
...
} catch (exception) {
    /* Handle any exceptions here. This example displays the exception to the user */
    alert(exception);
}

```

The **catch** block looks a little like a function definition; it takes a parameter that is populated with the details of the exception that has occurred. The actual contents of this parameter will depend on the specific error that triggered the exception.

Consuming the Response

You can retrieve the data returned by the server in response to the request by examining the **responseText** property of the request object.

Just as the body parameter passed to the server in the **send()** method can be any text, so the response from the server supplied in the **responseText** property can also be any valid text, whether formatted or not. This means that the server can return the data in HTML, JSON, XML, CSV, query string format, or just plain text. You can use the **getResponseHeader()** function of the request object to ascertain the type of data in the response. The type of the data returned is specified in the **Content-Type** header in the response ("text/html", "application/json", "text/xml", and so on). If the response is XML data, you can use the **responseXML** property to retrieve the well-formed XML response. The following code shows an example:

```

function getResponse(request) {
    var type = request.getResponseHeader("Content-Type");
    switch (type) {
        case "text/xml" :
            return request.responseXML;
        default :
            return request.responseText;
    }
}

```

- Determine the type of data in the response
- Read the response data from the **responseText** property

```

var request = new XMLHttpRequest();
...
var type = request.getResponseHeader();
switch(type) {
    case "text/xml":
        return request.responseXML;
    case "text/json":
        return JSON.parse(request.responseText);
    default:
        return request.responseText;
}

```

At first glance, it would seem that XML is the ideal format for data transfer, but remember that XML is verbose, and that the **XMLHttpRequest** object has to parse the data to make sure that it is valid before making it available in the **responseXML** property. This makes using XML slower than using more concise plain text responses.

Many servers return data in JSON format. For example, a server returning information about musical composers might pass back the following string:

```

{
    "surname": "Bach",
    "role": "composer",
    "firstNames": [
        "Johann", "Sebastian"
    ]
}

```

Remember that JSON is a serialized view of an object. To deserialize this data, you can use the **JSON.parse()** function, like this:

```
function getResponse(request) {  
    var type = request.getResponseHeader();  
    switch( type ) {  
        case "text/xml" :  
            return request.responseXML;  
        case "application/json" :  
            return JSON.parse(request.responseText);  
        default :  
            return request.responseText;  
    }  
}
```

If the **responseText** property does not contain valid JSON data, then **JSON.parse()** may throw an exception.

Handling an Asynchronous Response

If you are performing an asynchronous query (the default case), you should wait for the response to be available before attempting to read the **responseText** property. The **XMLHttpRequest** object provides the **readystatechange** event that you can use to detect whether data has been returned. This event occurs whenever the state of the **XMLHttpRequest** object changes. The **XMLHttpRequest** object has a **readyState** property that can have one of the following values:

- 0: The **XMLHttpRequest** object is has not been opened.
- 1: The **XMLHttpRequest** object has been opened.
- 2: The **XMLHttpRequest** object has sent a request.
- 3: The **XMLHttpRequest** object has started to receive a response.
- 4: The **XMLHttpRequest** object has finished receiving a response.

- Create an event handler for the **readystatechange** event
- Check that the **readyState** of the **XMLHttpRequest** object is set to 4

```
request.onreadystatechange = function () {  
    if (request.readyState === 4) {  
        var response = JSON.parse(request.responseText);  
        ...  
    }  
};
```

To receive data asynchronously, you can create a handler for the **readystatechange** event, examine the **readyState** property, and if a response has been received, read the **responseText** property of the request. The following code shows an example that uses the **onreadystatechange** callback to catch the **readystatechange** event.

```
request.onreadystatechange = function() {  
    if (request.readyState === 4) {  
        var response = JSON.parse(request.responseText);  
        ...  
    }  
};
```

Transmitting Data with a Request

HTTP **GET** requests return data to the browser from a server. You can also use HTTP **POST** requests to send data from the browser to a server. To transmit a request that includes data, specify the **POST** method rather than **GET** as the parameter to the **open()** function. Call the **send()** function as before, but the data to be sent to the server must be provided as the optional **body** parameter to the **send()** function. The body can be any text string that the server web method can parse.

- To send data to a server:
 1. Serialize the data
 2. Set the **Content-Type** property of the request header
 3. Transmit the data by using the HTTP **POST** method

```
var data = JSON.stringify( );
var request = new XMLHttpRequest();
var url = ...;
request.open("POST", url);
request.setRequestHeader("Content-Type", "text/plain");
request.send(data);
```

 **Note:** If the HTTP method is **GET**, the **body** can contain parameters that help to identify the data that the server should return. If there are no parameters, then **body** can be set to null.

```
request.open("POST", url);
request.send(body);
```

 **Note:** The actual format of the data expected by the server will depend on the server, but a common format is JSON, as described in module 3. You can serialize an object into JSON format by using the **JSON.stringify()** function.

You should also specify the format of the data to ensure that it is interpreted correctly by the server. You do this by specifying the format in the **Content-Type** property of the request header. For example, if you are sending data formatted by using **JSON.stringify()**, you can set the request header as follows:

```
request.setRequestHeader("Content-Type", "application/json");
```

As another example, if you are sending data entered by a user on an HTML5 form, you should specify that this data is encoded as form input, as follows:

```
request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Lesson 2

Sending and Receiving Data by Using the jQuery Library

The jQuery **ajax()** function provides a high-level wrapper around the **XMLHttpRequest** object that enables you to send requests and receive responses in a more concise and intuitive manner. In this lesson, you will learn how to use the jQuery library to send requests and handle the responses, and about the possible errors that can occur.

Lesson Objectives

After completing this lesson, you will be able to:

- Simplify your JavaScript code by using the jQuery library.
- Serialize data by using jQuery helper methods.

Using the jQuery Library to Send Asynchronous Requests

The jQuery library provides several functions that simplify the JavaScript needed to communicate asynchronously with the server. Behind-the-scenes functionality is based around the **XMLHttpRequest** object, so the concepts should be familiar.

 **Note:** The operations performed by using the jQuery library are inherently asynchronous.

To send an HTTP **GET** request to a server, use the jQuery **get()** function:

```
$ .get(url, body, callback);
```

The parameters are the URL to which the request is sent, the data for the body of the request (this is optional), and a callback function that runs when the response is received. The data returned in response to the request is passed as the first parameter to the callback. The following example shows how to send a request and handle the response by using the jQuery **get()** function:

```
var response;
$.get('http://contoso.com/resources/...', function(data) {
    response = data;
});
```

Note that this example uses an anonymous function to handle the response.

 **Note:** The jQuery library also includes the **post()** function, which operates in a similar manner to the **get()** function. The main difference is that it sends the request by using the HTTP **POST** method.

By default, if an error occurs during a call to the **get()** function, **the function fails silently**. To handle errors, you can chain a call to the jQuery **error()** function to the end of a request. This function takes a callback

as a parameter, and runs the callback if an error occurs while making the request. The following example shows a very simply error handler:

```
var response;
$.get('http://contoso.com/resources/...', function(data) {
    response = data;
}).error(function() {
    alert("error occurred during get operation");
});
```

To send a **GET** request that returns JSON data, use the **\$.getJSON()** method:

```
$.getJSON(url, body, callback);
```

The parameters are the same as those of the **get()** function. However, the data is passed to the callback in JSON format rather than as text.

To load a response directly into an element on a web page by using the jQuery library, you can use the **load()** function, as follows.

```
$('#container').load(url, body, callback);
```

In this example, the element with the **id** value of **container** is populated with the response received from the server at the specified URL. This is useful when the response is HTML or plain text. As with the **get()** function, the **body** parameter is optional; a null value triggers a **GET** operation, and a non-null value triggers a **POST** operation. The **callback** is also optional.

Using the jQuery **ajax()** Function

For really fine-grained control of asynchronous interactions with the web server, jQuery has a function called **\$.ajax()** that offers a comprehensive set of properties and events designed to give you full control over the request and its outcome

 **Note:** The **ajax()** function name comes from the acronym AJAX, which stands for Asynchronous JavaScript and XML; an AJAX interaction with a server is an asynchronous exchange of messages based on the **XMLHttpRequest** object.

- The jQuery **ajax()** function provides additional properties and finer control over HTTP requests

```
$.ajax({
    url: '/luckydip/enter',
    type: 'GET',
    timeout: 12000,
    dataType: 'text'
}).done(function(responseText) {
    $('#answer').text(responseText);
}).fail(function() {
    alert('An error has occurred – you may not have been entered');
});
```

The **ajax()** function creates an **XMLHttpRequest** object, sends requests, and receives asynchronous responses, but it hides from you much of the complexity for doing this. The **ajax()** function takes as its parameter an object that contains a set of properties. Each of these properties enables you to specify the various elements that constitute a request. You can chain on functions that indicate what happens when the request succeeds (**done()**) or fails (**fail()**). The **done()** and **fail()** functions expect you to provide a callback that runs when the request succeeds or fails. The following code shows an example:

```
$.ajax({
    url: '/luckydip/enter',
    type: 'GET',
    timeout: 12000,
    dataType: 'text'
```

```
}).done(function( responseText ){
    $('#answer').text( responseText );
}).fail(function() {
    alert('An error has occurred - you may not have been entered');
});
```

In this example, the **url** property is the URL of the web service method or remote resource. The **type** property can be set to **GET** or **POST**. The **timeout** property causes the call to abort after a time specified in milliseconds. The **dataType** property governs the response type you are expecting from the server, and can be set to a variety of values, including **text**, **xml**, **html**, **json**, and **script**. The **done** function is called when the call returns and the response data is parsed correctly. The **fail** function is called if the request fails for any reason.

In the example above, the handler sets an element with the ID **answer** to the text response from the server at the URL **/luckydip/enter**.



Additional Reading: The **\$.ajax()** function provides a comprehensive set of properties. For a complete description, visit <http://go.microsoft.com/fwlink/?LinkId=267723>.

Serializing Forms Data by Using jQuery

To send forms data to the server by using the **\$.ajax()** function, use the **POST** method and add a JSON-formatted **data** property containing the data to be sent to the parameter list.

```
$.ajax({
    url: '/luckydip/enterWithName',
    type: 'POST',
    timeout: 12000,
    dataType: 'text',
    data: {
        firstName: myForm.fname.value,
        lastName: myForm.lname.value
    }
}).done(...)
.fail(...)
};
```

- To include forms data in a request, use the **data** property:

```
$.ajax({
    url: '/luckydip/enterWithName',
    type: 'POST',
    timeout: 12000,
    dataType: 'text',
    data: {
        firstName: myForm.fname.value,
        lastName: myForm.lname.value
    }
});
```

- To retrieve input data directly from a form, use the **serializeArray()** function

Coding each field on a form in this way gives you precise control over which fields are sent and what the properties of the object are called.

You can also use the jQuery **serialize()** and **serializeArray()** methods, which will take the fields on a form and serialize the entire form in one statement. When using these methods, you do not get to rename the fields, so they will be named just as they are on the form.

The output from the methods is different. The **serialize()** function generates a list of fields and values formatted as a query string. The **serializeArray()** function generates a JSON-formatted array of field names and values. You can use the **serializeArray()** function like this:

```
$.ajax({
    url: '/luckydip/enterWithName',
    type: 'POST',
    timeout: 12000,
    dataType: 'text',
    data: {
        '#myForm').serializeArray();
    }
});
```

```
}).done(...  
).fail(...  
);
```

A data property set in this way produces a JSON array similar to this:

```
[  
  { name: "firstName", value: "Rachel" },  
  { name: "lastName", value: "Lopez" },  
]
```

Care should be taken when using these methods because they will not include fields that are hidden or that have a CSS **display** attribute set to **none**. Note that there is no submit button value in the array. This is because, from a technically accurate viewpoint, the form was not submitted by using the usual form submission mechanism.

 **Additional Reading:** For further discussion and a reference visit
<http://go.microsoft.com/fwlink/?LinkId=267724> and
<http://go.microsoft.com/fwlink/?LinkId=267725>.

Demonstration: Communicating with a Remote Data Source

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Communicating with a Remote Data Source

Scenario

You have been asked to modify the Schedule page for the ContosoConf website. Previously, the session data was provided as a hard-coded array of data and the JavaScript code for the page displayed the data from this array. However, session information is not static; it may be updated at any time by the conference organizers and stored in a database. A web service is available that can retrieve the data from this database, and you decide to update the code for the Schedule page to use this web service rather than the hard-coded data currently embedded in the application.

In addition, the conference organizers have asked if it is possible for conference attendees to be able to indicate which sessions they would like to attend. This will enable the conference organizers to schedule popular sessions in larger rooms. The Schedule page has been enhanced to display star icons next to each session. An attendee can click a star icon to register their interest in that session. This information must be recorded in a database on the server, and you send this information to another web service that updates the corresponding data in the database.

A session may be very popular, so the web service will return the number of attendees who have selected it. You will need to handle this response and display a message to the attendee when they have selected a potentially busy session.

Objectives

After completing this lab, you will be able to:

1. Write JavaScript code to retrieve data from a remote data source.
 2. Write JavaScript code to send data to a remote data source.
 3. Use the jQuery **ajax** method to simplify code that performs remote communications.
- Estimated Time: 60 minutes.
 - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
 - User Name: **Student**
 - Password: **Pa\$\$w0rd**

Exercise 1: Retrieving Data

Scenario

In this exercise, you will retrieve and display the list of sessions from a web service.

First, you will create a function that constructs an HTTP request to get session data from a remote data source running on a web server. The function will send the request asynchronously, and you will define a callback function that receives the session data when the web service replies. The session data will be a JSON string that requires parsing into a JavaScript object. You will use the existing **displaySchedule** function to display the sessions on the page.

Network connections to remote sources and web servers are not totally reliable. Therefore, you will need to make your code robust enough to handle errors that can occur when receiving data. For testing purposes, a version of the web service that generates errors is also available, and you will use this web service to verify the error handling capabilities of your code.

Finally, you will run the application and view the Schedule page to verify that it displays the list of sessions correctly, and also that it correctly handles errors.

The main tasks for this exercise are as follows:

1. Review the Schedule page.
2. Create the downloadSchedule function.
3. Add error handling to the downloadSchedule function.
4. Test the Schedule page.

► **Task 1: Review the Schedule page**

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod05\Labfiles\Starter\Exercise 1** folder.
4. In the **ContosoConf** project, open the **scripts\pages\schedule.js** file.
5. Review the JavaScript code.
 - Notice that the previously hard-coded array of session data (in the **schedule** variable) has been replaced with an empty array.
 - Also notice that the **createSessionElement** function has been modified to generate a star icon next to the session title.

 **Note:** The star icon is an `<a>` link element that is styled with a background image in the shape of a star. To do this, the `<a>` element has the **class** attribute set to **star**, and the **schedule.css** file in the **styles\pages** folder contains the following style:

```
.star {  
    display: inline-block;  
    width: 15px;  
    height: 15px;  
    cursor: pointer;  
    background-image: url(..\images\stars.png);  
    background-position: 0 0;  
}
```

► **Task 2: Create the downloadSchedule function**

1. In the **schedule.js** file, find the comment that starts with the following text:

```
// TODO: Create a function called downloadSchedule
```
2. Create an empty function named **downloadSchedule**. You will add code to this function in subsequent tasks to asynchronously get a list of sessions from a web service.
3. Within this function define a variable named **request** and assign it a new **XMLHttpRequest** object.
4. In the **downloadSchedule** function, use the **request** object to asynchronously retrieve data from the URL **/schedule/list** by performing a GET operation.
 - Call the **open()** method of the **request** object.
5. After the previous statement, create an empty callback that handles the response from the web service and attach it to the **onreadystatechange** property of the **request** object.

6. In the callback, perform the following operations:
 - o Ensure that the **readyState** property of the **request** object indicates the response is complete (it should have the value **4**).
 - o Parse the JSON string in the response into an object and assign this object to the **schedule** variable.
 - o Call the **displaySchedule** function to display the sessions on the page.
7. After you have created the callback, add a statement to send the request to the server:
 - o Use the **send()** method of the **request** object.
8. Add a statement to the end of the **schedule.js** file that calls the **downloadSchedule** function after initializing the event handlers for the check boxes and list on the Schedule page.

► **Task 3: Add error handling to the downloadSchedule function**

1. In the **onreadystatechange** callback function, after receiving the response (check that the **readyState** of the **request** object is set to 4), add code to test the request status code.

For non-200 status codes, the **response** object contains a **message** property. Display this message by using the built-in JavaScript **alert** function.

2. Also, handle the exceptional case of the remote data source returning invalid JSON data that cannot be parsed. In this case, display a general error message using the **alert** function.

► **Task 4: Test the Schedule page**

1. Run the application and view the **schedule.htm** page to verify that the list of sessions is displayed.
2. Close Internet Explorer.
3. In the **schedule.js** file, change the requested URL to **/schedule/list?fail**, save the file, and then refresh the **schedule.htm** page in Internet Explorer.



Note: The URL **/schedule/list?fail** generates errors that enable you to test that the exception handling parts of your code work correctly.

4. Run the application again and verify that the error message **Service currently unavailable** appears.
5. After testing, close Internet Explorer and change the URL back to **/schedule/list**.

Results: After completing this exercise, you will have modified the code for the Schedule page to displays the list of sessions retrieved from a web service, and to handle errors that can occur when communicating with a remote service.

Exercise 2: Serializing and Transmitting Data

Scenario

In this exercise, you will record the sessions that an attendee selects by transmitting this data to a web service. In addition, you will check for potentially busy sessions and inform the attendee accordingly.

First, you will create a function that creates an **XMLHttpRequest** object that posts data to a web service indicating the session that a user has selected. You will encode the content of this request and set the HTTP request headers appropriately. Next, you will handle the response and display a warning message if the response indicates that the attendee has selected a popular session that is likely to be busy. Finally,

you will run the application and view the Schedule page to verify that the busy session message is displayed.

The main tasks for this exercise are as follows:

1. Send the request to indicate the session that an attendee has selected.
2. Handle the web service response.
3. Test the Schedule page.

► **Task 1: Send the request to indicate the session that an attendee has selected**

1. Open the **ContosoConf.sln** solution in the **E:\Mod05\Labfiles\Starter\Exercise 2** folder.
2. In the **schedule.js** file, find the function named **saveStar**.
3. Within the function, add code to construct an **XMLHttpRequest** object and make a synchronous POST request to the following URL:

```
"/schedule/star/" + sessionId
```



Note: The **sessionId** variable is passed in as a parameter to the **saveStar** function; it contains the id of the session that the user has selected.

4. In the **saveStar** function, use the **setRequestHeader** function to set the request **Content-Type** header to the following:

```
application/x-www-form-urlencoded
```

5. Use the **Send** function to send the request, with the following body:

```
"starred=" + isStarred
```



Note: The **isStarred** parameter to the **saveStar** function is a Boolean value that indicates whether the attendee has selected or deselected the session on the **Schedule** page.

► **Task 2: Handle the web service response**

1. In the **saveStar** function, after the code creates the POST request but before it actually sends the request, if the attendee has selected the session (**isStarred** is true), create a callback for the **onreadystatechange** property of the **request** object that performs the following operations:
 - o If the request status is 200, then parse the response into a JSON object. This object should have a **starCount** property (the web service that sends the response back formats the data in this way).
 - o If **starCount** is greater than 50, display the following message by using the **alert** function:

```
This session is very popular! Be sure to arrive early to get a seat.
```

► **Task 3: Test the Schedule page**

1. Run the application and view the **schedule.htm** page.
2. Click the star next to the **New Technologies in Enterprise** session.
3. Verify that an alert is displayed (this session is popular).

4. Click the star next to **Diving in at the deep end with Canvas**.
5. Verify that no alert is displayed (this session is less popular).
6. Close Internet Explorer.

Results: After completing this exercise, you will have updated the Schedule page to send attendee selections to a web service, and to display a message when a session is popular.

Exercise 3: Refactoring the Code by Using the jQuery ajax Method

Scenario

The existing code using an **XMLHttpRequest** object works, but it is somewhat verbose. The **XMLHttpRequest** object also requires you to carefully set HTTP headers and encode the content appropriately; otherwise request data may not be transmitted correctly. In this exercise, you will refactor the JavaScript code for the Schedule page to make it simpler and more maintainable, by using the **jQuery ajax()** function.

First, you will refactor the **downloadSchedule** function by replacing the use of an **XMLHttpRequest** object with a call to the jQuery **ajax** method. Then you will refactor the **saveStar()** function in a similar manner. Using the **ajax()** function will simplify the code by automatically encoding the request content and setting HTTP headers. Finally, you will run the application and view the Schedule page to verify that it still displays sessions and responds to star clicks as before.

The main tasks for this exercise are as follows:

1. Refactor the **downloadSchedule** function.
2. Refactor the **saveStar** function.
3. Test the Schedule page.

► Task 1: Refactor the **downloadSchedule** function

1. Open the **ContosoConf.sln** solution in the **E:\Mod05\Labfiles\Starter\Exercise 3** folder.
2. Near the bottom of the **schedule.htm** file, before the reference to the **schedule.js** script, add a reference to the jQuery JavaScript file **jquery.min.js** located in the **scripts** folder.

This action makes the **jQuery** object available for use in the JavaScript code for the Schedule page.

3. In the **schedule.js** file in the **scripts/pages** folder, refactor the **downloadSchedule** function to use the jQuery **ajax()** function.
 - o The ajax options object must have a **type** property of **GET** and a **url** property of **/schedule/list**.
 - o Use the jQuery **done()** function to provide a callback function that handles the web service response; the response will contain a property called **schedule** that you should assign to the local **schedule** array variable, and then call the **displaySchedule()** function.



Note: The **done** callback is passed an argument containing the parsed JSON response object, so you do not need to use the **JSON.parse** method to extract the session data.

- o Use the jQuery **fail()** function to provide a callback function that handles any errors that might occur; simply display the message **Schedule list not available** in this callback.

► **Task 2: Refactor the saveStar function**

1. In the **schedule.js** file, refactor the **saveStar** function to use the jQuery **ajax()** function.
 - o The ajax options object must have a **type** property of **POST** and **url** property of **"/schedule/star/" + sessionId**.
 - o The data passed by using the ajax options object that specifies whether a session has been selected should be a JavaScript object and not a string.
 - o Use the jQuery **done()** function to implement a callback function that receives the web service response. In this callback, if the **starCount** property in the response object passed to the callback is greater than 50, display the message **This session is very popular! Be sure to arrive early to get a seat.**

► **Task 3: Test the Schedule page**

1. Run the application and view the **schedule.htm** page.
2. Verify that the list of sessions appears correctly.
3. Click the star icon next to **New Technologies in Enterprise** and verify that an alert is displayed.
4. Click the star icon next to **Diving in at the deep end with Canvas** and verify that no alert is displayed.
5. Close Internet Explorer.
6. In **schedule.js**, change the URL in the **downloadSchedule** function to be **/schedule/list?fail**.
7. Run the application and view the **schedule.htm** page. Verify that the error message **Schedule list not available** is displayed.
8. Close Internet Explorer.

Results: After completing this exercise, you will have refactored the JavaScript code that sends and receives data to use the jQuery **ajax** method.

Module Review and Takeaways

In this module, you have learned how to use the **XMLHttpRequest** object to send a request to a remote server and handle any response that is returned. You have seen how to perform these operations synchronously and asynchronously, and you have also learned how to catch and handle any error that might occur when a web application sends a request to a remote server.

You have seen how to use the jQuery library to simplify many of the operations associated with sending and receiving data, and in particular you have learned how to use the jQuery **ajax()** function to perform these tasks in a concise manner.

Review Question(s)

Test Your Knowledge

Question
In the onreadystatechange event handler for the XMLHttpRequest object, which property should you examine to ensure that data has been returned, and what value should this property contain?
Select the correct answer.
<input type="checkbox"/> The readyState property should be set to 0.
<input type="checkbox"/> The responseText property should be set to a non-null value.
<input type="checkbox"/> The readyState property should be set to 4.
<input type="checkbox"/> The status property should be set to 200 (HTTP OK).
<input type="checkbox"/> The HTTPResponse property should be set to 0.

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
If you use the jQuery get() function to retrieve data and you do not specify an error handling function, any failures while retrieving the data will cause your JavaScript code to stop with an unhandled exception. True or False?	

Module 6

Styling HTML5 by Using CSS3

Contents:

Module Overview	6-1
Lesson 1: Styling Text by Using CSS3	6-2
Lesson 2: Styling Block Elements	6-6
Lesson 3: Pseudo-Classes and Pseudo-Elements	6-14
Lesson 4: Enhancing Graphical Effects by Using CSS3	6-17
Lab: Styling Text and Block Elements by Using CSS3	6-24
Module Review and Takeaways	6-33

Module Overview

Styling the content displayed by a web page is an important aspect of making an application attractive and easy to use. CSS is the principal mechanism that web applications use to implement styling, and the features added to CSS3 support many of the new capabilities found in modern browsers.

Where CSS1 and CSS2.1 were single documents, the World Wide Web Consortium has chosen to write CSS3 as a set of modules, each focusing on a single aspect of presentation such as color, text, box model, and animations. This allows the specifications to develop incrementally, along with their implementations. Each specification defines properties and values that already exist in CSS1 and CSS2, and also new properties and values.

In this module, you will examine the properties and values defined in several of these modules, the new selectors defined in CSS3, and the use of pseudo-classes and pseudo-elements to refine those selections.

Objectives

After completing this module, you will be able to:

- Use the new features of CSS3 to style text elements.
- Use the new features of CSS3 to style block elements.
- Use CSS3 selectors, pseudo-classes, and pseudo-elements to refine the styling of elements.
- Enhance pages by using CSS3 graphical effects.

Lesson 1

Styling Text by Using CSS3

You have already seen a number of core CSS properties that style and transform text. In this lesson, you will look in more detail at the value ranges for some of these properties, and you will also be introduced to a number of new CSS3 properties that target text.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the different ranges of values you can apply to CSS font and text properties.
- Demonstrate the new CSS3 properties applicable to text and to blocks of text.

Fonts and Measurements

CSS3 adds the **@font-face** rule to support downloading fonts onto a user's computer so that the browser can use them to render text. CSS3 also includes a number of new ways to specify the size of a font and the spacing around it. These measurements also apply to boxes, columns, images, and positioning.

@font-face

Web designers have, until recently, been restricted when deciding how to render text to the set of fonts installed by default on different operating systems. This limitation has often resulted in using a combination of web-safe fonts for general text and the replacement of particular headings with a graphic of that heading's text in a different font.

The **@font-face** rule enables you to specify a font file to download, give it a name, and then use it in your CSS rules just like any other web-safe font such as Times, Arial, and Helvetica.

The following example shows how to use the **@font-face** rule to download the TrueType® version of the Roboto Regular font, and use it to style some paragraphs.

Using the **@font-face** Rule

```
@font-face {  
    font-family: 'RobotoRegular';  
    src: url('Roboto-Regular-webfont.ttf') format('truetype');  
    font-stretch: normal; // Default  
    font-weight: normal; // Default  
    font-style: normal; // Default  
    unicode-range: U+0-10FFFF; // Default  
}  
p {  
    font-family : RobotoRegular, "Segoe UI", Arial;  
    font-size: 14px;  
}
```

• CSS3 font and text properties support:

• External fonts

```
@font-face {  
    font-family: newGroovyFont;  
    src: url('CandaraPlus.ttf')  
}
```

• Absolute text sizes

```
font-size : 16pt;  
line-height : 0.5in;  
letter-spacing : 12mm;
```

• Relative text sizes

```
font-size : 1em;  
border-width : 30px;  
padding : 16rem;
```

The **@font-face** rule implements a combination of properties that define how a browser should use a font:

- **font-family** sets the name to be used for this font in other style sheet rules.
- **src** defines the URL to download the font file from and the type of font file being downloaded.
- **font-stretch** identifies how condensed or expanded the font is.
- **font-style** identifies whether the font is italicized, oblique, or normal.
- **font-weight** identifies the boldness of the font; bold, normal, or a value between 100 and 900.
- **unicode-range** identifies the range of Unicode characters the font supports.

Only the **font-family** and **src** properties are required, but if you have access to the font files for condensed, bold, or italic variants of the same font, you should prefer to use these rather than let the browser style the text. The purpose of the optional properties is to give the browser hints that these font variants are available, as shown in the following examples:

```
@font-face {
    font-family: 'RobotoRegular';
    src: url('Roboto-Regular-webfont.ttf') format('truetype');
}
@font-face {
    font-family: 'RobotoRegular';
    src: url('Roboto-Regular-B-webfont.ttf') format('truetype');
    font-weight: bold;
}
@font-face {
    font-family: 'RobotoRegular';
    src: url('Roboto-Regular-I-webfont.ttf') format('truetype');
    font-style: italic;
}
```

You should also note that while most web browsers support the **@font-face** rule, they do not all support the same font formats. You actually require four font file formats for complete support (**embedded-opentype**, **woff**, **truetype**, and **svg**). The following example shows how to download the font files that support these formats:

```
@font-face {
    font-family: 'RobotoRegular';
    src: url('Roboto-Regular-webfont.eot');
    src: url('Roboto-Regular-webfont.eot?#iefix') format('embedded-opentype'),
        url('Roboto-Regular-webfont.woff') format('woff'),
        url('Roboto-Regular-webfont.ttf') format('truetype'),
        url('Roboto-Regular-webfont.svg#RobotoRegular') format('svg');
}
```

 **Note:** You should not distribute font files around the web unless you have a license to do so. However, there are many open source fonts available, and several font webshops will sell you a font and the required license.

Measurements

When you set the **font-size** property, the most common units used for that property are **points** for print style sheets and **pixels** for screen style sheets. However, CSS3 defines several additional units of measurement that can be applied to boxes, columns, and images, and for positioning these items.

There are six units of absolute measurement:

- Centimeters.
- Millimeters (mm): 10 millimeters = 1 centimeter.

- Inches (in): 1 inch = 2.54 centimeters.
- Picas (pc): 1 pica = 12 points = 1/6th of an inch.
- Points (pt): 1 point = 1/72nd of an inch.
- Pixels (px): 1 pixel = 1/96th of an inch.

There are seven units of relative measurement. Four are 'font-relative':

- em: 1em = the current font size of the current element.
- ex: 1ex = the height of the font's lowercase x-height, or 0.5 em if not calculable.
- ch: 1ch = the width of the font's 0 (zero) character.
- rem: 1rem = the size of the font defined on the html element (16px default).

The other three are 'viewport-relative'. That is, they are relative to the size of the browser window object:

- vw: 1vw = 1% of the width of the viewport.
- vh: 1vh = 1% of the height of the viewport.
- vmin: vmin = the smaller of vh and vw.

You can also use the **calc()** function to calculate a measurement at runtime. For example:

```
img {
  max-height: calc(100vh - 5px);
  max-width: calc(100vw - 5px);
}
```

In this case, the height and width of an image is set to a maximum of the browser window height and width, minus 5px. This ensures you can always see the full image in your browser.

Implementing Text Effects

The core CSS typographic properties enable you to style text by setting the **letter-spacing**, **line-height**, **text-align**, **text-decoration**, and **text-transform** properties. However, the CSS3 Text module defines several more properties that provide even greater control over the layout of your text content. These properties include:

- The **text-indent** property, which indicates how far the first line of each new text block should be indented.

```
text-indent: 3rem;
```

- CSS3 includes further text styling support for:

- | | |
|-------------------------|--|
| • Paragraph indentation | <code>text-indent: 3rem;</code> |
| • Line wrapping | <code>hyphens: manual;</code>
<code>word-wrap: break-word;</code> |
| • Text spacing | <code>word-spacing: 2rem;</code> |
| • Shadow effects | <code>text-shadow: 2px 2px 0 red;</code> |



- The **hyphens** property, which indicates how the browser should hyphenate words when line-wrapping. Possible values are **none** (no hyphenating), **manual** (use the ­ sequence in your text to indicate where hyphens can be placed), and **auto**.

```
hyphens: manual;
-ms-hyphens : manual; // IE10
```

- The **word-wrap** property, which indicates whether the browser may break lines within words when line-wrapping. Possible values are **normal** (the default is no), and **break-word** (the browser may break

words at an arbitrary point). It has been renamed overflow-wrap in CSS3, but at this time browsers only recognize the old name.

```
word-wrap : normal;
```

- The **word-spacing** property, which enables you to set the spacing between words in a block of text. You can use both relative and absolute measurements.

```
word-spacing : 5px;  
word-spacing : 2rem;
```

- The **text-shadow** property, which enables you to apply shadowing to the selected text. A shadow is defined by four properties:

- x-offset: How far to the right the shadow starts. Use a negative value to move it to the left.
- y-offset: How far below the shadow starts. Use a negative value to move it up.
- blur (optional): How wide the blur of the shadow is. The default is 0.
- color: Can be any color value. The default is black.

```
text-shadow: 0 1px 0 #000; // not supported in Internet Explorer 9 and earlier  
versions.
```

 **Reference Links:** You can find the current draft of the CSS3 Text Module at <http://go.microsoft.com/fwlink/?LinkId=267726>.

Lesson 2

Styling Block Elements

The CSS box model defines how browsers handle every block of content on a page, and it is the basis of every CSS layout. In this lesson, you will learn about the additions in CSS3 to the core box model, and you will see a new way to navigate between blocks on screen. You will also look in more depth at the different ways to lay out blocks on a page, including the new CSS3 **Flexbox** method.

Lesson Objectives

After completing this lesson, you will be able to

- Describe the new CSS3 properties applicable to the box model.
- Describe how to use these properties to lay out the items on a web page.

New Block Properties in CSS3

The basic box model has not been changed in CSS3, but it does add several properties that modify the way in which boxes and their contents are presented and accessed.

Outlines

CSS defines an outline box in addition to the four concentric boxes (content, padding, border, and margin) that make up the box model. However, an outline does not add to the total width or height of the box. Instead, it is drawn above the margin box, and defined relative to the box's border.

Outlines can therefore overlap on a page.

The following example shows how to draw a border and an outline around a box. The outline is drawn 5px away from the border.

Adding an Outline

```
div {  
    border: 2px solid red;  
    outline: 2px solid green;  
    outline-offset: 5px;  
}  
/* The above code can also be written in full as */  
div {  
    border-width: 2px;  
    border-style: solid;  
    border-color: red;  
    outline-width: 2px;  
    outline-style: solid;  
    outline-color: green;  
    outline-offset: 5px;  
}
```

- CSS3 adds new box-level support for:

- Outlines

```
outline: 2px solid green;  
outline-offset: 5rem;
```

- Presentation

```
border-radius: 50% / 30%;  
overflow: hidden;  
resize: horizontal;
```

- Multiple column layouts

```
column-count: 3;  
column-gap: 5rem;  
column-rule: 1px solid black;
```

Outlines are defined by four properties:

- **outline-width** sets the width of the outline. Possible values are thin, medium (the default), and thick, or a specific measurement such as 2px or 1.5 rem.

- **outline-style** sets the line style of the outline. The most common values used are none, dotted, dashed, and solid.
- **outline-color** sets the color of the outline. Set it to any allowable color value, or invert (the default).
- **outline-offset** sets the distance between the outline and the border.

You can use **outline** as a shorthand property for the **outline-width**, **outline-style**, and **outline-color** properties, in that order. The **outline-offset** property is new in CSS3 and must be declared separately, as shown in the previous example.

Presentation

There are several new design-related box properties in CSS3.

- The **border-radius** property enables you to set rounded corners on a box's border by defining the radius of the circle or radii of an ellipse around which the corner will bend.

```
border-radius: 2em;    // circular corners with radius 2em
border-radius: 5px/10px; // elliptical corners with radii of 5px high and 10px wide
```

Note that border-radius is a shorthand property. You can set the radius of each of the four border corners individually by using the border-top-left-radius, border-top-right-radius, border-bottom-right-radius, and border-bottom-left-radius properties.

- The **overflow-x** and **overflow-y** properties enable you to set what happens when the content of an element is too wide or too high for the box that contains it. Possible values are:
 - **visible**: The content is not clipped and is rendered outside of the box. This is the default.
 - **hidden**: Only the content within the box is shown.
 - **scroll**: Only the content within the box is shown, but a scrollbar is displayed so the rest of the content can be viewed.

These are the same values as for the overflow property, which applies to both x- and y-axes.

```
overflow-x: hidden;
overflow-y: scroll;
```

- The **resize** property enables you to mark a text block as resizable, assuming that the **overflow** property has already been set to hidden or scroll. Possible values are **none** (the default), **both**, **horizontal**, and **vertical**, denoting in which axes the element should be resizable.

```
overflow: hidden;
resize: horizontal;
min-width: 60px;
max-width: 400px;
```

 **Note:** It's a good idea to set a minimum and maximum height for a block marked as resizable so that the user doesn't break the page layout completely by changing the size of the browser window.

Multiple Column Layout

The CSS3 Multi-Column module extends the CSS box model by adding properties to set the number of columns a box's content will be displayed in, as well as their width, padding (gap), and border (rule).

The following example shows how to style **section** elements to use a three-column layout with a 5rem gap between columns and a dotted line between each column.

Creating a Basic Cross-Browser Layout

```
section {
    text-align: justify;
    column-count : 3;
    column-gap : 5rem;
    column-rule : 1px solid black;
}
```

There are four main properties for creating multiple column layouts:

- **column-count** sets the number of columns to be used.
- **column-width** sets the width of the columns.
- **column-gap** sets the padding between the column.
- **column-rule** sets the properties of the line drawn between columns.

Support for multiple-column layouts is limited at present, although it is available in Internet Explorer 10.

 **Reference Links:** You can find the latest draft of the CSS3 Multi-Column Layout Module at <http://go.microsoft.com/fwlink/?LinkId=267727>.

Block Layout Models

CSS enables you to define the layouts and the types of boxes that you can display on a web page. The CSS **display** property determines the type of layout model used on a page. This property can have the following values:

- **block:** Block boxes are formatted down the page one after another and respect **padding**, **border**, and **margin** values.

```
display:block;
```

• CSS3 supports several block layout methods:

• Block	<code>display: block;</code>
• Inline	<code>display: inline;</code> <code>display: inline-block;</code>
• Table	<code>display: table;</code>
• Positioned	<code>position: relative;</code> <code>position: absolute;</code> <code>position: fixed;</code>
• Flexbox	<code>display: flexbox;</code>

- **inline:** Inline layout blocks are formatted one after another based on the baseline of their text content until they break down onto another line, and so on. Inline blocks ignore **height** and **width** values.

```
display:inline;
```

- **inline-block:** Inline-block layout blocks are formatted one after another based on the baseline of their **text** content, but they keep their **height** and **width** values.

```
display:inline-block;
```

- **table:** Table layout enables you to identify blocks on the page as tables, rows, columns, and cells. Blocks are aligned by their edges rather than their content, and sized to fit the computed table.

```
display:table;
```

- **flexbox:** Flexbox layout is new in CSS3 and designed to be far more fluid than the others. You choose in which direction boxes are laid out and how boxes are sized, depending on how any excess whitespace around blocks should be handled.

```
display: flexbox;      // for a block-level flexbox container
display: -ms-flexbox;
display: inline-flexbox; // for an inline flexbox container
display: -ms-inline-flexbox;
```



Note: Note that the display values for enabling flexbox layout were correct when written; the values may change before the CSS3 Flexbox Module is finalized.

All of these layout models (possibly with the exception of flexbox, depending on the final implementation) assume that blocks are arranged according to the *normal* flow of elements. For example, with the code below, the normal flow of elements would position **div1** on the page first then **div2** next to it, **div3** next to **div2**, and **div4** next to **div3**, and all four **div** elements would be placed inside the **article** element:

```
<article>
  <div id="1">One</div>
  <div id="2">Two</div>
  <div id="3">Three</div>
  <div id="4">Four</div>
</article>
```

These blocks all have the default CSS **position** value of **static**. You can take any of these blocks out of the normal flow by changing their **position** property to **relative**, **absolute**, or **fixed**.

If you set **position to relative**, you can use the **top**, **right**, **bottom**, and **left** properties to position the block relative to the position it would have been in. All other blocks stay in the same position they would have been in, as if the block was statically rather than relatively positioned.

```
position: relative;
top: -10px; // Top edge of block moved up 10 pixels from normal
left: 10px; // Left edge of block moved right 10 pixels from normal
```

If you set **position to absolute**, the block is taken completely out of the normal flow and positioned relative to the edges of its containing block. In the following styling, if **div2** were absolutely positioned, **div3** would be positioned next to **div1** in the normal flow.

```
position: absolute;
top: 25px; // Top edge of block 25 pixels down from top edge of container
right: 10px; // Right edge of block 10 pixels left from right edge of container
```

If you set **position to fixed**, the block is out of the normal flow and positioned relative to the edge of the browser window (technically, the viewport).

```
position: fixed;
bottom: 0; // Bottom edge of block in line with bottom edge of browser window
right: 0; // Right edge of block in line with right edge of browser window
```

Demonstration: Switching Between CSS Layout Models

In this demonstration, you will see how the different CSS layout modes and positioning values affect four simple boxes. You will use the F12 Developer Tools to make the changes between modes.

Demonstration Steps

Switch between layout modes in a web page

1. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.

2. In Visual Studio, on the **File** menu, point to **Open**, and then click **File**.
3. In the **Open File** dialog, browse to the **E:\Mod06\Democode** folder, click **positioning.html**, and then click **Open**.
4. Review the code with the students. This file contains an HTML **article** with four **div** elements. The file also contains styles for the **article** and **div** elements.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Positioning Demo</title>
    <style type="text/css">
        body {
            text-align: center;
        }
        article {
            padding: 10px;
            border: 2px solid red;
        }
        div {
            margin: 10px;
            padding: 5px;
            border: 2px solid black;
            width: 150px;
            height : 150px;
        }
        div:nth-child(odd) {
            font-size: 4rem;
        }
    </style>
</head>
<body>
    <article>
        <div id="one">One</div>
        <div id="two">Two</div>
        <div id="three">Three</div>
        <div id="four">Four</div>
    </article>
</body>
</html>
```

5. On the **File** menu, click **View in Brower (Internet Explorer)**.
6. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
7. Observe the four **div** elements laid out underneath each other, in order, within the **article** element. The **article** element has a red border to highlight its boundaries. This is block layout mode.

The web page looks like this:

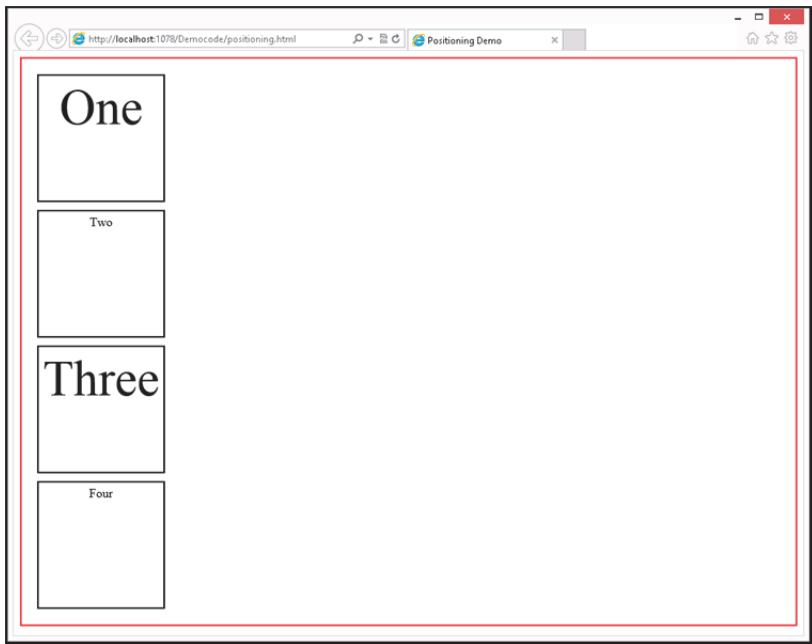


FIGURE 6.1:THE DIV ELEMENTS IN BLOCK LAYOUT MODE

8. Press **F12**.
9. In the F12 Developer Tools pane, press **Ctrl+P** to unpin the window. Position the F12 Developer Tools window so that you can see the Internet Explorer window at the same time.
10. Click the **CSS** tab to display the fully expanded version of the layout rules applied to the HTML content.
11. Right-click the **div** entry, and then click **Add attribute**.
12. Type **display: inline**, and then press ENTER.
13. In Internet Explorer, notice that the four **div** elements are now laid out side-by-side aligned by text baseline with height and width properties ignored. This is inline layout mode.

The web page now looks like this:

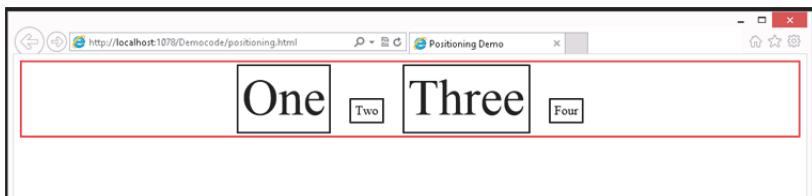


FIGURE 6.2:THE DIV ELEMENTS IN INLINE LAYOUT MODE

14. Resize the browser window to make it narrower, so you can see how blocks are wrapped onto the next line in inline layout mode.
15. In the F12 Developer Tools window, on the **CSS** tab, click the **display: inline** rule, change it to read **display:inline-block**, and then press ENTER.
16. Notice the layout is the same but the **height** and **width** properties are now preserved. This is inline-block mode.

 **Note:** If necessary, make the browser window wider so that blocks **One** and **Three** are on the same line.

The web page now looks like this:

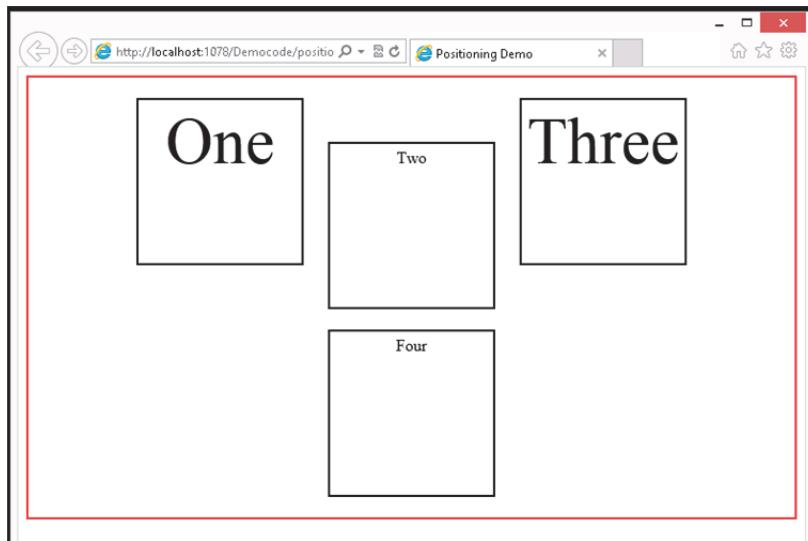


FIGURE 6.3:THE DIV ELEMENTS IN INLINE-BLOCK LAYOUT MODE

1. In Internet Explorer, resize the browser window so you can see how blocks are wrapped onto the next line in inline layout mode.
2. In the F12 Developer Tools window, on the **CSS** tab, click the **display: inline-block** rule. Change this rule to **display:-ms-flexbox**, and then press ENTER.
3. Switch to Internet Explorer to view the new layout. The **div** elements are displayed in a vertical column.
4. In the F12 Developer Tools window, on the **CSS** tab, click the **display: -ms-flexbox** rule, change it to **display:table-cell**, and then press ENTER.
5. Switch to Internet Explorer to view the new layout. The **div** elements are displayed in a horizontal table.

Switch between positioning modes in a web page

1. In the F12 Developer Tools window, on the **CSS** tab, clear the three checkboxes next to the display attributes for **body**, **article**, and **div**.
2. Right-click the **div** entry, and then click **Add rule after**.
3. Type **#three** and then press **Tab**.



Note: This action creates a new rule for the **<div>** element with the **id** property set to **three**. This is the **<div>** containing the text **Three**.

4. Type **position: relative** and then press ENTER.
5. Right-click the **#three** entry, and then click **Add attribute**.
6. Type **top: 2em**, and then press ENTER.
7. Right-click the **#three** entry, and then click **Add attribute**.
8. Type **left: 2em**, and then press ENTER.
9. In Internet Explorer, notice how the **three** box is positioned relative to its normal position.

10. In the F12 Developer Tools window, on the **CSS** tab, click the **position:relative** rule for the **#three** selector, change it to **position:absolute**, and then press ENTER.
11. In Internet Explorer, notice how the **three** box is now positioned relative to its containing **article** block.
12. In the F12 Developer Tools window, on the **CSS** tab, click the **position:absolute** rule for the **#three** selector, change it to **position:fixed**, and then press ENTER.
13. In Internet Explorer, notice how the **three** box is positioned relative to the browser window. Make the window small enough to require scrolling and see how the **three** box remains stationary when you scroll (it does not scroll into view).
14. Close Internet Explorer, and then close Visual Studio 2012.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3

Pseudo-Classes and Pseudo-Elements

You use CSS selectors to specify sets of elements to be styled based on element, class, id, and attribute. You can refine the set of elements to style in a CSS rule by concatenating them. In this lesson, you will learn how you can extend selectors to include runtime and navigation information by using pseudo-classes and pseudo-elements in selectors.

Lesson Objectives

After completing this lesson, you will be able to:

- Use pseudo-elements to add styles to text elements.
- Style hyperlinks and form elements based on their current state.
- Identify elements to style based on their position in a document.

Text Pseudo-Elements

Text-based pseudo-elements match elements that are not easily identifiable in the document tree for a page. The following list describes some of the more commonly used pseudo-elements. Notice that in a CSS rule you use a double colon to concatenate them to a text item.

- **first-letter** selects the first character of the first line of text content of an element.

```
p::first-letter{ }
```

- **first-line** selects the first line of text content of an element.

```
p::first-line{ }
```

- **before** selects the space immediately before an element.

```
p::before{ }
```

- **after** selects the space immediately after an element.

```
p::after{ }
```

- **selection** selects the part of the page that has been highlighted by the user.

```
::selection{ }
```

CSS pseudo-elements enable you to select:

- | | |
|--|---|
| • The first letter of a text element | <code>p::first-letter</code> |
| • The first line of a text element | <code>p::first-line</code> |
| • The space before or after a text element | <code>p::before</code>
<code>p::after</code> |
| • Text selected by the user | <code>::selection</code> |

You might use the **first-letter** and **first-line** pseudo-elements for styling text illuminations such as a drop-cap or a larger font for the start of a paragraph. The **selection** pseudo-element is useful for changing how content is highlighted on screen. The **before** and **after** pseudo-elements have several common uses. The most frequent is to add or remove content around an element. For example, *reset* stylesheets often normalize how browsers deal with **q** and **blockquote** elements by removing any smart quotes they might add around them.

```
blockquote::before, blockquote::after,
q::before, q::after {
    content: '';
    content: none;
}
```

Note that **before** and **after** are only applicable to elements that contain text content.

 **Note:** In CSS1 and CSS2, pseudo-elements start with a colon (:). In CSS3, pseudo-elements start with a double colon (::) to differentiate them from pseudo-classes. However most browsers still use single colons. Internet Explorer 10 recognizes single and double colons.

Link and Form Pseudo-Classes

Pseudo-classes are like pseudo-elements in that they match elements that are not part of the document tree. However, the pseudo-classes are not text elements; rather, they match against the result of user interaction with the page. In CSS rules you concatenate them to elements by using a single colon.

There are five pseudo-classes for hyperlinks:

- **a:link** selects all unvisited links.
- **a:visited** selects all visited links.
- **a:focus** selects all links in focus.
- **a:hover** selects all links with the cursor hovering over them.
- **a:active** selects all selected links.

CSS defines two sets of contextual pseudo-classes:

- | | |
|--|--|
| <ul style="list-style-type: none"> • Link classes | <pre>a:link
a:visited
a:focus
a:hover
a:active</pre> |
| <ul style="list-style-type: none"> • Form classes | <pre>input:enabled
input:disabled
input:checked</pre> |

If you define CSS rules that match against more than one of these pseudo-classes, it is important that you specify these pseudo-classes in the following order: link, visited, focus, hover, and active. If you reference them in a different order, some may cancel others out. Also, a simple selector can only contain more than one pseudo-class if the pseudo-classes aren't mutually exclusive. For example, the selectors **a:link:hover** and **a:visited:hover** are valid, but **a:link:visited** isn't because **:link** and **:visited** are mutually exclusive. A link element is either an unvisited link or a visited link.

 **Note:** There are several mnemonics for remembering the correct order (LVFHA) for link pseudo-classes. One is **Las Vegas fights Hell's Angels**. Another is **Let Victoria Free Her Armies**.

There are three pseudo-classes that you frequently use for forms elements:

- **input:enabled** selects all enabled input controls.
- **input:disabled** selects all user interface elements that are disabled.
- **input:checked** selects all user interface elements that are checked.

 **Note:** Forms elements also provide the **:valid** and **:invalid** pseudo-classes that you can use to select items that have been validated successfully or unsuccessfully. This subject was covered in module 4.

DOM-Related Pseudo-Classes

In addition to link and form-related pseudo-classes, other pseudo-classes are available that enable you to identify specific elements in a selected set based on their position in the DOM. For example:

- **:first-child** selects the item that's the first child of its parent. As an example, to find the first element in a list, use **li:first-child**.
- **:last-child** selects the list item that's the last child of its parent.
- **:only-child** selects a list item if it is the only child of its parent.
- **:nth-child(n)** selects a list item if it is the *n*th child of its parent.
- **:nth-last-child(n)** selects a list item if it is the *n*th child of its parent, counting backwards from its last child.

Use positional pseudo-classes to select a single element from a set based on:

- | | |
|----------------------|---|
| • Position | <code>p:first-child</code>
<code>p:nth-child(2)</code> |
| • Position and type | <code>p:last-of-type</code>
<code>p:nth-last-of-type(4)</code> |
| • Document structure | <code>:empty</code>
<code>:root</code>
<code>:not(p, h1)</code>
<code>:target</code> |

Another set enables you to select elements based on their position in the DOM *and their type*. For example:

- **:first-of-type** selects a list item if it is the first list item child of its parent.
- **:last-of-type** selects a list item if it is the last list item child of its parent.
- **:only-of-type** selects a list item if it is the only list item child of its parent.
- **:nth-of-type(n)** selects a list item if it is the *n*th list item child of its parent.
- **:nth-last-of-type(n)** selects a list item if it is the *n*th list item child of its parent, counting backwards from its last child.

Note that the numerical argument for nth-child, nth-last-child, nth-of-type, and nth-last-of-type can be set as a **simple integer**, the values **odd** and **even**, or a simple formula of the form $xn + y$ where **x** and **y** are integers.

A set of structural pseudo-classes is also available. These classes enable you to select elements based on the current structure of the document. These classes include:

- **E:root** selects a document's root element. For an HTML document, it will always select the `<html>` element.
- **E:empty** selects an element of type *E* if it has no children or text content.
- **E:target** selects an element of type *E* if it is the target of a referring URL.
- **E:not(s)** selects any element which does not match the selector strings.



Note: Remember: Pseudo-classes qualify the element they are attached to rather than specifying other elements. For example, **li:first-child** selects a list item that's the first child of its parent, and not the first child of the list item.

Lesson 4

Enhancing Graphical Effects by Using CSS3

The web is a far more colorful place now that most modern browsers have adopted CSS3. Previously, developers would need to use additional graphics to add background gradients or resort to various items of trickery to rotate and transform graphics elements. Many of these tricks were browser dependent, resulting in a poor experience for users running browsers for which the features were not designed.

The new color and background values, and the graphics capabilities provided by CSS3, now enable developers to perform these tasks in a standardized way that will work in any compliant browser. In this lesson, you will look at the new color values you can give to properties, and how you can incorporate multi-image backgrounds into a web page. You will also see how to perform simple transformations on an element and how to create simple shapes by using CSS.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use the new CSS3 color value sets.
- Use CSS3 to provide gradients and multi-image backgrounds.
- Combine CSS3 properties to create shapes, and use simple transforms to manipulate elements on the page.

Specifying Color Values

You use the CSS **color** property to modify the color of text content. CSS3 extends this functionality by enabling you to apply color backgrounds, borders, outlines, column rules, and more. It is important to be aware of the many different ways of specifying color values and how CSS3 implements transparency.

The CSS3 Color module defines several value ranges for the **color** property:

- Any of the 147 color keywords defined in the module. For example, red or green.

```
color: yellow;
```

- A red-green-blue (RGB) model value specified in three- or six-digit hexadecimal notation, a triplet of integers, or a triplet of percentage values. Each of the three values represents the amount of red, green, and blue is included in the color, with values for each between 0 and 255 (0 to 100%).

```
color: #ff00;           /* #rgb */
color: #ffff00;         /* #rrggbb */
color: rgb(255, 255, 0);
color: rgb(100%, 100%, 0%);
```

- A red-green-blue-alpha (RGBA) model value specified as either a triplet of integers, or a triplet of percentage values plus an opacity value of between 0 and 1 where 0 is completely transparent and 1 is completely solid (opaque).

CSS3 defines several different sets of color values:

- Keywords


```
color: red;
color: transparent;
color: currentColor;
```
- RGB \ RGBA


```
color: #ff0000;
color: rgb(255,0,0);
color: rgba(100%,0,0,0.5);
```
- HSL \ HSLA


```
color: hsl(240, 100%, 50%);
color: hsl(120, 100%, 50%, 0.5);
```

MCT USE ONLY. STUDENT LICENSED PROPERTY

```
color: rgba(255, 255, 0, 0.2); /* mostly transparent yellow */
color: rgba(100%, 100%, 0%, 0.8); /* mostly opaque yellow */
```

- The keyword **transparent**, which is the same value as `rgba(0,0,0,0)`.

```
color: transparent;
```

- A hue-saturation-lightness (**HSL**) model value specified as a triplet of numbers. The first is an integer value between 0 and 360 indicating the angle of the color circle (0 = red, 120 = green, 240 = blue). The second is a percentage value for saturation where 0% is a shade of grey and 100% is full color. The third is also a percentage value for lightness, where 0% is black, 100% is white and 50% is normal.

```
color: hsl(60, 100%, 50%);
```

- A hue-saturation-lightness-alpha (**HSLA**) model value specified as a quadruplet of numbers. The first three are those of the HSL model and the fourth is the same opacity value of between 0 and 1 as in the RGBA model.

```
color: hsla(60, 100%, 50%, 0.2); /* mostly transparent yellow */
```

- The keyword **inherit**. This indicates that the element should inherit the same color value as its parent element.
- The keyword **currentColor**. This indicates the same value should be used as that of the element's `color` property. Writing `color:currentColor` is the same as writing `color:inherit`.



Additional Reading: For a complete list of valid color names and a more in-depth description of the HSL model, read the current CSS3 Color Module Specification at <http://go.microsoft.com/fwlink/?LinkId=267728>

Defining Backgrounds and Effects

CSS enables you to set a variety of background properties on many elements. A number of properties are available that enable you to specify a color for the background or for an image, along with how that image is repeated (this was described in module 2). In CSS3, there are two significant additions to these possibilities.

Multi-Image Backgrounds

CSS3-compliant browsers support the use of multiple background images within an element. Consequently, every background image-related property in CSS3 now takes a comma-separated list of values rather than just one, as shown in the following example:

```
article {
    background-image: url('bluearrow.png'), url('greenarrow.png');
    background-repeat: repeat-x, repeat-y;
}
```

CSS3 supports:

- Multi-image backgrounds

```
article {
    background-image: url('bluearrow.png'), url('greenarrow.png');
    background-repeat: repeat-x, repeat-y;
}
```

Gradients

```
background: linear-gradient(direction, start-color, [mid-color-list], end-color);
background: radial-gradient(top right, ellipse, red, blue);
```

In this rule, `bluearrow.png` will be repeated left to right along the top edge of the `article` element and `greenarrow.png` will be repeated top to bottom along the left edge of the `article` element. The image

declared first in the list appears on top of the others, so bluearrow.png appears above greenarrow.png in the top left corner of the **article** element.

The following image shows this effect applied to the About page in the ContosoConf web application:

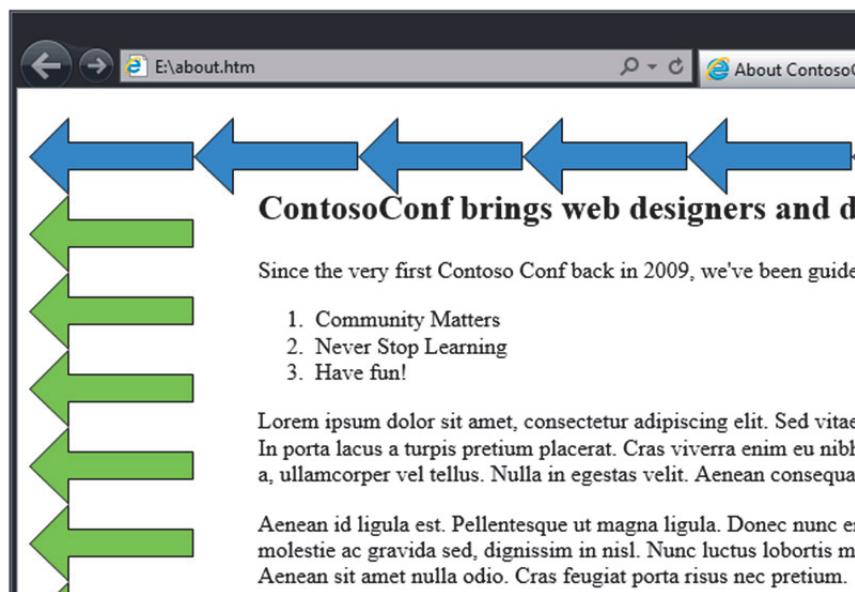


FIGURE 6.4:THE ABOUT PAGE STYLED WITH A REPEATED MULTI-IMAGE BACKGROUND

You can use the **background-position** to move each image around the other to achieve a specific effect.

Note that the **background** shorthand property also supports multiple images. The previous rule could also be written like this:

```
article {
    background: url('bluearrow.png') repeat-x, url('greenarrow.png') repeat-y;
}
```

Gradients

CSS3 enables you to define a color gradient as a value for any property that would take an image. Most obviously, this would apply to backgrounds. You can set two types of gradient:

- A **linear gradient**, which is a gradual change in color from the start color to the stop color. By default, the start color is displayed at the top of the background and the end color at the bottom, although the direction of the gradient may be changed.

```
background: linear-gradient(direction, start-color, [mid-color-list,] end-color);
```

The **direction** parameter is optional and is set in degrees; the default is 180deg. The **start-color** and **end-color** parameters can be set to any of the color values listed in the previous topic. You can also set any number of intermediate colors between the start and end, which the browser will space out evenly over the gradient line. The following CSS rule sets the background of the HTML page by using a linear gradient. Note that Internet Explorer uses the **-ms** prefix for the **linear-gradient** function.

```
html {
    background: -ms-linear-gradient(30deg, lightblue, green, yellow);
}
```

The following image shows this effect applied to the About page of the ContosoConf web application.

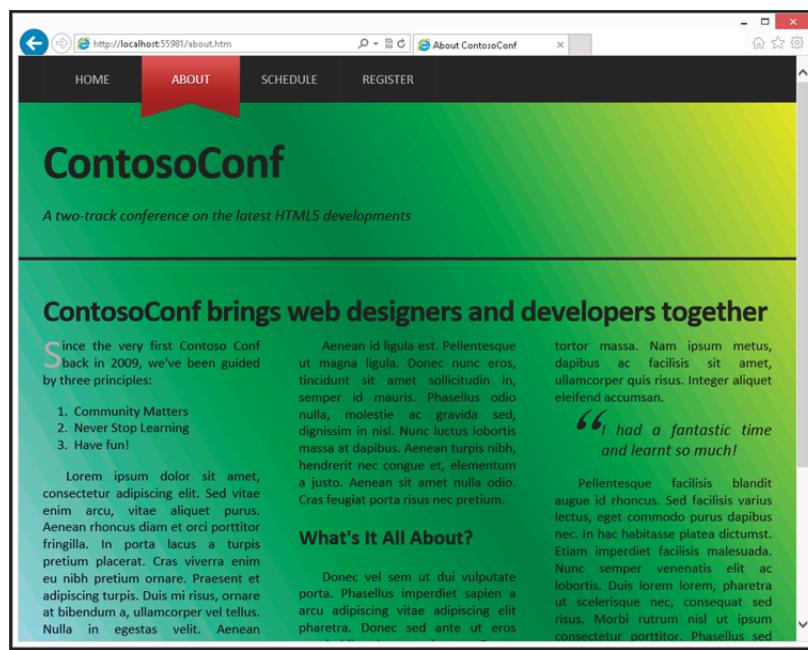


FIGURE 6.5:THE ABOUT PAGE STYLED WITH A LINEAR-GRADIENT BACKGROUND

- A **radial gradient**, which is a gradual change in color from a central point in the start-color outwards in either a circle or an elliptical shape to the end color at the edge of the shape. Any number of intermediate colors can be set in the list.

```
background: radial-gradient(position, shape, start-color, [mid-color-list,] end-color);
```

The **position** parameter is optional. Its default value is **center**. The **shape** parameter is also optional. Its default value is **circle**; the only other option is **ellipse**.

```
html {
    background: -ms-radial-gradient(top right, ellipse, red, blue);}
```

Note that most modern browsers, with the exception of Internet Explorer 9 and earlier versions, support gradients, but only with vendor-prefixes attached.

Implementing Transformations and Graphics

In addition to generating graphics by using gradients, CSS3 also enables you to apply graphical transformations such as rotation and skew to any element. By transforming stacked and adjacent color blocks, you can draw a number of shapes simply by using CSS.

Transforming Elements by Using CSS3

The CSS3 transform property can be applied to any block-level element. You set its value to either one of the following transformation functions, or to **none** to indicate no transform should be applied:

Using CSS3, you can:

- Transform, rotate, and skew elements

```
article {
    transform: rotate(30deg);}
```


- Generate shapes

```
#circle {
    width: 200px;
    height: 200px;
    background: blue;
    border-radius: 50%;}
```

- **translate3d(x,y,z)** : Moves the whole element by the distance **x** along the x-axis, **y** along the y-axis, and **z** along the z-axis. The values for **x**, **y**, and **z** can be any valid unit of measurement.

```
transform: translate3d(10px, 50px, 10px);
```

- **translate(x,y)** : 2d variant of **translate3d()**.
- **translateX(x)**, **translate(y)**, **translateZ(z)** : Single axis variants of **translate3d()**.
- **scale3d(x,y,z)** : Scales the element's size by a factor **x** along the x-axis, **y** along the y-axis, and **z** along the z-axis. The values for **x**, **y**, and **z** must be positive numbers (use a decimal value less than 1 to shrink an element in a given dimension).

```
transform: scale3d(2, 4, 0.5);
```

- **scale(x,y)** : 2d variant of **scale3d()**.
- **scaleX(x)**, **scaleY(y)**, **scaleZ(z)** : Single axis variants of **scale3d()**.
- **rotate3d(x,y,z,a)** : Rotates an element in 3d by angle **a** around the point with co-ordinates (**x,y,z**). Angle **a** is a value in degrees.
- **rotate(a)** : Rotates an element in 2d by angle **a** around its center.

```
transform: rotate(30deg);
```

- **skew(a,b)** : Skews an element by angle **a** along the x-axis and angle **b** along the y-axis. **a** and **b** are values in degrees between 0 and 180.
- **skewX(a)**, **skewY(b)** : Single axis-variants of **skew()**.

```
transform: skew(15deg, 15deg);
```

Note that most modern browsers, with the exception of Internet Explorer 9 and earlier versions, support transforms, but only with vendor-prefixes attached.

```
transform: rotate(30deg);
-ms-transform: rotate(30deg);           // IE10
```

 **Additional Reading:** For an in-depth discussion of 3D transforms in IE10, see <http://go.microsoft.com/fwlink/?LinkId=267729>.

Drawing Shapes by Using CSS3

One of the interesting results of combining **height**, **width**, and **border** values, the **before** and **after** pseudo-elements, and some rotation transforms, are the number of shapes that you can generate simply by using CSS. For example, to draw a square or rectangle, just combine the **height** and **width** properties with a **background** color:

```
HTML:
<div id='square'></div>
...
CSS:
#square {
    width: 200px;
    height: 200px;
    background: blue;
}
```

You can draw shapes such as circles and ovals by using the **border-radius** property to add curves to a square.

```
HTML:  
<div id='circle'></div>  
...  
CSS:  
#circle {  
    width: 200px;  
    height: 200px;  
    background: blue;  
    border-radius: 50%;  
}
```

You can draw triangles like this:

```
HTML:  
<div id='triangle-topleft'></div>  
...  
CSS:  
#triangle-topleft {  
    width: 0;  
    height: 0;  
    border-top: 200px solid blue;  
    border-right: 200px solid transparent;  
}
```

This HTML markup and styling draws the following shapes:

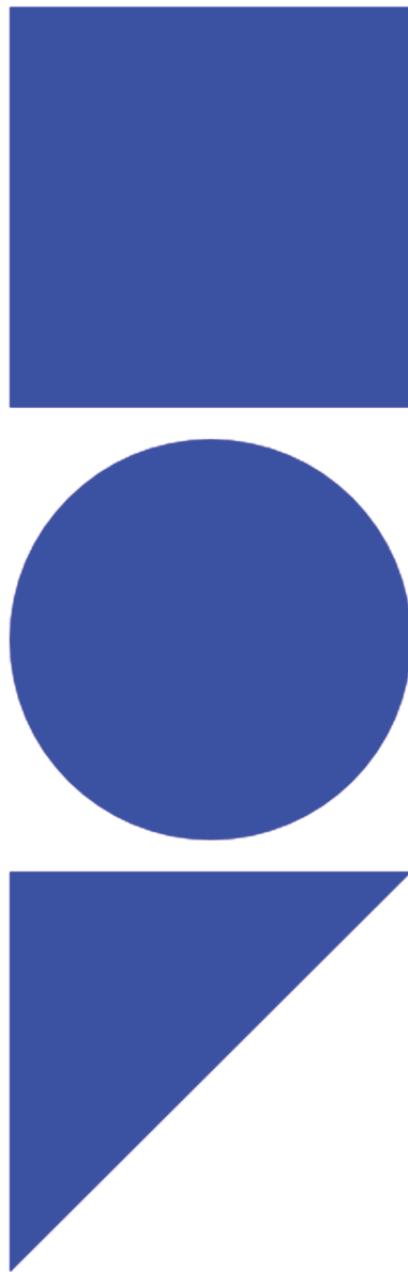


FIGURE 6.6:SHAPES DRAWN BY USING CSS STYLES

Adding 2D transforms such as rotations and skews enables you to create stars, parallelograms, and many more complex shapes.

Demonstration: Styling Text and Block Elements by Using CSS3

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Styling Text and Block Elements by Using CSS3

Scenario

The Contoso Conference web application needs to be visually appealing. A designer has produced mock-up designs of some of the pages that you have been asked to implement for the website.

You will be working on the Home and About pages. The HTML page structure has already been created. You will use CSS to style various parts of the pages, to make them match the designs. Much of the CSS that you create, such as the navigation links bar, will be reused by other pages.

Some aspects of the design are complicated and would have required images with previous versions of CSS. However, by using CSS3, you will not need to create any images.

Objectives

After completing this lab, you will be able to:

1. Implement advanced styling for text elements by using CSS.
 2. Style block elements by using CSS.
 3. Create graphical elements by using CSS.
- Estimated Time: 60 minutes
 - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
 - User Name: Student
 - Password: Pa\$\$w0rd

Exercise 1: Styling the Navigation Bar

Scenario

In this exercise, you will style the navigation bar for the website.

You will use CSS to style the navigation bar to look similar to the following image:

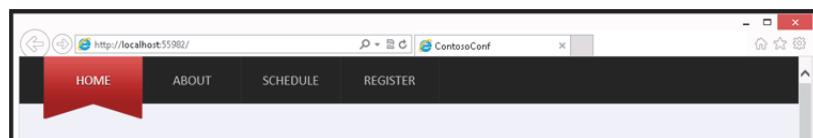


FIGURE 6.7:THE NAVIGATION BAR

The HTML markup for the navigation bar is simply a collection of `<a>` elements. The links are arranged as horizontally stacked blocks, which maximizes the click area. You will style the active page link with a linear gradient and red ribbon effect.

Finally, you will run the application, view the Home page, and verify that the navigation bar looks similar to the above image.



Note: The layout of the Home page has also changed slightly. The images of the speakers and the sponsor's logos have been laid out in a grid by using the Flexible Box Model display style.

The main tasks for this exercise are as follows:

1. Review the HTML structure.
2. Style the navigation bar and links.

3. Create graphics by using pseudo elements.

4. Test the navigation bar.

► Task 1: Review the HTML structure

1. Start **the MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution from the **E:\Mod06\Labfiles\Starter\Exercise 1** folder.
4. In the **ContosoConf** project, open the **index.htm** file.

Notice that the **<head>** has a link to the **/styles/nav.css** style sheet:

```
<link href="/styles/nav.css" rel="stylesheet" type="text/css" />
```

5. Find the **<nav>** element, at the start of the **<body>**.

Notice that the class of the **<nav>** element is **page-nav**, and that it contains a **<div>** element with the class **container**. Also notice that the Home link has the class **active** because it is the active page:

```
<nav class="page-nav">
  <div class="container">
    <a href="/index.htm" class="active">Home</a>
    ...
  </div>
</nav>
```

6. Run the application, view the Home page, and notice that the navigation links are currently unstyled.

The navigation bar looks like this:



FIGURE 6.8:THE UNSTYLED NAVIGATION BAR

7. Close Internet Explorer.

► Task 2: Style the navigation bar and links

1. Open the **nav.css** style sheet in the styles folder.
2. Add styles for the navigation bar as follows:
 - Replace the comment `/* TODO: nav.page-nav */` with a rule that styles **nav** elements that have the **page-nav** class (**nav.page-nav**). Set the background color to `#1d1d1d`, set the line height to `6rem`, and set the font size to `1.7rem`.
 - Replace the comment `/* TODO: nav.page-nav .container */` with a rule that styles elements that have the **container** class inside **nav.page-nav** elements (**nav.page-nav .container**). Style the container to use the Flexible Box Layout Model.



Reader Aid: Internet Explorer requires you to use **-ms-flexbox**; note the **-ms-**vendor prefix.

The navigation bar should look like this:

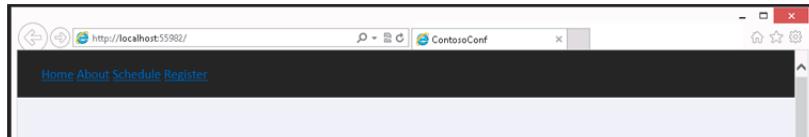


FIGURE 6.1:THE NAVIGATION BAR WITH INITIAL STYLING

3. Replace the comment /* TODO: nav.page-nav a */ with a style for the links for the navigation bar by using the **nav.page-nav a** selector, as follows:
 - Set the display style to **block**.
 - Set the minimum width to **9rem**.
 - Set the top and bottom paddings to **0**, and the left and right paddings to **1.8rem**.
 - Set the text alignment to **center**.
 - Transform the text of all links to upper case.
 - Set the link color to **#c3c3c3**.
 - Add a **1px** dotted border on the right of each link, using the color **#3d3d3d**.
 - Add a small, black text shadow.
 - Display the text for the links in uppercase.

The links in the navigation bar should look like this:

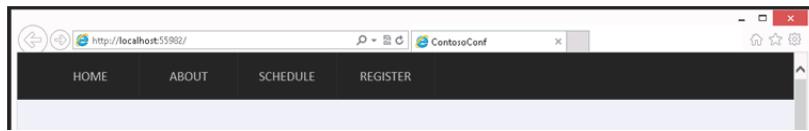


FIGURE 6.2:THE NAVIGATION BAR WITH STYLED LINKS

4. Replace the comment /* TODO: nav.page-nav a:first-child */ with a CSS rule that displays a **1px** dotted border with the color **#3d3d3d** to the left of the first link in the navigation bar. (Use the **first-child** pseudo-element to select the first link).
5. Replace the comment /* TODO: nav.page-nav a:hover */ with a CSS rule that applies when a mouse hovers over a navigation link. Make the text color **#e4e4e4** and the background color **black**. (Use the **hover** pseudo-element).
6. Replace the comment /* TODO: nav.page-nav .active */ with a CSS rule for the active navigation link. Make the text color white and the background a linear gradient from **#c95656** to **#8d0606**.
7. Replace the comment /* TODO: nav.page-nav .active:hover */ with a CSS rule that overrides the text color when the mouse hovers over the active link. Make the text white.

► Task 3: Create graphics by using pseudo elements

1. Replace the comments /* TODO: nav.page-nav active:before */ and /* TODO: nav.page-nav active:after */ with two CSS rules, using the **:before** and **:after** pseudo elements that generate triangle shapes below the active link, to simulate a ribbon effect.
 - Set the **display** property to **block**.
 - You may need to experiment with the **position**, **top**, **height**, **width**, **border-top**, **border-left**, **border-right**, and **margin-left** properties until you achieve the correct effect.
 - Set the color of the triangles to **#8d0606**

The following example CSS generates a black triangle:

```
.example:after {
    content: "";
    border-top: 100px solid #000;
    border-left: 100px solid transparent;
    display: block;
    position: absolute;
}
```

The navigation bar should look like this:

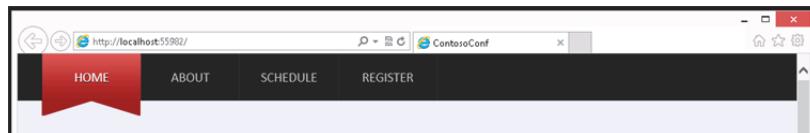


FIGURE 6.3:THE FULLY STYLED NAVIGATION BAR

► **Task 4: Test the navigation bar**

1. Run the application, view the **home.htm** page and verify that the navigation bar looks similar to the image shown in the previous task.
2. Use the navigation bar to move between pages (for example, move to the **About** page), and verify that the style of the active item in the navigation bar is displayed by using the ribbon effect.
3. Close Internet Explorer.

Results: After completing this exercise, you will have styled the navigation bar to match the design mockup.

Exercise 2: Styling the Register Link

Scenario

In this exercise, you will style the large **Register** link that appears in the header of the Home page. This link is unstyled, but you have been asked to make it stand out so that users will notice it.

A designer has provided the following image showing how the **Register** link should appear:

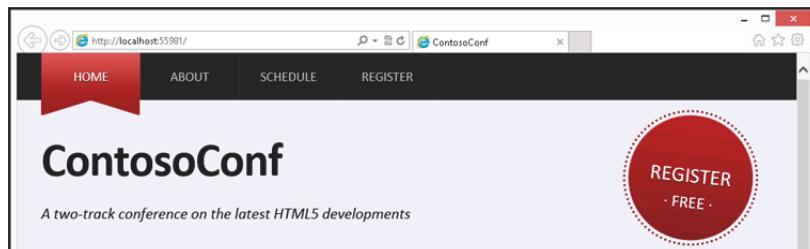


FIGURE 6.4:THE REGISTER LINK IN THE HEADER OF THE HOME PAGE

You will use a style to set the position of the **Register** link. You will modify the appearance of the text for the link, add a background gradient, rotate the link, and add the circular dotted border. Finally, you will run the application, view the Home page, and verify that the **Register** link is similar to that envisioned by the designer.

The main tasks for this exercise are as follows:

1. Review the HTML and CSS.
2. Position the Register link and set the text styling.

MATRICULE ONLY STUDENT USE PROHIBITED

3. Style the Register link background, shape, and rotation properties.
4. Test the Register link.

► **Task 1: Review the HTML and CSS**

1. Open the **Contoso.conf** solution in the **E:\Mod06\Labfiles\Starter\Exercise 2** folder.
2. In the **ContosoConf** project, open the **index.htm** file.
3. Notice that the **<head>** element contains a link to the **header.css** style sheet in the **styles** folder:

```
<link href="/styles/header.css" rel="stylesheet" type="text/css" />
```

4. Find the **<header>** element and review the contents. Notice that the header contains the HTML markup for the **Register** link. Also notice that the class of the header is **page-header** and that the class of the Register link is **register**:

```
<header class="page-header">
    <div class="container">
        <h1>ContosoConf</h1>
        <p class="tag-line">A two-track conference on the latest HTML5
        developments</p>
        <a class="register" href="/register.htm">
            Register<br />
            <span class="free">Free</span>
        </a>
    </div>
</header>
```

5. Run the application, view the **index.htm** page, and verify that the header's **Register** link is not styled.
6. Close Internet Explorer.

► **Task 2: Position the Register link and set the text styling**

1. Open the **header.css** file in the **styles** folder.
2. Find the currently empty CSS rule for the **register** class of the page header after the comment `/* TODO: header.page-header .register */`. In this rule, implement the following styling:
 - Use the block layout model for the **Register** link and give it a height of **10rem** and a width of **16rem**. Set the **right** property to **3.5rem**, the **top** property to **2rem**, and the **padding-top** property to **6rem**.
 - Position the **Register** link near the top-right of its container.
3. Modify the **header.page-header .register** rule to set the **font-size**, **text-alignment**, **text-decoration**, and **text-transformation** properties for the **Register** link:
 - Make the text appear in upper case with a font size of **2.7rem**.
 - Set the text color to white.
 - Add a **1px** black text shadow.
 - Set the text alignment to **center**.
 - Do not apply any additional text decoration.

If you run the application at this point, the Register link will look like this in the page header:

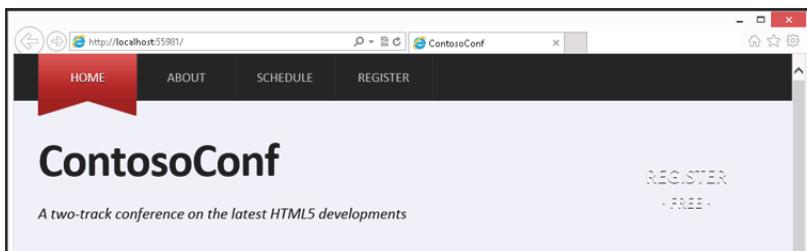


FIGURE 6.5:THE REGISTER LINK

- ▶ **Task 3: Style the Register link background, shape, and rotation properties**
1. The Register link is currently quite difficult to see. Modify the **header.page-header .register** rule as follows:
 - o Add a linear gradient background to the **Register** link. Use the colors **#a80000** and **#740404** (use the **-ms-linear-gradient** function to set the **background** property).
 - o Set the **-ms-border-radius** property to make the **Register** link circular.
 - o Rotate the **Register** link 6 degrees clockwise (use the **-ms-transform** property).
 2. Implement the **header.page-header .register:hover** CSS rule, to change the background linear gradient colors to **#bc0101** and **#8c0909** when the mouse hovers over the link.
 - o Set the **background** property by using the **-ms-linear-gradient** function.
 3. Implement the **header.page-header .register:before** CSS rule to create a **3px** circular, dotted border around the **Register** link. Specify the border color **#740404**.
 - o Set the **-ms-border-radius** property to create the circular effect.
 - o Make the border slightly wider and taller than the **Register** link (set the **height** and **width** properties to **16.8rem**).
 - o Adjust the position of the border to center it on the link (adjust the **top** and **right** properties by -0.7rem).
- ▶ **Task 4: Test the Register link**

1. Run the application and view the **index.htm** page:

The Register link should look like this:

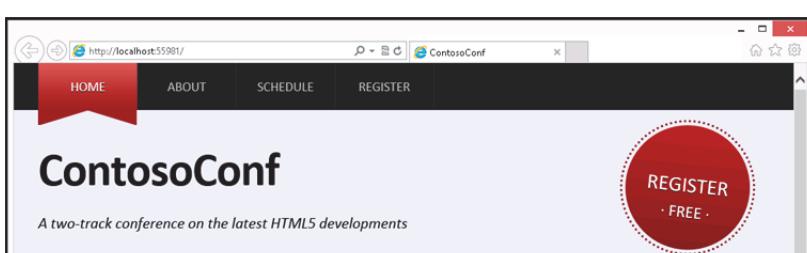


FIGURE 6.6:THE STYLED REGISTER LINK

2. Verify that the **Register** link changes color when the mouse hovers over it.
3. Close Internet Explorer.

Results: After completing this exercise, you will have styled the **Register** link in the header of the Home page.

Exercise 3: Styling the About Page

Scenario

In this exercise, you will style the **About** page. This page only contains text, but to make it look attractive you will use some advanced typography styling.

First you will flow the text over three columns and add a "drop cap" style to the first letter. Then you will style a testimonial quote. Finally, you will run the application, view the About page, and verify that it looks like the following image:

The About page should look like this when it is completed:

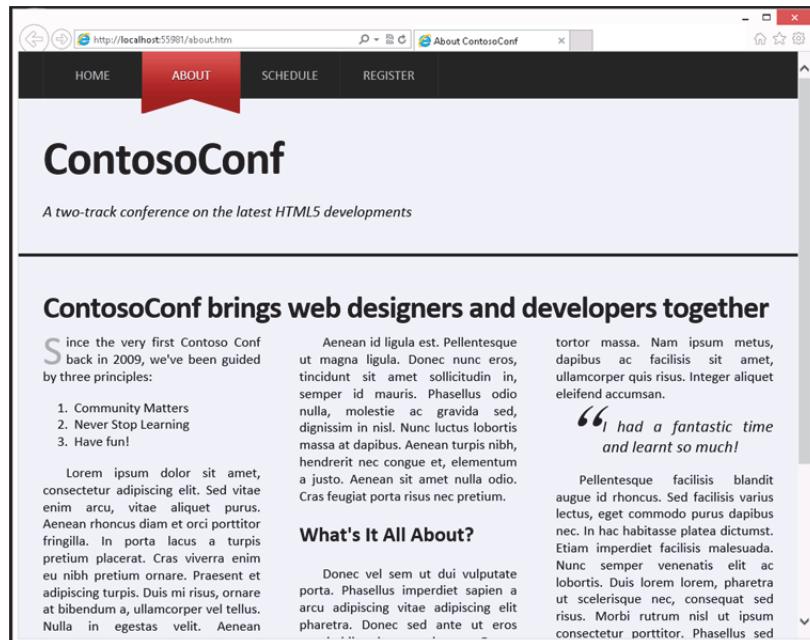


FIGURE 6.7:THE ABOUT PAGE

The main tasks for this exercise are as follows:

1. Review the HTML and CSS.
2. Define text columns.
3. Add a drop cap at the start of the text.
4. Indent paragraphs.
5. Style the block quote.
6. Test the About page.

► Task 1: Review the HTML and CSS

1. Open the **Contoso.conf** solution in the **E:\Mod06\Labfiles\Starter\Exercise 3** folder.
2. In the **ContosoConf** project, open the **about.htm** file.
3. In the **<head>** element notice there is a link to the **about.css** style sheet in the **/styles/pages** folder:

```
<link href="/styles/pages/about.css" rel="stylesheet" type="text/css" />
```
4. Review the **<article class="container">** HTML markup. This article contains the text that is displayed in the body of the **About** page. You will add styling to this markup:

```
<section class="page-section about">
```

```
<article class="container">
  ...
</article>
</section>
```

► **Task 2: Define text columns**

1. Open the **about.css** file in the **styles/pages** folder.
2. After the comment /* TODO: .about > article > section */, implement the **.about > article > section** CSS rule as follows:
 - Flow the content over three columns.
 - Add a gap of **5rem** between the columns.
 - Justify the text.

► **Task 3: Add a drop cap at the start of the text**

1. In **about.css**, after the comment /* TODO: Add drop cap styling */, add a CSS rule that makes the first letter of the first paragraph larger as follows:
 - Use the **first-letter** pseudo property to select the first letter of the text.
 - Make the first letter three times bigger than the rest of the text (use the **font-size** property).
 - Use the **float** property to make the rest of the text wrap around the first letter.
 - Set the **color** property to **#aaa**
 - Experiment with the **margin** and **line-height** properties until the layout of the text matches the following image:

The first paragraph of text should look like this:

Since the very first Contoso Conf back in 2009, we've been guided by three principles:

FIGURE 6.8:THE FIRST PARAGRAPH ON THE ABOUT PAGE

► **Task 4: Indent paragraphs**

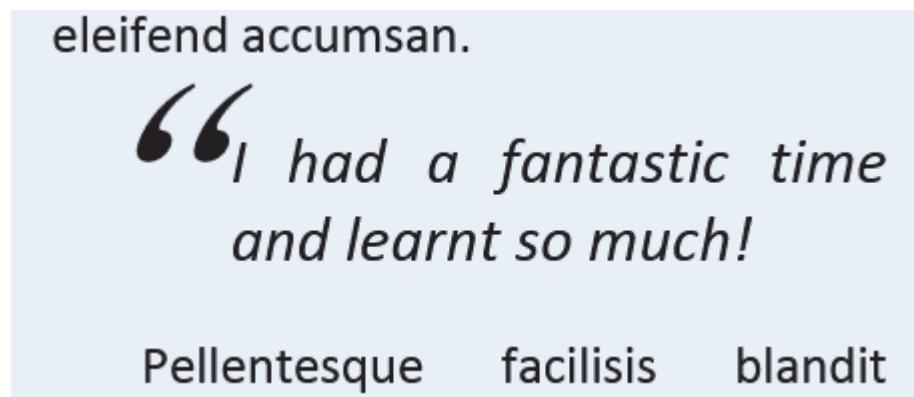
1. In **about.css**, after the comment /* TODO: Indent paragraphs */, add a CSS rule that uses the **text-indent** property to indent the first line of each paragraph by 3rem.
2. The first paragraph does not require indentation because it already has a drop cap effect; add a CSS rule specifically for the first paragraph (use the **.about p:first-child** selector), which resets the indentation and margin.

► **Task 5: Style the block quote**

1. In **about.css**, after the comment /* TODO: Blockquote */, add CSS rules that style **blockquote** elements to look like the following image:

The quotes should look like this:

MCT USE ONLY STUDENT USE PROHIBITED

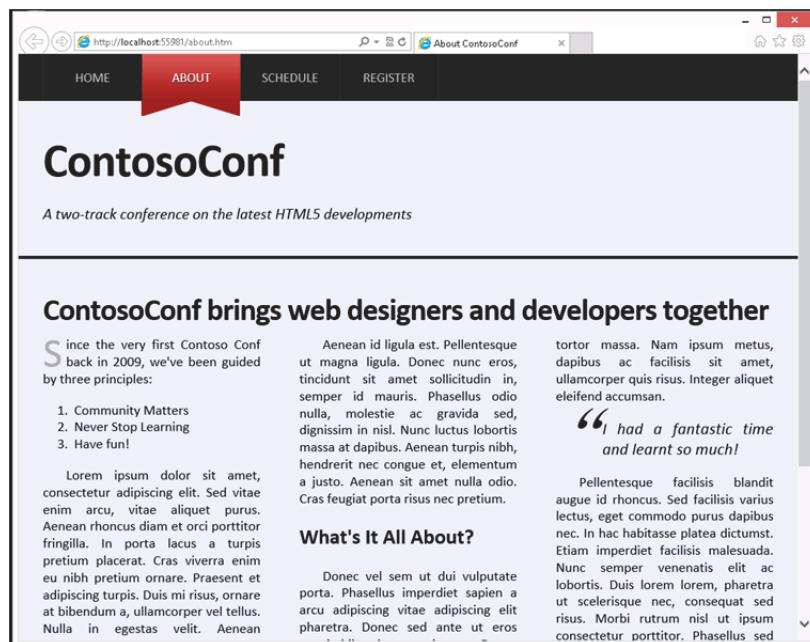
**FIGURE 6.9:A STYLED BLOCK QUOTE**

- Use the selector **.about blockquote** to create a rule that sets the **font-size (1.2em)**, **padding (0 0 0 6rem)**, **margin (0)**, **style (italic)** and position (**relative**) properties.
- Add another rule that uses the a **:before** pseudo element to generate the large double quotation mark; use the sequence '**\201C**' for the quotation mark, set the font size to **10rem**, and the font family to **serif**. Set the **position** property to **absolute** and experiment with values for the **top** and **line-height** properties ensure that the text appears with the appropriate spacing.

► Task 6: Test the About page

1. Run the application and view the **About** page.

The About page should look like this:

**FIGURE 6.10:THE ABOUT PAGE**

2. Close Internet Explorer.

Results: After completing this exercise, you will have styled the text on the **About** page.

Module Review and Takeaways

In this module, you have learned how to use the new features of CSS3 to provide advanced formatting for text, color, and backgrounds. You have also seen how CSS pseudo-elements and pseudo-classes offer a very fine level of control when it comes to selecting elements to style.

You have also learned how CSS3 offers alternative ways to position elements on a page that are in addition to the traditional box model, and how to use CSS properties in combination to generate simple graphics.

Review Question(s)

Test Your Knowledge

Question	
Which CSS rule can you use to download a font required by a web page?	
Select the correct answer.	
	@font-family
	@font-style
	@font-face
	@font
	You cannot download fonts by using CSS.

Question: What are the main differences between the CSS box model, flex-box, and multi-column layout?

Question: How do you select the first item in a list so that you can apply styling to it?

METHOD USE ONLY. STUDENT USE PROHIBITED

Module 7

Creating Objects and Methods by Using JavaScript

Contents:

Module Overview	7-1
Lesson 1: Writing Well-Structured JavaScript Code	7-2
Lesson 2: Creating Custom Objects	7-6
Lesson 3: Extending Objects	7-12
Lab: Refining Code for Maintainability and Extensibility	7-17
Module Review and Takeaways	7-22

Module Overview

Code reuse and ease of maintenance are key objectives of writing well-structured applications. If you can meet these objectives, you will reduce the costs associated with writing and maintaining your code.

This module describes how to write well-structured JavaScript code by using language features such as namespaces, objects, encapsulation, and inheritance. These concepts might seem familiar if you have experience in a language such as Java or C#, but the JavaScript approach is quite different and there are many subtleties that you must understand if you want to write maintainable code.

Objectives

After completing this module, you will be able to:

- Write well-structured JavaScript code.
- Use JavaScript code to create custom objects.
- Implement object-oriented techniques by using JavaScript idioms.

Lesson 1

Writing Well-Structured JavaScript Code

As web applications grow in size, it becomes increasingly important to properly structure your code so that you can maintain it more easily. This lesson will show you several techniques to help you achieve this goal.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the **scoping** rules for local variables, and describe how **hoisting** works.
- Use immediately **invoked functions**, **strict mode**, and **namespaces** to minimize global name clashes in a web application.
- Use **common global objects** and **functions** available in the standard JavaScript language.

Scoping and Hoisting

JavaScript has many syntactic similarities to other popular contemporary programming languages such as C#, C++, and Java. However, there are several subtle differences that often surprise experienced developers, because the syntax looks similar but has a different meaning.

One area of confusion for some developers concerns the concept of scoping and hoisting. The term scope describes the visibility of a variable.

When you declare a variable in JavaScript, it has one of two scopes:

- A variable has global scope if it defined outside of a function.
- A variable has function scope if it is defined inside a function.

Scope in JavaScript is subtly different from scope in C#, C++, or Java. Specifically, JavaScript does not support the concept of block-scoped variables. If you declare a variable inside a block, the visibility of the variable is elevated to function scope, as if the variable was declared at the start of the function. This language feature, whereby variables declared inside a block are elevated to function scope, is known as **hoisting**.

The following example demonstrates how scoping and hoisting work in JavaScript. Note the following points:

- The example declares a global variable named **num** and assigns it the value 7.
- The **demonstrateScopingAndHoisting()** function declares a local variable named **num** inside an **if** statement block, and assigns it the value 42. This variable is hoisted automatically to function scope, as if it was declared at the start of the **demonstrateScopingAndHoisting()** function.
- After the **if** statement, the local **num** variable is still in scope. Therefore the **alert()** function displays the local **num** variable (42), not the global **num** variable (7).

```
<script>
```

```

var num = 7;
function demonstrateScopingAndHoisting() {
    if (true) {
        var num = 42;
    }
    alert("The value of num is " + num);
}
</script>

```

Managing the Global Namespace

If you are familiar with C, C#, Java, Visual Basic, or almost any contemporary programming language, you will be aware of the concept of using namespaces to avoid name clashes.

Global name clashes can be a big problem in JavaScript. Any global variables that you declare in your code will be accessible by all the JavaScript code in the web application. Given this fact, it is almost inevitable that you will sooner or later define a variable name that clashes with another global variable, no matter how hard you try to adopt a naming convention aimed at ensuring the uniqueness of variable names. In addition, your JavaScript applications may use third-party libraries that define their own global variables, and you have little control (if any) over how these variables are named.

JavaScript provides several mechanisms that help you to avoid global name clashes, including:

- Immediately invoked functions
- Namespaces
- Strict mode

- Global name clashes can be problematic in JavaScript
 - Your global variables might conflict with other global variables elsewhere in the web application
- JavaScript provides several mechanisms to avoid global name clashes
 - Immediate functions
 - Namespaces
 - Strict mode

Immediately Invoked Functions

An immediately invoked function is a function that is defined and run immediately. You define an immediately invoked function by wrapping it in an anonymous function call that is immediately executed. The following example shows the syntax for an immediately invoked function:

```

(function() {
    // Variables defined inside the function disappear when the function finishes
    // - they will not conflict with variables defined by other scripts
    var localVar = ... ;
    var localVar2 = ...;
    // The same logic applies to functions
    // - they are destroyed when the immediately invoked function finishes
    function localFunc() {
        localVar = 99;
        ...
    }
    ...
    localFunc(); // Run localFunc
    ...
})();

```

An immediately invoked function is run as soon as it has been defined. Any variables and other functions created inside the function body are scoped to the immediately invoked function, and they disappear as

soon as the immediately invoked function has finished. In this way, you can guarantee that these variables and functions will not pollute the global namespace.

Many JavaScript programmers use this technique to ensure that the JavaScript code specific to a web page is isolated from the JavaScript code for other web pages.

Namespaces

JavaScript namespaces provide another way to avoid global name clashes. In JavaScript, a namespace is a global variable with variables and functions attached to it.

The following example shows how to define a namespace named **MyNamespace**, containing two functions and two variables:

```
var MyNamespace = {
    myFunction1: function(someParameters) {
        // Implementation code...
    },
    myFunction2: function(someParameters) {
        // Implementation code...
    },
    message: "Hello World",
    count: 42
}
```

Outside of the namespace declaration, you can access its functions and variables by using a qualified name that specifies the namespace. The following example shows how to access namespace members:

```
MyNamespace.myFunction1(someParameterValues);
MyNamespace.message = "Goodbye all";
```

Namespaces do not provide any privacy or encapsulation; the members defined inside a namespace are completely visible to external code, by using properly qualified names such as those shown in the previous example.

Strict Mode

The rules for declaring variables in JavaScript are fairly relaxed. If you omit the **var** keyword in a variable declaration, the variable is implicitly given global scope. This means you can accidentally declare a new global variable without realizing it, as illustrated in the following example:

```
function someFunction() {
    var errorCode = 100; // Declares a local variable named errorCode.
    count = 0;           // Implicitly declares a global variable named count;
    ...
}
```

To avoid accidentally declaring global variables by omitting the **var** keyword, you can use strict mode as follows:

```
function someFunction() {
    "use strict";
    // Other statements.
}
```

When you use strict mode, you will get an error if you try to declare a variable without using **var**; JavaScript will not automatically promote the variable to global scope.

Singleton Objects and Global Functions in JavaScript

One of the best known design patterns in object-oriented development is the singleton pattern. The singleton pattern describes how to ensure that there is only ever a single instance of a class in existence. Typical uses of the singleton pattern might include the following classes:

- A database driver manager class that is responsible for choosing which database driver to use to open a connection to a database.
- A screen manager class that is responsible for organizing the layout of windows in a single screen.
- A mathematical class that provides algebraic and trigonometric functions such as sin, cos, and tan.

JavaScript supports the singleton pattern, and there are several global singleton objects that come as part of the standard JavaScript library, such as **Math** and **JSON**.

- The **Math** object provides mathematical functions and constants. You access these functions and constant values directly through the **Math** object; you do not create a **Math** object first. The following example shows how to access some of the members of the **Math** object:

```
var radius = 100 * Math.random();
var circumference = 2 * Math.PI * radius;
var area = Math.PI * Math.pow(radius, 2);
```

- The **JSON** object provides methods for converting values to JavaScript Object Notation (JSON) strings, and for converting JSON strings back to values. The following example shows how to use the **JSON** object:

```
var anObject;
...
var anObjectAsJsonString = JSON.stringify(anObject);
var anObjectAgain = JSON.parse(anObjectAsJsonString);
```

JavaScript also provides a set of global common functions and properties that can be used with all the built-in JavaScript objects, such as **parseInt()**, **parseFloat()**, and **isNaN()**. The following example shows how to use these global functions:

```
var ageEnteredByUser;
var heightEnteredByUser;
...
var age = parseInt(ageEnteredByUser);
var height = parseFloat(heightEnteredByUser);
if (isNaN(age) || isNaN(height))
    alert("Invalid input");
```

 **Note:** The JavaScript global functions are actually functions belonging to the **Global** object. This is another singleton object. The functions of the **Global** object are intrinsic to JavaScript, and you do not need to qualify them with **Global**.

- JavaScript defines several singleton objects, such as:
 - **Math**
 - **JSON**

- JavaScript also defines global functions, such as:
 - **parseInt()**
 - **parseFloat()**
 - **isNaN()**

Lesson 2

Creating Custom Objects

In addition to the built-in JavaScript objects such as **String**, **Date**, **Math**, **JSON**, and **RegExp**, you can also create your own custom objects. For each object, you can specify the properties that the object needs to hold, and the functionality that the object needs to implement.

Lesson Objectives

After completing this lesson, you will be able to:

- Create custom objects that contain properties and methods.
- Use object literal notation to define the properties of objects.
- Define constructor functions to assign a common set of properties to objects.
- Use prototypes to implement object behavior.
- Use the **Object.create()** function to create objects based on an existing prototype.

Creating Simple Objects

JavaScript defines a standard object named **Object**. You can create a new object by using the following simple syntax:

```
var employee1 = new Object();
```

This statement creates an object with no properties and very limited functionality (it only has the methods provided by the **Object** type), and assigns the object to a variable named **employee1**. You can use the **employee1** variable to access the object.

- There are several ways to create new objects in JavaScript:

```
var employee1 = new Object();
var employee2 = {};
```

- You can define properties and methods on an object:

```
var employee1 = {};
employee1.name = "John Smith";
employee1.age = 21;
employee1.salary = 10000;

employee1.payRise = function(amount) {
    // Inside a method, "this" means the current object.
    this.salary += amount;
    return this.salary;
}
```

A simpler way to create an object is to use braces {}. The following example is semantically equivalent to the first example:

```
var employee2 = {};
```

Adding Properties to an Object

Empty objects are not very useful, so JavaScript enables you to add properties to an object by using the following syntax:

```
objectReference.propertyName = value;
```

A property can either hold a data value or refer to a function. If the property refers to a function, the property is known as a method. Inside a method, the keyword **this** refers to the object upon which the method was invoked (usually the object containing the method). The following example shows how to add some data properties to an **employee** object, and how to add a method to implement behavior on the object:

```
var employee1 = {};
employee1.name = "John Smith";
employee1.age = 21;
```

```
employee1.salary = 10000;
employee1.payRise = function(amount) {
    // Inside a method, "this" means the current object.
    this.salary += amount;
    return this.salary;
}
```

 **Note:** JavaScript does not support overloaded functions. This is because JavaScript has no notion of function signatures. If you define a function for an object that has the same name as an existing function, the new function will replace the existing function. The situation is analogous to assigning a new value to a data property; the old value is overwritten with the new value.

Accessing Properties on an Object

To access a property or invoke a method on an object, use the following syntax:

```
objectReference.propertyName = value;
objectReference.functionName(parameters);
```

The following example shows how to access data properties and invoke methods on an **employee** object:

```
var newSalary = employee1.payRise(1000);
document.write("New salary for employee1 is " + newSalary);
```

Using Object Literal Notation

Object literal notation provides a shorthand way to define an object and set its properties. Object literal notation has the following syntax:

```
var objectName = {
    property1: value1,
    property2: value2,
    ...
};
```

The following example shows how to create a new object and define its properties:

```
var employee1 = {
    name: "John Smith",
    age: 21,
    salary: 10000
};
```

You can also define methods as part of the object definition. Implement each method inline by using the syntax **function(){...}**. Within the function, the **this** keyword refers to the target object. The following example shows how to create an object that contains properties and methods. The methods reference properties that are part of the same object:

```
var employee2 = {
    name: "Mary Jones",
    age: 42,
    salary: 20000,
    payRise: function(amount) {
        this.salary += amount;
    }
};
```

- Object literal notation provides a shorthand way to create new objects and assign properties and methods:

```
var employee2 = {
    name: "Mary Jones",
    age: 42,
    salary: 20000,
    payRise: function(amount) {
        this.salary += amount;
        return this.salary;
    },
    displayDetails: function() {
        alert(this.name + " is " + this.age + " and earns " + this.salary);
    }
};
```

```
        return this.salary;
    },
    displayDetails: function() {
        alert(this.name + " is " + this.age + " and earns " + this.salary);
    }
};
```

Using Constructors

In JavaScript, a constructor is a function that assigns properties to the **this** object. A constructor in JavaScript plays a similar role to a class definition in C#, Java, or C++, because it enables you to define a common set of properties for objects of the same type.

You can use a constructor function to create a new object and initialize its values. The constructor function can take parameters to indicate the initial values for properties of the object. Inside the constructor, use the **this** keyword to set property values on the target object.

The following example defines a constructor function named **Account**, which assigns properties that represent a bank account:

```
var Account = function (id, name) {
    this.id = id;
    this.name = name;
    this.balance = 0;
    this.numTransactions = 0;
};
```

After you have defined a constructor function, you can use the constructor to create a new object by using the **new** keyword. You can pass parameters into the constructor function to specify the initial values for the object. When you create an object by using the **new** keyword, JavaScript performs the following steps:

1. It creates a new object.
2. It sets the **constructor** property of the new object to reference the constructor function.
3. It assigns **this** to refer to the new object.
4. It invokes the code in the constructor function on the new object, to set the properties on the object.

The following example creates two **Account** objects and sets their properties. After these statements, **acc1.constructor** and **acc2.constructor** both reference the **Account** constructor function.

```
var acc1 = new Account(1, "John");
var acc2 = new Account(2, "Mary");
```

- Constructor functions define the shape of objects

- They create and assign properties for the target object
- The target object is referenced by the **this** keyword

```
var Account = function (id, name) {
    this.id = id;
    this.name = name;
    this.balance = 0;
    this.numTransactions = 0;
};
```

- Use the constructor function to create new objects with the specified properties:

```
var acc1 = new Account(1, "John");
var acc2 = new Account(2, "Mary");
```

Using Prototypes

Constructors are just functions that create new objects. If you use the same constructor to create two objects, each object gets its own set of properties as defined by the constructor function. While this is useful for data properties, the same mechanism is not quite so useful for methods; each object gets its own copy of the methods defined by the constructor, despite the fact that they are logically all the same piece of code, and each copy of the method occupies its own space in memory and has a corresponding management overhead. It is clearly wasteful of resources if you create hundreds or thousands of instances of an object by using the same constructor.

- All objects created by using a constructor function have their own copy of the properties defined by the constructor
- All JavaScript objects, including constructors, have a special property named **prototype**
- Use the prototype to share function definitions between objects:

```
Account.prototype = {
    deposit: function(amount) {
        this.balance += amount;
        this.numTransactions++;
    },
    // Plus other methods...
};
```

Prototypes give you a way to share functions between objects created by using the same constructor. All JavaScript objects, including constructor functions, have a special property named **prototype**. The prototype is really just another object to which you can assign new properties and methods; you use it as a blueprint for creating new objects. It automatically provides a set of functions and other properties that are inherited from the prototype of the **Object** type by using a mechanism known as prototype chaining. Prototype chaining is described in more detail in the next lesson.

 **Note:** The prototype of the **Object** type provides useful functionality such as the **toString** method, which returns a string representation of an object.

The following example sets the prototype on the **Account** constructor function shown in the previous topic. The example adds methods to implement bank account behavior:

```
Account.prototype = {
    deposit: function(amount) {
        this.balance += amount;
        this.numTransactions++;
    },
    withdraw: function(amount) {
        this.balance -= amount;
        this.numTransactions++;
    },
    displayDetails: function() {
        alert(this.id + ", " +
            this.name + " balance $" +
            this.balance + "(" +
            this.numTransactions + " transactions)");
    }
};
```

When you create an object by using a constructor function, the new object delegates functionality to the **prototype** property of the constructor function. What this means is that the new object has access to all of the properties defined in the constructor function and all of the properties defined by the prototype for the constructor function. However, the properties defined by the prototype are shared by all instances of the object. All objects created by using a constructor function have their own set of properties that are defined by the constructor function, but they share the properties defined by the prototype with all other objects created by using the same constructor function. So, you can define shared functionality such as methods by using a prototype, and this functionality will not be duplicated. You can use the same feature to implement shared data properties (similar to *static* or *class* properties in other object-oriented languages).

The following example creates some objects by using the **Account** constructor function, and illustrates how you can invoke the methods defined in the **prototype** object of the **Account** constructor function. The important point to realize is that the data properties defined in the **Account** constructor (**id**, **name**, **balance**, and **numtransactions**) are specific to each object (**acc1** and **acc2**), whereas the methods defined by the prototype (**deposit**, **display**, and **withdraw**) are shared by all instances. The **this** object used by these functions references the appropriate instance:

```
var acc1 = new Account(1, "John");
var acc2 = new Account(2, "Mary");
acc1.deposit(100);
acc1.displayDetails();
acc2.withdraw(50);
acc2.displayDetails();
```

Using the Object.create Method

The **Object** object has a **create()** method that enables you to create an object based on an existing prototype, and optionally provide additional properties. This function enables you to implement an efficient form of inheritance based on prototypes.

The general syntax for the **Object.create()** method is as follows:

```
Object.create(prototypeObject,
propertiesObject)
```

- Use **Object.create()** to create an object based on existing prototype

- Pass in a prototype object
- Optionally pass in a properties object that specifies additional properties to add to the new object

```
var obj1 = Object.create(prototypeObject, propertiesObject);
```

- The new object has access to all the properties defined in the specified prototype

- It forms the basis of the approach used by many JavaScript developers to implement inheritance.

- The **prototypeObject** parameter specifies the object to use as the prototype for the new object. You can invoke the **Object.getPrototypeOf()** method if you want to obtain the prototype of an existing object to use here.
- The **propertiesObject** parameter is optional, and specifies an object whose properties will be added into the new object. This object takes the form of a collection of property descriptors. A property descriptor specifies the value stored in the property, and can also specify other attributes such as whether the property is read-only or can be updated; properties are read-only unless you set the **writable** attribute to **true**.

The following example creates an object by using a **null** prototype, and adds two simple properties:

```
var obj1 = Object.create(null, {
  prop1: {value: "hello", writable: true}, // read/write property
  prop2: {value: "world"} // read-only property
});
```

 **Note:** If you do specify a null prototype, the only functionality available in the new object will be that defined by the properties object. This means that methods normally inherited from **Object**, such as **toString** and the **+** operator, will not be available.

The next example creates an object called **obj2** by using the prototype of the **acc1** object defined in the earlier examples.

```
// Account constructor function, same as before.
```

```
var Account = function (id, name) { ... };
// Account prototype, same as before.
Account.prototype = { ... };
acc1 = new Account(...);
// Create an object by using the Account prototype.
var obj2 = Object.create(Object.getPrototypeOf(acc1));
```

The **acc1** object was created by using the **Account** constructor function which references the prototype that contains the **deposit**, **withdraw**, and **displayDetails** methods. Consequently, the **obj2** object has access to the same functions. However, because **obj2** was not created by using the **Account** constructor, it does not contain the **id**, **name**, **balance**, or **numTransactions** fields. Initially therefore, it looks as though this approach to creating an object has little value. But this is intentional, and it is actually useful to separate the prototype that defines the functionality of an object from the constructor that specifies the properties that an object contains. This approach is a fundamental part of the technique that many JavaScript developers use to implement inheritance that is described in the next lesson.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3

Extending Objects

Most object-oriented languages are based on the concept of classes, but JavaScript does not use classes. In JavaScript, everything is an object.

Despite the lack of classes in JavaScript, it is possible to implement object-oriented features such as inheritance and encapsulation. The concepts are similar to those in class-based languages such as C#, Java, and C++, but the underlying language mechanisms are quite different. In this lesson, you will learn how JavaScript implements inheritance and encapsulation.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement **encapsulation** in JavaScript.
- Implement **inheritance** in JavaScript.
- Add **functionality** to existing native objects.

Implementing Encapsulation

Encapsulation is an important principle in object-oriented development; it shields external code from the internal workings of a class.

Encapsulation helps to make code simpler, more self-contained, and maintainable by reducing dependencies on other classes.

Classic object-oriented languages have keywords such as **public**, **protected**, and **private** that specify the accessibility of members in a class. JavaScript does not have these keywords, but instead uses a technique known as closures to achieve encapsulation.

Closures enable you to define encapsulated variables for an object, and expose the variables through a set of public accessor functions. To implement closures, define a constructor function and add the following code:

1. Declare variables without using the **this** keyword. The absence of the **this** keyword means that the variables have local scope, and are visible only inside the constructor function.
2. Declare methods to get and set the values of the variables. Use the **this** keyword when declaring these methods to ensure they are visible to external code.

The following example shows how to use closures to achieve encapsulation. The **name** and **age** variables are effectively private to **Person** objects, whereas the **getName()**, **getAge()**, **setName()**, and **setAge()** methods are public.

Using Closures to Achieve Encapsulation

```
• To define private members for an object, declare
variables in the constructor and omit the this keyword
• To define public accessor functions for an object,
declare methods in the constructor and include the this
keyword
var Person = function(name, age)
{
    // Private properties.
    var _name, _age;

    // Public accessor functions.
    this.getName = function()
    {
        return _name;
    }
    ...
}
```

```
var Person = function(name, age)
{
    // Private properties.
    var _name, _age;
```

```

// Public methods defined in the constructor have access to private properties.
this.getName = function()
{
    return _name;
}
this.setName = function(name)
{
    _name = name;
}
this.getAge = function()
{
    return _age;
}
this.setAge = function(age)
{
    if (age > 0 && age < 100)
        _age = age;
}
// Constructor logic.
_name = name;
this.setAge(age);
}
// Public methods defined in the prototype do not have access to private properties.
Person.prototype =
{
    toString: function()
    {
        return this.getName() + " is " + this.getAge();
    }
}
// External code.
var person1 = new Person("Joe", 21);
alert(person1.toString()); // Displays "Joe is 21"
alert(person1._name); // Displays "undefined"

```

 **Note:** JavaScript also provides accessor properties enabling a programmer to encapsulate data behind **get** and **set** functions by using the **Object.defineProperty** function. For more information, visit <http://go.microsoft.com/fwlink/?LinkId=267730>.

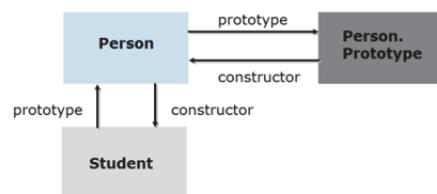
Implementing Inheritance by Chaining Prototypes

In class-based languages such as C#, Java, and C++, you implement inheritance by defining a class that extends from an existing class. In JavaScript, you implement inheritance by defining an object that extends an existing object.

You can use several JavaScript language mechanisms to implement inheritance. You can use the **Object.create** function to implement a form of inheritance that supports shared functions and instance data. Another common approach that gives you more refined control over the inheritance mechanism is to make use of constructor function prototypes, as follows:

1. Define the base constructor and prototype.
2. Define the derived constructor.

- Define the base constructor and prototype
- Define the derived constructor
- Set the **prototype** property of the derived constructor to an instance of the base object



3. Set the **prototype** property of the derived constructor to be an instance of the base object. This ensures that the derived object has access to all the members defined in the base prototype.
4. Reset the **constructor** property in the derived prototype so that it refers back to the derived constructor.



Note: A prototype has a reference to the constructor with which it is associated. If you don't set the **constructor** property of the derived prototype to the derived constructor, it will continue to reference the base constructor, which may cause problems later when your JavaScript code needs to resolve references to properties in the derived class.

The following example shows how to implement inheritance in JavaScript by using prototype chaining.

Implementing Inheritance by Using Prototype Chaining

```
// Base constructor.
var Person = function(name, age) {
    this.name = name;
    this.age = age;
}
// Base prototype.
Person.prototype = {
    haveBirthday: function() {
        this.age++;
    }
};
// Derived constructor.
var Student = function(name, age, subject) {
    this.name = name;
    this.age = age;
    this.subject = subject;
}
// Set the derived prototype to be the same object as the base prototype,
// and reset that derived prototype so that it uses the correct constructor.
Student.prototype = new Person();
Student.prototype.constructor = Student;
// Create a derived object and invoke any methods defined in the object or one of its
// parents. JavaScript uses prototype chaining to locate methods up the inheritance tree.
var aStudent = new Student("Jim", 20, "Physics");
aStudent.subject = "BioChemistry";
aStudent.haveBirthday();
alert(aStudent.age);
```

Adding Functionality to Existing Objects

You can use prototypes to extend the functionality of existing objects, including the built-in objects defined as part of the standard JavaScript language.

To extend the functionality of an object, follow these steps:

- Get the prototype for an object.
- Assign a new property to the object, to represent the new feature that you want to add.

- Get the prototype for an object
- Assign a new property to the object

```
var Point = function(x, y) {
    this.x = x;
    this.y = y;
}
```

```
Point.prototype.moveBy = function(deltaX, deltaY) { ... }
Point.prototype.moveTo = function(otherPoint) { ... }
```

```
var p1 = new Point(100, 200);
p1.moveBy(10, 20);
p1.moveTo(anotherPoint);
```

- Use the **apply** method to resolve references to **this** in generic functions



Note: Be careful not to accidentally replace a function in a prototype with another function with the same name; this can lead to unexpected behavior elsewhere in your application.

The following example shows how to add functionality to an existing object. The code defines a constructor function named **Point**, to represent a coordinate point that has **x** and **y** properties. The code then adds methods named **moveBy()** and **moveTo()** to the prototype object for **Point**, and shows how these methods are available on all **Point** objects.

Adding Functionality to an Object

```
var Point = function(x, y) {
    this.x = x;
    this.y = y;
}
Point.prototype.moveBy = function(deltaX, deltaY) {
    this.x += deltaX;
    this.y += deltaY;
}
Point.prototype.moveTo = function(otherPoint) {
    this.x = otherPoint.x;
    this.y = otherPoint.y;
}
var p1= new Point(100, 200);
p1.moveBy(10, 20);
var p2= new Point(25, 50);
p2.moveTo(p1);
alert("p2.x: " + p2.x + " p2.y: " + p2.y);
```

Using the **apply** Method With Generic Functions

As well as defining a prototype that is specific to a type, you can also create generic functions that you can use to implement common functionality for objects of almost any type. Consider the following simple global function that sets the **color** property of an object to the value specified as the parameter:

```
function SetColor(color) {
    this.color = color;
}
```

As it stands however, when you call **SetColor**, to which object does **this** refer? If you invoke it in the same way as any other global function, **this** will actually be undefined, and the function will fail with an exception when it attempts to set the **color** property of the undefined object. However, you can set the context for a function by using the **apply** method of the function. The **apply** method takes two parameters; an object that is used to resolve references to **this** in the function, and an argument list (as an array) that is passed as the parameter list to the function. The following example shows how to invoke the **SetColor** method for a **Point** object, as defined in the previous code example in this topic:

```
var p1= new Point(100, 200);
...
SetColor.apply(p1, ["red"]);
alert(p1.color); // Displays "red"
```

The **Point** object, **p1**, is passed as the **this** parameter, and the parameter list containing the single string "red" is passed as the **color** parameter to the **SetColor** method. The result is that the **color** property of **p1** is set to "red".

Demonstration: Refining Code for Maintainability and Extensibility

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Refining Code for Maintainability and Extensibility

Scenario

The existing JavaScript code for the ContosoConf website has been written without much high-level structure or organization. While this approach is fine for small pieces of code, it will not scale up for a larger project. An unstructured collection of functions and variables scattered throughout a JavaScript file can quickly become unmaintainable.

Before implementing more website features by using JavaScript, you decide to refactor the existing code to introduce better organizational practices. The resulting code will be more maintainable and provide a good pattern for implementing future website features.

Objectives

After completing this lab, you will be able to:

1. Implement good practices for writing JavaScript code.
 2. Refactor JavaScript code to use object-oriented principles.
- Estimated Time: 60 minutes
 - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
 - User Name: **Student**
 - Password: **Pa\$\$w0rd**

Exercise 1: Object Inheritance

Scenario

In this exercise, you will create a utility function named **Object.inherit**. The purpose of this function is to define a prototype object and initialization function, which can then be used to create new instances of a type.

The following example uses **Object.inherit** to define a **Circle** prototype. This prototype is then used to create a **Circle** object.

The **initialize** function is similar to a constructor in other object-oriented languages such as C# and Java; it initializes the state of a new object, based on the parameters passed to it.

Using a Prototype to Initialize an Object

```
var Circle = Object.inherit({
    initialize: function (radius) {
        this.radius = radius;
    },
    area: function () {
        return Math.PI * this.radius * this.radius;
    },
    circumference: function() {
        return 2 * Math.PI * this.radius;
    }
});
// Create and use a Circle object
var c = Circle.create(10);
alert(c.area());
alert(c.circumference());
```

The **inherit** function has been partially defined, but the JavaScript file containing it needs completing.

Next, you will prevent variables from leaking into global scope by using an immediately invoked function expression.

Then, you will switch the JavaScript code into strict mode. And finally, you will attach the **inherit** function to the built-in JavaScript **Object** object.

The main tasks for this exercise are as follows:

1. Review the **Object.inherit.js** JavaScript file.
2. Implement the **Inherit** function.
3. Scope the JavaScript code inside an immediately invoked function expression.
4. Use strict mode.

► Task 1: Review the **Object.inherit.js** JavaScript file

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod07\Labfiles\Starter\Exercise 1** folder.
4. Open the **Object.inherit.js** file in the **scripts** folder.
5. Review the JavaScript code and comments in this file. This file contains functions called **copyOwnProperties** and **inherit**.
 - You will implement the **inherit** function, using the TODO comments as guidance.
 - **copyOwnProperties** is a helper function that copies properties from one object to another. The **inherit** function will use this function.

► Task 2: Implement the Inherit function

1. Implement the **inherit** function, using the TODO comments for guidance.
 - Create and assign a variable named **factory** by using the **Object.create** function. Specify the current object as the prototype.
 - Add a method called **create** to **factory**. Leave the body of the method empty for now; you will implement it in the next step.
 - Copy the properties of **additionalProperties** into **factory**.
 - Return the value of **factory** from the **inherit** function.
2. Complete the implementation of the **factory.create** method in the **inherit** function:
 - Create a new object that uses **factory** as its prototype.
 - Call the **initialize** function of the new object (if it exists and it is a function), passing any arguments that were passed to **create**.



Note: Use the **typeof** operator to determine whether the **initialize** function exists, like this:

```
if (typeof instance.initialize === "function") {  
    ...  
}
```

The **typeof** operator was described in module 3, "Introduction to JavaScript"; it returns the type of an object as a string. If the object is a function, the **typeof** operator returns the string "function". If the object does not exist, the **typeof** operator returns the string "undefined".

- Use the **apply** method to call the **initialize** function, and specify the new object as the value to pass as the **this** parameter. This technique ensures that any references to **this** in the **initialize** function defined by the new object are resolved correctly and refer to the new object and not the **inherit** function.
3. Return the new object from the **create** function.
 4. At the end of the JavaScript file, make the **inherit** function available to other scripts by attaching it to the built-in **Object** object.
- **Task 3: Scope the JavaScript code inside an immediately invoked function expression**
1. In the **Object.inherit.js** file, enclose the JavaScript code in an immediately invoked function expression.
- **Task 4: Use strict mode**
1. In the **Object.inherit.js** file, modify the immediately invoked function to use strict mode.

Results: After completing this exercise, you will have written the **Object.inherit** utility function that you will use in later exercises to structure the code for the web application. You will also have modified the scope for this function, and specified that the code should run by using strict mode.

Exercise 2: Refactoring JavaScript Code to Use Objects

Scenario

The JavaScript code for the Schedule page has been partially refactored to be more maintainable. In this exercise, you will continue the refactoring process by updating the code for the Schedule page. You will use the **Object.inherit** utility function to define a **ScheduleList** prototype object. Then you will move the existing functions and variables relating to the schedule list into this prototype.

The main tasks for this exercise are as follows:

1. Define the **ScheduleList** factory.
2. Convert functions into methods of the **ScheduleList** object.
3. Create and use a **ScheduleList** object.
4. Test the application.

► **Task 1: Define the **ScheduleList** factory**

1. Open the **ContosoConf.sln** solution in the **E:\Mod07\Labfiles\Starter\Exercise 2** folder.
2. In the **schedule.js** file, in the **scripts\pages** folder, find the following comment:

```
// TODO: Create a ScheduleList factory object using the Object.inherit helper method.
```

3. After this comment, use the **Object.inherit** function to create an object named **ScheduleList**.
4. Add an **initialize** function to the object passed to **Object.inherit**.
 - The **initialize** function should take two parameters called **element** and **localStarStorage**.

- The **initialize** function should use these two parameters to create and populate properties also called **element** and **localStarStorage** for the **ScheduleList** object.
5. Remove the **element** and **localStarStorage** variables from the **ScheduleList** object (they have been replaced by the properties that you just defined).

► Task 2: Convert functions into methods of the ScheduleList object

1. In the **schedule.js** file, find the following comment:

```
// TODO: Refactor these functions into methods of the ScheduleList object.
```

2. Convert each of the functions following the comment into methods of **ScheduleList**.



Note: As an example of refactoring a function, the following code:

```
downloadDone: function (responseData) {  
    addAll(responseData.schedule);  
}
```

Should be moved to the **ScheduleList** object and converted to:

```
var ScheduleList = Object.inherit({  
    initialize: function (element, localStarStorage) {  
        ...  
    },  
    ...  
    downloadDone: function (responseData) {  
        this.addAll(responseData.schedule);  
    },  
    ...  
}
```

3. In the **startDownload** function, add the **context** option to the object passed to the **ajax** function and initialize it to **this**.



Note: This step ensures that the correct object is made the context of the ajax callbacks that occur when downloading is complete.

4. In the **addAll** function, specify that the **forEach** function should call the **add** method of the object referenced by **this**. Also, pass **this** as the second parameter to the **forEach** function.



Note: The second parameter to the **forEach** function specifies the object that any use of **this** will reference in the **add** function.

► Task 3: Create and use a ScheduleList object

1. Near the end of the **schedule.js** file, find the following JavaScript code:

```
// TODO: Replace the following code by creating a ScheduleList object  
//       and calling the startDownload method.  
element = document.getElementById("schedule");
```

```
localStarStorage = LocalStarStorage.create(localStorage);
startDownload();
```

2. Modify this code to create a **ScheduleList** object by using the **ScheduleList.create** function. Pass the DOM element and local star storage object as parameters to **ScheduleList.create()**.
3. Call the **startDownload** method of the **ScheduleList** object.

► **Task 4: Test the application**

1. Run the application and view the **schedule.htm** page.
2. Verify that the page looks similar to the image below:

The Schedule page should still display the list of sessions for the conference, like this:

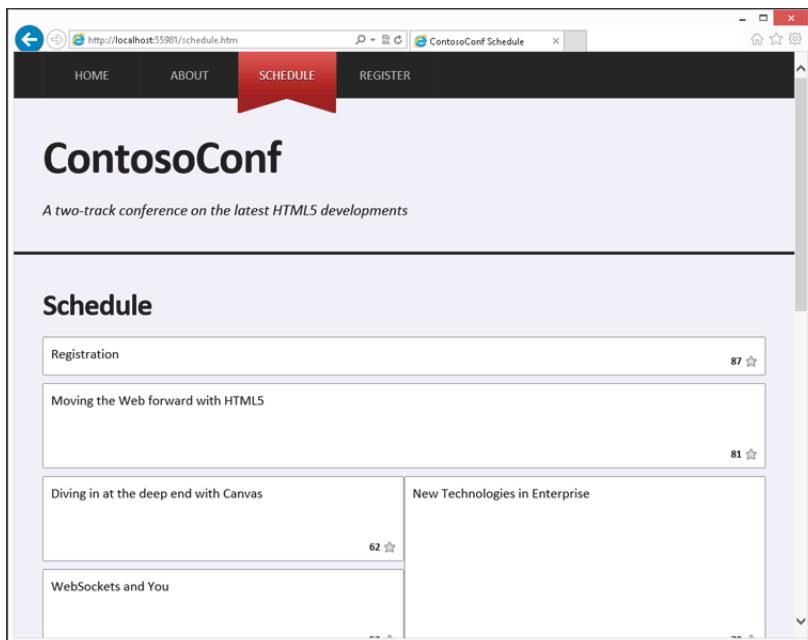


FIGURE 7.1:THE SCHEDULE PAGE



Note: The styling and layout of the Schedule page has been changed. This is performed by using the CSS rules implemented in the `schedule.css` style sheet and is not a result of any updates made to the JavaScript code.

3. Close Internet Explorer.

Results: After completing this exercise, you will have refactored the JavaScript code for the Schedule page to be more maintainable by using objects.

Module Review and Takeaways

In this module, you have seen how to follow an object-oriented approach to application design in JavaScript. First, you learned how to minimize global name clashes by using namespaces, strict mode, and immediate functions. Next, you saw how to create custom objects by using object literal syntax, constructors, and prototypes. Finally, you saw how to implement encapsulation and inheritance in JavaScript.

Review Question(s)

Question: How can you guard against name clashes in JavaScript?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
If you modify the prototype object for a constructor function, the changes are only visible to new objects that you create by using that constructor function; existing objects created by using the constructor function will be unaffected. True or False?	

Test Your Knowledge

Question
Which of the following statements is true?
Select the correct answer.
<input type="checkbox"/> JavaScript uses the public, private, and protected keywords to implement encapsulation.
<input type="checkbox"/> JavaScript does not support encapsulation.
<input type="checkbox"/> JavaScript uses closures to achieve encapsulation.
<input type="checkbox"/> JavaScript uses prototype chaining to achieve encapsulation.
<input type="checkbox"/> JavaScript uses the Object.create() function to implement encapsulation.

Module 8

Creating Interactive Pages by Using HTML5 APIs

Contents:

Module Overview	8-1
Lesson 1: Interacting with Files	8-2
Lesson 2: Incorporating Multimedia	8-7
Lesson 3: Reacting to Browser Location and Context	8-12
Lesson 4: Debugging and Profiling a Web Application	8-17
Lab: Creating Interactive Pages with HTML5 APIs	8-22
Module Review and Takeaways	8-29

Module Overview

Interactivity is a key aspect of modern web applications, enabling you to build compelling web sites that can quickly respond to the actions of the user, and also adapt themselves to the user's location.

This module describes how to create interactive HTML5 web applications that can access the local file system, enable the user to drag-and-drop data onto elements in a web page, play multimedia files, and obtain geolocation information.

Objectives

After completing this module, you will be able to:

- Access the local file system, and add drag-and-drop support to web pages.
- Play video and audio files in a web page, without the need for plugins.
- Obtain information about the current location of the user.
- Use the F12 Developer Tools in Internet Explorer to debug and profile a web application.

Lesson 1

Interacting with Files

HTML5 enables a web application to interact with the local file system, and supports drag-and-drop operations that enable the user to drag files or HTML elements onto a web page. In this lesson, you will learn how to use the HTML5 File APIs and how to add drag-and-drop support to a web page.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the HTML5 file types.
- Describe the **FileReader** interface for reading files on the local file system.
- Use **FileReader** to read a **text** file.
- Use **FileReader** to read a **binary** file.
- Add **drag-and-drop** support to a web page.

HTML5 File Interfaces

HTML5 has a standardized File API that enables an HTML page to interact with local files on the browser file system. This API defines four key interfaces:

- A **Blob** interface represents immutable raw data. A **Blob** has a **type** attribute that indicates the media type of the data, such as "**text/plain**". **Blob** also has a **slice()** method that returns a portion of a **Blob** between a specified start and end offset. Slicing a blob is useful if you want to incrementally upload a large file to a server.
- The **File** interface inherits from **Blob**, and represents an individual file. A **File** has two additional read-only attributes that describe the state of the file:
 - The **name** attribute indicates the name of the file without any path information, expressed as a string.
 - The **lastModifiedDate** attribute indicates the last modification date of the file, expressed as a **Date** object.
- The **FileList** interface is a collection of **File** objects. There are two common ways to obtain a **FileList** object in a web page:
 - Define an **<input type="file">** element in your web page, and handle the **change** event. The **change** event indicates the files that the user selected in the element.
 - Implement drag-and-drop support in your web page, by handling the **drop** event on an element. When the user drops files on the element, the drop event indicates the files that the user dropped onto the element.

- The HTML5 File API enables a web application to access the local file system
- There are four key interfaces:

- **Blob** – immutable raw binary data
- **File** – readonly information about a file
- **FileList** – an array of files
- **FileReader** – methods for reading data from a file or blob



Additional Reading: For detailed information about the HTML5 File API, see <http://go.microsoft.com/fwlink/?LinkId=267731>.

- The **FileReader** interface enables an application to read a file or blob into a JavaScript variable.

The FileReader Interface

The **FileReader** interface provides three methods for reading data:

- **readAsText()** reads a file or blob and makes the contents available as plain text. This method is useful if you want to read the contents of a text file.
- **readAsDataURL()** reads a file or blob and makes the contents available as a data URL. This method is useful if you want to read the contents of a binary file, such as an image.
- **readAsArrayBuffer()** reads a file or blob and makes the contents available as an **ArrayBuffer**. An **ArrayBuffer** represents a finite number of bytes that can be used to store numbers of any size, such as an array of 8-bit integers or 32-bit floating point numbers.

The **FileReader** interface provides methods for reading a file or blob:

- **readAsText()** – used for reading text files
- **readAsDataURL()** – used for reading binary files
- **readAsArrayBuffer()** – used for reading data into a buffer array

FileReader reads data asynchronously and fires events:

progress	abort	loadend
load	error	

A **FileReader** object reads contents asynchronously, and fires events to indicate the progress of the loading operation. The following list describes the events that you typically handle:

- The **progress** event occurs repeatedly while data is being loaded, to indicate progress.
- The **load** event indicates that the data has been successfully loaded. The file contents are available through the **result** attribute on the **FileReader** object. The result is a **Blob** object if you invoked **readAsText()** or **readAsDataURL()**, or an **ArrayBuffer** object if you invoked **readAsArrayBuffer()**.
- The **abort** event indicates that data loading has been aborted, perhaps due to a call to the **abort()** method.
- The **error** event indicates that an error occurred during loading. Errors can occur for several reasons, such as attempting to read an unreadable file, attempting to read a file that is too large, or attempting multiple simultaneous reads on the same file.
- The **loadend** event indicates that the read operation has completed, either successfully or unsuccessfully.

Reading a Text File

The **FileReader readAsText()** method enables you to read a text file on the local file system. The following example shows how to read a text file.

The web page includes an **<input>** element that enables the user to select a text file on the local file system. As soon as the user has selected the file, the **onLoadTextFile()** function reads the file and displays its contents as plain text in a **<textarea>** element on the web page.

Reading a Text File

To read a text file:

1. Get a File or Blob object, either by using an **<input type="file">** element or by drag-and-drop.
2. Create a **FileReader** object and handle events such as **load** and **error**.
3. Invoke **readAsText()** on the **FileReader** object.
4. In the **load** event handler function, access the text content in the **result** property of the event target.
5. In the **error** event handler function, implement appropriate error handling.

```
<input type="file" id="theTextFile" onchange="onLoadTextFile()" />
<textarea id="theMessageArea" rows="30" cols="40"></textarea>
<script type="text/javascript">
    function onLoadTextFile() {
        var theFileElem = document.getElementById("theTextFile");
        // Get the File object selected by the user, and make sure it is a text file.
        if (theFileElem.files.length != 0 && theFileElem.files[0].type.match(/text.*/)) {
            // Create a FileReader and handle the onload and onerror events.
            var reader = new FileReader();
            reader.onload = function(e){
                var theMessageAreaElem = document.getElementById("theMessageArea");
                theMessageAreaElem.value = e.target.result;
            };
            reader.onerror = function(e){
                alert("Cannot load text file");
            };
            // Read text file (the second parameter is optional - the default encoding is
            "UTF-8").
            reader.readAsText(theFileElem.files[0], "ISO-8859-1");
        } else {
            alert("Please select a text file");
        }
    }
</script>
```

Reading a Binary File

The **FileReader readAsDataURL()** method enables you to read a binary file on the local file system. The following example shows how to read an image file.

The web page includes an **<input>** element that enables the user to select an image file on the local file system. As soon as the user selects the file, the **onLoadBinaryFile()** function reads the file and assigns its contents in an **** element on the web page.

Reading a Binary File

To read a binary file:

1. Get a File or Blob object, either by using an **<input type="file">** element or by drag and drop.
2. Create a **FileReader** object and handle events such as **load** and **error**.
3. Invoke **readAsDataURL()** on the **FileReader** object.
4. In the **load** event handler function, access the text content in the **result** property of the event target.
5. In the **error** event handler function, implement appropriate error handling.

```
<input type="file" id="theBinaryFile" onchange="onLoadBinaryFile()" />
<img id="theImage"></img>
<script type="text/javascript">
    function onLoadBinaryFile() {
        var theFileElem = document.getElementById("theBinaryFile");
        // Get the File object selected by the user (and make sure it is an image file).
        if (theFileElem.files.length != 0 && theFileElem.files[0].type.match(/image.*/))
{
            // Create a FileReader and handle the onload and onerror events.
            var reader = new FileReader();
            reader.onload = function(e){
                var theImgElem = document.getElementById("theImage");
                theImgElem.src = e.target.result;
            };
            reader.onerror = function(e){
                alert("Cannot load binary file");
            };
            // Read the binary file.
            reader.readAsDataURL(theFileElem.files[0]);
        } else {

```

```

        alert("Please select a binary file");
    }
</script>

```

Implementing Drag-and-Drop

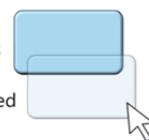
HTML5 supports drag-and-drop. You can drag elements on a web page or files in the file system, and drop them on another element in the web page.

To make an element draggable, follow these steps:

- Set the **draggable** attribute for the element to **true**.
- Handle the **dragstart** event.
- In the **dragstart** event handler function, configure data transfer information for the drag-and-drop operation. You can do this through the **dataTransfer** attribute on the **event** object, which represents a **DataTransfer** object.

• HTML5 supports drag-and-drop

- The user can drag HTML elements, or files from the local file system
- The user can drop items onto drop-enabled target elements



• To support drag and drop operations

- Enable drag support on HTML elements, if required
- Enable drop support on HTML drop target elements
- Handle dragover and drop events on HTML drop target elements

The **DataTransfer** object defines attributes and methods that enable you to control all aspects of the drag-and-drop operation:

- **setData(mimeType, data)** specifies the MIME type of the data being transferred, and the data itself. It is possible to store several different items in the **DataTransfer** object, as long as they all have different MIME types.



Note: Internet Explorer 10 does not currently support the full range of MIME types with the **setData()** function; you can only specify "text" or "URL".

- **setDragImage(imgElement, x, y)** specifies a custom mouse cursor image that the browser will use during the drag-and-drop operation.
- **effectAllowed** restricts what type of drag-and-drop operation is allowed, such as **copy**, **move**, or **link**.
- **dropEffect** specifies the feedback that the user receives when the mouse hovers over a target element. The **dropEffect** attribute can be **none**, **copy**, **move**, or **link**.

The following example shows how to make a **<div>** element draggable:

Making a **<div>** Element Draggable

```

<div draggable="true" ondrag="handleDrag(event)">
    <b>Some content</b> to be dragged.
</div>
<script type="text/javascript">
    function handleDrag(event) {
        event.dataTransfer.effectAllowed = "copy";
        event.dataTransfer.setData("text/plain", event.target.innerHTML);
    }
</script>

```

To enable a control to act as a drop target, it must handle the **drop** event. To accept the data being dropped on the element, do this:

- In the **drop** event handler function, get the data being dropped by invoking the **getData()** method on the **dataTransfer** attribute of the object that raised the event. The **getData()** method expects the MIME type of the data to be retrieved to be provided as a parameter. If there is no data that has this type available, then the **getData()** method returns a null value.

 **Note:** Internet Explorer 10 currently only recognizes the "text" or "URL" types with the **getData()** function.

When a user drops data on a control that can handle drop events, the default behavior is to navigate to the data that was dropped. For example, if a user drops a URL, the browser will move to that URL; if a user drops an image, the browser will display the image. If the user drops a piece of text, the browser will attempt to treat this text as a URL and most likely generate an invalid URL error. Therefore, in most cases you will probably want to disable the default behavior, as follows:

- In the **drop** event handler function, invoke the **preventDefault()** method on the object that raised the event.
- You can also prevent further drop events from bubbling up the DOM tree by calling the **stopPropagation()** method on the event object.

Controls can also handle the **dragover** event that occurs when a user drags an item over the control without dropping it. It is common to use this event to change the cursor to indicate that the control can act as a drop target.

The following example shows how to make an **<input>** element accept HTML content being dropped on it.

Dropping Content on a **<div>** Element

```
<input ondragover="handleDragOver(event)" ondrop="handleDrop(event)" />
<script type="text/javascript">
    function handleDragOver(event) {
        event.stopPropagation();
        event.preventDefault();
        event.dataTransfer.dropEffect = "copy"; // Display a "copy" cursor
    }
    function handleDrop(event) {
        event.stopPropagation();
        event.preventDefault();
        event.target.innerHTML = e.dataTransfer.getData("text/plain");
    }
</script>
```

Lesson 2

Incorporating Multimedia

In this lesson, you will learn how to use the **<video>** and **<audio>** tags in HTML5 to play video and audio multimedia files without the need for plugins (that is, natively).

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how to play video files.
- Support multiple video formats.
- Interact with a video in code.
- Describe how to play audio files.

Playing Video Content by Using the **<video>** Tag

HTML5 provides a **<video>** tag that enables you to play video in a web page, without the need for plugins (natively). The ability to play video natively is especially beneficial in mobile and smartphone platforms, where some plugins are not supported.

The simplest way to use the **<video>** tag is to set the **src** attribute to the video that you want to play. For example:

```
<video src="MyVideo.mp4"></video>
```

The source of a video can be a local file or a reference to a resource on a remote URL.

HTML5 enables a web application to play video files natively, without requiring plugins

Use the **<video>** tag and set the attributes:

- **src**
- **width** and **height**
- **poster**
- **controls**
- **autoplay**
- **loop**
- **muted**

```
<video src="MyVideo.mp4"
       width="300" height="200"
       poster="MyPoster.jpg"
       autoplay="autoplay"
       muted="muted"
       controls="controls"
       loop="loop">
</video>
```

The **<video>** tag also supports the following attributes, which enable you to customize the behavior and appearance of the video player on the web page:

- The **width** and **height** attributes enable you to specify the size of the video player on the web page, in pixels.
- The **poster** attribute enables you to specify an image to be displayed on the web page while the video is being downloaded, or until the user presses the play button.
- The **controls** Boolean attribute specifies that the video player should display video control buttons such as Play, Pause, and Mute.
- The **autoplay** Boolean attribute specifies that the video will start playing automatically as soon as the video content has been loaded.
- The **loop** Boolean attribute specifies that the video will start over again as soon as it has finished playing.
- The **muted** Boolean attribute specifies that the audio output for the video will be muted.

The following example shows how to use the **<video>** tag attributes.

This example creates a video player that is 300 pixels wide and 200 pixels high. An image named "MyPoster.jpg" will be displayed while the video is being downloaded. As soon as the video has been

downloaded, it will start playing immediately with the audio muted. The video player will display controls to enable the user to control the video player. As soon as the video has finished, it will restart automatically.

Creating a Video Player

```
<video src="MyVideo.mp4"
       width="300" height="200"
       poster="MyPoster.jpg"
       autoplay="autoplay"
       muted="muted"
       controls="controls"
       loop="loop" >
</video>
```

Supporting Multiple Video Formats

The HTML5 specification does not stipulate what video formats are supported by web browsers. For example, most browsers support the .mp4 format, some browsers support the .webm format, while some support the .ogg format.

For maximum portability, you can provide several versions of your video in different formats, so that the video can be played successfully on different browsers. To support multiple video formats, include one or more **<source>** tags within the **<video>** tag. The **<source>** tag has two attributes:

- The **src** attribute specifies a video source.
- The **type** attribute specifies the video MIME type, such as video/mp4, video/webm, and video/ogg.

If you include multiple **<source>** tags within a **<video>** tag, the video player will play the first video format supported by the browser. You can also embed Microsoft Silverlight® or Adobe® Flash content as a fallback, plus a simple text message for browsers that do not support the **<video>** tag.

The following example shows how to create a video player that supports multiple types of content.

Supporting Multiple Formats in a Video Player

```
<video poster="MyPoster.jpg" autoplay controls>
  <source src="MyVideos/MyVideo.mp4" type='video/mp4' />
  <source src="MyVideos/MyVideo.webm" type='video/webm' />
  <source src="MyVideos/MyVideo.ogg" type='video/ogg' />
  <!-- You can embed Flash or Silverlight content here, as a fallback -->
  <!-- You can also display some simple text in case the browser does not support the
      video tag -->
  Cannot play video. Download video <a href="MyVideos/MyVideo.webm">here</a>
</video>
```

Interacting with Video in JavaScript Code

You can create **<video>** elements programmatically by using Document Object Model (DOM) functions in JavaScript code.

The following JavaScript code creates a **video** object, sets its attributes, and then adds it to an existing element on the web page.

Creating a Video Player by Using JavaScript Code

```
function
createVideoElement(nameOfFile) {
    // Create a video object and set its
    properties.
    var newVideo = document.createElement("video");
    newVideo.src = nameOfFile;
    newVideo.loop = true;
    newVideo.autoplay = true;
    newVideo.controls = true;
    newVideo.poster = "ImageLoading.png";
    // Add the video object to an existing element on the web page.
    var hostElem = document.getElementById("videoDir");
    hostElem.appendChild(newVideo);
}
```

- An application can interact with a **video** object in JavaScript code:

```
var newVideo = document.createElement("video");
newVideo.src = nameOfFile;
newVideo.loop = true;
newVideo.controls = true;
newVideo.poster = "ImageLoading.png";
...
newVideo.addEventListener("loadeddata", function(){
    newVideo.play();
}, false);
```

The **video** object has properties and functions that enable you to control video playback programmatically:

- The **play()** function plays the video.
- The **pause()** function pauses the video.
- The **paused** property indicates whether the video is currently paused.
- The **currentTime** property gets and sets the current time in the video.
- The **duration** property gets the total duration of the video.
- The **volume** property gets and sets the volume of the video.
- The **playbackRate** property gets and sets the playback rate of the video. A **playbackRate** of 1 indicates normal speed.

The following example plays a video if it is paused, or pauses a video if it is playing. The example also displays the time and duration of a video before it starts playing.

Playing a Video

```
if (aVideo.paused) {
    alert("Video current time: " + aVideo.currentTime + ", total duration: " +
aVideo.duration);
    aVideo.play();
}
else {
    aVideo.pause();
}
```

The **video** object has events that enable you to check whether content is available, to check the state of video playback, and to monitor the current playing position in a video. The following list describes some common **events**:

- The **loadedmetadata** event fires when the **video** object gets enough information about the content to know the duration. The event handler function can read the **duration** property on the **video** object, to tell the user the duration of the video.
- The **loadeddata** event fires when all video data has been loaded. The event handler function can then invoke the **play()** function on the **video** object, in order to play the video.
- The **timeupdate** event fires during playback of a video, to indicate the current time. The event handler function can read the **currentTime** property on the **video** object, to tell the user the current time in the video.

The following example shows how to handle the **loadedmetadata**, **loadeddata**, and **timeupdate** events on a **video** object.

Handling Video Events

```
aVideo.addEventListener("loadedmetadata", function() {
    alert("Video duration: " + aVideo.duration);
}, false);
aVideo.addEventListener("loadeddata", function() {
    aVideo.play();
}, false);
aVideo.addEventListener("timeupdate", function() {
    alert("Video current time: " + aVideo.currentTime);
}, false);
```

Playing Audio Content by Using the <audio> Tag

HTML5 provides the **<audio>** tag that enables you to play audio natively in a web page, without the need for plugins.

The simplest way to use the **<audio>** tag is to set the **src** attribute to the audio file that you want to play:

```
<audio src="MyAudio.mp3"></audio>
```

You can also play audio by using JavaScript code. The technique is very similar to that for playing video.

- Use the **<audio>** tag to play audio files natively, without requiring plugins:

```
<audio src="MyAudio.mp3"></audio>
```

- The JavaScript API for audio is similar to the API for video

The **<audio>** tag has many attributes in common with the **<video>** tag, including:

- **controls**
- **autoplay**
- **loop**

There are other similarities between the **<audio>** and **<video>** tags:

- The **<audio>** tag supports the **<source>** child tag, to enable you to provide alternative audio file formats.
- The **audio** object has **play()** and **pause()** functions, to play or pause the audio playback.
- The **audio** object has properties to control playback, such as **controls**, **autoplay**, **loop**, **paused**, **currentTime**, **duration**, **volume**, **muted**, and **playbackRate**.

- The **audio** object has events to indicate progress of playback, such as **loadedmetadata**, **loadeddata**, and **timeupdate**.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3

Reacting to Browser Location and Context

In this lesson, you will learn how to use the HTML5 Geolocation API to determine the current location of the browser. You will also see how to use the Page Visibility API and properties of the `navigator` object in the DOM to obtain information about the context in which a web page is running.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the HTML5 Geolocation API.
- Request geolocation information.
- Process geolocation information.
- Handle geolocation errors.
- Detect the network and execution context for a page.

The HTML5 Geolocation API

Location awareness is a key feature in many web applications, especially on tablets, mobile devices, and smartphones. The HTML5 Geolocation API enables a web application to detect the current location of the device, expressed in longitude, latitude, and altitude.

Typical uses of the Geolocation API include:

- Locating the current position of the user, and giving instructions on how to get to a new location.
- Tracking the movement of a user to see how far they have travelled in a certain time period.

The HTML5 specification does not stipulate a particular mechanism for obtaining geolocation information. Instead, when a web application makes a geolocation request, the API interacts with underlying device capabilities to retrieve the information. The device can use any of the following techniques to determine location:

- **IP address.** The user's IP address is looked up, and the physical address of the registrant is retrieved. This is a rudimentary technique that can be very inaccurate.
- **GPS positioning.** This technique is popular on mobile devices, and involves receiving GPS data from satellites to determine the location of the device. This technique can be very accurate, but it can be slow and it might not work when the device is indoors.
- **Wi-Fi.** This technique involves triangulating the device location based on the user's distance from a number of known Wi-Fi access points. This technique can be very accurate in urban areas, where there are a lot of Wi-Fi access points in the vicinity.
- **Cell phone location.** This technique involves triangulating the device's location based on the user's distance from a number of known cell phone towers. This technique is relatively accurate in urban areas, but it does require a device that has access to a cell phone or wireless modem.

- The Geolocation API enables a browser to determine the longitude and latitude of its current location 
- A host device can use several techniques to obtain geolocation information:
 - IP address
 - GPS positioning
 - Wi-Fi
 - Cell phone location
 - User-defined location information

- User-defined location information. An application can ask the user to enter their address, postal code, or some other details that the application can use to provide location-aware services. This technique can be useful if the user is able to supply more accurate location data than other techniques.

Requesting Geolocation Information

The HTML5 Geolocation API supports two types of position request:

- One-shot position request. To perform a one-shot position request, invoke the **navigator.geolocation.getCurrentPosition()** method.
- Repeated position updates. To start receiving position updates, invoke the **navigator.geolocation.watchPosition()** method. This method returns a watch ID value. To stop receiving position updates, invoke **navigator.geolocation.clearWatch()**, and pass the watch ID as a parameter.

- To make a one-shot request for position information:

```
navigator.geolocation.getCurrentPosition(myPositionCallbackFunction,
                                         myPositionErrorCallbackFunction,
                                         {enableHighAccuracy: true, timeout: 5000});
```

- To receive repeated position information updates:

```
var watchID =
  navigator.geolocation.watchPosition(myPositionCallbackFunction,
                                       myPositionErrorCallbackFunction,
                                       {enableHighAccuracy: true, maximumAge: 10000});
...
navigator.geolocation.clearWatch(watchID);
```

The **getCurrentPosition()** and **watchPosition()** methods take the following parameters:

- A call-back function, which will be called by the browser when location data is available. In the case of **getCurrentPosition()**, the call-back function will be called once. In the case of **watchPosition()**, the call-back function will be called repeatedly until you invoke **clearWatch()**.
- An optional error call-back function, which will be called by the browser if any errors occur.
- An optional **PositionOptions** object, which enables you to configure the geolocation request. You can specify the following properties on the **PositionOptions** object:
 - enableHighAccuracy**. A Boolean property that gives a hint that you want the device to use the most accurate technique to obtain geolocation information. The default value for the **enableHighAccuracy** property is false.
 - timeout**. The maximum elapsed time in milliseconds that the browser is permitted to get location data. If data is not available in this time, the error call-back function is called. The default value for the **timeout** property is infinity.
 - maximumAge**. How old location data can be in milliseconds, before the browser must try to refresh the data. The default value for the **maximumAge** property is zero seconds.

The following example shows how to make a one-off request for geolocation information. The example requests the highest accuracy information available, and specifies that the information must be available within five seconds, or it will time out.

```
navigator.geolocation.getCurrentPosition(myPositionCallbackFunction,
                                         myPositionErrorCallbackFunction,
                                         {enableHighAccuracy: true, timeout: 5000});
```

The following example shows how to make repeated requests for geolocation information. The example requests the highest accuracy information available, and specifies that the data can be up to 10 seconds old before it must be refreshed by the browser.

```
var watchID = navigator.geolocation.watchPosition(myPositionCallbackFunction,
```

```
myPositionErrorCallbackFunction,  
{enableHighAccuracy: true, maximumAge: 10000} );
```

The final example shows how to stop receiving geolocation information.

```
navigator.geolocation.clearWatch(watchID);
```

Processing Geolocation Information

When geolocation information is available, the browser calls the call-back function that you specified when you invoked **getCurrentPosition()** or **watchPosition()**. The call-back function receives an object that has a **coords** property, which has the following properties to provide coordinate information:

- **latitude**, in degrees. A latitude of 0 degrees represents the equator. A positive latitude indicates a location in the Northern Hemisphere, and a negative latitude indicates a location in the Southern Hemisphere.
- **longitude**, in degrees. A longitude of 0 degrees represents the Greenwich Meridian. A positive longitude indicates a location east of Greenwich, and a negative longitude indicates a location west of Greenwich.
- **accuracy**, in meters.

Geolocation properties include:

- **latitude**
- **longitude**
- **accuracy**



Geolocation data may include the following optional properties:

- **altitude**
- **altitudeAccuracy**
- **heading**
- **speed**



The **coords** property also has the following parameters, which might be **null** if the device is incapable of determining the values:

- **altitude**, in meters above sea level.
- **altitudeAccuracy**, in meters.
- **heading**, in degrees relative to the North direction.
- **speed**, in meters/second.

The following example shows how to implement a call-back function for geolocation information:

Processing Geolocation Information

```
function myPositionCallbackFunction(position) {  
    var latitude = position.coords.latitude;  
    var longitude = position.coords.longitude;  
    var accuracy = position.coords.accuracy;  
    var heading = position.coords.heading;  
    var speed = position.coords.speed;  
    var altitude = position.coords.altitude;  
    var altitudeAccuracy = position.coords.altitudeAccuracy;  
    // Add code here, to process the information.  
}
```

Handling Geolocation Errors

If a geolocation error occurs, the browser calls the error call-back function that you specified when you invoked **getCurrentPosition()** or **watchPosition()**. The error call-back function receives an error object that has the following properties:

- **code**: The error code can be one of the following values:
PositionError.PERMISSION_DENIED,
PositionError.POSITION_UNAVAILABLE, or
PositionError.TIMEOUT.
- **message**: A string that identifies the reason for the error.

If an error occurs during a geolocation request, the following properties are available:

- **code**
 - **PositionError.PERMISSION_DENIED**
 - **PositionError.POSITION_UNAVAILABLE**
 - **PositionError.TIMEOUT**
- **message**

```
function myPositionErrorCallbackFunction(error){
    var errorMessage = error.message;
    var errorCode = error.code;
    // Add code here, to process the information.
}
```

The following example shows how to implement an error call-back function for geolocation information:

Handling Geolocation Errors

```
function myPositionErrorCallbackFunction(error) {
    var errorMessage = error.message;
    var errorCode = error.code;
    // Add code here, to process the information.
}
```

Detecting the Context for a Page

HTML5 provides the **Page Visibility API** that enables you to determine if a web page is currently visible. Depending on whether a page is visible or not, you can choose to modify its behavior. For example:

- If a web-based email client is visible, it might check the server for new mail every few seconds. When hidden, it might scale checking email to every few minutes.
- If a web-based puzzle game is hidden, the game could be paused until it becomes visible again.

- **Page Visibility API**
 - Enables an application to determine whether a page is currently visible.
- **Offline detection**
 - Enables an application to detect whether the browser has a live connection to a server.
- **User agent information**
 - Enables an application to obtain the user agent string for the browser.

The **Page Visibility API** consists of two properties and an event:

- The **document.hidden** property describes whether the page is visible or not.
- The **document.visibilityState** property indicates the detailed page visibility state, such as **PAGE_VISIBLE** or **PAGE_PREVIEW**.
- The **visibilitychange** event fires any time the visibility state of the page changes.

HTML5 also enables a browser to determine if it has a live network connection. You can use this information to modify the appearance of a web page, in order to give the user visual feedback about whether the user is online or offline. To determine whether the **browser** is online or offline, use the following properties and events:

- The **navigator.onLine** property, which indicates whether the browser is online or offline.
- The **online** event, which fires when the browser was offline and becomes online.
- The **offline** event, which fires when the browser was online and becomes offline.

HTML5 enables a web application to determine the browser type, also commonly known as the user agent. You can use this information to access JavaScript or HTML features that are only supported on a particular type or version of browser. To determine the user agent, use the following property:

- The **navigator.userAgent** property, which returns a string indicating the user agent string for the browser. You can then use the string **indexof()** function to test for a particular browser type.

 **Reader Aid:** There are several third-party libraries available that perform automatic feature detection of the browser that is running your JavaScript code. The most popular of these libraries is called Modernizr. After you have added the Modernizr library to a web application, you can invoke it as each page loads to detect whether the browser supports the features required by that page (such as video and audio support, for example). Modernizr creates a JavaScript object that encapsulates the features available; you can query the properties of this object in your code and take an appropriate course of action. Modernizr also adds classes to the HTML object in the DOM that you can reference from CSS rules, to enable you to style elements according to which HTML features are available in the browser.

Lesson 4

Debugging and Profiling a Web Application

In this lesson, you will learn about the **Navigation Timing API** to evaluate the network performance of a web page.

Visual Studio 2012 provides a large number of debugging and profiling tools, but you can perform many similar tasks for a live web application by using the F12 Developer Tools provided with Internet Explorer. In this lesson, you will also learn how to use these tools to profile the performance of HTML5 web pages, and to debug JavaScript code.

Lesson Objectives

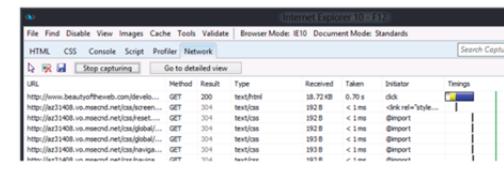
After completing this lesson, you will be able to describe how to use the HTML5 APIs and the F12 Developer Tools to debug Java Script code and to profile the performance of web pages.

Overview of the F12 Developer Tools in Internet Explorer 10

Download speed for web applications can be a critical factor in user satisfaction. Even a relatively small delay of a few hundred milliseconds can be unacceptable for some users. HTML5 provides the Navigation Timing API that enables you to determine the download speed of your web applications. To use the Navigation Timing API, use the following properties on the **window.performance** object:

- **navigation**: indicates how the user navigated to the page.
- **timing**: provides data for navigation and page load events.

- The Navigation Timing API enables an application to determine the download speed for a web page:
 - **window.performance.navigation**
 - **window.performance.timing**
- The F12 Developer Tools provide debugging and profiling capabilities in Internet Explorer



Additional Reading: For more information about the timing data available in the **window.performance.timing** object, see <http://go.microsoft.com/fwlink/?LinkId=267732>.

Another useful technique for developers is to use the **console.log()** function to write status information to the browser console. Writing status information to the console can be a helpful way of tracing the path through a web application, and can also help you to diagnose potential problems in the code. To view the console window in Internet Explorer 10, follow these steps:

- Press F12 or click the **Tools** icon, and then click **F12 Developer Tools**.
- In the Developer Tools panel, click the **Console** menu item.

The Developer Tools capabilities in Internet Explorer 10 provide a wide range of useful features for developers, including:

- Exploring HTML document structure.
- Determining which CSS styles apply to elements in the document.
- Debugging JavaScript code.

- Profiling application performance.
- Observing network communications.

Demonstration: Using the F12 Developer Tools to Debug JavaScript Code

In this demonstration, you will see how to use the F12 Developer Tools to view and debug JavaScript code in a live web application.

Demonstration Steps

Set a Breakpoint in JavaScript code

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, browse to the file **E:\Mod08\Democode\Document.html**.
4. If the message **Internet Explorer restricted this webpage from running scripts or ActiveX controls** appears, click **Allow blocked content**.
5. Press F12 to display the Developer Tools window.
6. If the F12 Developer Tools window appears as a pane in Internet Explorer, in the F12 Developer Tools pane, click **Unpin**.



Note: This action causes the F12 Developer Tools to appear in a standalone window.

7. In the F12 Developer Tools window, click **Script**.
8. Click in the margin next to line 12, in order to create a breakpoint on the first statement inside the **onload()** function.

The **Script** window should look like this:

The screenshot shows the F12 Developer Tools window titled "Hello HTML5 - F12". The "Script" tab is selected. The code editor displays the following HTML and JavaScript:

```
file:///E:/Mod08/Democode/Document.html
7 <link rel="stylesheet" href="MyStylesheet.css">
8
9 <script type="text/javascript">
10
11     function onload() {
12         var hyperlinks = document.querySelectorAll("a");
13
14         for (var i = 0; i < hyperlinks.length; i++) {
15             hyperlinks[i].style.backgroundColor = "yellow";
16         }
17     }
18
19     window.addEventListener("load", onload, false);
20
21 </script>
22
23 </head>
24
25 <body>
26
27 <header>
28     <h1>Hello HTML5</h1>
29     <h2>Enjoy the ride!</h2>
30     <h3>Fancy eh!</h3>
31 </header>
32
33 <div id="MyMainContainer">
34
35     <nav>
36         <h3>Links</h3>
37         <a href="http://www.microsoft.com">Microsoft</a>
38         <a href="http://www.beautyoftheweb.com/">Beauty of the Web</a>
39     </nav>
40
41     <section>
42
43         <article>
44             <header>
45                 <h1>Welcome!</h1>
46             </header>
47         </article>
48     </section>
49
50 </div>
```

A red circular breakpoint marker is placed on line 12, indicating a breakpoint is set on the first statement of the **onload()** function.

FIGURE 8.1:THE SCRIPT WINDOW WITH A BREAKPOINT SET

Step Through JavaScript Code and Examine Variables

1. On the F12 Developer Tools toolbar, click **Start debugging**.
2. In the F12 Developer Tools window, verify that the debugger pauses execution at the breakpoint.
3. In the right pane, click the **Locals** tab.
4. On the F12 Developer Tools toolbar, click the **Step over** button several times to step through the code. In the **Locals** tab, verify that the value of the **i** variable changes as execution progresses.
5. Close the F12 Developer Tools window.
6. Close Internet Explorer.

Demonstration: Using the F12 Developer Tools to Profile a Web Application

In this demonstration, you will see how to examine the network traffic for a web application, and how to capture profiling information to enable you to identify the hot spots in a web application.

Demonstration Steps

Examine the Network Traffic for a Web Application

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, browse to the website <http://www.beautyoftheweb.com/>.
4. Press F12 to display F12 Developer Tools window.
5. In the F12 Developer Tools window, click **Network**.
6. On the F12 Developer Tools toolbar, click **Start capturing**.
7. In Internet Explorer, click the **touchgallery** icon in the navigation bar.
8. Return to the F12 Developer Tools window.

The network traffic should look like this (the actual values displayed may vary):

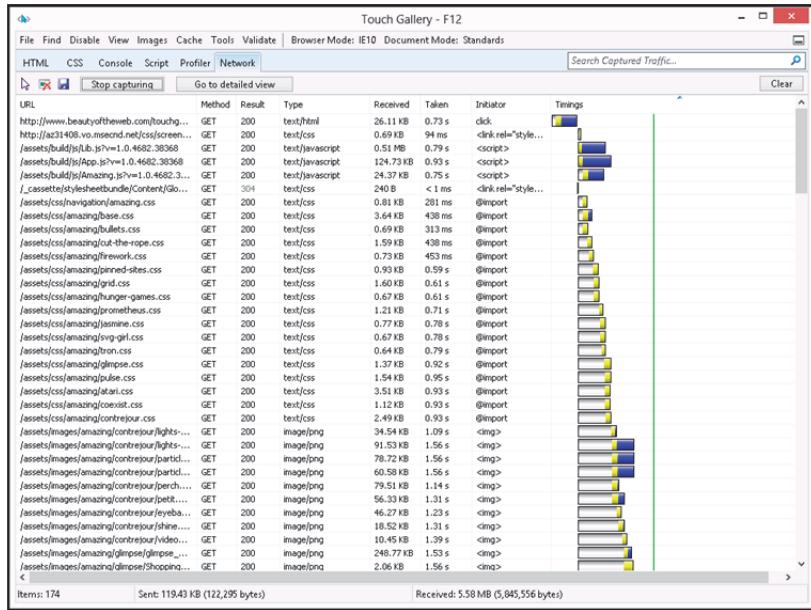


FIGURE 8.2:THE NETWORK TRAFFIC CAPTURED FOR A WEB PAGE

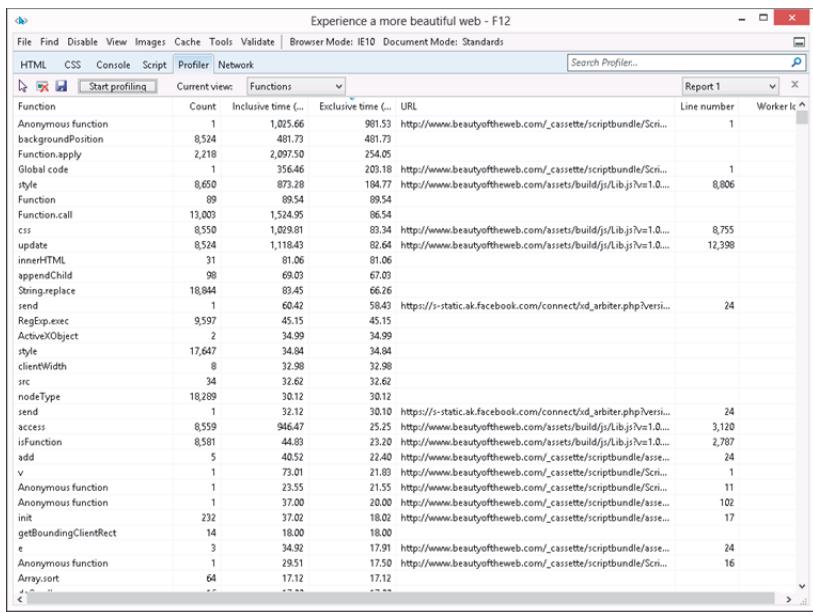
9. Click the first line of the network capture, and then on the toolbar click **Go to detailed view**.
10. Click each of the following tabs and show the data that they contain:
 - **Request headers.**
 - **Request body.**
 - **Response headers.**
 - **Response body.**
 - **Cookies.**
 - **Initiator.**
 - **Timings.**

11. On the F12 Developer Tools toolbar, click **Stop capturing**.

Capture Profile Data for a Web Application

1. In the F12 Developer Tools window, click **Profiler**.
2. On the F12 Developer Tools toolbar, click **Start profiling**.
3. In Internet Explorer, click the **videos** icon in the navigation bar.
4. Return to the F12 Developer Tools window.
5. On the F12 Developer Tools toolbar, click **Stop profiling**.

The captured profile data should look like this:

**FIGURE 8.3:PROFILE DATA FOR A WEB PAGE**

6. On the F12 Developer Tools toolbar, in the **Current view** list, click **Call tree**.
7. In the **Search Profiler** box, type **onWindowEvent** and then press **ENTER**.

In the captured data, examine the work performed by the **onWindowEvent** event handler.

8. Double-click **onWindowEvent** to display the code for this event handler in the **Script** window.
9. Press **Ctrl+Alt+O** to display the tools menu, and then select **Format JavaScript** to display the code in an easier to read format.
10. Close the F12 Developer Tools window.
11. Close Internet Explorer.

Demonstration: Creating Interactive Pages with HTML5 APIs

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Creating Interactive Pages with HTML5 APIs

Scenario

The ContosoConf organizers want to highlight the latest HTML5 technologies in order to create an interactive experience for people visiting the conference website. Specifically, the conference organizers have asked you to add the following features to the application:

- Conference speakers need a way to generate their badges. A web page should be added that enables a speaker to drag-and-drop a profile picture to start creating their badge.
- A video from a previous conference is available. This video should be available on the Home page.
- The Location page should be customized to display information about the visitor's current physical location.

Objectives

After completing this lab, you will be able to:

1. Add video to an HTML page.
 2. Use a drag-and-drop operation to create interactive pages, and to read files by using the File API.
 3. Get the location of the user by using the Geolocation API.
- Estimated Time: 60 minutes
 - Virtual Machines: 20480-SEA-DEV11, MSL-TMG1
 - User Name: Student
 - Password: Pa\$\$w0rd

Exercise 1: Dragging and Dropping Images

Scenario

In this exercise, you will begin working on the Speaker Badge page. This page will eventually enable conference speakers to create a badge displaying their name, photo, and ID barcode. In this exercise, you will implement drag-and-drop support so that an image of a speaker can be dropped onto the web page and displayed.

You will add event listeners to handle drag-and-drop events. Then you will use the File API's FileReader object to read a file as a data URL, which is then displayed on the page. Finally, you will run the application and test the Speaker Badge page.

The main tasks for this exercise are as follows:

1. Review the HTML markup and JavaScript code for the Speaker Badge page.
2. Add drag-and-drop event listeners.
3. Handle the drop event.
4. Read the image by using a FileReader.
5. Test the Speaker Badge page.

► Task 1: Review the HTML markup and JavaScript code for the Speaker Badge page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.

3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod08\Labfiles\Starter\Exercise 1** folder.
4. Open **speaker-badge.htm** and review the HTML markup for this page. Notice the **** element that will display the speaker's image when you have implemented the drag-and-drop functionality:

```
<img style="width: 300px; height: 300px; border: 1px solid #000"/>
```

5. Also notice that the page has script references for jQuery and the **/scripts/pages/speaker-badge.js** file:

```
<script src="/scripts/jquery.min.js" type="text/javascript"></script>
<script src="/scripts/pages/speaker-badge.js" type="text/javascript"></script>
```

6. Open the **speaker-badge.js** file in the **scripts/pages** folder, and review the JavaScript code. This code defines a **SpeakerBadgePage** object that controls the page:

```
var SpeakerBadgePage = Object.inherit({
  ...
});
```

- You will add code to the **initialize** method of **SpeakerBadgePage** object to configure drag and drop event handlers.
- The **handleDragOver** function has already been implemented; it simply changes the cursor to a "copy" cursor as an item is dragged over an element.
- You will add code to the **handleDrop** function to display the image in the file that the user drops onto an element.
- The **isImageType** function is a utility function that indicates whether the MIME type passed in as the parameter represents a valid image file.
- You will complete the code in the **readFile** method to read the contents of the file dropped on an element by the user.
- The **displayImage** function is another utility function that displays an image in the control referenced by the **imageElement** variable. The **imageElement** variable is created by the **initialize** method, and refers to the **** element on the speaker-badge.htm page.

► Task 2: Add drag-and-drop event listeners

1. In the **speaker-badge.js** file in the **initialize** function, after the comment that starts with // TODO: Add event listeners for element "dragover" and "drop" events, add event listeners for the **dragover** and **drop** events of the **element** variable. These handlers should call the **handleDragOver** and **handleDrop** methods.

 **Note:** The **element** variable contains a reference to the **** element that displays the speaker's image.

► Task 3: Handle the drop event

1. In the **speaker-badge.js** file, find the **handleDrop** function. Notice that the event that invoked the function is passed in as the parameter.
2. In this method, add code to perform the following tasks:

- After the comment // TODO: Get the files from the event, create a variable containing the array of files from the **dataTransfer** property of the event.
- After the comment // TODO: Read the first file in the array, add code to ensure that the array is not empty
- After the comment // Check the file type is an image, call the **isImageType()** function to ensure the first file is a type of image.



Note: The **isImageType()** function checks that the file specified as the parameter is a jpeg, jpg or png file.

- After the comment // Use this.readFile to read the file, then display the image, add code to start reading the first file by calling the **readFile** function.



Note: The **readFile** function is not yet complete. You will implement the missing parts of this function in the next task.

- When this operation is complete, call the **displayImage()** function.
 - If a non-image file was dropped, display an alert with the following message:

Please drop an image file.

► Task 4: Read the image by using a FileReader

1. In the **speaker-badge.js** file, review the **readFile** method.
2. Use the File API to implement the missing functionality in the **readFile** method:
 - After the comment // TODO: Create a new FileReader, create a new **FileReader** object and assign it to a variable named **reader**.
 - After the comments // TODO: Assign a callback function for reader.onload and // TODO: In the callback use reading.resolveWith(context, [fileDataUrl]); to return the file data URL, add code to assign a callback function to the **onload** property of the **reader** object.
 - In the callback, resolve the **reading** deferred object, passing it the result of reading the file.
 - After the comment // TODO: Start reading the file as a DataURL, add code that uses the **reader** object to start reading the file as a data URL.



Note: Reading a file is an asynchronous operation, so the **readFile** method returns a jQuery deferred object that is resolved after the file has been read.

► Task 5: Test the Speaker Badge page

1. Run the application and view the **speaker-badge.htm** page.



Note: The Speaker Badge page is not accessible from the menu bar in the application because this feature is only intended for use by speakers. To view this page, you must navigate directly to the speaker-badge.htm page on the web site.

2. Drag-and-drop the file **mark-hanson.jpg** in the **E:\Mod08\Labfiles\Resources** folder onto the Speaker Badge page's empty image element.
3. Verify that the speaker's photo is displayed within the image element.

The page should look like this:

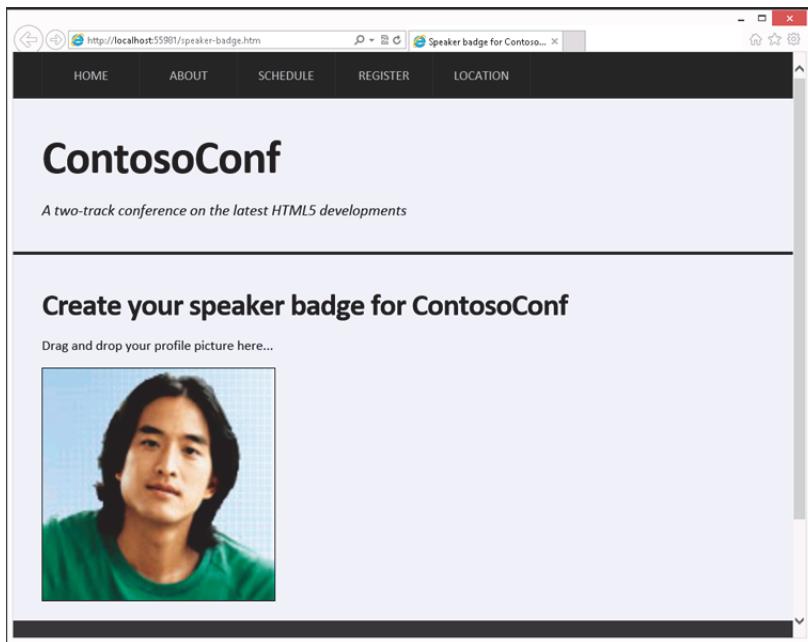


FIGURE 8.4:THE SPEAKER BADGE PAGE WITH THE SPEAKER'S PHOTO

4. Close Internet Explorer and File Explorer.

Results: After completing this exercise, you will have implemented functionality that enables the user to drag-and-drop an image from File Explorer onto the web page.

Exercise 2: Incorporating Video

Scenario

In this exercise, you will add a video to the website Home page. You will add custom controls that enable a user to play and pause the video, and then you will handle video events to display how much playback time has elapsed. Finally, you will run the application, view the Home page, and verify that it plays the video correctly.

The main tasks for this exercise are as follows:

1. Add a video player to the Home page.
2. Add controls to the video player.
3. Control the video by using JavaScript code.
4. Display the video elapsed time.
5. Test the video player.

► Task 1: Add a video player to the Home page

1. Open the **ContosoConf.sln** solution in the **E:\Mod08\Labfiles\Starter\Exercise 2** folder.

MCT USE ONLY. STUDENT - SEE PROHIBITED

2. Open the **index.htm** file and find the following comment:

```
<!-- TODO: Add video tag here -->
```

3. Add a **<video>** element, using the video source specified in the **index.htm** file.

► Task 2: Add controls to the video player

1. In the **index.htm** file after the **<video>** element, add a **<div>** element with the class **video-controls** and a style of **display:none**.
 - o This **<div>** element should contain two buttons; **Play** and **Pause**.
 - o Set the **class** of the **Play** button to **video-play** and set the **class** of the **Pause** button to **video-pause**.
2. Also add a **** element with the class **video-time** to the **<div> element**; you will use this span to display the elapsed time for the video.

 **Note:** The JavaScript code used by this page references the play, pause, and elapsed time elements on the page by using the classes specified in this task.

3. Add a reference to the **video.js** script in the **/scripts/pages** folder to the **index.htm** file.

► Task 3: Control the video by using JavaScript code

1. The custom video control elements are hidden when the page initially loads. Open the **video.js** file in the **scripts\pages** folder and review the code in this file.
2. Complete the **ready** function to display the video control elements (use the **block** style). The video controls are available in the **controls** variable.
3. Near the bottom of the file, add an event listener for the **loadeddata** event of the video, which calls the **ready** function.
4. In the **video.js** file, complete the **play** and **pause** functions.
 - o In the **play** method, start the video playing, hide the play button, and display the pause button.
 - o In the **pause** method, pause the video, hide the pause button, and display the play button.

 **Note:** The **video** variable holds a reference to the video player. Display or hide a control by setting the **style.display** property of the control.

The **video** variable holds a reference to the video player.

Display or hide a control by setting the **style.display** property of the control.

5. Add click event listeners for the play and pause buttons, which call the **play** and **pause** functions, respectively.

► Task 4: Display the video elapsed time

1. Complete the **updateTime** function, to display the elapsed time of the video in the time DOM element.
 - o Use the **formatTime** helper function to convert the total seconds into HH:MM:SS format.

2. Near the end of the `video.js` file, add a **timeupdate** event listener for the `video` element that invokes the **updateTime** function.

► **Task 5: Test the video player**

1. Run the application and view the **index.htm** page.
2. Wait for the video to load and the **Play** button to appear.
3. Click **Play**, verify that the video begins to play and that the elapsed time is displayed.
4. Click **Pause** and verify that the video pauses.



Note: You will not hear any sound because the virtual machine does not support audio.

Results: After completing this exercise, you will have added a video player to the Home page.

Exercise 3: Using the Geolocation API to Report the User's Current Location

Scenario

In this exercise, you will modify the `Location` page to react to the current geographic location of the user viewing the page.

You will use the Geolocation API to get the visitor's current location, and then you will calculate and display the distance to the conference venue. Finally, you will run the application and verify that this feature is working as expected.

The main tasks for this exercise are as follows:

1. Review the HTML markup and JavaScript code.
2. Get the current location of the user viewing the page.
3. Display the distance to the conference venue.
4. Test the `Location` page.

► **Task 1: Review the HTML markup and JavaScript code**

1. Open the **ContosoConf.sln** solution in the **E:\Mod08\Labfiles\Starter\Exercise 3** folder.
2. Open the **location.htm** file and review the HTML markup for this page. Notice that the page includes the following empty heading:

```
<h2 id="distance"></h2>
```

- You will display the distance of the user to the conference venue in this heading later in this exercise.
3. Review the JavaScript code for this page in the **location.js** file, and note that the conference venue location is stored in a variable named **conferenceLocation**:

```
var conferenceLocation = {  
    latitude: 47.6097, // decimal degrees  
    longitude: 122.3331 // decimal degrees  
};
```

► **Task 2: Get the current location of the user viewing the page**

1. Near the end of the **location.js** file find the comment:

```
// TODO: Get current position from the geolocation API.
```

2. After this comment, add code that uses the Geolocation API to get the current location of the user viewing the page.
 - o Use the callback function **updateUIForPosition** to process the location.



Note: The **updateUIForPosition** function will display the distance of the user from the conference venue. This function is not yet fully implemented; you will complete it in the next task.

- Use the callback function **error** to handle any errors that might occur.

► **Task 3: Display the distance to the conference venue**

1. In the **location.js** file, complete the **updateUIForPosition** function:

- o Create a variable named **distance**.
- o Use the **distanceFromConference** function to calculate the distance to the conference venue. This function requires the coordinates of the user's current location as a parameter.

► **Task 4: Test the Location page**

1. Run the application and view the **location.htm** page.



Note: If Internet Explorer displays the message **localhost wants to track your physical location**, click **Allow once**.

2. Verify that the page displays the distance to the conference venue in miles.

Results: After completing this exercise, you will have a Location page that displays the distance of the user from the conference venue.

Module Review and Takeaways

In this module, you have seen how to use HTML5 APIs to access the local file system, support drag-and-drop data operations, play multimedia files natively in a web page, and obtain geolocation information. You have also learned how to debug and profile web applications by using the F12 Developer Tools in Internet Explorer.

Review Question(s)

Question: What methods are provided by the **FileReader** interface for reading files on the local file system?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
HTML5 browsers are guaranteed to support the .mp4 video format. True or false?	

Question: What methods are provided by the **navigator.geolocation** object for obtaining geolocation information?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
The F12 Developer Tools require that you have Visual Studio installed on your computer before you can use them to debug JavaScript code. True or False?	

MCT USE ONLY STUDENT USE PROHIBITED

Module 9

Adding Offline Support to Web Applications

Contents:

Module Overview	9-1
Lesson 1: Reading and Writing Data Locally	9-2
Lesson 2: Adding Offline Support by Using the Application Cache	9-10
Lab: Adding Offline Support to Web Applications	9-15
Module Review and Takeaways	9-21

Module Overview

Web applications have a dependency on being able to connect to a network to fetch web pages and data. However, in some environments a network connection may be intermittent. In these situations, it might be useful to enable the application to continue functioning by using data cached on the user's device. HTML5 provides a choice of new client-side storage options, including **session storage** and **local storage**, and a resource caching mechanism called the ***Application Cache***.

In this module, you will learn how to use these technologies to create robust web applications that can continue running even when a network connection is unavailable.

Objectives

After completing this module, you will be able to:

- Save data locally on the user's device, and access this data from a web application.
- Configure a web application to support offline operations by using the Application Cache.

Lesson 1

Reading and Writing Data Locally

One key strategy that a web application can use to cache data is to store it locally in the file system of the user's device. This is not a new strategy, and it has formed the basis of several mechanisms used by web servers to simulate user sessions. This lesson describes some of the technologies that are currently available.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how a web application can use **cookies** to maintain simple state information.
- Use the **Session Storage API** to save session state information in a web application.
- Use the **Local Storage API** to persist information between sessions.
- Use **storage events** to notify an application of changes made to stored data.
- Describe how the **Indexed Database API** can implement a structured data store on the user's device.

Maintaining Session State Information by Using Cookies

At their best, web applications are stateless, meaning that a web server has no notion of a continuous client session. When a user connects to a website, the web server simply serves the page requested by the user's browser. If the user views another page, the web server retrieves that page and sends it to the user's browser. As far as the web server is concerned, the two requests are completely independent. This mechanism can pose a problem if the second page requires data that the user entered on the first. To facilitate the seamless connection among web pages, the

browser needs a mechanism to store information representing the state of the client's session. The browser can transmit this data as part of any request to view a page in the website so that the server can associate the page request with data previously entered by the user. This is the basic premise behind the *cookie* protocol.

A cookie is nothing more than a text file with a small number of fields that can be used to identify the user's session and cache information about the user. Cookies can persist data between browser sessions in order to provide a seamless experience when a user revisits a web site. A good example of this continuity is the Amazon® website, where even if you have not logged in, Amazon can recognize you, and personalizes your pages according to its analysis of your interests and preferences. This personalization is achieved by using cookies; information about a user's preferences is stored locally on the user's computer, and this information is transmitted as part of each request sent to the Amazon website.

Cookies can only store a very small amount of data, **up to 4 KB**. There is an important reason for this limitation. Cookies are designed to be sent to and from the web server on each and every page request. If they were any bigger, their bandwidth requirements could have a significant impact on the responsiveness of the web application. However, just as the web has evolved, so has the use of cookies. Increasingly, they are used to store far more than just session tokens. Cookies have become a place to store important user profile data, page history, and other data.

• **Cookies:**

- Were designed to implement session tokens
- Are sent to the server on every page request
- Are small files of limited size, up to 4 KB
- Are open to abuse
- Have no synchronization or concurrency mechanism

• Cookies were not designed for general-purpose data storage

The grave danger in keeping user data in cookies is that information can be shared unintentionally (or otherwise) between websites, and in recent times they have become notorious for betraying users' personal information without their consent. Throughout the European Union, recent legislation forces websites hosted in these countries to notify a user if they use cookies, and to enable the user to disable them.

Another issue to be aware of is that users often open several tabs when browsing a single site. The data held in cookies is shared between these tabs, but **cookies have no defined synchronization or concurrency mechanism**. This can lead to undesirable effects if the values held in a cookie are changed in one tab, and are then used in the second tab mid-process.

Since cookies are designed to associate a user session with a web application, they are mapped to URLs either at the root level or a subdirectory level, and to individual pages, betraying their allegiance to a time when web servers were essentially just file servers, and applications were essentially just HTML pages in folders. This is no longer an accurate paradigm. For example, the model-view-controller (MVC) pattern implemented by many common web applications **disassociates the physical server file structure from the URLs** used to fetch pages and other resources.

In an increasingly semantic web, cookies are becoming less useful, and clearly raise some significant concerns if misused. Cookies were never designed to be used as a wholesale storage technology, and HTML5 introduces new client-side storage paradigms to overcome many of shortcomings of cookies. These technologies include the Session Storage API and the Local Storage API.

Persisting Session Data by Using Session Storage

Session storage is a browser-persistence mechanism that can store text data on the device running the browser. As the name implies, data kept in this store lasts for the duration of the current user session. When the user closes the browser, the session store is cleared automatically.

Session storage is widely supported and has been implemented by the major browsers, including Internet Explorer, since 2009. You access session storage by using the **sessionStorage** property of the **window** object. You can test whether a browser implements session storage by querying for the presence of this property, like this:

```
if( window.sessionStorage ){
    ...
}
```

Session storage associates each item of session data with a unique key value; you provide this key value when you store a value, and use this same key value to retrieve the data. The Session Storage API provides three ways to store and retrieve data:

- **setItem** and **getItem** functions. The **setItem** function expects you to provide the key and the data to store. The **getItem** function uses the key to return the data. If there is no data with the specified key, the value returned is **null**.

- Use the **sessionStorage** object to store and retrieve text data for a session:

```
sessionStorage.setItem("myKey", "some text value");
var textFromSession1 = sessionStorage.getItem("myKey");

sessionStorage["myKey"] = "some text value";
var textFromSession2 = sessionStorage["myKey"];

sessionStorage.myKey = "some text value";
var textFromSession3 = sessionStorage.myKey;
```

- Session data is only available in the session that creates it

- Session storage is cleared when the user finishes the browser session

```
sessionStorage.setItem("myKey", "some text value");
var textFromSession = sessionStorage.getItem("myKey");
```

- name-key pair. You can use array notation, and specify the key value as the array index.

```
sessionStorage["myKey"] = "some text value";
var textFromSession2 = sessionStorage["myKey"];
```

- pseudo-properties. You can add a property for each key to the **sessionStorage** object.

```
sessionStorage.myKey = "some text value";
var textFromSession3 = sessionStorage.myKey;
```



Note: You do not have to prefix these calls with the **window** object. You can call them directly because **window** is the default calling context.

If you need to persist objects in session storage, serialize them as JSON strings by using the **JSON.stringify()** function.

Objects in the session store are also accessible as an array of items with a **long** index. You can test the **length** property to find out how many keys are in the **sessionStorage** object. You can retrieve the key for each object by using the **key()** function. This information can be very useful for iterating over items, as shown in the next code example, which creates a list of the keys in the **sessionStorage** object and displays them in a **<div>** element :

```
var listDiv = document.getElementById("myList");
for(var i=0; i<sessionStorage.length; i++)
{
    listDiv.innerHTML += "<br />" + sessionStorage.key(i);
}
```

To remove an item from session storage, use the **removeItem()** method.

```
sessionStorage.removeItem("myKey");
```

To clear a session store before the end of the session, call the **clear()** method:

```
sessionStorage.clear();
```

Remember that if you don't clear the data for a session, it will be cleared automatically when the user closes the browser window. Consequently, session storage may be of limited use for applications where you need to preserve user data between sessions. This is where the Local Storage API may prove more useful.



Additional Reading: For more information about the session storage API, visit <http://go.microsoft.com/fwlink/?LinkId=267733>.

Persisting Data Across Sessions by Using Local Storage

Local Storage also enables you to store data items on the client browser, but unlike session storage, items in local storage are persisted even after the browser session has ended. The data is stored in the file system of the device running the browser. It is removed when the web application deletes it, or when the user requests that the browser remove it. This mechanism is browser dependent.

Data persisted in local storage is available across different web pages, although it is only available to web pages running as part of the website that stored the data; you cannot use local storage to share data between pages originating from different websites.

You access local storage by using the **localStorage** property of the window object. You can detect whether the browser supports local storage by querying for the presence of this property, as follows:

```
if( window.localStorage ){
    ...
}
```

The Local Storage API is very similar to the Session Storage API. You can store and retrieve data by using the **setItem()** and **getItem()** functions, a name-pair key, or pseudo-properties.

```
localStorage.setItem("myKey","some text value");
var textData = localStorage.getItem("myKey");
localStorage["myKey"] = "some text value";
var textData = localStorage["myKey"];
localStorage.myKey = "some text value";
var textData = localStorage.myKey;
```

You can determine the number of items in local storage by using the **length** property, and iterate over items and retrieve data by using the **key()** function, as shown in the following example:

```
var listDiv = document.getElementById("myList");
for(var i=0; i<localStorage.length; i++)
{
    listDiv.innerHTML += "<br />" + localStorage.key(i);
}
```

To remove an item from the store, call the **removeItem()** function:

```
localStorage.removeItem("myKey");
```

To remove all items from the store, call the **clear()** function:

```
localStorage.clear();
```

Objects stored in local storage do not have a specified size restriction. Instead, the store is free to grow to an upper size limit decided by the user and configured in the browser settings. This makes it ideal to store large amounts of data useful to the client-side application. As with the session storage API, you should serialize objects as text by using the **JSON.stringify()** function before storing them.

Using local storage instead of relying on server round trips has a significant impact on the user experience. Data found on the client can be displayed almost instantly, improving the responsiveness of the website.

- Use the **localStorage** object to persist data across sessions and web pages:

```
localStorage.setItem("myKey", "some text value");
var textData = localStorage.getItem("myKey");

localStorage["myKey"] = "some text value";
var textData = sessionStorage["myKey"];

localStorage.myKey = "some text value";
var textData = sessionStorage.myKey;
```

- Data is persisted until it is explicitly removed



Additional Reading: For more information about the local storage API, visit <http://go.microsoft.com/fwlink/?LinkId=267734>.

Handling Storage Events

The storage API to which both session and local storage conform includes a single event called **storage**. You can use this event to notify a web page of changes to data held in the store; it fires whenever data is modified.

The following example shows how to subscribe to this event:

```
function myStorageCallback( e ) {
    alert("Key:" + e.key + " changed to "
+ e.newValue);
}
window.addEventListener("storage",
myStorageCallback, true);
```

- Use the **storage** event to notify a web page of changes made to session and local storage:

```
function myStorageCallback( e ) {
    alert( "Key:" + e.key + " changed to " + e.newValue );
}
...
window.addEventListener("storage", myStorageCallback, true );
```

- Properties of the event object:

key	- name of the value which has changed
oldValue	- the original value
newValue	- the new value
url	- the origin of the event
storageArea	- a reference to the store that has changed

The event object passed to the event handler includes the following properties:

- **key**: The name of the value which has changed.
- **oldValue**: The original value before the change.
- **newValue**: The new value.
- **url**: The document whose script is the origin of the event.
- **storageArea**: A reference to the store that has changed (session or local).

The event model enables web pages to listen for storage events and to update themselves when data they use from the store changes state. For example, in Internet Explorer, if you have the same web page open in multiple tabs, the data on each tab can remain synchronized in order to reduce the scope for inconsistencies caused by changes made in different tabs.

Internet Explorer also defines the **storagecommit** event. The **storage** event fires when data changes, but the **storagecommit** event occurs when data held in local storage is actually written to disk.

```
function myStorageCommitCallback( e ) {
    alert("Data written to disk");
}
window.addEventListener("storagecommit", myStorageCommitCallback, true);
```



Reader Aid: For more information about handling storage events, visit <http://go.microsoft.com/fwlink/?LinkId=267735>.

Storing Structured Data by Using the Indexed Database API

The Indexed Database API, or IndexedDB, provides an efficient mechanism for storing, retrieving, and searching for structured data held locally on the device running the browser. You access IndexedDB by using the **indexedDB** property of the **window** object.

You store data in a named database; you connect to a database by creating a request object that references the **open()** function. If the database does not exist, the **open()** function creates it. The IndexedDB API is asynchronous, and you use the **onsuccess** event to capture the value returned by functions such as **open()**. If a function fails, you can determine the reason for the failure by handling the **onerror** event. The following example shows how to open a database and obtain a reference that you can use for storing and retrieving data. The **db** variable holds a reference to the database, if it is opened successfully:

```
var db; // Reference to the database to use
var openRequest = indexedDB.open("contosoDB");
openRequest.onsuccess = function(event) {
    db = event.target.result;
};
openRequest.onerror = function(event) {
    alert("Error " + event.target.errorCode + " occurred while opening the database");
};
```

 **Note:** Request objects execute when they go out of scope; that is, when the current JavaScript block finishes. For this reason, in the example code above, it is perfectly safe to assign the **onsuccess** and **onerror** callbacks after setting the **openRequest** variable because the **open()** function will not run (and invoke the callbacks) until control reaches the end of the JavaScript block.

A database holds one or more object stores, which are analogous to tables in a relational database. You define an object store by using the **createObjectStore()** function; you specify an object to add to the store together with the name of the key property that the IndexedDB API can use to retrieve the object. The following example creates an object store for holding the details of conference attendees. The object store is initialized with the details of the first attendee—Rachel Valdez. The **id** property is the key to the object store:

```
var attendee = {
    id: "1",
    name: "Rachel Valdez",
    password: "Pa$$w0rd"
};
var attendeeStore = db.createObjectStore("attendees", { keyPath: "id" });
```

 **Additional Reading:** You should only create a new object store in the context of a **VERSION_CHANGE** transaction. At the time of writing, the IndexedDB API specification is currently fluid in this area. Some vendors initialize a **VERSION_CHANGE** transaction by using the **setVersion()** function of a database, while other vendors start a **VERSION_CHANGE** transaction

implicitly if you specify the version number as a second parameter for the **open()** function of the **indexedDB** object; this is the approach that Internet Explorer 10 takes:

```
// Create version 2 of the database and start a version change transaction
var openRequest = indexedDB.open("contosoDB", 2);
Internet Explorer triggers the upgradeneeded event, which runs in the context of the version
change transaction. You can use this event to add the object store to the database:
openRequest.onupgradeneeded = function(event) {
    var db = event.target.result;
    var attendee = {
        id: "1",
        name: "Rachel Valdez",
        password: "Pa$$w0rd"
    };
    var attendeeStore = db.createObjectStore("attendees", { keyPath: "id" });
}
```

For more information about opening a database and initiating a version change transaction, see the latest version of the IndexedDB specification at <http://go.microsoft.com/fwlink/?LinkId=267736>.

You can use the **add()** function to store additional records in an object store. The next example adds the details for Eric Gruber to the store. Notice that the **add()** function is asynchronous:

```
var newAttendee = {
    id: "2",
    name: "Eric Gruber",
    password: "Pa$$w0rd"
};
var addRequest = attendeeStore.add(newAttendee);
addRequest.onsuccess = function(event) {
    // Attendee was successfully added
}
addRequest.onerror = function(event) {
    // Handle failure
};
```

If a record with the same key value as the new item already exists, the **add()** function fails and raises the **error** event.

To modify an existing record, use the **put()** function, as follows:

```
var updatedAttendee = {
    id: "2",           // Id of existing attendee
    name: "Eric Gruber",
    password: "P@ssw0rd" // Change the password
};
var updateRequest = attendeeStore.put(updatedAttendee);
updateRequest.onsuccess = function(event) {
    // Attendee was successfully updated
}
updateRequest.onerror = function(event) {
    // Handle failure
};
```

You can use the **delete()** function to remove an object from the object store. Specify the key of the object as the parameter. If there is no matching object, the **error** event is raised:

```
var deleteRequest = attendeeStore.delete("1"); // Remove the details for Rachel Valdez
deleteRequest.onsuccess = function(event) {
    // Attendee was successfully deleted
```

```
}

deleteRequest.onerror = function(event) {
    // Handle failure
};
```

To find data in the object store, you can use the **get()** function and specify the key of the object to retrieve. Again, if there is no matching object, the **error** event occurs:

```
var attendee;
var getRequest = attendeeStore.get("2"); // Retrieve the details for Eric Gruber
getRequest.onsuccess = function(event) {
    // Attendee details are available in event.target.result
    attendee = event.target.result;
}
getRequest.onerror = function() {
    // Handle failure
};
```

The IndexedDB API defines many more features. For example, you can create transactions if you need to batch operations together, you can use a cursor to fetch multiple records from an object store, and you can define indexes to speed up queries that retrieve objects by using specified properties.

 **Note:** To use the IndexedDB API, Internet Explorer must be configured to allow website caches and databases. This option is available on the **Settings** page of the **Internet Options** dialog in Internet Explorer; click the **Caches and databases** tab, and then select **Allow website caches and databases**.

 **Additional Reading:** For more information about using the IndexedDB API, visit <http://go.microsoft.com/fwlink/?LinkId=267737>.

Lesson 2

Adding Offline Support by Using the Application Cache

The session storage, local storage, and IndexedDB APIs provide a programmer-centric model for storing and managing data locally on a user's device. In addition, HTML5 features a mechanism for caching web pages and other resources so that once they are downloaded to the client machine, the browser defers to this cache and not the network. After the page is downloaded, there is no need to retrieve it again. This saves on network resources, and enables developers to build web applications that do not even need web connectivity once they are client-side. The fact that a web page or other resource is cached is transparent to the JavaScript code running on a web page, although the Application Cache API includes functions and objects that enable JavaScript code to detect whether a page is running online or offline, and to refresh the cached resources if a network connection is available.

In this lesson, you will learn how to configure and use the HTML5 application cache in a website.

Lesson Objectives

After completing this lesson, you will be able to:

- Configure the application cache.
- Detect the state of the application cache.
- Refresh the application cache.
- Test for network connectivity.

Configuring the Application Cache

Large parts of web applications are simply file-based resources such as HTML pages, CSS files, JavaScript files, and images. There is little point in downloading such static files over and over, and yet this is what a typical browser session does.

The application cache is a client-side storage mechanism that enables the developer to explicitly declare which static files should be cached by the browser. You can use this mechanism to create websites that run just as well offline as they do online, making it ideal for developing robust mobile applications that have to run with intermittent network connections.

The cache manifest file specifies the data that the web browser should retain in the application cache. This file is a list of resources divided into separate sections labeled **CACHE**, **NETWORK**, and **FALLBACK**. The information in these sections determines how the browser responds when a request is made for a resource when the web application is running in the online or offline states.

To add a manifest file to an application, create a new text file and store it in the root folder of the web application. In this file, list all the static resources that should be downloaded and cached. This list will most likely include any graphics and images in the application, any HTML pages that do not have URL data dependencies, CSS files and JavaScript files, and so on. An example manifest file looks like this:

- The application cache manifest file specifies the resources to cache, and how they should be updated:

```
CACHE MANIFEST
CACHE:
index.html
verification.js
site.css
graphics/logo.jpg
```

- Each web page that uses cached resources should reference the manifest file:

```
<html manifest="appcache.manifest">
```

```
NETWORK:
/login

# alternatives paths
FALLBACK:
/ajax/account/ /noCode.htm
```

CACHE MANIFEST

CACHE:

```
index.html  
verification.js  
site.css  
graphics/logo.jpg  
NETWORK:  
login  
# alternatives paths  
FALLBACK:  
ajax/account/    noCode.htm
```

This file should have the .manifest file extension.

 **Note:** The **.manifest** extension file is of a new MIME type called **text/cache-manifest**. You may need to configure the web server to serve this type of file by adding this new MIME type, if it is not already set up.

The cache manifest file starts with the line **CACHE MANIFEST**; if this line is missing, the file may not be recognized as a manifest file. The file can contain the following sections:

- **CACHE:** Resources listed in this section are downloaded once, when the web page is initially loaded into the user's browser. Thereafter, the cached version of these resources will be used and they will not be updated from the server.
- **NETWORK:** Resources listed in this section will always be downloaded if the network is available. They are not cached.
- **FALLBACK:** Resources listed in this section are not cached, but you provide an alternative URL for them should the server become unavailable. In the example shown, all URLs prefixed with the **ajax/account/** path will be replaced with the **noCode.htm** file if they cannot be retrieved. The alternative resources, such as the **noCode.htm** file in the example, *are* cached.

 **Note:** Be sure to add the colon after the **CACHE**, **NETWORK**, and **FALLBACK** keywords, otherwise they might not be recognized. If you do not specify any sections, then **CACHE** is assumed.

You can add **comments** to the manifest file by creating a new line starting with the pound symbol, #.

To use the application cache, each web page must reference the manifest file that lists the resources to cache. A website can contain multiple manifest files, and different web pages can reference different manifest files. You specify the name of the manifest file to use in a web page by adding the **manifest** attribute to the **<html>** element.

```
<html manifest="appcache.manifest">
```

 **Note:** As with the IndexedDB API, to use an application cache, Internet Explorer must be configured to allow website caches and databases. This option is available on the **Settings** page of the **Internet Options** dialog in Internet Explorer; click the **Caches and databases** tab, and then select **Allow website caches and databases**.

Monitoring with the Application Cache

The contents of the application cache are managed by the browser by using the configuration specified in the cache manifest file. However you can monitor the application cache from JavaScript code by using the Application Cache API.

The application cache is accessible to JavaScript code through the **applicationCache** property of the **window** object. The application cache object returned by this property introduces a comprehensive event model covering the lifecycle and behavior of the cache. These **events** include the following:

- **checking**: This event fires when the browser examines the application cache for **updates**.
- **downloading**: This event fires when the browser **starts downloading** resources to the application cache.
- **updateready**: This event fires when the new version of the cached objects for a web page have been **downloaded**.
- **obsolete**: This event fires if the manifest file is **no longer available** and the application cache is no longer valid for the current web page.
- **cached**: This event fires when the application cache is **ready and available** for use.
- **error**: This event fires if an **error** occurs while downloading resources to cache, or when checking for resources to download.
- **noupdate**: This event fires if no changes were found after checking the manifest for updates.
- **progress**: This event fires as **each resource** specified in the manifest is **downloaded** to the application cache.

The following example shows how to catch the **error** event of the application cache:

```
applicationCache.addEventListener( "error", function() {
    alert( "Error while downloading resources to the application cache");
}, true );
```

 **Note:** Just as with the local storage and session storage objects, you do not have to reference the **applicationCache** property from the **window** object, because that is the default context.

The application cache also implements a numeric **status** property that can be tested independently of the event model. This property can have one of the following values:

Status	Meaning	Description
0	UNCACHED	The page is not associated with a cache. No resources have been downloaded
1	IDLE	All cached resources have been downloaded. The

Status	Meaning	Description
		cached event has been fired.
2	CHECKING	The cache is being checked for updates to download. The checking event has been fired. If no updates are found, the noupdate event is fired.
3	DOWNLOADING	Resources are being downloaded to the cache. The downloading event has been fired.
4	UPDATEREADY	The cache has been updated with new resources, and all resources have been downloaded. The updateready event has been fired.
5	OBSOLETE	The manifest is missing and no cache is available. The obsolete event has been fired.

 **Additional Reading:** For more information about using the Application Cache API to monitor the application cache, visit <http://go.microsoft.com/fwlink/?LinkId=267738>.

Triggering Resource Updates by Using the Manifest

The behavior of the cache depends entirely on updates to the manifest file. Making a change to a resource on the server no longer guarantees the browser will get the latest version of the file; if a web page caches a resource, the resource will be loaded from the application cache even if there is a newer version on the server.

To force an update to get the new version of an existing resource, you must make a significant change to the manifest file. Simply updating the last modified date is not enough to trigger a complete refresh on the browser. The best way to force an update is to add a comments field in the manifest with a version number, for example:

```
#version=1.2.3
```

A change to this comment will be acknowledged as a change to the manifest file, for example:

```
#version=1.2.4
```

You can also use the **update()** function of the **applicationCache** object to initiate a check for updates, similar to the one performed when a web page is first loaded. Any existing cached resources will continue to be used until the page is reloaded or you invoke the **swapCache()** function of the **applicationCache** object.

In the code example below, the **swapCache()** function is called if the cache has been updated with new resources (status code 4 is the **UPDATEREADY** status). This code forces the web page to use the new resources.

```
applicationCache.update();
```

- A web page may continue to use cached resources even if newer versions are available
- To force an update:
 - Make a significant change to the manifest file, or
 - Initiate a check for updates by using the **update()** function, and then use the **swapCache()** function

```
applicationCache.update();
...
if (applicationCache.status == 4) {
  applicationCache.swapCache();
}
```

```
...  
if (applicationCache.status == 4) {  
    applicationCache.swapCache();  
}
```

Testing for Network Connectivity

Sometimes it makes sense to temporarily hide or disable functionality in a web application if no network connection is available. For example, if a web page expects a user to enter and submit data to a server by using a form, then this feature will not work unless an active link to the server is available. In situations such as this it does not make sense to cache the web page.

You can detect the network connectivity in an application by using the **onLine** property of the **navigator** object. This is a Boolean property that is true if a network connection is available, false otherwise. The **navigator** object also provides two events called **online** and **offline**. These events fire when the network state changes.

The following code example shows how to detect the network status of a page when it loads. The **onload** event handler examines the **onLine** property of the **navigator** object and displays the status in the **statusDiv** div on the page. The **online** and **offline** event handlers fire and update the displayed status if the network connectivity changes.

```
var s;  
function onlineStatus() {  
    s.innerHTML = "Online.";  
}  
function offlineStatus() {  
    s.innerHTML = "Offline.";  
}  
onload = function() {  
    s = document.getElementById("statusDiv");  
    if( navigator.onLine ) {  
        onlineStatus();  
    } else {  
        offlineStatus();  
    }  
    window.addEventListener("online", onlineStatus, true);  
    window.addEventListener("offline", offlineStatus, true);  
}
```

- Sometimes it is better to disable functionality that requires a network connection
- Use the **onLine** property of the **navigator** object to detect the network status
- Handle the **online** and **offline** events of the **window** object to track changes to network connectivity

Demonstration: Adding Offline Support to Web Applications

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Adding Offline Support to Web Applications

Scenario

The conference organizers are concerned that the venue has poor Wi-Fi coverage in some locations, meaning that attendees might not always be able to access the conference website on their tablets and laptops. The Schedule page is especially important because attendees need to know when sessions are running.

You have decided to make parts of the web application available offline by using the offline web application features of HTML5. After an attendee has visited the online website once, their browser will have downloaded and cached the important pages. If a Wi-Fi connection is unavailable, the attendee will still be able to view the website by using the cached information.

Objectives

After completing this lab, you will be able to:

1. Use the Application Cache API to make web pages available offline.
2. Use the Local Storage API to persist user data locally between browser sessions.
 - Estimated Time: 60 minutes
 - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
 - User Name: **Student**
 - Password: **Pa\$\$w0rd**

Exercise 1: Caching Offline Data by Using the Application Cache API

Scenario

In this exercise, you will make the **Home**, **About**, **Schedule**, and **Location** pages available offline.

First, you will complete the creation of an application manifest file, which lists all of the files that should be cached for offline access. Next, you will reference the manifest file from the **Home**, **About**, **Schedule**, and **Location** pages. Then, you will write JavaScript code that adapts the page navigation, hiding links to pages that are not available offline. Finally, you will run the application and view the **Schedule** page. You will stop the web server and then reload the **Schedule** page to verify that it works offline.

The main tasks for this exercise are as follows:

1. Configure the application cache manifest.
2. Detect offline mode by using JavaScript code.
3. Test the application.
 - ▶ **Task 1: Configure the application cache manifest**
 1. Start the **MSL-TMG1** virtual machine if it is not already running.
 2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
 3. Configure Internet Explorer to enable support for application caching:
 - Start Internet Explorer.
 - In the Internet Explorer, press F10 to display the menu bar.
 - On the **Tools** menu, click **Internet options**.

- In the **Internet Options** dialog box, click **Settings**.
 - In the **Website Data Settings** dialog box, click the **Caches and databases** tab.
 - Select the **Allow website caches and databases** check box, and then click **OK**.
 - In the **Internet Options** dialog box, click **OK**.
 - Close Internet Explorer.
4. Start Visual Studio and open the **ContosoConf.sln** solution from the **E:\Mod09\Labfiles\Starter\Exercise 1** folder.
 5. Open the **appcache.manifest** file. This file contains a partial application manifest that lists the resources a web browser must cache for offline access.
 6. After the comment near the start of the file, insert the following URLs to cache the **Home**, **About**, **Schedule**, and **Location** pages:

```
/index.htm  
/about.htm  
/location.htm  
/schedule.htm
```

7. Add the **manifest="/appcache.manifest"** attribute to the **<html>** element of the **Home**, **About**, **Schedule**, and **Location** pages. These are the pages that will operate offline by using the application cache.

► Task 2: Detect offline mode by using JavaScript code

1. Open **offline.js** file in the **scripts** folder. This script contains methods that hide or show links depending on whether a page is currently running in online or offline mode. The pages that support offline mode reference this script.
2. After the comment // TODO: if currently offline, hide navigation links that require online, add JavaScript code to detect if the web browser is currently offline, and call the **hideLinksThatRequireOnline()** function.
 - Examine the **navigator.onLine** property to determine whether the browser is online or offline.
3. After the comment that starts with // TODO: add onoffline and ononline events to document.body, handle the **ononline** and **onoffline** events for the document body; call the **hideLinksThatRequireOnline** and **showLinks** functions as appropriate.
4. After the comment // TODO: also handle the applicationCache error event to hide links, add an event listener to the **applicationCache error** event, which calls the **hideLinksThatRequireOnline** function.

► Task 3: Test the application

1. Run the application and view the **index.htm** page.
2. Expand the Windows notification area, right click **IIS Express**, and then click **Exit**. Allow IIS Express to exit when prompted.

 **Note:** When IIS Express first starts running, the IIS Express icon may appear in the Windows task bar rather than the notification area. If this occurs, right-click the **IIS Express** icon and then click **Exit**.

The following image shows the IIS Express icon.

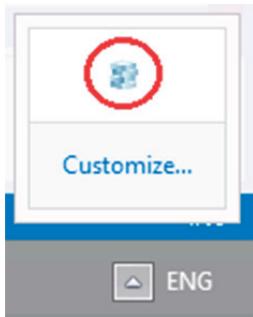


FIGURE 9.1:THE IIS EXPRESS ICON IN THE WINDOWS NOTIFICATION AREA

3. In Internet Explorer, view the **Schedule** page. Verify that the page loads and displays the schedule information. Also verify that, after a brief delay, the **Register** link in the navigation bar is removed. This occurs because the **Register** page is not cached (it is not included in the **appcache.manifest** file), and registration requires an active connection to the web site.
4. View the **About** page and verify that this page displays correctly, and that the **Register** link disappears from the navigation bar.
5. Close Internet Explorer.

Results: After completing this exercise, you will have modified the web application and made the **Home**, **About**, **Schedule**, and **Location** pages available offline.

Exercise 2: Persisting User Data by Using the Local Storage API

Scenario

Currently, the **Schedule** page is able to display sessions when offline, but this information does not include whether the attendee selected the session by using the star icon. This information is stored on a remote server that is inaccessible when the application is running offline. However, attendees also need to see the sessions they have selected. This information should be saved locally on an attendee's computer, so it is available offline.

In this exercise, you will update the JavaScript code for the Schedule page to record an attendee's selected sessions. You will create an object that wraps the Local Storage API. Then you will use this wrapper to save information locally about starred sessions. Finally, you will run the application, view the Schedule page, and verify that the starred sessions are persisted even when the web server is not available.



Note: The Local Storage API is very simple. It can only save and load string key-value pairs. Persisting more complex data requires serialization. In this exercise, you will use a wrapper around the Local Storage API.

The main tasks for this exercise are as follows:

1. Observe the current behavior of the Schedule page.
2. Save information about starred session to local storage.
3. Load information about starred session from local storage.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Use the local storage wrapper to save and load data in the Schedule page.
5. Test the application.
6. Reset Internet Explorer caching.

► **Task 1: Observe the current behavior of the Schedule page**

1. In Visual Studio, open the **ContosoConf.sln** solution in the **E:\Mod09\Labfiles\Starter\Exercise 2** folder.
2. Run the application and view the **schedule.htm** page.
3. Close IIS Express by using the icon in the Windows notification area.
4. On the Schedule page, click some of the stars, and then refresh the page.
5. Notice that information about which sessions are starred is not saved when the application is running offline (the yellow stars turn white again).
6. Close Internet Explorer.

► **Task 2: Save information about starred session to local storage**

1. In the **scripts** folder, open the **LocalStarStorage.js** file. The code in this file wraps the Local Storage API to provide a higher-level interface used by the application code.



Note: The Schedule page will use the **addStar**, **removeStar**, **isStarred**, and **initialize** functions defined in this file. These functions in turn call the **save** and **load** methods that you will implement in this task.

2. Implement the **LocalStarStorage save** method.
 - The IDs of the starred sessions to be saved are held in the array **this.sessions**.
 - Convert the array to a string by using the **JSON.stringify** function, and save this string to local storage with the key **stars**.

► **Task 3: Load information about starred session from local storage**

1. In the **LocalStarStorage.js** file, implement the **LocalStarStorage load** method.
 - Get the **stars** item from local storage.
 - Parse the JSON string into an array and assign it to **this.sessions**.
 - Handle the cases when the local storage returns no data or invalid JSON. In these situations, set **this.sessions** to be an empty array.

► **Task 4: Use the local storage wrapper to save and load data in the Schedule page**

1. In the **scripts\pages** folder, open the **schedule.js** page.
2. In the **initialize** method, find the following comment:

```
// TODO: Check if item is starred
```

3. Add JavaScript code after this comment to:
 - Detect whether the session identified by **this.id** is starred (use the **isStarred** method of the **localStarStorage** object).

- If the session is starred, tag the element with the **starred** CSS class so that it is rendered correctly (use the statement `this.element.classList.add(this.starredClass);`).

4. In the **unsetStar** method, find the following comment:

```
// TODO: remove the star from the item
```

5. After this comment, add JavaScript code to remove the star from the session identified by **this.id**. Use the **removeStar** method of the **localStarStorage** object.

6. In the **setStar** method, find the following comment:

```
// TODO: add a star to the item
```

7. After this comment, add JavaScript code to add a star to the session identified by **this.id**. Use the **addStar** method of the **localStarStorage** object.

► Task 5: Test the application

1. Open the `appcache.manifest` file and insert a version comment near the top of the file. As follows:

```
CACHE MANIFEST  
# version 2  
...
```

 **Note:** The JavaScript files are listed in **appcache.manifest**, and they will have been cached by the web browser. Making a change to the **appcache.manifest** file forces the browser to download the updated scripts.

2. Run the application and view the **schedule.htm** page.
3. In Internet Explorer, refresh the page to ensure the updated scripts are used.
4. Close IIS Express by using the icon in the Windows notification area.
5. On the **Schedule** page, click some of the stars, and then refresh the page.
6. Verify that the same sessions are still starred.
7. Close Internet Explorer.

► Task 6: Reset Internet Explorer caching

1. Configure Internet Explorer to disable support for application caching:
 - Start Internet Explorer.
 - In the Internet Explorer, press F10 to display the menu bar.
 - On the **Tools** menu, click **Internet options**.
 - In the **Internet Options** dialog box, click **Settings**.
 - In the **Website Data Settings** dialog box, click the **Caches and databases** tab.
 - Clear the **Allow website caches and databases** check box, and then click **OK**.
 - In the **Internet Options** dialog box, click **OK**.
 - Close Internet Explorer.

Results: After completing this exercise, you will have updated the Schedule page to locally record starred sessions.

Module Review and Takeaways

In this module, you have seen several techniques that you can use to store data locally on the device that is running the browser. These techniques include session storage for storing session data that is automatically removed when the user's browsing session completes; local storage, which provides for more persistent data on the user's device; and the Indexed Database API, which enables the browser to store and retrieve data in a more structured manner.

You have also learned how to use the application cache of HTML5 to configure web pages and to enable them to cache resources locally on the user's device.

Review Question(s)

Question: What are the primary differences between data retained on a user's device by using session storage and by using local storage?

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
You configure a web page to use the application cache to cache a resource locally. If the resource is modified at the web server, then the browser automatically downloads the latest version. True or false?	

Module 10

Implementing an Adaptive User Interface

Contents:

Module Overview	10-1
Lesson 1: Supporting Multiple Form Factors	10-2
Lesson 2: Creating an Adaptive User Interface	10-6
Lab: Implementing an Adaptive User Interface	10-13
Module Review and Takeaways	10-21

Module Overview

One of the most enduring features of the web is its temporary nature. For the first time, the monopoly of the keyboard and mouse is coming under challenge, and that means questioning how user interfaces are designed. You may develop a web application on a computer with a large, high-resolution monitor, a mouse, and a keyboard, but other users might view and interact with your application on a smartphone or a tablet without a mouse, or have a monitor with a different resolution. Users may also want to print pages of your application.

In this module, you will learn how to build a website that adapts the layout and functionality of its pages to the capabilities and form factor of the device on which it is being viewed. You will see how to detect the type of device being used to view a page, and learn strategies for laying out content that effectively targets particular devices.

Objectives

After completing this module, you will be able to:

- Describe the requirements in a website for responding to different form factors.
- Create web pages that can adapt their layout to match the form factor of the device on which they are displayed.

Lesson 1

Supporting Multiple Form Factors

The number of different devices and form factors now capable of viewing a website has increased greatly over the past few years. But is it a good idea to create a 'one size fits all' website? What are the alternatives and what are the differences to look for in a website targeted at traditional desktop users, versus mobile or touch users? In this lesson, you will consider some of the issues involved in supporting multiple form factors in a website.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain why a website should be able to respond appropriately to users viewing the site by using different devices.
- Describe the changes that are typically necessary to enable a website to be viewed by using different devices.

Why Design an Adaptive User Interface?

At the heart of every website is the content that it displays. The design of a website should present this content to make it easy to use and navigate. However, while many developers commonly design websites for use on a desktop or a laptop with a screen resolution typically exceeding 1280 x 1024 pixels, with a mouse and keyboard for input, users visiting these sites may access them by using a smartphone or tablet, or they may wish to print out pages for later reference offline. Consequently, it is becoming increasingly necessary to design websites for devices with capabilities that are

unknown when the application is built. These capabilities include the screen **resolution**, device **orientation**, **input method** (mouse, finger, voice, keyboard, pen), and so on.

- The core of a web site is its content
- Implement an adaptive user interface so that a website can present itself better:
 - To smartphones
 - To tablets
 - As printed matter
 - As a spoken page
- Monitor site use to detect how users access your website over time, and modify the design if necessary
- Create platform-specific websites if the user statistics suggest this would be beneficial

The developer's ability to select the content displayed on a website might be limited; it is typically driven by commercial decisions based on the requirements of the organization for which the website is constructed. However, as a developer, you can make design choices that enhance a website so that it responds intelligently to the form factor of the user's device. By fine-tuning the layout of a website to the viewport of the browser, increasing the target area on links for smaller screens and touch-based devices, and removing unnecessary navigation aids and headings when printing out a page, you can optimize the reading experience for the device. The adaptive user interface features of HTML5 and CSS, such as media queries and conditional comments, enable you to design and implement a website that reacts in this manner.

 **Note:** The adaptive user interface features of HTML5 and CSS3 enable developers to design applications that follow the responsive web design approach. The term responsive web design was coined in 2010 by Ethan Marcotte. For more information, visit <http://go.microsoft.com/fwlink/?LinkId=267739>.

It is also important to review the statistics of the users visiting the website, together with their behavior and their devices, so that you can adapt the website design and structure to best meet their needs; patterns of use may change over time as new devices become more mainstream.

One common dilemma is whether to build a second site specifically for users of a certain device type. Many organizations build shadow websites for smartphones or other mobile devices, and use features of the browser to detect the device capabilities and redirect the user to the mobile version of a web page, if necessary. This approach requires a nontrivial amount of initial effort and design based on building two sets of layout for the same content, and optimizing this layout for each set of pages. The benefit of this approach is that users with smartphones will get a faster, more usable site than if the developer follows the 'one size fits all' strategy. Ultimately, the question of whether to follow an adaptive approach or to implement mobile versions of content should be decided based on user statistics and logs, rather than an initial guess before the website has been launched.

Considerations for Supporting Different Types of Device

Most issues concerning how to build an adaptive user interface for a website are really considerations about the aspects and capabilities of the devices that a website should target and to which it should react. The following sections summarize these issues, and describe some actions that can be taken to enhance the user experience of your websites for differing device capabilities.

Screen Resolution

The most obvious difference between a smartphone and a desktop monitor is the screen resolution, and therefore the potential size of the browser window or viewport. As part of a responsive design, your site must be usable on any screen, whether it is 480, 1280, or 2560 pixels wide. Your website may adapt different types of content by using the strategies in the following list:

- **Text.** Change the font size for lower viewport sizes so that text remains readable. Decrease the font size as the viewport size increases. You can also try to adapt the way in which the browser deals with line breaks by switching on hyphenation, and add hints by using the `<wbr>` tag, as well as CSS hyphenation and wrapping properties.

- Creating an adaptive user interface requires considering the following items:
 - Screen resolution
 - Display density
 - Input method
 - Browser capabilities
- Some users may want to print the contents of a page
- Visually impaired users might require screen readers

The `<wbr>` tag is similar to the `
` tag except that it specifies that the browser should only insert a line break if it is necessary to prevent text from wrapping onto the next line. You can embed it in the middle of a long word or place name as follows:

```
<p>The place with the longest name in Wales is Llanfair<wbr />pwllgwyngyll<wbr />gogerychwyrndrobwll<wbr />llantysilio<wbr />gogogoch</p>
```

 **Note:** Module 6, "Styling HTML5 by Using CSS3", described how to format text by using the `text-indent`, `hyphens`, `word-wrap`, and `word-spacing` properties in CSS.

- **Images.** Shrink and grow image size to a minimum/maximum width by setting width and height properties relative to the screen or viewport size. Alternatively, you can create different-sized versions of your images and display the appropriate image in the browser window when the size of the window falls within a certain range.



Note: You can use the JavaScript expression `document.documentElement.offsetHeight` to determine the height of the viewport for the current document, and the expression `document.documentElement.offsetWidth` to determine the width. You can also use media queries to detect the resolution of the display device. This approach is described in the next lesson.

- **Layout.** The most common response to viewport size is to change the layout of the website, switching to a one-column layout for low viewport sizes and to high-column layouts for much larger ones. You can also use 'fluid columns' that change their width as the viewport width is changed, but above and below a certain point, extra-narrow or extra-wide columns look unappealing (the optimum column width is 480 - 600px). In these situations, you should change the number of columns.



Note: Module 6, "Styling HTML5 by Using CSS3", described how to change the layout of a web page as the width increases or decreases.

- **Navigation.** If necessary, consider changing the main navigation menu from a list to a drop-down menu. This modification saves space and increases functionality.

When you are testing a website to see how it responds to different screen sizes, you can use the Internet Explorer F12 Developer Tools to accurately resize your browser window to a given window size. On the **Tools** menu, click **Resize** to set a custom size, or pick one of the preset sizes.

Display Density

The resolution of a screen has varied widely for a while, but only recently has the density of a screen (measured in dots per inch, or dpi) started to vary enough to make it worthwhile for browsers to detect and react to it. Increased screen density enables applications to present data with much greater clarity by using common screen resolutions. However, a larger screen density can cause the operating system to scale up all elements on a page, which can result in blurry images. To counter this problem, you can adopt the following strategies for text and image data:

- **Text.** Increase the font size for higher-density displays.



Note: You can also use media queries to detect the density of the display device. This approach is described in the next lesson.

- **Images.** Use a graphics editor to create copies of your graphics and images at a higher dpi, or, failing that, a bigger size. Alternatively, you can recreate the images as Scalable Vector Graphics (SVG), which will scale up perfectly.



Note: Module 11, "Creating Advanced Graphics", describes how to use SVG to create scalable graphics.

Input Method

When it comes to designing the user experience of a website for a touch-based device, remember that a finger will never be as accurate as a mouse pointer. This has an impact on the features that a user uses to interact with your application:

- **Buttons and Links:** Make buttons and links larger than normal so users can easily identify them and tap them with a finger. Again, consider changing list-based menus into one larger drop-down menu to enhance usability.

- **Hovering:** Hover states do not exist on a touch-based device. There is no way to simulate a mouse pointer over a link, so don't use the **hover** pseudo-class.
- **Screen Orientation:** Touch-based devices are nearly all handheld and can usually change their orientation depending on how the user holds them. Make sure your design works in both **landscape and portrait mode**. Switching between the two is effectively a change in resolution, so treat it as such.

Browser Capabilities

Different browsers often have very different levels of functionality. For example, Internet Explorer 10 implements many (but not all) of the features specified by HTML5 and CSS3, while browsers from other vendors implement a frequent cycle of releases to react to standards changes. You also need to consider the mobile versions of browsers (including mobile versions of Internet Explorer) that have their own levels of compliance.

The website at <http://go.microsoft.com/fwlink/?LinkId=267741> provides information on which browsers support which features of CSS3 and HTML5.



Note: You can examine the **window.navigator.userAgent** property in JavaScript to detect information about the current browser displaying a web page. Alternatively, you can use a feature-detection library such as Modernizr.

Printers

Users may want to print copies of selected parts of a web page, such as a map or directions to an organization's office, for later use offline. The form factor to consider here is A4 paper, where navigation, site logos, and design are irrelevant. The only thing that matters is the presentation of the content. You can create a print style sheet to apply the following changes:

- **Non-content:** Remove the page header, footer, navigation menus, background colors, CSS graphics, transforms, and animations.
- **Text:** Set the size of your fonts to values in points as you would a Microsoft Word document, their color to dark grey, and remove any text shadow.
- **Links and Abbreviations:** Expand links and abbreviations on the page.
- **Columns:** Lay out your content in one column, unless it includes an index or glossary, in which case two columns is acceptable.

These techniques are described in more detail in the next lesson.

Screen Readers

Using the web is an entirely different experience for the visually impaired. Reading web content and navigation are both achieved aurally. When using aural properties, the canvas consists of a three-dimensional physical space (sound surrounds) and a temporal space (one may specify sounds before, during, and after other sounds). The CSS properties also allow you to vary the quality of synthesized speech (voice type, frequency, inflection, and so on), the direction it comes from, and to modify pauses, cues, and volume.

Lesson 2

Creating an Adaptive User Interface

In this lesson, you will see how to use CSS to detect different types of devices and to implement a user interface that can adapt to the form factor and capabilities of the device on which your application is running.

Lesson Objectives

After completing this lesson, you will be able to:

- Use media types to target different types of device, and to apply appropriate style sheet rules.
- Use media queries to identify certain properties of a device, and to apply appropriate style sheet rules.
- Identify the version of Internet Explorer being used to view a web page, and to apply appropriate style sheet rules.
- Create a style sheet suitable for printing the content displayed by a web page.

CSS Media Types

Developers have had some ability to tailor styles to types of devices, since HTML4 identified a list of media descriptors with which you could qualify your style sheets by using the **media** attribute of the **<link>** element. HTML4 recognized the following media types:

- **speech**: Speech synthesizers.
- **braille**: Braille tactile feedback screen readers.
- **embossed**: Braille printers.
- **handheld**: Mobile devices (described as "small screen, monochrome, bitmapped graphics, limited bandwidth" in the HTML4 specification, dated 1999!).
- **print**: Print preview screens and printer output.
- **projection**: Projectors.
- **screen**: Computer screens.
- **tty**: Teletypes.
- **tv**: Televisions and other low-resolution devices with limited ability to scroll.
- **all**: Applicable to all devices.

- HTML uses the **media** attribute to qualify the use of a style sheet for a type of device

- screen
- print
- braille
- speech
- all
- ...

```
<link rel="stylesheet" type="text/css" href="print.css" media="print" />
```

- CSS defines the **@media** rule to perform the same task

```
@media print, projection {  
    ..print_only_rules..  
}
```

Many websites continue to use these attributes today, especially when it comes to attaching a print style sheet to a page. For example, the two elements shown in the following example link first to a style sheet containing rules for any device (primarily screens) and then to a second style sheet only for printers, to optimize the content for paged media.

```
<link rel="stylesheet" type="text/css" href="core.css" media="all" />  
<link rel="stylesheet" type="text/css" href="print.css" media="print" />
```

The **@media** CSS rule performs a similar task to the **media** attribute. It enables you to identify a set of styling rules for a media type within an existing style sheet, rather than by creating a separate style sheet specific to each media type. You use the **@media** rule like this:

```
@media print {
    header, footer { display: none; }
    ...
}
```

In this example, the **header** and **footer** CSS rules apply only to printed media.

You can combine multiple media types together if you need to apply the same styling rules to them. The following example shows how to style the header and footer for printers and projectors:

```
@media print, projection {
    header, footer { display: none; }
    ...
}
```

 **Additional Reading:** For more information about media types, visit <http://go.microsoft.com/fwlink/?LinkId=267742>.

Detecting Device Capabilities by Using Media Queries

Media queries in HTML5 and CSS3 enhance the concept of media types; they enable developers to inspect the physical characteristics of a device, including device height and width, orientation, and resolution. Although media queries cannot tell you the exact type of device, you can use them to infer this information. For example, at the time of writing you could reasonably assume that a device with a screen width of 480px or less is a mobile phone, and use this knowledge to style content accordingly.

A media query has two parts:

- A media type, such as screen, print, speech, and so on.
- A set of parentheses containing the query, which comprises a device characteristic, a colon, and then a target value.

• Use media queries to detect the capabilities of a device

- width
- height
- orientation
- resolution
- ...
- vendor-specific

```
@media screen
and (max-device-width: 800px)
and (orientation: portrait) {
    ...
}
```

• Write styles for a base device and use media queries to override them based on the properties of a device

The following examples show how to define a media query in an HTML **<link>** element and in a CSS rule. If the device viewing the corresponding page matches the query criteria, it will apply the CSS styles associated with the query:

```
<link rel="stylesheet" type="text/css" href="mobile.css"
      media="screen and (max-device-width: 480px)" />
@media screen and (max-device-width: 480px) {
    article {
        column-count: 1;
    }
    ...
}
```

You can include multiple queries by concatenating them with and, or, and not. For example:

```
@media screen and (max-device-width: 480px) and (resolution:300dpi) {  
    ...  
}
```

You can test for 13 device characteristics in a media query. The following list summarizes these characteristics.

- **width, height:** The width and height of the viewport (usually the browser window).
- **device-width, device-height:** The width and height of the active device screen (or paper, if printing).
- **orientation:** Whether the device is in portrait or landscape mode.
- **resolution:** The pixel density (in dpi or as a ratio) of the target device.
- **aspect-ratio:** The width to height ratio of the viewport.
- **device-aspect-ratio:** The width-to-height ratio of the device screen (or paper).
- **color:** The bits per color of the target display.
- **color-index:** The total number of colors the target device screen can show.
- **monochrome:** The bits per pixel in a monochrome frame buffer.
- **scan:** The scanning method of a TV. Possible values are progressive and interlace.
- **grid:** The display type of the output device: grid or bitmap.

All of these characteristics except **scan** and **grid** also allow you to query for minimum and maximum values as well. For example, you can specify **min-width**, **max-width**, **min-resolution**, **max-resolution**, and so on.

By using media queries, you can implement a responsive design for your website that takes into account the size of the viewport displaying the web pages.

When using media queries, it is good practice to decide what all of your styles will be for either a large or a small viewport size, and then use media queries at the end of a page to override the primary styles for gradually decreasing or increasing sizes. This code example takes a mobile-first approach, and then overrides fonts and columns for larger viewports.

Implementing a Mobile-First Approach with Media Queries

```
/* Start with mobile styles - max width assumed at 480px*/  
article {  
    column-count: 1;  
    line-height: 1.6;  
    font-size: 12px;  
    width: 98%;  
}  
@media (max-width: 600px) {  
    .article {  
        font-size: 14px;  
        width: 90%;  
    }  
}  
@media (max-width: 800px) {  
    .article {  
        column-count : 2;  
        line-height: 1.5;  
        width: 70%;  
    }  
}  
/*note min-width here, not max-width*/  
@media (min-width: 1200px) {  
    .article {
```

```

column-count : 3;
width: 60%;
font-size: 16px;
line-height : 1.7;
}
}

```

 **Note:** Note that some browsers support vendor-prefixed properties. These properties are ignored by browsers that do not recognize them.

 **Additional Reading:** For more information about media queries, visit <http://go.microsoft.com/fwlink/?LinkId=267743>.

Detecting an Older Version of Internet Explorer by Using Conditional Comments

One of the main issues in developing a website front end over the past few years has been the widely varying levels of support for CSS, most notably in the different editions of Internet Explorer. For example, Internet Explorer 6 incorrectly implements the basic box model, while Internet Explorer 8 and earlier versions ignore all of the new elements in HTML5.

To enable you to build web applications that can run in different versions of Internet Explorer, Microsoft has implemented **conditional comments**.

These comments enable you to add rules to a style sheet to target specific versions of Internet Explorer. A conditional comment is written as an ordinary HTML comment (so that it is ignored by other vendors' browsers), but with a conditional expression enclosed in square brackets followed by a section of HTML markup, and a closing **endif** comment. The conditional expression can detect the version of Internet Explorer by using the **IE** operand together with a version number. For example, the following code detects whether the user is running Internet Explorer 9 :

```

<!--[if IE 9]>
<p>Welcome to Internet Explorer 9.</p>
<![endif]-->

```

- Conditional comments enable you target specific versions of Internet Explorer prior to version 10

- To link style sheets:

```
<!--[if gte IE 9 ]>
<link href="ie9.css" rel="stylesheet" />
<![endif]-->
```

- To add classes for styling:

```
<!--[if lt IE 7 ]>
<html class="ie6">
<![endif]-->
```

- To run scripts:

```
<!--[if IE]>
<script src="http://contoso.com/scripts/iescript.js"></script>
<![endif]-->
```

 **Note:** Conditional comments are not supported by Internet Explorer operating in standards mode. Use conditional comments to detect versions of Internet Explorer prior to version 10 operating in quirks mode.

You can use the **!** operator to reverse the sense of a condition. You can also use the **IE** operand in isolation to determine whether the user is running another vendor's browser, like this:

```

<!--[if !(IE)]>
<p>You are not using Internet Explorer.</p>
<![endif]-->

```

You can also use operators such as **lt** (less than), **gt** (greater than), **lte** (less than or equal), and **gte** (greater than or equal) to detect a range of values, as shown in the following example, which detects whether the user is running a version of Internet Explorer prior to Internet Explorer 9:

```
<!--[if lt IE 9]>
<p>Please upgrade to Internet Explorer 9 or later.</p>
<![endif]-->
```

The next example shows how to use conditional comments to load an appropriate style sheet for the version of Internet Explorer that runs on the user's operating system:

```
<html>
<head>
<link href="styles.css" rel="stylesheet" />
<!--[if IE 8]> <link href="ie8.css" rel="stylesheet" /><![endif]-->
<!--[if lt IE 8]> <link href="ie7.css" rel="stylesheet" /><![endif]-->
<!--[if gte IE 9]> <link href="ie9.css" rel="stylesheet" /><![endif]-->
<!--[if IEMobile7]> <link href="winPhone7.css" rel="stylesheet" /><![endif]-->
<!--[if !(IE)]> <link href="otherBrowsers.css" rel="stylesheet" /> <![endif]-->
...
</head>
...
</html>
```

In this example,

- The page loads a style sheet called styles.css.
- If the browser is Internet Explorer 8, it also loads ie8.css.
- If the browser is Internet Explorer 7, 6, or older, it also loads ie7.css.
- If the browser is Internet Explorer 9 or 10, it also loads ie9.css.
- If the browser is Internet Explorer for Windows Phone 7, it also loads winPhone7.css.
- Finally, if the browser is not Internet Explorer, it also loads otherBrowsers.css.

It is also possible to define inline conditional comments so that all styles can be contained in a style sheet:

```
<!--[if lt IE 7 ]> <html class="ie6"> <![endif]-->
<!--[if IE 7 ]> <html class="ie7"> <![endif]-->
<!--[if IE 8 ]> <html class="ie8"> <![endif]-->
<!--[if IE 9 ]> <html class="ie9"> <![endif]-->
<!--[if (gt IE 9)|(IE)|(IE Mobile7)]><!--> <html> <!--<![endif]-->
```

If you follow this approach, you can prefix styles with the version of Internet Explorer to which the style applies. For example, the following code sets box model properties for an **article** element, and then adds a correction for Internet Explorer 6:

```
article {
    width: 200px;
    margin: 10px;
    border: 5px solid red;
    padding: 10px;
}
ie6 article {
    width: 250px;
}
```

You can also use conditional comments to include scripts in a web page, as follows:

```
<!--[if IE]>
<script src="http://contoso.com/scripts/iescript.js"></script>
```

```
<! [endif]-->
```



Additional Reading: For more information about conditional comments in Internet Explorer, visit <http://go.microsoft.com/fwlink/?LinkId=267744>.

Defining Style Sheets for Printing

Print-specific styles enable your website content to be printed correctly for easy reading, without spurious screen artifacts such as navigation and side menus taking up space on the printed page. You can use the **print** media type to identify the appropriate styles to use. You can create these styles in a separate style sheet, or you can use the **@media** rule to add them inline in an existing style sheet, as shown in the following examples.

```
<link rel="stylesheet" type="text/css"
      href="print.css" media="print" />
@media print {
    .. print styling rules go here ..
}
```

- Add print styles to control how content is printed
 - Specify the **print** media type in CSS rules
- Perform the following optimizations
 - Remove page headers, footers, navigation, background, graphics, and animations
 - Set the size of the font and remove text effects
 - Expand the text for links and abbreviations
 - Lay out the content in one column
 - Define the target size and layout of the printed page

The styles for a printer often implement the following rules:

- Remove the page header, footer, navigation menus, background colors, CSS graphics, transforms, and animations.

```
header, footer, nav {
    display : none;
}
article {
    background : none;
}
.highlight {
    transform : none;
}
```

- Set the size of your fonts to values in points as you would for a Word document, set their color to dark grey, and remove any effects such text-shadow.

```
article, p, li {
    font-size : 14pt\1.5; color: #222; text-shadow : none;
}
```

- Expand any links and abbreviations on the page so that the URL of the link or the expanded text of the abbreviation is printed to the right of the text.

```
a:after {
    content: " (" attr(href) ")";
}
abbr:after {
    content: " (" attr(title) ")";
}
```

- Lay out the content in one column, unless it includes an index or glossary, in which case two columns is acceptable.

```
article {  
    column-count : 1;  
}  
#glossary {  
    column-count : 2;  
}
```

- Define the target size of the printed page, the margins around facing pages, and the minimum number of lines in a paragraph printed at the top (widows) and the bottom (orphans) of the page. You can use the **@page** rule to achieve this, as follows:

```
@page {  
    size: A4;  
    orphans: 3;  
    widows: 3;  
}
```

The **@page** rule also enables you to specify different layouts for the left hand and right hand pages in double-sided documents by using the **:left** and **:right** pseudo-classes, as shown below:

```
@page :left {  
    margin-left: 3cm;  
    margin-right: 4cm;  
}  
@page :right {  
    margin-left: 4cm;  
    margin-right: 3cm;  
}
```



Additional Reading:

For more information about using CSS to optimize pages for printing, visit
<http://go.microsoft.com/fwlink/?LinkId=267745>.

Demonstration: Implementing an Adaptive User Interface

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Implementing an Adaptive User Interface

Scenario

Most conference attendees are expected to use a laptop to view the live version of the Contoso Conference website, but some may wish to print a hard copy of some of the information. Other attendees might use smartphones or other handheld devices to view the website. The different requirements and form factors of a printer or a handheld device compared to a laptop make it necessary for the user interface of the web application to detect device capabilities and adapt itself accordingly. For example, some website elements, such as the header, are not necessary for printing, while the smaller screens of smartphones are not ideal for viewing full-sized websites.

Objectives

After completing this lab, you will be able to:

- Create style sheets that apply only when printing a web page.
- Use CSS media queries to enable a web page to adapt to different device form factors.

Lab Setup

Estimated Time: 60 minutes

- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa\$\$w0rd**

Before you start this exercise, make sure that you have disabled caching in Internet Explorer, as follows:

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In the Internet Explorer, press F10 to display the menu bar.
4. On the **Tools** menu, click **Internet options**.
5. In the **Internet Options** dialog box, click **Settings**.
6. In the **Website Data Settings** dialog box, click the **Caches and databases** tab.
7. Clear the **Allow website caches and databases** check box, and then click **OK**.
8. In the **Internet Options** dialog box, click **OK**.
9. Close Internet Explorer.

Exercise 1: Creating a Print-Friendly Style Sheet

Scenario

In this exercise, you will add a style sheet for formatting web pages in a style suitable for printing. You will ensure that this style sheet is used only when a page is being printed.

In the style sheet, you will add rules to override the layout of the website, removing the header and footer, and reformatting the **About** page to display the information in a single wide column. To test the application, you will view the **About** page and verify that the print preview displays a correctly formatted version of the page.

The main tasks for this exercise are as follows:

1. Review the existing application.

2. Create a style sheet for printing web pages.
3. Link the print style sheet to the About page.
4. Test the application.

► **Task 1: Review the existing application**

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod10\Labfiles\Starter\Exercise 1** folder.
4. Run the application and view the **about.htm** page.
5. In Internet Explorer, press F10 to display the menu bar.
6. On the **File** menu, click **Print preview**. Notice that the print preview attempts to display the page header and that the text columns are too narrow.

The Print Preview version of the **About** page looks like this:



FIGURE 10.1:THE ABOUT PAGE IN PRINT PREVIEW MODE

7. Close the **Print Preview** window and then close Internet Explorer.
 8. In Visual Studio, examine the about.htm page and verify that the **<nav>** element is marked with the **page-nav** class, the **<header>** element is marked with the **page-header** class, and the **<footer>** element is marked with the **page-footer** class, as follows:

```
<nav class="page-nav">  
<header class="page-header">  
<footer class="page-footer">
```

You will use these classes to style the elements when the page is printed

► Task 2: Create a style sheet for printing web pages

1. In the ContosoConf project, add a new style sheet named **print.css** to the **styles** folder.
 - o Use the **Add New Item** wizard and use the Style Sheet template on the **Web** tab.
 2. In the **print.css** style sheet, delete the existing contents and add a CSS rule to hide the **nav** element that has the **page-nav** class, the **header** element that has the **page-header** class, and the **footer** element that has the **page-footer** class.

- Set the **display** property of these elements to **none**.
3. Add a CSS rule for the **.container** class, which overrides the CSS from **site.css**.
 - When printing, container elements should have no maximum width or padding.
 4. The **about.css** style sheet in the **styles/pages** folder contains the following CSS rule to display the page text in columns.

```
.about > article > section {  
    text-align: justify;  
    /* Columns Layout */  
    column-count: 3;  
    column-gap: 5rem; ;  
}
```

In **print.css**, add a CSS rule that removes the columns, so that the text displays as a single block.

- You only need to set the **column-count** property in this rule.

► Task 3: Link the print style sheet to the About page

1. Add a link to **print.css** in the **<head>** section of the **About** page, where indicated by the comment **<!-- TODO: Add print.css <link> here -->**. Set the **media** attribute of the link so that the style sheet is used only when printing.

► Task 4: Test the application

1. Run the application and view the **About** page.
2. Refresh the page to ensure that the most recent version of the **About** page is loaded.
3. In Internet Explorer, open the **Print Preview** window and verify that the preview does not display the navigation, header, or footer, and that the text has no columns.

The Print Preview version of the **About** page should look like this:

About ContosoConf

Page 1 of 2

ContosoConf brings web designers and developers together

Since the very first Contoso Conf back in 2009, we've been guided by three principles:

1. Community Matters
2. Never Stop Learning
3. Have fun!

LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT. SED VITAE ENIM ARCU, VITAE ALIQUET PURUS. AENEAN RHONCUS DIAM ET ORCI PORTTITOR FRINGILLA. IN PORTA LACUS A TURPIS PRETUM PLACERAT. CRAS VIVERRA ENIM EU NIBH PRETUM ORNARE. PRAESENT ET ADIPISCING TURPIS. DUIS MI RISUS, ORNARE AT BIBENDUM A, ULLAMCORPER VEL TELLUS. NULLA IN EGESTAS VELIT. AENEAN CONSEQUAT MI SED TELLUS IACULIS LAOREET. DONEC ET ODIO VEL FELIS COMMODO PORTTITOR.

AENEAN ID LIGULA EST. PELLentesQUE UT MAGNA LIGULA. DONEC NUNC EROS, TINCidunt SIT AMET SOLlicitUDIN IN, SEMPER ID MAURIS. PHASEllUS ODIO NULLA, MOlestie AC GRAVida SED, DIGNissim IN NISL. NUNC Luctus LOBORTIS MASSA AT DAPIBUS. AENEAN TURPIS NIBH, HENDERIT NEC CONGUE ET, ELEMENTUM A JUSTO. AENEAN SIT AMET NULLA ODIO. CRAS FEUGIAT PORTA RISUS NEC PRETUM.

What's It All About?

DONEC VEL SEM UT DUI VULPUTATE PORTA. PHASEllUS IMPERdET SAPIEN A ARCU ADIPISCING VITAE ADIPISCING ELIT PHARETRA. DONEC SED ANTE UT EROS MATTIS BIBENDUM NON IN ERAT. DONEC SAGITTIS, MASSA EU ACCUMSAN ELEIFEND, EROS JUSTO CURSUS JUSTO, ID CONSEQUAT MAURIS DIAM ID MAGNA. VIVAMUS QUIS TORTOR MASSA. NAM IPsum METUS, DAPIBUS AC FACILISIS SIT AMET, ULLAMCORPER QUIS RISUS. INTEGER ALIQUET ELEIFEND ACCUMSAN.

“*I had a fantastic time and learnt so much!*

PELLentesQUE FACILISIS BLANDIT AUGUE ID RHONCUS. SED FACILISIS VARIUS LECTUS, EGET COMMODO PURUS DAPIBUS NEC. IN HAC HABITASSE PLATEA DICTUMST. ETIAM IMPERdET FACILISIS MALESUADA. NUNC SEMPER VENENATIS ELIT AC LOBORTIS. DUIS LOREM LOREM, PHARETRA UT SCelerisque NEC, CONSEQUAT SED RISUS. MORBI RUTRUM NISL UT IPsum CONSEQUETUR PORTTITOR. PHASEllUS SED NUNC ID DIAM TEMPUS CONGUE IN A LEO.

PROIN FEUGIAT, TURPIS ID TEMPOR TEMPOR, LOREM LIBERO MALESUADA.

<http://localhost:56789/about.htm>

8/16/2012

4. Close the **Print Preview** window and then close Internet Explorer.

Results: After completing this exercise, you will have added a style sheet that implements a print-friendly format for web pages.

Exercise 2: Adapting Page Layout to Fit Different Form Factors

Scenario

In this exercise, you will create a style sheet that enables the pages in the Contoso Conference website to adapt to different device form factors.

First, you will view the application running in a small window to simulate a small device, such as a smartphone. Then you will use CSS media queries to define rules that change the website layout to better suit small devices.

Finally, you will run the application again and verify that the website layout adapts to large and small screen sizes.

The main tasks for this exercise are as follows:

1. Simulate the web application running on a small device.
2. Implement styles for hand-held devices and smartphones.
3. Test the application.

► **Task 1: Simulate the web application running on a small device**

1. In Visual Studio, open the **ContosoConf** solution in the **E:\Mod10\Labfiles\Starter\Exercise 2** folder.
2. Run the application and view the **index.htm** page.
3. Resize Internet Explorer to **480 × 800**, approximating the form factor of a Windows Phone 8 device.



Note: Use the F12 Developer Tools to change the size of the device viewing the website. The **Resize** command on the **Tools** menu enables you to change the size of the browser.

4. Notice that the website navigation bar does not fit on the screen and that the big **Register** link overlaps the header text.

The **Home** page should look like this:



FIGURE 10.2:THE HOME PAGE

5. Close Internet Explorer.
- **Task 2: Implement styles for hand-held devices and smartphones**
1. Open the **mobile.css** style sheet in the styles **folder**. This style sheet is referenced in the `<head>` element of each page in the website. This style sheet is currently empty, but you will use it to specify the styles for hand-held devices and smartphones.

2. Add a CSS media query that targets screens that are less than or equal to 480 pixels in width.
3. In the media query, add a rule for **nav.page-nav .container** that uses a flexbox to display the contents of the navigation bar, as follows:

```
display: -ms-flexbox;  
-ms-flex-wrap: wrap;  
-ms-flex-pack: center;
```

4. Add another rule that hides the **:before** and **:after** pseudo elements of the **.active** navigation link.
 - o Set the **display** property to **none**.
5. Add another rule that sets a **.5rem** margin around each navigation link. This rule should also set a **1px** dotted border with the color **#3d3d3d** completely around the link, instead of just on the right.
6. The default layout of the website header is not suitable for screens less than 720 pixels wide. Add another media query to **mobile.css** that targets screen widths up to 720 pixels.
7. In this media query, add rules that perform the following actions:
 - o Reduce the height of the page header (set the **height** to **auto**).
 - o Hide the large **Register** link in the page header.
 - o Reduce the **font-size** of the **<h1>** element in the page header to **3rem**.

► Task 3: Test the application

1. Run the application and view the **Home** page.
2. Use the F12 Developer Tools to resize Internet Explorer to various sizes to test that the media queries adapt the user interface correctly.
 - o As a minimum, try the sizes 1280x1024 and 480x800.
3. Close Internet Explorer.

Results: After completing this exercise, you will have a website that adapts to different screen sizes.

Module Review and Takeaways

In this module you have learned why it is necessary to create websites that can dynamically adapt to different devices and browsers. You have seen the main considerations when tailoring a page to different form factors and browsers, and you have learned—by using media types, media queries, and conditional comments—how to detect the form factor and browser in CSS and HTML5. And finally, you have learned how to implement a basic style sheet suitable for printing content.

Review Question(s)

Question: What are the main device characteristics used by media queries to detect whether a client device is a hand-held tablet?

Question: How can you style content to adapt to the type and form factor of the device on which a user is viewing your web site?

Module 11

Creating Advanced Graphics

Contents:

Module Overview	11-1
Lesson 1: Creating Interactive Graphics by Using SVG	11-2
Lesson 2: Drawing Graphics by Using the Canvas API	11-19
Lab: Creating Advanced Graphics	11-26
Module Review and Takeaways	11-33

Module Overview

High-resolution, interactive graphics are a key part of most modern applications. Graphics can help to enhance the user's experience by providing a visual aspect to the content, making a website more attractive and easier to use. Interactivity enables the graphical elements in a website to adapt and respond to user input or changes to the environment, and is another important element in retaining the attention of the user and their interest in the content.

This module describes how to create advanced graphics in HTML5 by using Scalable Vector Graphics (SVG) and the Microsoft Canvas API. You will learn how to use SVG-related elements such as `<rect>`, `<ellipse>`, and `<polyline>` to display graphical content on a web page. You will also learn how to enable the user to interact with SVG elements through the use of events such as keyboard events and mouse events.

The Canvas API is somewhat different than SVG. The Canvas API provides a `<canvas>` element and a set of JavaScript functions that you can invoke to draw graphics onto the canvas surface. You will learn how to use the Canvas API, and also find out when it is more appropriate to use Canvas or SVG.

Objectives

After completing this module, you will be able to:

- Use SVG to create interactive graphical content.
- Use the Canvas API to generate graphical content programmatically.

Lesson 1

Creating Interactive Graphics by Using SVG

SVG is an XML grammar that has been incorporated into HTML. SVG enables you to incorporate interactive graphical content in your web pages. SVG comprises a set of elements that you enclose in an **<svg>** element, and that become part of the Document Object Model (DOM) for your web page.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the elements of SVG.
- Create an **<svg>** element and embed simple graphical elements inside it.
- Use SVG to draw circles and ellipses.
- Use SVG to draw complex shapes.
- Apply fill styles and strokes to SVG elements.
- Use gradients and patterns to fill SVG elements.
- Use SVG to draw text.
- Apply transformations to SVG elements.

What is SVG?

SVG defines a set of elements that represent common types of drawing shapes. For example, there are SVG-related elements named **<rect>**, **<text>**, **<ellipse>**, **<polygon>**, and **<path>**. SVG implements two-dimensional vector graphics that scales to the size of the user's browser window and the resolution of the screen on which it is running.

SVG uses a retained mode model. When you add SVG-related elements to an HTML5 web page, the elements are kept in the DOM tree for the web page, and a user can interact with them in the same way as they would with HTML elements such as **<h1>**, **<div>**, and **<button>**. As a developer, you can perform the following actions:

- SVG enables you to draw 2D vector graphics
 - It defines XML elements to represent a wide range of shapes
- SVG uses a "retained mode" model
 - The objects tree is kept in memory
 - Rendering speed depends on the number of elements
- You can perform the following operations on SVG-related elements:
 - Access elements through DOM
 - Style elements with CSS
 - Handle user-interaction events

- Call **DOM** functions such as **document.querySelector()** and **document.getElementById()** to locate and manipulate SVG-related elements in the document in JavaScript code.
- Apply **CSS** styles such as colors, borders, and transformations to SVG-related elements.
- Handle **events** such as mouse events and keyboard events on SVG-related elements. For example, you can use SVG to create a complex graphical figure, and then handle mouse click events to enable the user to interact with specific elements within the figure.

When you add SVG-related elements to your web page, the performance of the page depends on the number of elements. The more elements you add, the more objects the browser has to create and add to the DOM tree, and the longer it will take the browser to render the page. If you want to create extremely complex graphical figures, the Canvas API might be a better option.

Creating SVG Graphics

To use SVG in a web page, add an **<svg>** element and specify an XML namespace as follows:

```
<svg xmlns="http://www.w3.org/2000/svg">
  ...
</svg>
```

The **<svg>** element can contain any number of SVG-related elements, such as **<rect>** or **<ellipse>**. Each of these elements has a set of properties that enable you to configure its appearance within the **<svg>** element. The following list describes some of the common properties that you can set on SVG-related elements:

- **x** and **y**: The position of the shape within the **<svg>** element, relative to the left hand side and the top of the **<svg>** element, respectively.
- **width** and **height**: The width and height of the shape.
- **fill** and **stroke**: The fill color and stroke color of the shape.

The following example creates an **<svg>** element that contains two rectangles. The first rectangle is red and has rounded corners, as specified by the **rx** and **ry** attributes. The second rectangle is yellow, and partially obscures the first rectangle because it is defined after it in the **<svg>** element:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="50" y="50" width="100" height="75" rx="20" ry="20" fill="red" stroke="blue" />
  <rect x="75" y="75" width="100" height="75" fill="yellow" stroke="blue" />
</svg>
```

The rectangles generated by this code look like this:



FIGURE 11.1:RECTANGLES DRAWN BY USING AN <SVG> ELEMENT

You can define CSS styles for **<svg>** elements, and for the elements that are contained inside an SVG element. The following example defines a style sheet rule for all **<svg>** elements. The rule specifies that all **<svg>** elements have a dark blue border, a background color of light green, a width of 300 pixels, and a height of 200 pixels:

```
<style type="text/css">
  svg {
    border: 2px solid darkblue;
    background-color: lightgreen;
    width: 300px;
    height: 200px;
  }
</style>
```

The **<svg>** element from the previous example looks like this after the styling shown above is applied:

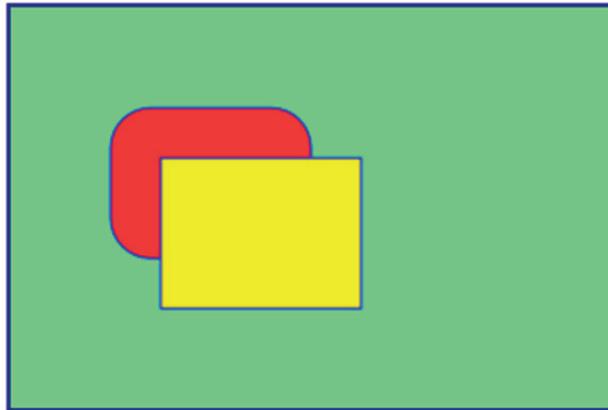
- Use an **<svg>** element and embed child elements that define the graphics:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="50" y="50" width="100" height="75"
    rx="20" ry="20" fill="red" stroke="blue" />
  <rect x="75" y="75" width="100" height="75"
    fill="yellow" stroke="blue" />
</svg>
```



- Style SVG elements by using CSS:

```
<style type="text/css">
  svg {
    border: 2px solid darkblue;
    background-color: lightgreen;
    width: 300px;
    height: 200px;
  }
</style>
```



Drawing Circles and Ellipses

SVG defines **<circle>** and **<ellipse>** elements that enable you to draw circles and ellipses in an **<svg>** element.

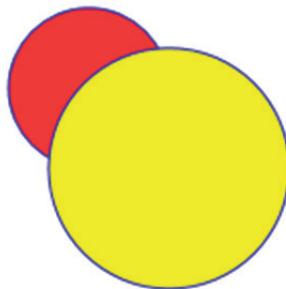
Circle

The **<circle>** element has **cx** and **cy** properties that specify the location of the center point for the circle inside the **<svg>** element. **<circle>** also has an **r** attribute that specifies the radius of the circle.

The following example creates two circles. The first circle is red and the second circle is yellow. The second circle partially obscures the first circle because it is defined after it in the **<svg>** element:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle cx="120" cy="80" r="40" stroke="blue" fill="red" />
  <circle cx="160" cy="120" r="60" stroke="blue" fill="yellow" />
</svg>
```

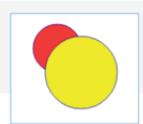
The circles generated by this code look like this:



To draw circles:

```
<circle cx="120" cy="80" r="40"
  stroke="blue" fill="red" />

<circle cx="160" cy="120" r="60"
  stroke="blue" fill="yellow" />
```



To draw ellipses:

```
<ellipse cx="150" cy="60" rx="110" ry="30"
  stroke="blue" fill="red" />

<ellipse cx="150" cy="140" rx="110" ry="30"
  stroke="blue" fill="yellow" />
```



FIGURE 11.3:CIRCLES DRAWN BY USING AN **<SVG>** ELEMENT

Ellipse

The `<ellipse>` element has `cx` and `cy` properties that specify the location of the center point. `<ellipse>` also has `rx` and `ry` attributes that specify the radius of the ellipse in the X and Y directions. If `rx` and `ry` are the same, the ellipse will appear as a circle.

The following example creates two ellipses. The first ellipse is red and the second ellipse is yellow. The ellipses have the same shape because they have the same `rx` and `ry` properties. However, the ellipses have different `cy` properties, so they appear at different vertical offsets within the `<svg>` element:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <ellipse cx="150" cy="60" rx="110" ry="30" stroke="blue" fill="red" />
  <ellipse cx="150" cy="140" rx="110" ry="30" stroke="blue" fill="yellow" />
</svg>
```

The ellipses generated by this code look like this:

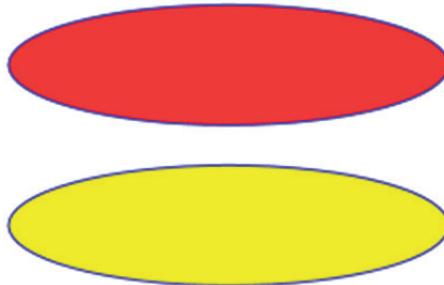


FIGURE 11.4: ELLIPSES DRAWN BY USING AN <SVG> ELEMENT

Drawing Complex Shapes

SVG defines **<polyline>**, **<polygon>**, and **<path>** elements that enable you to draw complex shapes in an **<svg>** element.

Polyline

The **<polyline>** element creates a line drawing comprising a series of connected points. The points are specified by the **points** attribute, which defines a comma-separated series of X and Y coordinates. A polyline does not connect the last point back to the first point. The **<polyline>** element has various attributes such as stroke and fill, which enable you to configure the appearance of the polyline.

The following example creates a polyline that draws a jagged blue line:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <polyline points="105 100, 120 100, 125 90, 135 110, 145 90, ...
    155 110, 165 90, 175
    110, 180 100, 195 100" fill="none" stroke="blue" />
</svg>
```

The polyline generated by this code looks like this:



FIGURE 11.5: POLYLINE DRAWN BY USING AN <SVG> ELEMENT

Polygon

The **<polygon>** element is similar to the **<polyline>** element, except that a polygon connects the last point back to the first point to form a closed shape.

The following example creates a polygon that draws a yellow block arrow pointing upwards:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <polygon points="110 70, 150 40, 190 70, 190 160, 150 130, 110 160" fill="yellow"
stroke="blue" />
</svg>
```

The polygon generated by this code looks like this:



FIGURE 11.6:POLYGON DRAWN BY USING AN <SVG> ELEMENT

Path

The **<path>** element enables you to draw a complex shape as a series of path segments. The **<path>** element has a **d** attribute that defines the outline of the shape. The **d** attribute comprises a series of drawing commands as follows:

- **M:** Move to a new location, without drawing a line.
- **L:** Draw a line from the current location to a new location.
- **A:** Draw an elliptical arc.
- **Q:** Draw a quadratic Bezier curve. A quadratic Bezier curve is a curve that joins two points, and has one turning point along its journey.
- **C:** Draw a cubic Bezier curve. A cubic Bezier curve is a curve that joins two points, and has two turning points along its journey.
- **Z:** Close the current path by connecting the last point back to the first point.

The following example creates a simple path that draws a solid red triangle with a blue outline. The **M** command moves the current location to (150, 50). The first **L** command draws a line from the current location to (250, 150). The next **L** command draws a line from the current location to (50, 150). The **Z** command closes the path, by drawing a line back to the starting point of (150, 50):

```
<svg xmlns="http://www.w3.org/2000/svg">
  <path d="M 150 50 L 250 150 L 50 150 Z" fill="red" stroke="blue" />
</svg>
```

The filled path generated by this code looks like this:

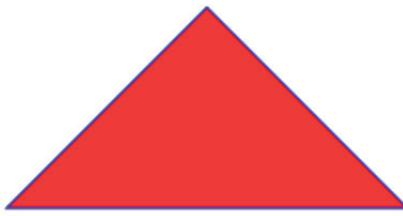


FIGURE 11.7:FILLED PATH DRAWN BY USING AN <SVG> ELEMENT



Additional Reading: For more information about the `<path>` element, including details on how to draw arcs and Bezier curves, see <http://go.microsoft.com/fwlink/?LinkId=267746>.

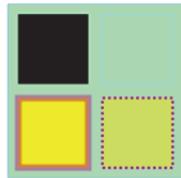
Specifying Fill Styles and Stroke Styles

SVG-related elements have a series of attributes that enable you to specify how to fill an element and how to draw its outline. These attributes include:

- **stroke** and **fill**: Specify the outline color and the fill color of the shape, respectively.
- **stroke-opacity** and **fill-opacity**: Specify the opacity of the outline and the filling of the shape, respectively. The opacity is a fractional value between 0.0 and 1.0. A value of 0.0 means completely transparent, and a value of 1.0 means completely opaque. The default opacity is 1.0.
- **stroke-width**: Specifies the width of the outline, either as an absolute size or as a percentage of the shape size.
- **fill-rule**: Specifies how to determine what side of a path is inside the shape. The **fill-rule** attribute is important in complex overlapping shapes, because it enables SVG to determine which parts of the shape to fill with the fill color. There are two possible values for **fill-rule**: **nonzero** and **evenodd**.

• You can set the fill style or stroke style for an element

```
stroke="color"
fill="color"
stroke-opacity="opacity-fraction"
fill-opacity="opacity-fraction"
stroke-width="width"
fill-rule="nonzero" | "evenodd"
stroke-dasharray="dash-gap series"
```



Additional Reading: For more information about fill rules, visit <http://go.microsoft.com/fwlink/?LinkId=267747>.

- **stroke-dasharray**: Specifies a pattern of dashes and gaps to use when drawing the outline. The **dasharray** attribute is a series of numbers specified by spaces or commas. The first number specifies the length of a dash; the second number specifies the length of a gap; the third number specifies the length of the next dash; the fourth number specifies the length of the next gap; and so on. Numbers can be expressed either as absolute values or as percentages.

The following example creates a rectangle with a black interior and no outline:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="10" y="10" width="50" height="50" fill="black"/>
```

```
</svg>
```

The next example creates a rectangle with no interior and a light blue outline:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <rect x="70" y="10" width="50" height="50" fill="none" stroke="lightblue"/>
</svg>
```

The next example creates a rectangle with a yellow interior and a thick, semi-transparent purple outline:

```
<svg xmlns="http://www.w3.org/2000/svg">
<rect x="10" y="70" width="50" height="50"
      fill="yellow"
      stroke="purple"
      stroke-width="5"
      stroke-opacity="0.5" />
</svg>
```

The final example creates a rectangle with a semi-transparent yellow interior and a dashed purple outline:

```
<svg xmlns="http://www.w3.org/2000/svg">
<rect x="70" y="70" width="50" height="50"
      fill="yellow"
      fill-opacity="0.5"
      stroke="purple"
      stroke-width="2"
      stroke-dasharray="2 2" />
</svg>
```

The following image shows the different fill and stroke styles generated by using these code examples:



FIGURE 11.1:FILL AND STROKE STYLES FOR SHAPES DRAWN BY USING AN <SVG> ELEMENT

Using Gradients and Patterns

SVG provides three elements that enable you to specify gradients and patterns that you can use to fill a shape or draw an outline:

- **<linearGradient>**
- **<radialGradient>**
- **<pattern>**

The following sections describe these elements in more detail.

The <linearGradient> Element

The <linearGradient> element creates a linear

- You can define a linear or radial gradient for shapes
- Patterns can specify image files

```
<polygon points="50,50 250,50 150,200" fill="url(#gradient1)" />
<ellipse cx="150" cy="50" rx="100" ry="20" fill="url(#wales)" />
<circle cx="150" cy="250" r="50" fill="url(#gradient2)" />
```



gradient of colors that can be applied to a shape. The **<linearGradient>** element has four attributes to define the start and end locations of the linear gradient on the target shape:

- **x1** and **y1**: Specifies the start point of the linear gradient on the target shape.
- **x2** and **y2**: Specifies the end point of the linear gradient on the target shape.

You can specify any number of color stops in a linear gradient. Each color stop is specified as a **<stop>** child element and has two attributes:

- **offset**: Specifies the location of the color stop along the linear gradient.
- **stop-color**: Specifies the color to apply at this location along the linear gradient.

The following example shows how to define a linear gradient. The linear gradient starts at the top left corner of the shape to which it is applied, because the **x1** and **y1** values are 0%. The linear gradient ends at the top right corner of the target shape, because the **x2** value is 100% and the **y2** value is 0%. The linear gradient therefore defines a horizontal line across the top of the target shape. There are three color stops that are applied along this line, creating a linear gradient that ranges from yellow to green, and then from green to red:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <linearGradient x1="0%" y1="0%" x2="100%" y2="0%">
    <stop offset="0.2" stop-color="yellow" />
    <stop offset="0.5" stop-color="green" />
    <stop offset="1.0" stop-color="red" />
  </linearGradient>
</svg>
```

 **Note:** A gradient is a reusable resource that you can apply to multiple shapes, as described in the section "Creating Reusable Gradients and Patterns" later in this topic.

The following image shows this linear gradient applied to a rectangle:

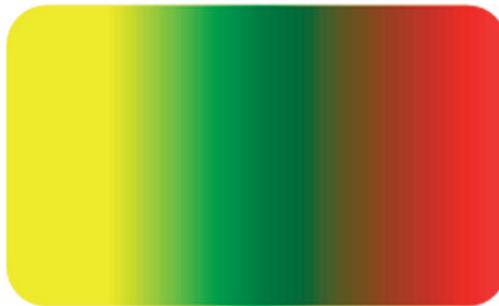


FIGURE 11.2:A LINEAR GRADIENT APPLIED TO A RECTANGLE

The **<radialGradient>** Element

The **<radialGradient>** element creates a radial gradient of colors that can be applied to a shape. The **<radialGradient>** element has two attributes named **fx** and that define the focal point of the radial gradient on the target shape.

You can specify any number of color stops in a radial gradient. Each color stop has **offset** and **stop-color** attributes; the **offset** attribute represents a percentage distance from **(fx,fy)** to the edge of the outermost circle.

The following example shows how to define a radial gradient. The radial gradient is focused on a point 30% from the top left corner of the target shape, because the **fx** and **fy** values are 0.3. There are two color

stops that are applied as concentric circles centered on (fx, fy), creating a radial gradient that ranges from yellow to red:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <radialGradient id="gradient2" fx="0.3" fy="0.3">
    <stop offset="0.1" stop-color="yellow" />
    <stop offset="0.7" stop-color="red" />
  </radialGradient>
</svg>
```

The following image shows this radial gradient applied to a rectangle:

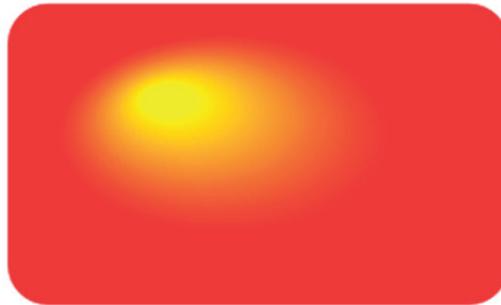


FIGURE 11.3:A RADIAL GRADIENT APPLIED TO A RECTANGLE

The `<pattern>` Element

The `<pattern>` element creates a pattern that can be applied to a shape. The `<pattern>` element has attributes that specify the width and height of the pattern. The content to display in the pattern is specified as a child element of the `<pattern>` element.

The following example shows how to define a pattern based on an image file named "wales.png" which contains an image of the Welsh flag.

```
<svg xmlns="http://www.w3.org/2000/svg">
  <pattern patternUnits="userSpaceOnUse" width="100" height="100">
    <image xlink:href="wales.png" x="0" y="0" width="100" height="100" />
  </pattern>
</svg>
```

 **Note:** The `patternUnits` attribute controls how the image in a pattern is displayed. The value `userSpaceOnUse` causes the image to be repeatedly tiled without any spacing.

The following image shows this pattern applied to a rectangle:



FIGURE 11.4:A PATTERN APPLIED TO A RECTANGLE

MCIT '11ICE ONLY. STUDENT USE PROHIBITED

Creating Reusable Gradients and Patterns

A web page might use a particular gradient or pattern at several different places. Rather than defining each gradient or pattern separately, you can define them once within a `<defs>` element, and then refer to them by their `id` when you want to apply them to elements in your page.

The following example defines three reusable gradients and patterns, and gives each one a unique `id`. The example then creates several shapes. The shapes make use of the gradients and patterns by referencing their `id` values:

```
<svg xmlns="http://www.w3.org/2000/svg">
    <!-- Define some gradients and patterns, for use later -->
    <defs>
        <linearGradient id="gradient1" ... />
        <radialGradient id="gradient2" ... />
        <pattern id="wales" ... />
    </defs>
    <!-- Draw shapes that make use of the gradients and patterns -->
    <polygon points="50,50 250,50 150,200" fill="url(#gradient1)" />
    <ellipse cx="150" cy="50" rx="100" ry="20" fill="url(#wales)" />
    <circle cx="150" cy="250" r="50" fill="url(#gradient2)" />
</svg>
```

The result of this code looks like this:

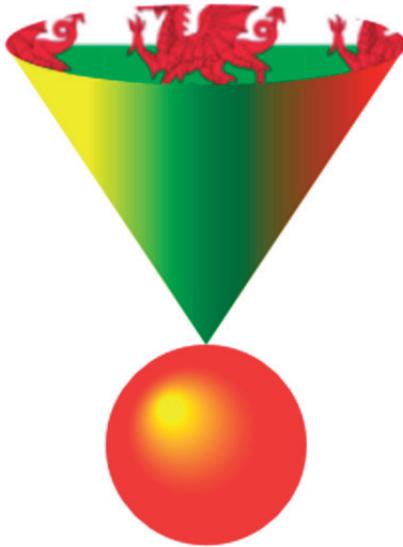


FIGURE 11.5: PATTERNS AND GRADIENTS APPLIED TO A SET OF SHAPES

Drawing Graphical Text

SVG provides a `<text>` element that enables you to draw graphical text. You specify the text between the `<text>` start tag and the `</text>` end tag.

SVG provides ways to customize the appearance of text:

- Text style.
- Text decorations.
- Text paths.
- Text span.

The following sections describe how to apply these customizations.

Text Style

The `<text>` element has a range of attributes that enable you to specify the text style. For example, you can set the **fill**, **stroke**, and **stroke-width** attributes to define how the filling and outline for the text. You can also set the **font-size**, **font-family**, and **font-weight** attributes to specify the font for the text. The following example shows how to use these attributes:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <text x="20" y="50"
    fill="yellow" stroke="purple" stroke-width="2"
    font-size="36" font-family="verdana" font-weight="bold">
    Styled text
  </text>
</svg>
```

The resulting text looks like this:

A large, bold, yellow text "Styled text" with a purple outline and a thick stroke width of 2 pixels.

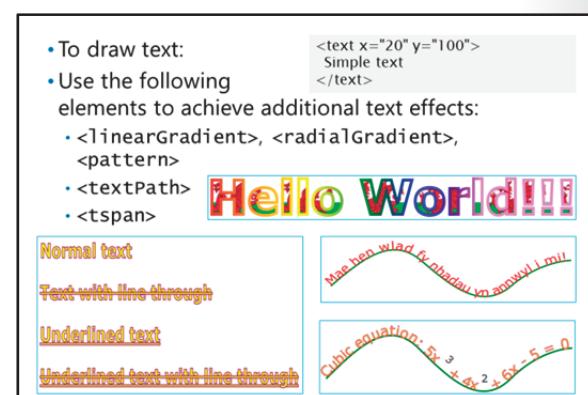
FIGURE 11.6: TEXT DRAWN AND STYLED BY USING THE `<TEXT>` ELEMENT OF AN `<SVG>` ELEMENT

Note: You can use gradients and patterns to fill text, using the techniques described in the previous topic, "Using Gradients and Patterns"; just set the **fill** attribute to reference an appropriate gradient or pattern.

Text Decorations

The `<text>` element has a **text-decoration** attribute that takes a space-separated list of text decorations. The following text decorations are supported:

- **underline**
- **overline**
- **line-through**
- **blink**



The following example shows how to set text decorations on `<text>` elements. The `<text>` elements are grouped inside a `<g>` element. The `<g>` element enables you to group SVG shapes together and handle the result as though it were a single shape. In this case, the `<g>` element defines a common set of styles and property values that apply to all `<text>` elements in the group:

```
<svg xmlns="http://www.w3.org/2000/svg" id="decoratedText">
  <g fill="yellow" stroke="purple" stroke-width="2" font-size="36" font-
family="verdana" font-weight="bold">
    <text x="20" y="50">Normal text</text>
    <text x="20" y="150" text-decoration="line-through">Text with line
through</text>
    <text x="20" y="250" text-decoration="underline">Underlined text</text>
    <text x="20" y="350" text-decoration="underline line-through">
      Underlined text with line through
    </text>
  </g>
</svg>
```

The text generated by this code looks like this:

Normal text

Text with line through

Underlined text

Underlined text with line through

FIGURE 11.7:TEXT DRAWN AND DECORATED BY USING THE `<TEXT>` ELEMENT OF AN `<SVG>` ELEMENT

Text Paths

A `<text>` element can contain a `<textPath>` child element that indicates a path to use as the baseline for the text when it is displayed on the web page. For example, you can create a `<textPath>` that draws text around the perimeter of another shape on the page.

The following example shows how to use `<textPath>`. In this example, a `<path>` element is created to represent a wavy line. The `<text>` element has a `<textPath>` child element that links to the `<path>` element by using an XLink expression. An XLink expression enables you to reference a fragment of XML or HTML code by using its identifier, as shown in the following example:

```
<svg xmlns="http://www.w3.org/2000/svg" id="textPath">
  <path id="wavyPath1"
    fill="none" stroke="green" stroke-width="5"
    d="M 50 250
      C 150 150 250 50 350 150
      C 450 250 550 350 650 250
      C 750 150 850 150 850 150" />
  <text font-size="46" fill="red" font-family="Verdana">
    <textPath xlink:href="#wavyPath1">
      Mae hen wlad fy nhadau yn annwyl i mi!
    </textPath>
  </text>
</svg>
```

```
</svg>
```

The text and path drawn by this code looks like this:

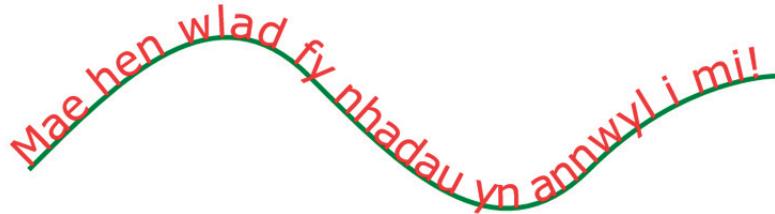


FIGURE 11.8:TEXT DRAWN BY USING TEXT PATH

Text Spans

A **<textSpan>** element can contain any number of **<tspan>** child elements that partition the text into a series of discrete sections. Each section can be styled individually.

The following example shows how to partition a **<text>** element by using multiple **<tspan>** elements. The **<tspan>** elements specify how to draw that particular part of text:

```
<svg xmlns="http://www.w3.org/2000/svg" id="textSpans">
  <defs>
    <path id="wavyPath2"
      d="M 50 250
          C 150 150 250 50 350 150
          C 450 250 550 350 650 250
          C 750 150 850 150 850 150" />
  </defs>
  <text font-size="46" fill="orange" font-family="Verdana" stroke="purple">
    <textPath xlink:href="#wavyPath2">
      <tspan>Cubic equation: 5x</tspan>
      <tspan dy="-30" fill="green" font-size="33">3</tspan>
      <tspan dy="+30"> + 4x</tspan>
      <tspan dy="-30" fill="green" font-size="33">2</tspan>
      <tspan dy="+30"> + 6x - 5 = 0</tspan>
    </textPath>
  </text>
</svg>
```

The result looks like this (the path is the same as the previous example, but the text elements show a combination of font sizes and fill colors):

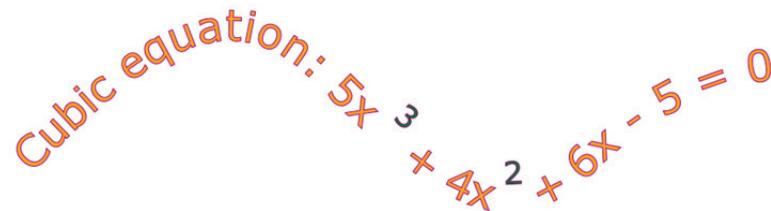


FIGURE 11.9:TEXT CONSISTING OF SEVERAL <TSPAN> ELEMENTS

MCY USE ONLY STUDENT LICENSE PROHIBITED

Transforming SVG Elements

Transformations enable you to relocate, resize, rotate, and reshape an element. You can use transformations in conjunction with JavaScript code to create animated graphics.

Transformations

To transform an element, set the **transform** attribute to one of the following values:

- **rotate(angle, cx, cy)**: Rotates the shape by the specified angle, about the specified (cx, cy) center point.
- **translate(dx, dy)**: Translates the shape by the specified distance in the X and Y directions.
- **scale(sx, sy)**: Scales the shape by the specified fraction in the X and Y directions.
- **skewX(angle)**: Skews the shape by the specified angle in the X direction.
- **skewY(angle)**: Skews the shape by the specified angle in the Y direction.



Additional Reading: For more information about SVG transformations, visit <http://go.microsoft.com/fwlink/?LinkId=267748>.

You can apply multiple transformations on a shape by using nested `<g>` elements and then applying the `transform` attribute on each shape. The following example shows how to perform multiple transformations on a rectangle. The example defines two rectangles; the first rectangle is displayed without transformation, and the second rectangle is displayed with a translation, a scaling of 0.5 to make it half the original size, and a rotation of 20 degrees about a center point of (160, 160):

```

<svg xmlns="http://www.w3.org/2000/svg" id="transformations" >
    <defs>
        <pattern id="checkerPattern" width="80" height="80"
        patternUnits="userSpaceOnUse">
            <rect fill="red" x="0" y="0" width="40" height="40" />
            <rect fill="blue" x="40" y="0" width="40" height="40" />
            <rect fill="blue" x="0" y="40" width="40" height="40" />
            <rect fill="red" x="40" y="40" width="40" height="40" />
        </pattern>
    </defs>
    <rect x="0" y="0" width="200" height="200" fill="url(#checkerPattern)"
    stroke="orange" stroke-width="5" />
    <g transform="translate(200, 200)">
        <g transform="scale(0.5)">
            <g transform="rotate(20, 160, 160)">
                <rect x="0" y="0" width="200" height="200" fill="url(#checkerPattern)"
                stroke="orange" stroke-width="5" />
            </g>
        </g>
    </g>
</svg>

```

- To transform SVG elements, set the `transform` attribute to a transformation function

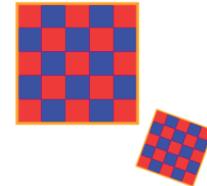
`rotate(angle, cx, cy)`

`translate(dx, dy)`

`scale(sx, sy)`

`skewX(angle)`

`skewY(angle)`



- To perform a transformation on several elements enclose elements in a `<g>` element, and transform the `<g>`

The rectangles drawn by this code look like this:

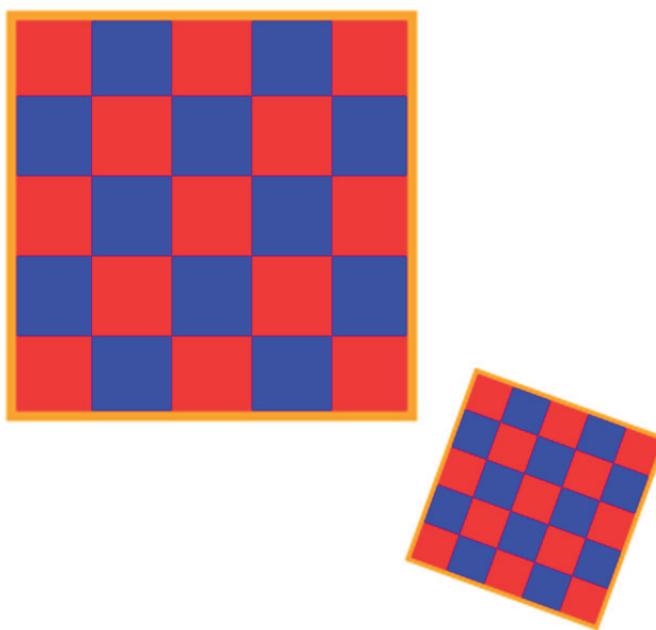


FIGURE 11.10:A RECTANGLE ROTATED, TRANSFORMED, AND SCALED BY USING AN <SVG> ELEMENT

Demonstration: Using SVG Transformations and Events

In this demonstration, you will see how to apply transformations to SVG elements, animate SVG elements, and handle events on SVG elements.

Demonstration Steps

Transform SVG Elements

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod11\Democode\SvgDocument.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. In Internet Explorer, click the **Transformations** button.
6. Right-click the web page in Internet Explorer, and then click **View source**.
7. In the source window, locate the `<!-- Demonstrate transformations -->` comment and review the **<svg>** element:
 - o The **<transform>** elements translate the square by 200 units in the X and Y axes, scale it by a factor of 0.5, and rotate it.
8. Close the source window.

Handle Events on SVG Elements

1. In Internet Explorer, click the **Events** button.

2. Hover the mouse over the red shape on the left side of the window. Verify that the shape changes to a yellow fill color and a dotted green border.
3. Move the mouse off the shape. Verify that it reverts to a red fill color with no outline.
4. Hover the mouse over the blue shape on the right side of the window. Verify that the shape changes to a yellow fill color and a dotted green border.
5. Move the mouse off the shape. Verify that it reverts to a blue fill color with no outline.
6. Click the red shape. Verify that a message box appears, indicating that the shape represents Alaska. Close the message box.
7. Click the blue shape. Verify that a message box appears, indicating that the shape represents Hawaii. Close the message box.
8. Right-click in the browser window, and then click **View source**.
9. In the source window, locate the **<!-- Demonstrate events -->** comment and review the **<svg>** element:
 - o The **<path>** elements contain the data that defines the two maps.
 - o Each **<path>** element responds to the **onmousedown** event and uses JavaScript code to display the appropriate message.
10. In the source window, locate the **path:hover** CSS rule near the top of the document. This CSS rule defines the style for all **<path>** elements when the user hovers over them with the mouse.
11. Close the source window.
12. Close Internet Explorer.

Lesson 2

Drawing Graphics by Using the Canvas API

The Canvas API comprises a `<canvas>` element and an associate set of JavaScript functions that enable you to draw shapes onto the canvas. The Canvas API is an alternative to SVG graphics and is useful if you want to perform one-off graphical operations in a web page. However, whereas SVG graphics are defined by using HTML5 elements, the Canvas API is programmatic and requires you to draw graphics by writing JavaScript code.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how the Canvas API works.
- Create a `<canvas>` element on a web page and use the Canvas API to draw simple shapes and lines.
- Use the Canvas API to draw complex shapes.
- Fill shapes by using gradients and patterns.
- Apply transformations and animations to canvas drawings.

What is the Canvas API?

The Canvas API enables you to draw graphical shapes onto a bitmap area on the web page. To use the Canvas API, you add a `<canvas>` element to the page and then call JavaScript functions to draw shapes on the canvas drawing surface.

The Canvas API uses a "fire-and-forget" model. When you call JavaScript functions to draw shapes on a canvas, it is as if you are painting shapes on a piece of paper. As soon as you have drawn the shapes, the drawing is complete. The shapes are not retained in the DOM tree, so there is no way to interact with the shapes afterwards. For

example, you cannot access shapes by using the DOM, you cannot apply CSS styles to canvas shapes, and you cannot handle events on canvas shapes.

The performance of the Canvas API depends on the size and resolution of the device screen. The larger the device, and the higher the resolution, the more pixels have to be painted and the slower the drawing will be rendered. If you want to create drawings with comprise relatively few elements but that target large high-resolution devices, SVG might be a better option.

- The Canvas API enables you to draw onto a bitmap area
 - Immediate mode: "fire and forget"
 - It does not remember what you drew last
- JavaScript APIs and drawing primitives
 - Simple API: 45 methods, 21 attributes
 - Rectangles, lines, fills, arcs, Bezier curves, ...
- No DOM support
 - A canvas is a "black box"

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Canvas API

To use the Canvas API, the first step is to add a `<canvas>` element to your web page. You can also define a CSS style for `<canvas>` elements, if required.

It is typical to define an `id` attribute on a `<canvas>` element, so that you can locate it easily in JavaScript code by calling a DOM function such as `document.getElementById()`. It is also quite common to define fallback content between the `<canvas>` start tag and the `</canvas>` end tag; this content will be displayed by browsers that do not recognize the `<canvas>` element.

To draw graphics on a canvas, you must write JavaScript code. Follow these steps:

1. Get a reference to the `<canvas>` element by calling a DOM function such as `document.getElementById()`.
2. Call `getContext('2d')` on the canvas object, to get the two-dimensional drawing context for the canvas.
3. Invoke methods in the context object, to draw shapes on the canvas surface.

The following example shows how to create a canvas and draw a rectangle on it. The example sets the `context.fillStyle` property to set the ambient fill color to red for subsequent drawing operations. The example then calls the `context.fillRect()` method to draw a solid rectangle. The rectangle will be filled with the ambient fill color, which is red. The canvas itself has a dark blue border and a light green fill color, due to the CSS style rule at the top of the code:

```
<style>
    canvas {
        border: 2px solid darkblue;
        background-color: lightgreen;
    }
</style>
...
<h1>Getting started with canvas</h1>
<canvas id="myCanvas">No canvas support in this browser</canvas>
<script>
    var canvas = document.getElementById('myCanvas');
    var context = canvas.getContext('2d');
    context.fillStyle = "red";
    context.fillRect(20, 20, canvas.width - 40, canvas.height - 40);
</script>
```

The rectangle drawn by this code look like this:



• Create a `<canvas>` element
<h1>Getting started with canvas</h1>
<canvas id="myCanvas">No canvas support</canvas>

• Define styles for the `<canvas>` element
canvas {
 border: 2px solid darkblue;
 background-color: lightgreen;
}

• Write JavaScript code to draw on the canvas
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
context.fillStyle = "red";
context.fillRect(20, 20, canvas.width - 40, canvas.height - 40);

Getting started with canvas



FIGURE 11.11:A RECTANGLE DRAWN BY USING THE CANVAS API

Note that the Canvas API also provides a **context.strokeRect()** function that draws the outline of a rectangle but does not fill its interior.

The Canvas API has a range of functions for drawing shapes and lines, including:

- **arc()** and **arcTo()**: Draw an arc.
- **quadraticCurveTo()**: Draw a quadratic Bezier curve.
- **bezierCurveTo()**: Draw a cubic Bezier curve.

 **Additional Reading:** For more information about the full set of functions available in a canvas two-dimensional context, see <http://go.microsoft.com/fwlink/?LinkID=267749>.

Drawing Paths

You can draw complex shapes by using a path. The Canvas API has a **beginPath()** function that enables you to create a path connecting a series of points. You can then call functions such as **moveTo()** and **lineTo()** to move to new locations, and optionally you can call **closePath()** to connect the final point back to the first point.

When you are ready to render the path, you can call the **stroke()** function to draw the outline of the path. You can also call the **fill()** function to draw the interior of the path.

- Use a path to draw a complex shape
- Use the **beginPath**, **moveTo**, **lineTo**, and **closePath** functions to define the shape
- Draw the shape by using the **stroke** function
 - Specify the style by using the **strokeStyle** function
- Fill the shape by using the **fill** function
 - Specify the style by using the **fillStyle** function

The following example shows how to draw a triangle path by using the Canvas API. The triangle has a blue outline color, because the **strokeStyle** property is set to **rgb(0, 0, 255)**. The triangle has a semi-transparent red fill color, because the **fillStyle** property is set to **rgba(255, 0, 0, 0.75)**.

```
<canvas id="myCanvas">
    Sorry, your browser does not support canvas.
</canvas>
<script>
    // Get the canvas element and its drawing context.

```

```
var canvas = document.getElementById('myCanvas');
var context = canvas.getContext('2d');
// Clear any existing content in the canvas.
context.clearRect(0, 0, canvas.width, canvas.height);
// Set the stroke color and the fill color.
context.strokeStyle = "rgb(0, 0, 255)";
context.fillStyle = "rgba(255, 0, 0, 0.75)";
// Create a path in absolute coordinates.
context.beginPath();
context.moveTo(60, 30);
context.lineTo(100, 90);
context.lineTo(20, 90);
// Close the path.
context.closePath();
// Draw the path as a stroked shape;
context.stroke();
// Draw the path as a filled shape.
context.fill();
</script>
```

The triangle drawn by this code looks like this:

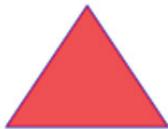


FIGURE 11.12:A TRIANGLE DRAWN BY USING THE CANVAS API

Using Gradients and Patterns

The Canvas API provides functions that enable you to specify gradients and patterns that you can use to fill a shape or draw an outline:

- **createLinearGradient()**
- **createRadialGradient()**
- **createPattern()**

The following sections describe these functions in more detail.

The **createLinearGradient()** Function

The **createLinearGradient()** function creates a linear gradient. **createLinearGradient()** has parameters that define the start and end locations of the linear gradient, and returns a linear gradient object. You can add color stops to the linear gradient by calling the **addColorStop()** function. You can then set the linear gradient as the ambient fill style or stroke style for a canvas context, by setting the **fillStyle** or **strokeStyle** property. You can also use the gradient to define the fill color or the stroke color for text.

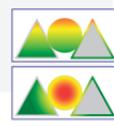
The **createRadialGradient()** Function

The **createRadialGradient()** function creates a radial gradient. **createRadialGradient()** has parameters that define the focal point of the radial gradient. You can add color stops to the gradient and apply it to a context in the same way that you would for a linear gradient.

- To fill a shape with a gradient:

```
var grad = ctx.createLinearGradient(x1, y1, x2, y2);
var grad = ctx.createRadialGradient(startCx, startCy, startRad,
endCx, endCy, endRad);

grad.addColorStop(fraction1, color1);
grad.addColorStop(fraction2, color2);
...
ctx.fillStyle = grad;
```



- To fill a shape with a pattern:

```
var image = document.getElementById("anImageElement");
var pattern = ctx.createPattern(image, "repeat");
ctx.fillStyle = pattern;
```



The **createPattern()** Function

The **createPattern()** function creates a pattern, typically based on an image or some other content on the web page. You can apply a pattern in the same way that you would for a linear gradient or a radial gradient.

The following example shows how to create a linear gradient, a radial gradient, and a pattern. The pattern used is based on the **wales** image (this image is the flag of Wales in the United Kingdom).

```
<canvas id="myCanvas">
    Sorry, your browser doesn't support canvas.
</canvas>

<script>
    // Get the canvas element and its drawing context.
    var canvas = document.getElementById('myCanvas');
    var context = canvas.getContext('2d');
    context.lineWidth = 5;
    demoLinearGradient();
    demoRadialGradient();
    demoPattern();
    function demoLinearGradient()
    {
        var linearGradient = context.createLinearGradient(0, 0, 0, canvas.height);
        linearGradient.addColorStop(0, "red");
        linearGradient.addColorStop(0.4, "yellow");
        linearGradient.addColorStop(1, "green");
        drawShapes(linearGradient);
    }
    function demoRadialGradient()
    {
        var radialGradient = context.createRadialGradient(canvas.width/2, canvas.height/2,
10, canvas.width/2, canvas.height/2, 100);
        radialGradient.addColorStop(0, "red");
        radialGradient.addColorStop(0.4, "yellow");
        radialGradient.addColorStop(1, "green");
        drawShapes(radialGradient);
    }
    function demoPattern()
    {
        var image = document.getElementById("wales");
        var pattern = context.createPattern(image, "repeat");
        drawShapes(pattern);
    }
    function drawShapes(theStyle)
    {
        // Clear any existing content in the canvas.
        context.clearRect(0, 0, canvas.width, canvas.height);
        // Draw a filled triangle, using the specified style.
        context.fillStyle = theStyle;
        context.strokeStyle = "rgb(200, 200, 200)";
        context.beginPath();
        context.moveTo(70, 30);
        context.lineTo(130, 140);
        context.lineTo(10, 140);
        context.closePath();
        context.fill();
        context.stroke();
        // Draw a stroked circle, still using the specified style.
        context.beginPath();
        context.arc(canvas.width/2, canvas.height/2, 50, 0, 2*Math.PI);
        context.fill();
        // Draw a stroked triangle, using the specified style.
        context.fillStyle = "rgb(200, 200, 200)";
        context.strokeStyle = theStyle;
        context.beginPath();
        context.moveTo(230, 30);
```

```
    context.lineTo(290, 140);
    context.lineTo(170, 140);
    context.closePath();
    context.fill();
    context.stroke();
}
</script>
```

The shapes generated by this code look like this:



FIGURE 11.13:SHAPES FILLED WITH A PATTERN FROM AN IMAGE FILE

Transforming Shapes

The Canvas API enables you to transform the coordinates of the canvas context so that subsequent drawing operations are performed on a transformed coordinate system. To use transformations in the Canvas API, use the following functions:

- **rotate(angle)**: Rotates the coordinate system by the specified angle clockwise in radians.
- **translate(dx, dy)**: Translates the coordinate system by the specified distance in the X and Y directions.
- **scale(sx, sy)**: Scales the coordinate system by the specified fraction in the X and Y directions.
- **transform(scaleX, skewX, scaleY, skewY, translateX, translateY)**: Adjusts the current transformation matrix to perform scaling, skewing, and translation.
- **setTransform(scaleX, skewX, scaleY, skewY, translateX, translateY)**: Sets a new transformation matrix to perform scaling, skewing, and translation.

- To rotate the canvas context:
`ctx.rotate(clockwiseAngleInRadians);`

- To translate the canvas context:
`ctx.translate(deltaX, deltaY);`

- To scale the canvas context:
`ctx.scale(xScaleMultiple, yScaleMultiple);`

- To adjust the current transformation matrix:
`ctx.transform(scaleX, skewX, scaleY, skewY, translateX, translateY);`

- To set a new transformation matrix:
`ctx.setTransform(scaleX, skewX, scaleY, skewY, translateX, translateY);`



Additional Reading: For more information about Canvas transformations, visit <http://go.microsoft.com/fwlink/?LinkId=267750>.

Demonstration: Performing Transformations by Using the Canvas API

In this demonstration, you will see how use transformations to rotate, skew, and translate graphics.

Demonstration Steps

Perform Simple Transformations

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod11\Democode\CanvasDocument.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. In Internet Explorer, click the **Separate Transformations** button.
6. Right-click the web page in Internet Explorer, and then click **View source**.
7. In the source window, locate the **demoSeparateTransformations** function and review the code:
 - o The **demoSeparateTransformations** function uses the **drawShape** function to draw a triangle filled with an image of the Welsh flag.
 - o Before calling the **drawShape** function, the code transforms the context; it moves the canvas to the right and down by half the width and height of the canvas, then it scales the context by a different value in the X and Y dimensions, and then rotates the context by $\pi/4$ radians.
 - o When the **drawShape** function is called, the image is transformed according to the context settings.

 **Note:** If time allows, comment out each of the transformations and run the code again. Then uncomment each transformation one at a time, so that students can see the effects of each one.

Perform Matrix Transformation

1. In Internet Explorer, click the **Matrix Transformations** button.
2. In the source window, locate the **demoMatrixTransformations** function and review the code:
 - o This function is similar to the previous one in that it transforms the context and then calls the **drawShape** function to display the image.
 - o The difference is that this function uses the **transform** function to perform a matrix transformation, scaling, skewing, and translating the context in a single function call.
3. Close the source window.
4. Close Internet Explorer.

Demonstration: Creating Advanced Graphics

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Lab: Creating Advanced Graphics

Scenario

The conference organizers would like a venue map displayed on the website. Conference attendees will use the map to find out more about the rooms of the conference facility. Therefore, the map should be interactive, responding to mouse clicks. The floor plans are available in a vector format, so they can be displayed in a resolution-independent format.

Conference speakers need badges with their photo, name, and ID. The ID is in the form of a barcode to make it easy for security personnel to scan and verify the holder's identity before allowing backstage access. You have been asked to create a web page that enables a speaker to create a badge.

Objectives

After completing this lab, you will be able to:

- Create graphics by using SVG, interactively style SVG graphics, and handle SVG graphics events.
- Draw graphics by using the Canvas API.

Estimated Time: 60 minutes

- Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
- User Name: **Student**
- Password: **Pa\$\$w0rd**

Exercise 1: Creating an Interactive Venue Map by Using SVG

Scenario

In this exercise, you will create an interactive conference venue map.

First, you will complete the partially completed SVG mark-up of the venue map. Next, you will add interactive styling to the SVG by using CSS. Then you will handle SVG element click events to display extra information about conference rooms. Finally, you will run the application, view the Location page, and verify that the venue map is interactive.

The main tasks for this exercise are as follows:

1. Review the incomplete HTML markup for the venue map.
2. Complete the SVG venue map.
3. Add interactivity to the venue map.
4. Test the application.

► Task 1: Review the incomplete HTML markup for the venue map

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution in the **E:\Mod11\Labfiles\Starter\Exercise 1** folder.
4. Open the **location.htm** file.
5. Verify that the page contains the following **<svg>** element of the venue map, and two hidden **<div>** elements containing room information:

```
<svg viewBox="-1 -1 302 102" width="100%" height="230">
    <!-- Room A -->
    <g id="room-a" class="room">
        <rect fill="#fff" x="0" y="0" width="100" height="100"/>
        <text x="13" y="55" font-weight="bold" font-size="20">ROOM A</text>
    </g>
    <!-- Room B -->
    <!-- The outline of the building -->
    <polyline fill="none" stroke="#000" points="135,95 140,100 0,100 0,0 100,0 100,80
130,80 130,70 110,70 110,30 190,30 190,70 170,70 170,80 200,80 200,0 300,0 300,100
160,100 165,95"/>
    <text x="150" y="55" font-size="12" style="text-anchor: middle">Registration</text>
</svg>
```

Also notice that the script references the **location-venue.js** script in the **scripts/pages** folder:

```
<script src="/scripts/pages/location-venue.js" type="text/javascript"></script>
```

- Run the application and view the **Location** page. Notice that the details for Room B are missing:

The venue map currently looks like this:

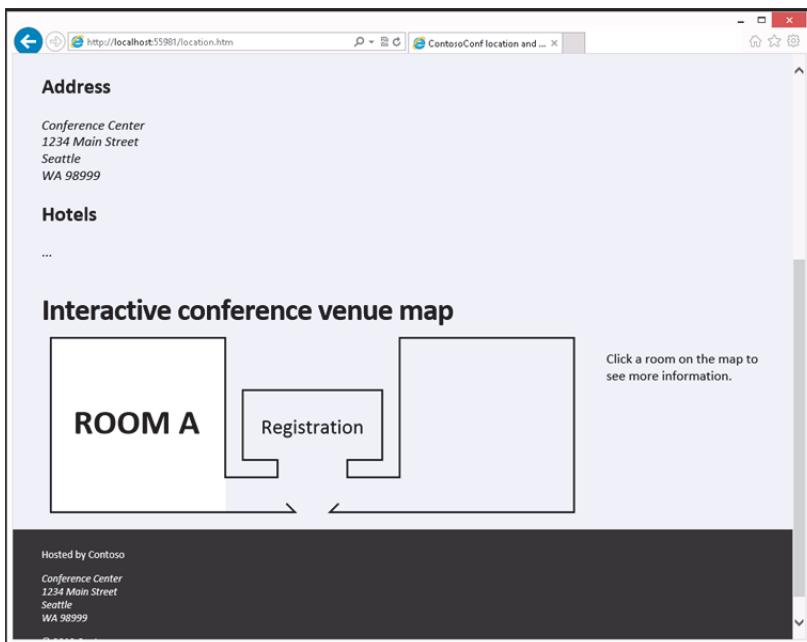


FIGURE 11.1:THE INCOMPLETE VENUE MAP

- Close Internet Explorer.

► Task 2: Complete the SVG venue map

- In the **location.htm** file, add the SVG elements for Room B by using **room-b** as the group element id. The missing elements are a filled rectangle and the text with the name of the room. Use the SVG elements for Room A as a guide.

 **Note:** You may need to view the page in Internet Explorer and experiment with the coordinate values to get elements in the correct location.

► Task 3: Add interactivity to the venue map

1. The venue map should be interactive; a user should be able to view more information about a room by clicking on it in the map. In the location.htm file, find the following **<div>** elements. These elements contain the information about each room, but they are hidden by default:

```
<div id="room-a-info" style="display: none">
  <h2>Room A</h2>
  <p>Capacity: 250</p>
  <p>Popular sessions in here:</p>
  <ul>
    <li>Diving in at the deep end with Canvas</li>
    <li>Real-world Applications of HTML5 APIs</li>
    <li>Transforms and Animations</li>
  </ul>
</div>
<div id="room-b-info" style="display: none">
  <h2>Room B</h2>
  <p>Capacity: 230</p>
  <p>Popular sessions in here:</p>
  <ul>
    <li>Building Responsive UIs</li>
    <li>Getting to Grips with JavaScript</li>
    <li>A Fresh Look at Layouts</li>
  </ul>
</div>
```

2. Notice that each **<div>** is named after the room with the suffix **-info**.



Note: The information for each room is hard-coded into the HTML markup. However, to make the page even more dynamic, you could retrieve information about popular sessions from the web service used in the **Schedule** page.

3. The room should change color when the mouse moves over it. Open the **location.css** style sheet in the **styles\pages** folder. This style sheet contains CSS for the **location.htm** page.
4. Add a CSS rule that targets **rect** elements when the cursor is hovering over **.room** elements.
5. In this rule, set the **fill** property to **#b1f8b0**
6. Open the **location-venue.js** file in the **scripts\pages** folder. This JavaScript file contains the **showRoomInfo** function that displays information about a room. The id of the room is specified as the parameter to this function.
7. Add **click** event listeners for the SVG **room** elements, which call the **showRoomInfo** function.
 - o After the comment // TODO: Get the room elements in the svg element, use the **querySelectorAll** function of the document object to find all elements that have the class set to **.room** and assign them to the **rooms** variable.
 - o After the comment // TODO: Add a click event listener for each room element, iterate through the list of elements in the **rooms** variable and add the **click** event handler for each item.

► Task 4: Test the application

1. Run the application and view the **location.htm** page.
2. Move the cursor over each room and verify that the fill color changes.

The venue map should look like this:

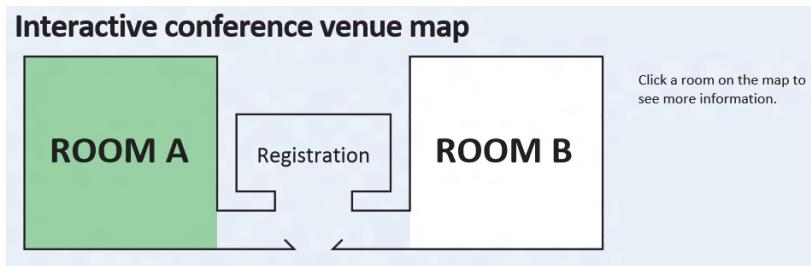


FIGURE 11.2:THE VENUE MAP WITH ROOM A HIGHLIGHTED

3. Click each room and verify that the correct information is displayed next to the venue map.

The information displayed for Room B should look like this:

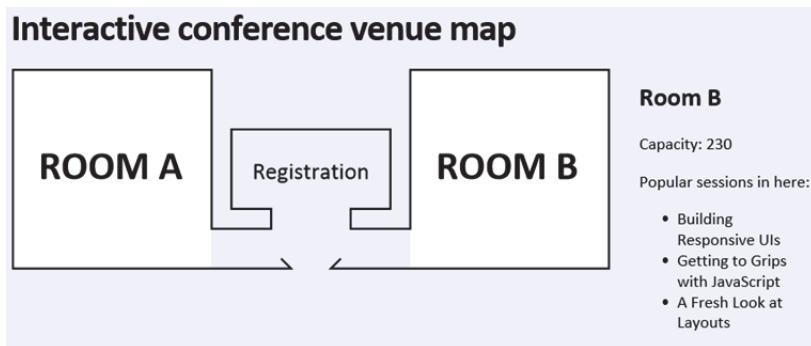


FIGURE 11.3:THE DETAILS FOR ROOM B

4. Close Internet Explorer.

Results: After completing this exercise, you will have a venue map that displays extra information when clicked.

Exercise 2: Creating a Speaker Badge by Using the Canvas API

Scenario

In this exercise, you will use the Canvas API to draw the elements of a conference speaker's badge.

First, you will create a canvas element on the speaker badge page. Next, you will write JavaScript code to implement methods that draw parts of the badge. Finally, you will run the application and test the speaker badge page.

The main tasks for this exercise are as follows:

1. Create the canvas element.
2. Draw the details for the badge.
3. Test the application.

► Task 1: Create the canvas element

1. In Visual Studio, open the ContosoConf.sln solution in the **E:\Mod11\Labfiles\Starter\Exercise 2** folder.
2. Open the **speaker-badge.htm** file. This page contains a section that enables the user to create their speaker badge. Previously, you used an **** element to drag and drop an image of the speaker onto this page. This **** element has been removed because you are going to modify the page to use a canvas instead. Using a canvas provides more scope for customizing the image.

MCT USE ONLY. STUDENTS USE PROHIBITED

3. Find following comment:

```
<!-- TODO: Add canvas here -->
```

4. Add a <canvas> element, with a **width** of **500** and a **height** of **200** after the comment.
5. Add a solid black border to the canvas by using a CSS style. Set the width to **1px**.
6. Add the following custom attributes to the canvas element:

```
data-speaker-id="234724"  
data-speaker-name="Mark Hanson"
```



Note: The custom data attributes provide a convenient way to store application-specific data in an HTML element. They will be used by the JavaScript code that draws the elements for the badge on the canvas.

In this exercise, these details are hard-coded into the HTML markup, but you could also write JavaScript code to dynamically populate these attributes.

► Task 2: Draw the details for the badge

1. Open **speaker-badge.js** file in the **scripts\pages** folder. The JavaScript code in this file contains a refactored version of the drag-and-drop code that you created during an earlier lab exercise. Notice that the canvas element on the page has been assigned to **this.canvas**:

```
this.canvas = element.querySelector("canvas");
```

The file also contains a number of other functions that will draw the various elements of the speaker's badge.



Note: The other drawing methods of the speaker badge page object will use the context, assigned to **this.context**, to do their drawing.

2. In the **drawBadge** method after the comment // TODO: Get the canvas's (**this.canvas**) context and assign to **this.context**, assign **this.context** to be the canvas's 2D context.
3. After the comment // TODO: Draw the following by calling the helper methods of 'this', add code to perform the following tasks:
 - Draw the background (use the **drawBackground** function).
 - Draw the top text (use the **drawTopText** function; this function simply generates the text **ContosoConf 2012 Speaker**, which is displayed at the top of the badge).
 - Draw the speaker name (use the **drawSpeakerName** function).
 - If the **image** variable references a valid image, then draw the speaker's image (use the **drawSpeakerImage** function and pass the **image** variable as the parameter), otherwise draw a placeholder (use the **drawImagePlaceHolder** function).
 - Draw the bar code (use the **drawBarCode** function and pass the value in the **speakerId** variable as the parameter).

4. In the **drawBackground** method, after the comment // TODO: Fill the canvas with a white rectangle, add statements to fill the canvas with a white rectangle.
 - o Set the **fillStyle** of the context to "**white**".
 - o Use the **fillRect** method of the context to draw and fill the rectangle; the **width** and **height** properties of the rectangle should be the same as those of the canvas.
5. In the **drawSpeakerImage** method, after the comment // TODO: Draw the image on the canvas, add code to draw the image on the canvas at the coordinates **(20, 20)**, with size **160 × 160**.
 - o Use the **drawImage** method of the context.
 - o Note that the image is not always square, so calculate the source coordinates and size that will display the central square portion only. Use the **Math.min** function to calculate the minimum of the image's **width** and **height** properties, like this:

```
var size = Math.min(image.width, image.height);
```

The following image highlights the central square portion of a rectangle.



FIGURE 11.4:THE CENTRAL SQUARE PORTION OF A RECTANGLE

6. In the **drawSpeakerName** method, after the comment // TODO: Draw this.speakerName on the canvas, add code to draw the speaker's name on the canvas:
 - o The speaker's name is available in the **this.speakerName** property.
 - o Configure the following properties of the canvas to style the text before drawing it.
 - i. **font: 40px sans-serif.**
 - ii. **fillStyle: black.**
 - iii. **textBaseline: top.**
 - iv. **textAlign: left.**
 - v. **fillText: the speaker's name.**

► Task 3: Test the application

1. Run the application and view the **speaker-badge.htm** page.
2. Drag and drop the **mark-hansen.jpg** image file from the **E:\Mod11\Labfiles\Resources** folder onto the canvas.
3. Verify that the image is drawn on the canvas, along with the speaker's details.

The speaker badge should look like this:

MCT USE ONLY. STUDENT USE PROHIBITED



FIGURE 11.5:THE SPEAKER BADGE FOR MARK HANSON

4. Close Internet Explorer.

Results: After completing this exercise, you will have a Speaker Badge page that enables a conference speaker to create their badge.

Module Review and Takeaways

In this module, you have seen how to use SVG and the Canvas API to draw graphical content in a web page.

SVG uses a retained drawing model, which means SVG elements are retained in the DOM tree. You can access SVG elements by using DOM functions, you can style SVG elements by using CSS style rules, and you can handle user-interaction events on SVG elements.

The Canvas API uses a fire-and-forget drawing model, which means shapes are drawn on the canvas but are not retained in the DOM tree. You cannot access shapes in a canvas by using DOM functions, or style the shapes, or handle any events on them. Nonetheless, the Canvas API is very useful if you need to draw static graphical images on the web page.

Review Question(s)

Test Your Knowledge

Question
Which of the following statements about SVG is false?
Select the correct answer.
You can use SVG to draw complex shapes, and fill them with gradients and patterns.
SVG elements are parsed by the browser when the page is first loaded, and they are then discarded from memory.
You can create SVG elements dynamically by using DOM functions such as <code>document.createElement()</code> .
You can handle events on SVG elements.
SVG elements must be enclosed in an <code><svg></code> container element on a web page.

Question: When might you consider using the Canvas API instead of using SVG?

Module 12

Animating the User Interface

Contents:

Module Overview	12-1
Lesson 1: Applying CSS Transitions	12-2
Lesson 2: Transforming Elements	12-7
Lesson 3: Applying CSS Keyframe Animations	12-16
Lab: Animating the User Interface	12-22
Module Review and Takeaways	12-26

Module Overview

Animations are a key element in maintaining the interest of a user in a website. Implemented carefully, animations improve the usability of a web page and provide useful visual feedback on user actions.

This module describes how to enhance web pages by using CSS animations. You will learn how to apply transitions to property values. Transitions enable you to specify the timing of property changes. For example, you can specify that an element should change its width and height over a five-second period when the mouse pointer hovers over it. Next, you will learn how to apply 2D and 3D transformations to elements. Transformations enable you to scale, translate, rotate, and skew elements. You can also apply transitions to transformations, so that the transformation is applied gradually over a specified animation period.

At the end of this module, you will learn how to apply keyframe animations to elements. Keyframe animations enable you to define a set of property values at specific moments during an animation. For example, you can specify the color and position of an element at 0 percent, 33 percent, 66 percent, and 100 percent of the animation period.

Objectives

After completing this module, you will be able to:

- Apply transitions to animate property values to HTML elements.
- Apply 2D and 3D transformations to HTML elements.
- Apply keyframe animations to HTML elements.

Lesson 1

Applying CSS Transitions

Transitions enable you to specify the timeframe for property value changes. Transitions improve the user interface by making property changes smooth and graceful, instead of suddenly changing from one value to another. In this lesson, you will learn how to implement transitions by using CSS.

Lesson Objectives

After completing this lesson, you will be able to:

- Apply simple transitions to element property values by using CSS.
- Configure transition information.
- Detect the end of a transition.

Applying Simple Transitions by Using CSS

When you define CSS style rules for elements, the style rules apply immediately by default. For example, the following style rules specify that `<div>` elements will have a width of 300px normally, but a width of 600px when the user hovers over them. The change from 300px to 600px will take effect as soon as the user hovers over a `<div>` element. Likewise, the reverse change from 600px to 300px will take effect immediately as the user moves the mouse away from the `<div>` element:

```
div {  
    width: 300px;  
}  
div:hover {  
    width: 600px;  
}
```

• A CSS3 **transition** enables you to define a transition for one or more properties
• The browser interpolates between initial value and final value over a specified duration

```
div {  
    width: 300px;  
    background-color: yellow;  
    transition: width 2s, background-color 3750ms;  
}  
div:hover {  
    width: 600px;  
    background-color: red;  
}
```



CSS transitions enable you to define a gradual change in property values. To define a transition, you can set the CSS **transition** property and specify the following information:

- The CSS property for which you want to define a transition.
- The duration of the transition.

The following example shows the same styles as before, except that the width of `<div>` elements changes over a period of two seconds when the mouse moves over them. When the mouse moves away, the reverse transition is automatically applied over the same period.

```
div {  
    width: 300px;  
    transition: width 2s;  
}  
div:hover {  
    width: 600px;  
}
```

If you want to apply multiple transitions simultaneously, specify a comma-separated list of CSS property names and durations for the **transition** property. The following example defines transitions for various CSS properties for **<div>** elements. When the user hovers over a **<div>**, the **width**, **height**, and **font-size** properties transition to their new values in two seconds, and the **background-color** property transitions to its new value in 3.75 seconds. When the user moves the mouse away from a **<div>**, the **width**, **height**, and **font-size** properties transition to their original values in two seconds, and the **background-color** property transitions to its original value in 3.7 seconds:

```
div {
    width: 400px;
    height: 60px;
    background-color: yellow;
    transition: width 2s, height 2s, font-size 2s, background-color 3750ms;
}
div:hover {
    width: 600px;
    height: 80px;
    font-size: large;
    background-color: red
}
```

 **Additional Reading:** For detailed information about CSS transitions, visit <http://go.microsoft.com/fwlink/?LinkId=267751>.

Configuring Transitions

CSS3 defines five properties that you can use to configure a transition:

- **transition-property**: The name of the CSS property to which the transition is applied.
- **transition-duration**: The length of time that the transition takes. The default value is zero seconds, which means the transition happens all at once. This has the same effect as not applying a transition.
- **transition-timing-function**: Specifies how the speed of the transition varies during its execution. Possible values include "**linear**", "**ease**", "**ease-in**", "**ease-out**", and "**ease-in-out**". The default value is "**ease**".
- **transition-delay**: The length of time that must elapse before the transition starts. The default value is zero seconds, which means the transition starts immediately.
- **transition**: Shorthand property for the other four transition properties, in the order **transition-property** **transition-duration** **transition-timing-function** **transition-delay**. If any of these values are not specified, the default values described above apply.

- You can configure transitions by using separate properties:

transition-property	Target property of the transition
transition-duration	Duration of the transition
transition-timing-function	Defines how the transition speed varies
transition-delay	Delay before the transition starts
transition	Shorthand notation for all properties

```
div {
    width: 400px;
    height: 60px;
    transition-property: width, height;
    transition-duration: 2s, 2s;
    transition-timing-function: ease-in;
    transition-delay: 1s;
}
```

 **Additional Reading:** For further information about CSS transitions properties, see <http://go.microsoft.com/fwlink/?LinkId=267751>.

If you want to apply multiple transitions simultaneously, specify a comma-separated list of values for each transition property. For example, if you want to animate the width, height, and font size, specify **transition-property: width, height, font-size**. If you want the width and height transitions to last two seconds, and the font size transition to three seconds, specify **transition-duration: 2s, 2s, 3s**. If you specify fewer values than the number of properties being transitioned, the final value is repeated for the remaining properties. For example, if you specify **transition-property: width, height, font-size** and you specify **transition-duration: 2s, 3s**, then the width transition will take two seconds and the height and font size transition will both take three seconds.

The following example defines transitions for various CSS properties for **<div>** elements. The example animates the **width**, **height**, **font-size**, and **background-color** CSS properties for a **<div>** when the user hovers over it. The **width**, **height**, and **font-size** transitions take two seconds to complete, and the **background-color** property takes 3.75 seconds to complete. All of the transitions use an **ease-in** timing function, and there is a delay of one second before the transitions commence.

```
div {  
    width: 400px;  
    height: 60px;  
    background-color: yellow;  
    transition-property: width, height, font-size, background-color;  
    transition-duration: 2s, 2s, 2s, 3750ms;  
    transition-timing-function: ease-in;  
    transition-delay: 1s;  
}  
div:hover {  
    width: 600px;  
    height: 80px;  
    font-size: large;  
    background-color: red  
}
```

Detecting the End of a Transition

When a transition has finished, a **transitionend** event is fired on the element with the properties that have changed. This event is useful if you need to implement functionality that runs only when a transition has completed, such as implementing a series of animations. If you apply multiple transitions to the same element, a separate **transitionend** event is fired for each transition when it ends.

The **event argument for the transitionend** event handler function has two properties:

- **propertyName**: The name of the CSS property for which the transition has completed, such as **width**, **height**, or **font-size**.
- **elapsedTime**: The elapsed time of the transition in seconds, excluding any delay that occurred before the transition started.

```
anElement.addEventListener("transitionend", onTransitionend, true);  
  
function onTransitionend(e) {  
    var nameOfProperty = e.propertyName;  
    var elapsedTime = e.elapsedTime;  
    ...  
}
```

The following example shows how to handle the **transitionend** event on all the **<div>** elements in the document (this example assumes that **<div>** elements are styled with transitions as described in the previous topic). The event-handler function for the **transitionend** event adds a message to a **<select>** drop-down list, to indicate the name of the property whose transition has completed and the elapsed time of the transition.

```
<script>
    function onLoad() {
        var divElements = document.querySelectorAll("div");
        for (var i = 0; i < divElements.length; i++) {
            divElements[i].addEventListener("transitionend", onTransitionend, true);
        }
    }
    var messagesElement = document.querySelector("messages");
    function onTransitionend(e) {
        messages.add(new Option(e.propertyName + ", " + e.elapsedTime));
    }
</script>
...
<body onload="onLoad()">
    <div>Text appearing in a div</div>
    <div>More text appearing in a div</div>
    <select id="messages"></select>
</body>
```

The following image shows the typical messages generated by the code in the previous example:

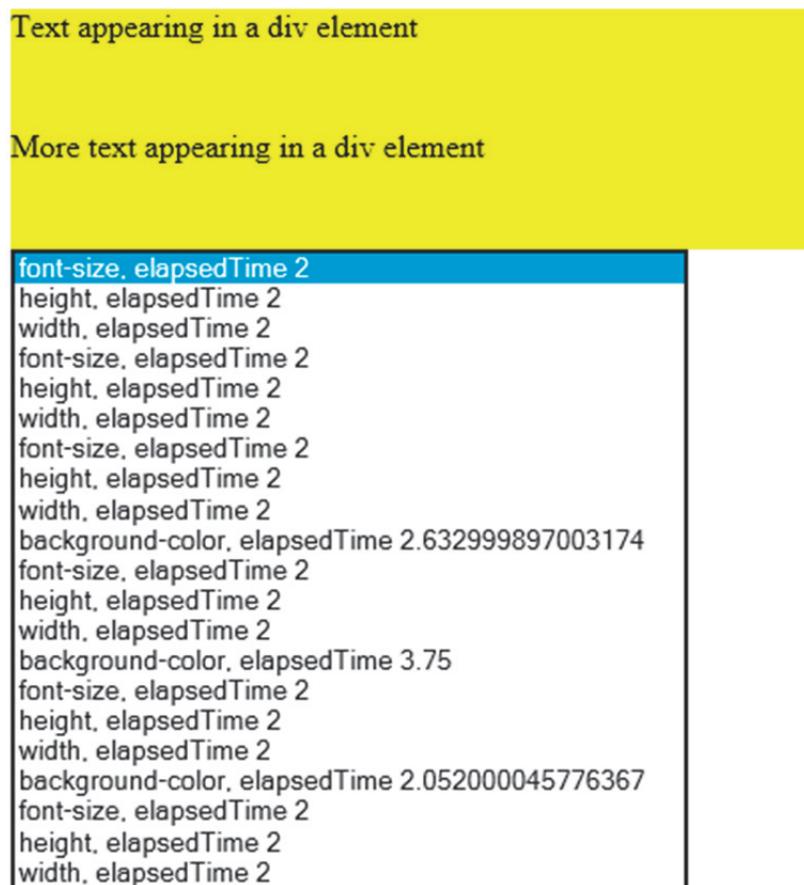


FIGURE 12.1:MESSAGES GENERATED BY HANDLING THE TRANSITIONEND EVENT

Demonstration: Using CSS Transitions

In this demonstration, you will see how to apply CSS transitions to HTML elements. You will also see how to handle the **transitionend** event to detect when a transition has ended.

Demonstration Steps

Apply CSS Transitions to HTML Elements

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod12\Democode\Transitions.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. Hover over the upper rectangle on the web page. Verify that the following transitions are applied simultaneously to the rectangle:
 - o The width, height, and font size increase over a period of two seconds.
 - o The background color transitions to red over a period of 3.75 seconds.
6. Move the mouse away from the upper rectangle. Verify that the rectangle reverts to its original appearance, over the same period of time.
7. Repeat the previous two steps for the second rectangle. Verify that the same animations apply.
8. Right-click the page in Internet Explorer and then click **View source**. Note that:
 - o The first **div** rule defines default CSS properties for all **<div>** elements.
 - o The **div.simple** rule defines a **transition** property that applies to transitions on the **width**, **height**, **font-size**, and **background-color** CSS properties.
 - o The **div.complex** rule defines similar transitions, but it uses separate **transition-property**, **transition-duration**, **transition-timing-function**, and **transition-delay** properties.
 - o The **div:hover** rule defines the final values for the **width**, **height**, **font-size**, and **background-color** CSS properties when the user hovers over a **<div>**.

Handle the **transitionend** Event

1. In Internet Explorer, expand the drop-down list box. Verify that it displays messages for all the transitions that have ended. The list includes events for the original transitions when you hover over a rectangle, as well as events for the reverse transitions when you move the mouse away from a rectangle.
2. Switch to the source window.
3. In the JavaScript code, note that:
 - o The **onLoad()** function sets up event-handlers that call the **onTransitionend()** function when the **transitionend** event is raised on all the **<div>** elements.
 - o The **onTransitionend()** function displays information about the **transitionend** event, by using the **propertyName** and **elapsedTime** properties of the event argument.
4. Close the source window.
5. Close Internet Explorer.

Lesson 2

Transforming Elements

CSS transformations enable you to apply 2D and 3D transformations to elements. You can apply translations, rotations, scaling, and skewing transformations. You can also specify 3D-related transformation properties such as a perspective and back-face visibility.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how CSS implements transformations.
- Apply 2D transformations to elements by using CSS.
- Apply 3D transformations to elements by using CSS.
- Apply transitions to CSS transformations.

Applying Transformations by Using CSS

CSS provides the **transform** property that you can use with a style rule to enable you perform 2D and 3D transformations on an element on a web page. When you set the **transform** property, you can specify one or more of the following transformation functions:

- **translate()**, **translate3d()**, **translateX()**, **translateY()**, and **translateZ()**: Translate the element in 2D or 3D space.
- **scale()**, **scale3d()**, **scaleX()**, **scaleY()**, and **scaleZ()**: Scale the element in 2D or 3D space.
- **rotate()**, **rotate3d()**, **rotateX()**, **rotateY()**, and **rotateZ()**: Rotate the element in 2D or 3D space.
- **skew()**, **skewX()**, and **skewY()**: Skew the element along the X-axis or along the Y-axis.
- **matrix()**, **matrix3d()**: Perform a 2D or 3D transformation by using a matrix to specify the transformation.
- **perspective()**: Define a perspective for an element that has been transformed in 3D.
- **none()**: Cancel any transformations that apply on an element.

Types of transformation supported by CSS:

- Translating



- Rotating



- Scaling



- Skewing



The following code shows a web page that displays a single button. The transform property in the styling rule for the button moves the button to the right and down the page, and applies a 45 degree skew.

Applying a CSS Transformation

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <style>
            button {
                font-family: verdana, arial;
                width: 300px;
                height: 40px;
            }
        </style>
    </head>
    <body>
        <button>Click Me!</button>
    </body>
</html>
```

```
background-color: yellow;  
border-radius: 5px;  
transform: translate(60px, 100px) skew(45deg);  
}  
</style>  
</head>  
<body>  
    <button type="button">Click Me</button>  
</body>  
</html>
```

The button looks like this when it is rendered by a browser:



FIGURE 12.2:THE TRANSFORMED BUTTON



Additional Reading: For detailed information about CSS transformations, visit <http://go.microsoft.com/fwlink/?LinkId=267753>

Applying 2D Transformations

You can use the **transform** property to perform one or more transformations functions on an element. The default origin for a transformation is the center of the target element. To change the origin of transformation, set the **transform-origin** property and specify the following values:

- Horizontal position: A length, a percentage of element width, or **left**, **center**, or **right**.
- Vertical position: A length, a percentage of element height, or **top**, **center**, or **bottom**.

This topic describes how to perform the following 2D transformations:

- Translate
- Scale
- Rotate
- Skew

To perform a 2D translation:
`translate(tx, [ty], [tz])` `translateX(tx)` `translateY(ty)`

To perform a 2D scaling transformation:
`scale(sx, [sy])` `scaleX(sx)`

To perform a 2D rotation:
`rotate(angle)`

To perform a 2D skew transformation:
`skew(anglex, [angley])` `skewX(anglex)` `skewY(angley)`

Example

```
div.rotate1 {  
    transform: rotate(10deg);  
    transform-origin: left top;  
}
```

Translating Elements

A 2D translation enables you to move elements within a page, along the horizontal and vertical axes. To perform a 2D translation, use one of the following functions:

- **translate(tx, ty)**
- **translateX(tx)**
- **translateY(ty)**

Note the following points about the **tx** and **ty** parameters:

- **tx** specifies a horizontal translation to the left (if negative) or the right (if positive). The value can be an absolute length or a percentage of the element width.
- **ty** specifies a vertical translation upwards (if negative) or downwards (if positive). The value can be an absolute length or a percentage of the element height. If you call **translate()** and omit the **ty** parameter, the default value for **ty** is 0.

The following CSS rule translates a **<div>** element with the class **translate1**. The element is moved 60 pixels to the right:

```
div.translate1 {
    transform: translate(60px);
}
```

Scaling Elements

A 2D scaling transformation enables you to resize an element. To perform a 2D scaling transformation, use one of the following functions:

- **scale(sx, sy)**
- **scaleX(sx)**
- **scaleY(sy)**

Note the following points about the **sx** and **sy** parameters:

- **sx** specifies a horizontal scaling factor. A value of 1.0 represents the normal scale.
- **sy** specifies a vertical scaling factor. A value of 1.0 represents the normal scale. If you call **scale()** and omit the **sy** parameter, the default value for **sy** is the same as the value you specified for **sx**.

The following CSS rule scales a **<div>** element with the class **scale1**. The element is 30 percent larger in all dimensions:

```
div.scale1 {
    transform: scale(1.3);
}
```

Rotating Elements

To perform a 2D rotation, use the following function:

- **rotate(angle)**

Note the following points about the **angle** parameter:

- **angle** specifies the angle of rotation anticlockwise (if negative) or clockwise (if positive) about the transformation origin. You can specify the angle in degrees or radians.

The following CSS rule rotates a **<div>** element with the class **rotate1**. The element is rotated 10 degrees clockwise about its top left corner:

```
div.rotate1 {  
    transform: rotate(10deg);  
    transform-origin: left top;  
}
```

Skewing Elements

To perform a 2D skewing transformation, use one of the following functions:

- **skew(anglex, angley)**
- **skewX(anglex)**
- **skewY(angley)**

Note the following points about the **anglex** and **angley** parameters:

- **anglex** specifies a skew angle about the X axis as a clockwise skew (if negative) or an anticlockwise skew (if positive).
- **angley** specifies a skew angle about the Y axis as a clockwise skew (if negative) or an anticlockwise skew (if positive). If you call **skew()** and omit the **angley** parameter, the default value for **angley** is 0.

The following CSS rule skews a `<div>` element with the class **skew1**. The element is skewed by 30 degrees anticlockwise about the X axis:

```
div.skew1 {  
    transform: skew(30deg);  
}
```

 **Note:** In many browsers, you might need to use the vendor-prefixed versions of the **transform** property. For example, in Internet Explorer you can specify **-ms-transform**, although the latest release of Internet Explorer 10 recognizes the **transform** without the prefix.

Demonstration: Performing 2D Transformations

In this demonstration, you will see how to perform 2D transformations by using CSS. You will see how to apply translations, scaling transformations, rotations, and skewing transformations. You will also see the effect of changing the origin of transformation.

Demonstration Steps

Perform 2D Translations

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod12\Democode\2DTranslations.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. Verify that the browser displays a series of rectangles. Each rectangle demonstrates how to perform a 2D translation by using the **translate()**, **translateX()**, or **translateY()** functions. The text message inside each rectangle describes the transformation for that rectangle.
6. Right-click in the browser window, and then click **View source**.

MCT USE ONLY. STUDENT USE PROHIBITED

7. In the source window, scroll down to the bottom of the document. Note that the body of the document has a series of `<div>` elements, and each element has a distinct CSS class.
8. Scroll back up to the top of the code and locate the `<style>` element. The CSS rules implement translations for all the `<div>` elements. The CSS rules set the standard `transform` property, as well as the following vendor-specific properties:
 - `-ms-transform`: Perform a transformation on Internet Explorer 9.
 - `-webkit-transform`: Perform a transformation on Webkit-based browsers such as Chrome and Safari.
 - `-moz-transform`: Perform a transformation on Mozilla browsers.
 - `-o-transform`: Perform a transformation on Opera browsers.
9. Close the source window.

Perform 2D Scaling Transformations

1. In Internet Explorer, open the file **E:\Mod12\Democode\2DScaling.html**.
2. Verify that the browser displays a series of rectangles. Each rectangle demonstrates how to perform a 2D scaling transformation by using the `scale()`, `scaleX()`, or `scaleY()` functions. The text message inside each rectangle describes the transformation for that rectangle.
3. Right-click in the browser window, and then click **View source**.
4. In the source window, scroll down to the bottom of the document. Note that the body of the document has a series of `<div>` elements, and each element has a distinct CSS class.
5. Scroll back up to the top of the code and locate the `<style>` element. The CSS rules implement scaling transformations for all the `<div>` elements. The CSS rules set the standard `transform` property, as well as vendor-specific properties.
6. Close the source window.

Perform 2D Rotations

1. In Internet Explorer, open the file **E:\Mod12\Democode\2DRotations.html**.
2. Verify that the browser displays a series of rectangles. Each rectangle demonstrates how to perform a 2D rotation by using the `rotate()` function. The fourth rectangle also shows how to perform multiple transformations, and how to change the origin of the transformation to the top left of the target element.
3. Right-click in the browser window, and then click **View source**.
4. In the source window, scroll down to the bottom of the document. Note that the body of the document has a series of `<div>` elements, and each element has a distinct CSS class.
5. Scroll back up to the top of the code and locate the `<style>` element. The CSS rules implement rotations for all the `<div>` elements. The CSS rules set the standard `transform` property, as well as vendor-specific properties. The final CSS rule shows how to apply multiple transformations, specifically a translation followed by a rotation. The final CSS rule also shows how to set change the origin of the transformation by setting the `transform-origin` property and its vendor-specific equivalent properties.
6. Close the source window.

Perform 2D Skewing Transformations

1. In Internet Explorer, open the file **E:\Mod12\Democode\2DSkewing.html**.
2. Verify that the browser displays a series of rectangles. Each rectangle demonstrates how to perform a 2D skewing operation by using the `skew()`, `skewX()`, or `skewY()` functions.

3. Right-click in the browser window, and then click **View source**.
4. In the source window, scroll down to the bottom of the document. Note that the body of the document has a series of **<div>** elements, and each element has a distinct CSS class.
5. Scroll back up to the top of the code and locate the **<style>** element. The CSS rules implement skewing transformations for all the **<div>** elements.
6. Close the source window.
7. Close Internet Explorer.

Applying 3D Transformations

CSS3 provides a set of functions that also enable you to perform transformations in 3D space. These functions are similar to their 2D counterparts, but generally take an additional parameter specifying the Z dimension (or depth). This topic describes how to perform the following 3D transformations:

- Translation
- Scaling
- Rotation

Translating Elements

To perform a 3D translation, use one of the following functions:

- **translate3d(tx, ty, tz)**
- **translateZ(tz)**

The **tz** parameter specifies the translation in the Z axis, which is perpendicular to the plane of the screen. You must specify an absolute value, rather than a percentage. In the **translate3d()** function, the **ty** and **tz** parameters are optional; they both default to 0.

Scaling Elements

To perform a 3D scaling transformation, use one of the following functions:

- **scale3d(sx, sy, sz)**
- **scaleZ(sz)**

The **sz** parameter specifies the scaling factor in the Z axis. In the **scale3d()** function, the **sy** and **sz** parameters are optional; they both default to 0.

Rotating Elements

To perform a 3D rotation, use the following function:

- **rotate3d(xnum, ynum, znum, angle)**

The **angle** parameter specifies the angle of rotation anticlockwise (if negative) or clockwise (if positive) about the direction vector specified by the first three parameters. For example:

- **rotate3d(1, 0, 0, 60deg)** performs a rotation about the X axis.
- **rotate3d(0, 1, 0, 60deg)** performs a rotation about the Y axis.
- **rotate3d(0, 0, 1, 60deg)** performs a rotation about the Z axis.

To perform a 3D translation:
translate3d(tx, [ty], [tz]) **translateZ(tz)**

To perform a 3D scaling transformation:
scale3d(sx, [sy], [sz]) **scaleZ(sz)**

To perform a 3D rotation:
rotate3d(xnum, ynum, znum, angle) **rotateZ(angle)**

You must also specify a perspective:

- Use the **perspective()** function, or
- Specify **perspective** and **perspective-origin** properties



Setting the Perspective and the Perspective Origin

When you specify a 3D transformation, you must also define a perspective. The perspective sets the viewer's position relative to the object being transformed and defines how content gets smaller as the Z value varies. If you do not specify a perspective, all points in the Z axis are flattened into the same 2D plane without any perception of depth. There are two ways to specify perspective:

- Call the **perspective()** function every time you use the transform property.
- Set the perspective CSS property on the parent element, to apply the perspective to each of the child elements.

You can also set the **perspective-origin** CSS property to shift the viewpoint away from the center of the element.

The following example shows how to define perspective for a 3D transformation. The example defines **<div>** elements named **child1** and **child2** in a parent element named **parent**. Note the following CSS rules:

- The **#parent** CSS rule defines a **perspective** of 300px, which means the disappearing point will be 300 pixels to the right of the perspective origin. The **perspective-origin** property moves the perspective origin 100 pixels to the left and 50 pixels upwards.
- The **#child1** CSS rule rotates a **<div>** element by 30 degrees clockwise about the Y axis.
- The **#child2** CSS rule rotates a **<div>** element by 30 degrees clockwise about the Y axis, and then translates the shape by 250 pixels in the X direction.

```
<style>
    #parent {
        perspective: 300px;
        perspective-origin: -100px -50px;
    }
    #child1 {
        background-color: orange;
        position: absolute;
        transform-origin: 0px 0px;
        transform: rotateY(30deg);
    }
    #child2 {
        background-color: orange;
        position: absolute;
        transform-origin: 0px 0px;
        transform: rotateY(30deg) translate(250px);
    }
</style>
...
<div id="parent">
    <div id="child1">This is child1</div>
    <div id="child2">This is child2</div>
</div>
```

The following image shows how the browser renders two **<div>** elements.



FIGURE 12.3: RENDERING ELEMENTS WITH A 3D PERSPECTIVE

Defining Transitions for Transformations

You can define a transition for transformations, so that the transformation is applied gradually over a specified time period. To define a transition for a transformation, follow these steps:

- Define a 2D or 3D transformation for an element by using the **transform** CSS property.
- Set the **transition** property so that it defines a transition for the **transform** CSS property.

The following example defines a rotation for a **<div>** element named **container**. When the user hovers over the element, the element will rotate by 90 degrees clockwise. The example also defines a transition for the transformation, so that the rotation will take five seconds to complete:

```
<style>
    #container {
        transition: transform 5s;
    }
    #container:hover {
        transform: rotate(90deg);
    }
</style>
...
<div id="container">
    ...
</div>
```

You can define a transition for a transformation:

- Set the **transform** property on an element
- Set the **transition** property so that it defines a transition for the **transform** property

```
<style>
    #container {
        transition: transform 5s;
    }
    #container:hover {
        transform: rotate(90deg);
    }
</style>
...
<div id="container">
    ...
</div>
```

Demonstration: Performing 3D Transformations

In this demonstration, you will see how to perform 3D transformations. You will also see how to define a transition for a transformation.

Demonstration Steps

Perform 3D Transformations that Include Transitions

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod12\Democode\3DTransformations.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. Verify that the browser displays a cube. There are six faces to the cube, each of which displays text and has a different background color. The front face is partially transparent, so that it does not completely obscure the other faces.
6. Hover over the cube. Verify that it rotates by 90 degrees over a period of five seconds. Then move the mouse off the cube, and verify that the cube rotates smoothly back to its original position.
7. Right-click in the browser window, and then click **View source**.
8. In the source window, scroll down to the bottom of the document. Note that the body of the document has six **<div>** elements that represent the six faces of the cube. These **<div>** elements are contained in a parent **<div>** element named **container**.
9. Scroll back up to the top of the code and locate the **<style>** element. Note the following CSS rules:
 - **#container**: Specifies a perspective for all the child elements of the **container** element, and a transition of five seconds for transformations.
 - **#container:hover**: Specifies a rotation of 90 degrees when the user hovers over the **container** element.
 - **#rightFace**: Transforms the **rightFace** element in 3D space, so that it appears on the right side of the cube.
 - **#leftFace**: Transforms the **leftFace** element in 3D space, so that it appears on the left side of the cube.
 - **#topFace**: Transforms the **topFace** element in 3D space, so that it appears on the top of the cube.
 - **#bottomFace**: Transforms the **bottomFace** element in 3D space, so that it appears on the bottom of the cube.
 - **#backFace**: Transforms the **backFace** element in 3D space, so that it appears at the back of the cube.
 - **#frontFace**: Transforms the **frontFace** element in 3D space, so that it appears at the front of the cube. The background color is partially transparent.
10. Close the source window.
11. Close Internet Explorer.

Lesson 3

Applying CSS Keyframe Animations

CSS keyframe animations enable you to define an animation as a series of steps. For each step, or keyframe, you can define a set of property values that you want to apply at that stage in the animation. When the browser performs the animation, it interpolates property values between the keyframes, to give you the effect of a smooth transition between values.

Lesson Objectives

After completing this lesson, you will be able to:

- Define keyframes for CSS animations.
- Configure keyframe animation properties.
- Start keyframe animations programmatically.
- Handle the events that occur during a keyframe animation.

Defining a Keyframe Animation

CSS enables you to define keyframe animations for element property values of HTML elements. You can define a series of rule-sets that specify the property values at distinct stages of the animation. Keyframe animations enable you to implement sophisticated user interface effects that would previously have required video or third-party plugins.

To use keyframe animations in a web page, the first step is to define a **@keyframes** CSS rule. Within this rule, you define a series of rule-sets that apply at different points during the animation. These points are specified as percentages of the elapsed time of the duration of the animation. The first rule-set is designated by the value **0%** or **from**. The final rule-set is designated by the value **100%** or **to**. You can define as many rule-sets as you want, and the ordering of the rule-sets is irrelevant.

The following example defines a keyframe animation named **ballmovement**. The keyframe animation has four rule-sets, which describe the color and location of a ball on a pool table during an animation; the ball also changes color as it moves:

```
@keyframes ballmovement {  
    0% {  
        left: 0px;  
        top: 0px;  
        background-color: yellow;  
    }  
    33% {  
        left: 100px;  
        top: 160px;  
    }  
    66% {  
        left: 200px;  
        top: 0px;  
    }  
}
```

- Define property values that apply at distinct points during the animation

```
@keyframes name_of_animation {  
    0% /* or from */  
    ... properties to at the start of the animation ...  
}  
  
50%  
... properties to apply after 50% of the animation ...  
  
100% /* or to */  
... properties to apply at the end of the animation ...  
}
```

```

100% {
    left: 300px;
    top: 160px;
    background-color: purple;
}

```

 **Note:** Most browsers currently do not support the standard **@keyframes** CSS rule name. Instead, you must use vendor-specific prefixes such as **@-ms-keyframes** for Microsoft Internet Explorer 10.

You perform the animation by specifying the **animation-name** property in a style rule. This property expects the name of the keyframe animation, like this:

```

#ball.animate {
    animation-name: ballmovement;
}

```

In this example, any elements named **ball** that have the class **animate** will be animated.

 **Note:** You can write JavaScript code to apply the **animate** class to the ball when you want the animation to play. You can also write JavaScript code to remove the **animate** class from the ball when you want the animation to stop.

 **Additional Reading:** For detailed information about CSS animations, visit <http://go.microsoft.com/fwlink/?LinkId=267754>.

Configuring Keyframe Animation Properties

After you have defined a keyframe animation, the next step is to configure the animation properties. You must specify when the animation is applied and how long it lasts. Optionally, you can also specify a delay before the animation starts, a repeat count, and whether the animation should reverse itself upon completion.

To configure a keyframe animation, define a CSS rule that applies the keyframe animation to an element in the document. Inside the CSS rule, you can set the following properties to configure the keyframe animation:

- **animation-name:** The name of the animation that you want to apply to the target element.
- **animation-duration:** The duration of the animation.
- **animation-delay:** An optional delay that occurs before the animation starts.
- **animation-timing-function:** Optional information about how the animation progresses over one cycle. Possible values include "linear", "ease", "ease-in", "ease-out", and "ease-in-out". The default value is "ease". You can define this property for the whole animation, or just for specific steps in the **@keyframe** animation definition.
- **animation-iteration-count:** Optional iteration count. The default value is 1.

Apply the animation to a target element in a CSS

```

CSS_rule_to_apply_animation{
    animation-name: name_of_animation,
    animation-duration: duration_of_animation,
    ...
}

```

Keyframe animation properties:

- **animation-name**
- **animation-duration**
- **animation-delay**
- **animation-timing-function**
- **animation-iteration-count**
- **animation-direction**

- **animation-direction:** Optional information about the direction the animation should play. The default value is **normal**, which means the animation always plays in forward direction from start to end. The other possible value is **alternate**, which means the animation reverses itself each time it plays if the iteration count is more than 1.

The following example configures keyframe animation for an element named **ball** that has a CSS class named **animate**. The example applies the **ballmovement** keyframe animation described in the previous topic. The animation will last 10 seconds and will start after an initial delay of three seconds. The animation will use a linear timing function, which causes the browser to interpolate values linearly between keyframe steps. The animation will play twice; the first cycle will be in the forward direction, and the second cycle will be in the reverse direction:

```
#ball.animate {  
    animation-name: ballmovement;  
    animation-duration: 10s;  
    animation-delay: 3s;  
    animation-timing-function: linear;  
    animation-iteration-count: 2;  
    animation-direction: alternate;  
}
```

 **Note:** As with keyframes rules, most browsers currently do not support the standard property names for keyframe animations. Instead, you must use vendor-specific prefixes such as **-ms-animation-name** and **-ms-animation-duration** for Microsoft Internet Explorer 10.

Starting a Keyframe Animation Programmatically

In most scenarios, you will start keyframe animations programmatically in response to an event. For example, if you want the animation to start as soon as the web page has loaded, you can use the **load** event of the **<body>** element to trigger the animation. If you want to enable users to start the animation themselves, provide a button and handle the **click** event.

The following example handles the **click** event on a button. When the user clicks the button, the **startAnimation()** function adds the **animate** class to the **ball** element. This causes the **#ball.animate** CSS rule to apply, which triggers the **ballmovement** keyframe animation:

- Common technique:

- Add a CSS class to the target element
- Trigger the keyframe animation based on the CSS class

```
<style>  
    @keyframes ballmovement {  
        ...  
    }  
  
    #ball.animate {  
        animation-name: ballmovement;  
        ...  
    }  
</style>  
...  
<script>  
    function startAnimation() {  
        var ball = document.getElementById("ball");  
        ball.classList.add("animate");  
    }  
</script>
```

```
<style>  
    ...  
    @keyframes ballmovement {  
        ...  
    }  
    #ball.animate {  
        animation-name: ballmovement;  
        ...  
    }  
</style>  
...  
<script>  
    function startAnimation() {  
        var ball = document.getElementById("ball");  
    }  
</script>
```

```

        ball.classList.add("animate");
    }
</script>
...
<body>
...
<div id="ball"></div>
<button id="button" onclick="startAnimation()">Start Animation</button>
...
</body>

```

Handling Keyframe Events

Three events occur when a keyframe animation runs. These events are:

- **animationstart**: Indicates that a keyframe animation has started.
- **animationiteration**: Indicates that an iteration of the keyframe animation has completed.
- **animationend**: Indicates that a keyframe animation has ended.

You can use these events to sequence animations.

For example, you can use the **animationend** event to trigger another animation on a different element.

If you handle these events, the event-handler function will receive an event argument that has the following properties:

- **animationName**: The name of the animation, such as **ballmovement** in the previous examples.
- **elapsedTime**: The total elapsed time of the animation so far, excluding any delay before the animation started.

The following example shows how to handle the keyframe animation events for an element named **ball**:

```

<script>
...
var ball = document.getElementById("ball");
// Handle the event that occurs when the ball animation starts.
ball.addEventListener("MSAnimationStart", function (e) { ... }, false);
// Handle the event that occurs for each iteration.
ball.addEventListener("MSAnimationIteration", function (e) { ... }, false);
// Handle the event that occurs when the ball animation ends.
ball.addEventListener("MSAnimationEnd", function (e) { ... }, false);
...
</script>

```

Keyframe animations raise the following events:

- **animationstart**
- **animationiteration**
- **animationend**

The event-handler function receives an event argument with the following properties:

- **animationName**
- **elapsedTime**



Note: Internet Explorer 10 uses the names **MSAnimationStart**, **MSAnimationIteration**, and **MSAnimationEnd** for these events.

Demonstration: Implementing KeyFrame Animations

In this demonstration, you will see how to define a keyframe animation and apply the animation to an HTML element. You will also see how to control the animation programmatically, and how to handle keyframe animation events.

Demonstration Steps

Define and Run a Keyframe Animation

1. On the Windows 8 **Start** screen, click the **Desktop** tile.
2. On the Windows taskbar, click **Internet Explorer**.
3. In Internet Explorer, open the file **E:\Mod12\Democode\KeyframeAnimations.html**.
4. If a message box appears asking if you want to allow blocked content, click the **Allow blocked content** button.
5. Verify that a green rectangle appears on the page, with a small white circle in the top left corner. The green rectangle represents a pool table and the white circle represents a ball. There is also a button that enables you to start the animation.
6. Click **Start Animation**.
 - After three seconds, the ball starts moving diagonally on the pool table. The color of the pool table also changes to blue, and a message appears at the bottom of the page to indicate the start time of the animation.
 - As the animation proceeds, the ball appears to bounce off the sides of the pool table and the color of the ball varies from white to yellow, then to orange, then to red, and then finally to purple.
 - When the ball reaches the bottom right corner of the pool table, a message appears to indicate that the first iteration of the animation has completed. The next iteration of the animation begins; this iteration plays the animation in reverse, so that the ball ends up in its original position and with its original color.
 - At the end of the animation, the pool table reverts to green and messages appear at the bottom of the page to indicate the elapsed time of the animation.
7. Right-click in the browser window, and then click **View source**.
8. In the source window, scroll down to the bottom of the document. Note that the body of the document has a **<div>** element named **pooltable** that represents the pool table, and a nested **<div>** named **ball** that represents the ball on the pool table. There is also a **<button>** element to start the animation, and a **<div>** named **messageLabel** where messages are displayed.
9. Scroll back up to the top of the file and locate the **<style>** element. Note the following CSS rules:
 - **#pooltable**: Specifies the initial appearance of the pool table.
 - **#pooltable.animate**: Specifies a different color for the pool table during an animation. There is JavaScript code elsewhere in the document that programmatically adds the **animate** class to the **pooltable** element when an animation starts, to cause the pool table to turn blue during an animation.
 - **#ball**: Specifies the initial appearance of the ball.
 - **@-ms-keyframes ballmovement**: Defines a keyframe animation named **ballmovement**. The first rule-set specifies the original color and location of the ball. Each subsequent rule-set simulates the

ball hitting one of the sides of the pool table, and causes the ball to change color during each part of its journey. The final rule-set specifies the final color and location of the ball.

- **#ball.animate:** Applies the **ballmovement** keyframe animation to a ball when the ball has the **animate** class. There is JavaScript code elsewhere in the document that programmatically adds the **animate** class to the **ball** element when the user clicks the **Start Animation** button, to trigger the animation.

10. Locate the `<script>` element. Note that:

- The **init()** function, invoked as soon as the page has loaded, establishes event-handler functions for the **MSAnimationStart**, **MSAnimationIteration**, and **MSAnimationEnd** events on the **ball** element:
- The **MSAnimationStart** event-handler function is called when an animation starts on the **ball** element. The function adds the **animate** class to the pool table, so that the pool table becomes blue. The function also displays a message to indicate the time when the animation started.
- The **MSAnimationIteration** event-handler function is called when each iteration of the animation has completed. The function displays a message to indicate the elapsed time of the animation.
- The **MSAnimationEnd** event-handler function is called when an animation ends on the **ball** element. The function enables the button, removes the **animate** class from the ball and the pool table, and displays a message to indicate the elapsed time of the animation.
- The **startAnimation()** function is Invoked when the user clicks the **Start Animation** button. The function disables the button, and adds the **animate** class to the ball to trigger the animation. The animation starts three seconds later, due to the **-ms-animation-delay: 3s;** property in the **#ball.animate** CSS rule.

11. Close the source window.

12. Close Internet Explorer.

Demonstration: Animating the User Interface

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Animating the User Interface

Scenario

You have been asked to make the Contoso Conference web site more engaging by adding some animation.

You decide to animate the **Register** link, displayed on the **Home** page. When the user moves the mouse over this link, you will make it rotate slightly to highlight it.

The **Feedback** page contains a form that enables an attendee to provide their assessment of the conference and to make additional comments. This information is submitted by the **Feedback** page to a data-collection service. You have decided that you can make this page more interesting by animating the stars as the user moves the mouse over them, and by making the feedback form fly away when the user submits their feedback.

Objectives

After completing this lab, you will be able to:

1. Animate HTML elements by using CSS transitions.
 2. Animate HTML elements using CSS keyframes, and trigger animations and handle animation events by using JavaScript code.
- Estimated Time: 60 minutes
 - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
 - User Name: **Student**
 - Password: **Pa\$\$w0rd**

Exercise 1: Applying CSS Transitions

Scenario

In this exercise, you will use CSS transitions to animate parts of the conference website.

First you will animate the star icons on the **Feedback** page so they react when moving the cursor over them. Next, you will rotate the **Register** link on the **Home** page as the mouse traverses it. Finally, you will run the application, view the **Feedback** and **Home** pages, and verify that the elements are animated correctly.

The main tasks for this exercise are as follows:

1. Review the Feedback page.
2. Animate the stars on the Feedback form.
3. Animate the Register link on the Home page.
4. Test the application.

► Task 1: Review the Feedback page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution from the **E:\Mod12\Labfiles\Starter\Exercise 1** folder.

4. In the **ContosoConf** project, examine the contents of the **feedback.htm** file. This page contains an HTML form that collects conference attendee feedback:

```
<form method="post" action="/send-feedback">
    <div class="field feedback-question">
        <label>How would you rate the speaker's knowledge of the topic?</label>
        <input name="question" type="range" min="1" max="5"/>
    </div>
    <div class="field feedback-question">
        <label>How well could you hear the speaker?</label>
        <input name="question" type="range" min="1" max="5"/>
    </div>
    <div class="field feedback-question">
        <label>How useful did you find the information in this talk?</label>
        <input name="question" type="range" min="1" max="5"/>
    </div>
    <div class="field feedback-question">
        <label>How would you rate the overall session?</label>
        <input name="question" type="range" min="1" max="5"/>
    </div>
    <div class="field comments">
        <label>Any additional comments?</label>
        <textarea name="comments" cols="30" rows="5"></textarea>
    </div>
    <div class="field actions">
        <button type="submit">Send Feedback</button>
    </div>
</form>
```

This page also references the **feedback.css** style sheet in the **/styles/pages** folder, and the **feedback.js** JavaScript file in the **/scripts/pages** folder:

```
<link href="/styles/pages/feedback.css" rel="stylesheet" type="text/css" />
<script src="/scripts/pages/feedback.js" type="text/javascript"></script>
```

5. Run the application and view the **Feedback** page.
6. Note that the input elements for the form have been converted into star icons. This feature is implemented by the JavaScript code in the **feedback.js** and **StarRatingView.js** files.
7. Close Internet Explorer.

► Task 2: Animate the stars on the Feedback form

1. In the **styles\pages** folder, open the **feedback.css** style sheet.
2. Add a CSS property to the **.star:hover, .star.hover** rule that transforms the stars to be 1.3 times larger when the mouse moves over them.
 - o Add a **scale** transform to the rule.
3. Specify that the transformation should take 0.5 seconds to perform.
 - o Add a **transition** to the rule.
4. When the mouse is no longer hovering over a star, the star should not suddenly jump back to its original size, so add a transition to the **.star** CSS rule that reverts the star back to its original size over 0.5 seconds.
5. When a star has the **.selected** CSS class, it should remain scaled. Add a transform to the **.star.selected** rule that scales the star by a factor of 1.3.

► **Task 3: Animate the Register link on the Home page**

1. The **index.htm** page has large **Register** link in the header. This link is styled by using the rules in the **header.css** style sheet. In this style sheet, add CSS properties to animate the **Register** link when the cursor hovers over it, as follows:
 - Rotate the link to 16 degrees and scale it by a factor of 1.1 in both dimensions.
 - Transition the transformations over a period of one second.
 - When the mouse moves away, return the **Register** link to its original state over a period of one second.

► **Task 4: Test the application**

1. Run the application and view the **Feedback** page.
2. Move the mouse over the stars. Verify that their size changes and they remain larger when selected.
3. View the **Home** page.
4. Move the mouse over the **Register Free** link in the header and verify that it rotates and enlarges.
5. Close Internet Explorer.

Results: After completing this exercise, the **Register** button will rotate and the feedback stars will animate when the mouse moves over them.

Exercise 2: Applying Keyframe Animations

Scenario

In this exercise, you will create a keyframe animation to animate the form on the **Feedback** page. The form will fly off the page when the user submits the form.

First, you will define a keyframe animation by using CSS. Next, you will use the keyframe animation in a CSS rule. Then you will add this CSS rule to the **Feedback** form to trigger the animation when the form is submitted. You will handle an animation event to show a message when the animation is complete.

Finally, you will run the application, view the **Feedback** page, and verify that the form animates correctly when the user submits it.

The main tasks for this exercise are as follows:

1. Define a keyframe animation.
2. Trigger the animation.
3. Test the application.

► **Task 1: Define a keyframe animation**

1. In Visual Studio, open the **ContosoConf.sln** solution in the **E:\Mod12\Labfiles\Starter\Exercise 2** folder.
2. In the **feedback.css** style sheet in the **styles\pages** folder, find the following comment:

```
/* TODO: Add key frame animation named "send"
   At 0% scale(1)
   At 50% scale(.8)
   At 100% translate(0, -1000px)
*/
```

3. After this comment, define a keyframe animation named **send**. The animation should perform the following operations:
 - Reduce the size of the targeted element, using a scale transform, to be 0.8 times the original size.
 - Slide the target element up off the page. A translation of 1000px up is enough to move the feedback form out of view.
1. The **sending** CSS class will be added to the feedback form when it is submitted (you will write the JavaScript code to do this in the next task). In the **feedback.css** style sheet, add CSS properties to the **.sending** rule that apply the **send** animation (where indicated by the comment in this rule):
 - Set the animation duration to be one second.
 - Ensure that the animation runs only once, and that it maintains the properties set by the last keyframe after completion.

► Task 2: Trigger the animation

1. Open the **feedback.js** file in the **scripts\pages** folder.
2. A submit event listener has already been added to the feedback form. This event listener calls the **formSubmitting** function. In the **formSubmitting** function, after the comment // TODO: Trigger the animation by adding the "sending" CSS class to the form, add the CSS class **sending** to the **form** element.



Note: Adding this class to the **form** element triggers the animation that you defined in the previous task.

3. After the feedback form has finished animating, the **feedback-sent** <div> in the **Feedback** form should be displayed. This <div> displays the message **Thanks for the feedback**. The **animationEnded** function displays this <div>. In this function, after the comment // TODO: Add listener for the animationend event, add an **animationend** event listener to the **form**, which calls **animationEnded**.

► Task 3: Test the application

1. Run the application and view the **Feedback** page.
2. Click **Send Feedback**.
3. Verify that the form shrinks and flies off the top of the page.
4. Close Internet Explorer.

Results: After completing this exercise, submitting the conference feedback form will trigger an animation that makes the form fly off the page.

Module Review and Takeaways

In this module, you have learned how to create animated content by using CSS3 transitions, transformations, and keyframe animations.

CSS transitions enable you to define a time span for property changes. The browser interpolates the property from its initial value to its final value over the specified time span, to give the user the effect of a smooth transition from the original property value to the new property value.

CSS transformations enable you to translate, scale, rotate, or skew an element. CSS supports 2D transformations in the X and Y directions, and 3D transformations in the X, Y, and Z directions.

CSS keyframe animations enable you to specify a set of property values to apply to a target element at distinct steps in the animation. You express the steps as percentages of the elapsed time for the animation. You can start animations programmatically and handle events that occur as the animation progresses.

Review Question(s)

Question: What happens if you do not set the **transition-duration** property of a CSS transition?

Test Your Knowledge

Question	
Which of the following operations can you NOT perform by using a CSS transformation?	
Select the correct answer.	
	Rotate
	Translate
	Animate
	Scale
	Skew

Question: What are the steps for implementing a keyframe animation?

Module 13

Implementing Real-time Communication by Using Web Sockets

Contents:

Module Overview	13-1
Lesson 1: Introduction to Web Sockets	13-2
Lesson 2: Using the WebSocket API	13-4
Lab: Performing Real-time Communication by Using Web Sockets	13-10
Module Review and Takeaways	13-17

Module Overview

Web pages request data on demand from a web server by submitting HTTP requests. This model is ideal for building interactive applications, where the functionality is driven by the actions of a user. However, in an application that needs to display constantly changing information, this mechanism is less suitable. For example, a financial stocks page is worthless if it shows prices that are even a few minutes old, and you cannot expect a user to constantly refresh the page displayed in the browser. This is where web sockets are useful. The Web Sockets API provides a mechanism for implementing real-time, two-way communication between web server and browser.

This module introduces web sockets, describes how they work, and explains how to create a web socket connection that can be used to transmit data in real time between a web page and a web server.

Objectives

After completing this module, you will be able to:

- Describe how using web sockets helps to enable real-time communications between a web page and a web server.
- Use the Web Sockets API to connect to a web server from a web page, and exchange messages between the web page and the web server.

Lesson 1

Introduction to Web Sockets

Sockets are a long-established communication mechanism for establishing a bi-directional network connection between two applications. The socket protocol enables a server to listen for connection requests at an advertised or well-known address. When a client connects to this address, a negotiation occurs and the two parties establish a private communications channel over a separate connection, leaving the server free to listen for further requests from other clients. The client application and server exchange messages over their private channel, and when the conversation completes, either party can close the connection.

The Web Sockets API enables web pages and web servers to exploit the socket protocol. In this lesson, you will learn why the Web Sockets API is a useful addition to the communications mechanisms available for building web applications, and what happens when a page connects to a web server by using a web socket.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the problems that web sockets are intended to solve.
- Describe how a client application connects to a server and exchanges data by using a web socket.

The Problem of Web-based Real-time Communications

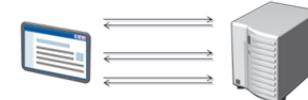
When creating real-time web applications, the need for continuous communication between the web page in a browser and the web server is paramount. As soon as a user views a web page, the data it displays might already be obsolete; stock values may have fallen or risen, or the tickets for a concert may have sold out. A great deal of data is time-dependent. Users must be able to trust the information on the page in front of them without having to refresh it constantly.

There are two established ways to **implement real-time communications**:

- **Continuous polling.** The page connects to the server and sends an AJAX request to the server for new data. The server instantly responds, indicating that the data has not changed since the last request, or sends back the new data. The page then closes the connection. This process is repeated every few seconds.
- **Long polling.** The page connects to the server, setting the connection timeout value to a very long period of time (up to several hours, depending on the application), and then sends a request to the server for new data. The server only replies if it has new data to send. The connection is closed when either the timeout period is reached or new data is sent to the page. The process then starts again. This mechanism has an advantage over the continuous polling approach in that the overhead of opening and closing many short-lived network connections is reduced, but the cost is the need to maintain an open network channel to the server. The server may be able to support only a limited number of concurrent network connections, and a web page may not be able to connect to a server when this limit is reached.

- Common techniques for implementing real-time communications include:

- Continuous polling
- Long polling



- Real-time communication solutions between web page and server have two main issues

- Additional HTTP headers increase message size
- HTTP is not full-duplex

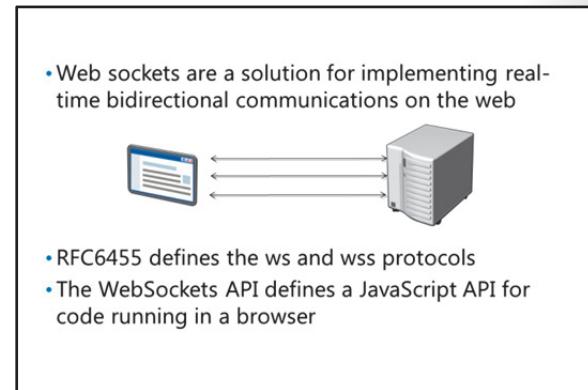
Both of these methods are well understood, but they share a common set of **disadvantages**:

- Requests are sent over HTTP. This means a lot of header information is added to both request and response, which in turn produces unwelcome delays in transmission when operating in real time.
- HTTP can send data only one way at a time. This is known as "half-duplex" transmission. Real-time communication implies that both client and server can send and receive messages at the same time, or "full-duplex" transmission.

What is a Web Socket?

Web sockets provide a simple, lightweight method for enabling full-duplex, real-time communications between a client and a server without relying on HTTP. Browser vendors implement web sockets according to two specification documents:

- **RFC6455**, which defines the transport mechanism for sending and receiving messages with the minimum amount of additional information needed to identify sender and recipient.
- **Web Sockets API**, which is a World Wide Web Consortium (W3C) specification that details a JavaScript API enabling web pages to use web sockets. It is not part of the W3C HTML5 standard, but it is part of the wider family of HTML5 standards defined by the Web Hypertext Application Technology Working Group (WHATWG).



The Web Sockets API is only a draft standard at this time, but most modern browsers support it. Internet Explorer 10 is the first version of Internet Explorer to support web sockets.

How Do Web Sockets Work?

The web sockets protocol defined in **RFC6455** states that there are four steps to exchanging data between a client and a server:

1. The client requests a connection to the server over **HTTP** or **HTTPS**.
2. If the server responds positively, both the client and server switch to the web sockets protocol (known as **WS**) or **WSS** (the secure variant of WS), and a persistent bi-directional socket connection is created between the two.
3. The client and server send and receive messages over the open connection. The format of the data in the messages is entirely up to the client and server; the client just needs to construct messages in a format that the server expects, and vice versa.
4. The client or the server explicitly closes the connection, or a timeout value is reached.

All four steps are carried out transparently by the browser in response to methods implemented by the Web Sockets API.

Additional Reading: You can read RFC6455 at

<http://go.microsoft.com/fwlink/?LinkID=267756> and the latest copy of the WebSockets API at <http://go.microsoft.com/fwlink/?LinkID=267755>.

Lesson 2

Using the WebSocket API

This lesson introduces the WebSocket API. It describes how you can use the WebSocket API to implement the client-side logic for a page that opens and closes connections to a WebSocket server, and uses these connections to exchange messages with the server.

Lesson Objectives

After completing this lesson, you will be able to write JavaScript code in a web page that:

- Connects to and disconnects from a web server by using a web socket.
- Sends messages to a server by using a web socket.
- Receives messages from a server by using a web socket.

Connecting to a Server by Using a Web Socket

The WebSocket API defines the **WebSocket** object for establishing a connection between a client application and a server. The **WebSocket** object provides the methods that a client object uses to connect to a server, and to send and receive messages. The **WebSocket** object also contains a number of properties that maintain information about the state of the current connection.

The **WebSocket** object is also available as the **WebSocket** property of the **window** object. You can detect whether the browser running a web page supports web sockets by using the following code:

```
if (window.WebSocket) {  
    alert("WebSocket is supported");  
} else {  
    alert("WebSocket is not supported");  
}
```

- The **WebSocket** object contains the functionality required to communicate with a server through a web socket
 - Establish a connection by creating a new web socket:

```
var socket =  
    new WebSocket('ws://websocketserver.contoso.com/bookings');
```
 - Check that the socket was opened successfully before using it:

```
socket.onopen = function() {  
    alert("Connection to server now open!");  
};  
...  
};
```
 - Use the **close()** function to terminate the connection:

```
socket.close();
```

Opening a Connection

The start of all communication with a web socket server is the opening handshake over HTTP between the client code running in a web page and the server. The **WebSocket** constructor enables you to create a new connection and to specify the URL of the server to connect to. This URL uses the **ws** or **wss** scheme to indicate that it is a web socket address:

```
var socket = new WebSocket('ws://websocketserver.contoso.com/bookings');
```

The default port for the **ws://** protocol is port 80 (like http). If the server is listening for connection requests on a different port, simply specify the port as part of the URL. For example:

```
var socket = new WebSocket("ws://localhost:55981/bookings");
```

If you need to establish an encrypted connection, you can use the secure web sockets protocol **wss://**. This protocol uses port 443 by default.

```
var socket = new WebSocket('wss://secure.websocketserver.contoso.com/bookings');
```

The initial handshake over HTTP is performed automatically, and if the server accepts the request from the client a new connection will be established by using the web socket transport protocol.

The **WebSocket** API is asynchronous. This is because it can take time to establish a connection, and after a connection has been opened, messages can be received at any time over that connection. After you have created a **WebSocket** object, you should not attempt to use it until it is ready. You can determine the state of the **WebSocket** object by examining the **readyState** property. This property can have the following values:

- **CONNECTING** (0), which indicates that a **WebSocket** object has been created, but a connection is still being made between page and server.
- **OPEN** (1), which indicates that a connection between page and server has been established.
- **CLOSING** (2), which indicates that the closing handshake is in progress.
- **CLOSED** (3), which indicates that the connection between page and server has been closed or could not be established.

The following example loops until a socket is in the open state:

```
while (socket.readyState != 1) {
    ... // wait until the socket is open before continuing
    ...
}
```

A better way to detect when a connection has been opened successfully is by handling the **open** event of the **WebSocket** object. At this point, you can start to send and receive messages over the connection.

You can bind to this event by using the **onopen** callback.

```
socket.onopen = function() {
    // Web Socket Server is connected
    alert("Connection to server now open!");
    //send message etc ...
};
```

 **Note:** You can also choose to use the **addEventListener()** method to bind to the events fired by the **WebSocket** object, like this:

```
function sendMessage() {
    // Create a message and send it to the server
    ...
};

socket.addEventListener("open", sendMessage);
```

If an error occurs while connecting to the server, the **error** event fires (this event also fires if an error occurs while disconnecting from, or sending a message to, the server). The error message is available as the **data** property of the **event** object. You can bind to this event by using the **onerror** callback.

```
socket.onerror = function(event) {
    // An error has occurred
    alert("An error has occurred: " + event.data);
};
```

Closing a Connection

To close the connection to the server, call the **close()** function of the **WebSocket** object. This function takes two optional parameters, **code** and **reason**, which enable you send the server an exit status code (described in RFC6455) and a text-based reason for closing the connection.

```
socket.close();
socket.close(1000, "No Error. All communication finished with.");
```

The **close** event fires when a connection has been closed. The event object has three properties:

- **wasClean**, which is a Boolean value indicating whether the connection was closed cleanly (true) or experienced a problem (false).
- **code**, which is the exit status code (laid out in RFC6455) indicating why the connection was closed.
- **reason**, which is a text string giving a reason why the connection was closed.

The following code shows an example that detects whether a connection was closed successfully or not:

```
socket.onclose = function(event) {
    // Connection has been closed
    if (event.wasClean) {
        alert("Connection closed OK");
    } else {
        alert("Connection closed with issues. Code " + event.code);
    }
};
```

Sending Messages to a Web Socket

After you have established a connection to a server through a web socket, you can send a message to the server by using the **send()** function of the **WebSocket** object, like this:

```
var message = ...; // Message to be sent
socket.send(message);
```

When the **send()** function is called, the message data is placed in a buffer and transmitted asynchronously.

- Use the **send()** function to send messages to a server

```
var message = ...;
socket.send(message);
```

- Use the **bufferedAmount** property to determine:
 - If there is a backlog before sending
 - If the message has been sent
- Handle the **error** event to determine whether an error has occurred
- Messages can be text, binary, or array data

 **Note:** The **bufferedAmount** property of the

WebSocket object reports the number of bytes of data queued for sending that have yet to be sent. It is set to zero when a connection is opened. It is not reset to zero when the connection is closed. Before sending a message, you may want to check the value in the **bufferedAmount** property to see if any previous messages are still being sent, and delay sending the message if the previous one has not yet been completed.

If an error occurs during sending, the **error** event occurs.

Message data can be sent as one of four object types:

- Any type of **UTF8 text data**; plain text, JSON, base64-encoded, and so on. The following example uses the **JSON.stringify()** function to serialize an object as text and to send it:

```
function sendRequest(socket, text) {
    socket.send(JSON.stringify({ request : text }));
}
```

- Blobs, such as files or images. The example below sends a file specified by a field in an HTML5 form:

```
function sendFile(socket) {
    var file = document.querySelector('input[type="file"]').files[0];
    socket.send(file);
}
```

- An **ArrayBuffer** object. This object represents a buffer used to store raw binary data. The data can then only be accessed by using a typed array view such as **Uint8Array** and **Int32Array**. The following example sends an array of integers:

```
function sendRawData(socket) {
    var numbers = new Uint8Array([8,6,7,5,3,0,9]);
    socket.send(numbers.buffer);
}
```

 **Note:** A **Uint8Array** represents an array of 8 bit unsigned integers. An 8 bit unsigned integer can hold a value between 0 and 255. An **Int32Array** is similar, except that it represents an array of 32 bit signed integers. A 32 bit signed integer can hold a value between -2147483648 and 2147483647.

- An **ArrayBufferView** object. This object represents a typed array view that can be used to access binary data in an **ArrayBuffer**. The following example sends the data from an image displayed by using a **<canvas>** element:

```
function sendCanvasImage(socket, canvas) {
    var image = canvas.getImageData(0,0,50,50);
    var binArray = new Uint8Array(image.data.length);
    for (var i=0; i<image.data.length; i++) {
        binArray[i] = image.data[i];
    }
    socket.send(binArray);
}
```

After you have sent a message, you can see whether it was transmitted successfully by checking if the **bufferedAmount** property of the **WebSocket** object is zero.

Receiving Messages From a Web Socket

The WebSocket protocol is bidirectional, and the server that you are connected to may send you a message at any time. The **message** event fires when a message is received from the server, giving you the opportunity to receive and process the message. The event object passed to the **message** event handler has two properties:

- **type**, which indicates whether the type of message received is text or binary data.
- **data**, which contains the message data.

The following code shows an example:

```
socket.onmessage = function(event) {
    // Message has been received
    if (event.type == "Text") {
        handleTextMessage(event.data);
    } else {
        handleBinaryMessage(socket.binaryType, event.data);
    }
};
```

- The **message** event fires when a message is received from a server:
 - Examine **event.type** to determine whether the message is text or binary
 - Read **event.data** to retrieve the message
- ```
socket.onmessage = function(event) {
 if (event.type == "Text") {
 handleTextMessage(event.data);
 } else {
 handleBinaryMessage(socket.binaryType, event.data);
 }
};
```
- Before receiving a message, set the **binaryType** property of the web socket object to indicate the expected format for binary data

There are no headers on the message, so it is assumed that you will know how to deal with both text and binary messages. For example, a text message may be JSON that needs parsing:

```
function handleTextMessage(text) {
 var message = JSON.parse(text);
 if (message.request) { // do something }
}
```

Alternatively, it may be a binary message. If so, the message will be an instance of either a blob or an **ArrayBuffer**. You can specify how binary messages should be presented to your application by the browser by setting the **binaryType** property of the **WebSocket** object; it is the responsibility of the browser to format the data according to the value specified in the **binaryType** property before raising the **message** event. The **binaryType** property is set to **blob** when a connection is first opened, but you can change it to **arrayBuffer** if your code expects to receive data in this format.

The following code example shows how to handle different types of binary data. If the **binaryType** property is set to **arrayBuffer**, the data is assumed to be an array of **integer** data. If it is set to **blob**, then the message is assumed to contain image data for a canvas:

```
function handleBinaryMessage(binaryType, data) {
 if (binaryType == "arrayBuffer") {
 var binArray = new Uint8Array(data);
 ...
 } else {
 var canvas = document.getElementById("serverCanvas");
 var image = canvas.getImageData(0,0,50,50);
 for (var i=8; i<image.data.length; i++) {
 image.data[i] = binArray[i];
 }
 canvas.putImageData(image.data,0,0);
 ...
 }
}
```

## Demonstration: Performing Real-time Communication by Using Web Sockets

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

MCT USE ONLY. STUDENT USE PROHIBITED

# Lab: Performing Real-time Communication by Using Web Sockets

## Scenario

During conference sessions, attendees may wish to ask questions. Distributing microphones among members of the audience can be problematic and slow, so you have been asked to add a page to the website that enables attendees to submit questions. Speakers can either respond immediately or later, depending on the nature of the questions and the session.

On the website, all questions must be displayed in real-time without reloading the page, so that all attendees and the speaker can see what has been asked. To support this requirement, a web socket server has been created. You need to update the web application to send the details of questions to the socket server, and also to receive and display questions submitted by other attendees.



**Note:** The web socket server is implemented by using ASP.NET and C#. The details of how this server works are outside the scope of this lab.

Conference organizers are concerned about people asking inappropriate questions. Therefore, a back-end moderation system is also being developed. Conference attendees should be able to report a question that they think is inappropriate. Administrators can then mark this question for removal. The web socket server will transmit a message to all connected clients, and the web page must be updated to remove the question.

## Objectives

After completing this lab, you will be able to:

1. Create a web socket that connects to a server and receives messages.
  2. Serialize and send messages to a web socket.
  3. Send and receive different types of messages by using a web socket.
- Estimated Time: 90 minutes
  - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
  - User Name: **Student**
  - Password: **Pa\$\$w0rd**

## Exercise 1: Receiving Messages from a Web Socket

### Scenario

In this exercise, you will review the new **Live** page and JavaScript. You will write JavaScript that creates a web socket and connect the socket to the server. Then you will handle messages received from the web socket. You will parse the JSON serialized messages into objects that contain new questions to display on the page. Finally, you will run the application, view the **Live** page and verify that it displays the questions sent by the socket server.

The main tasks for this exercise are as follows:

1. Review the Live page.
2. Receive messages by using a web socket.
3. Display questions received as messages.

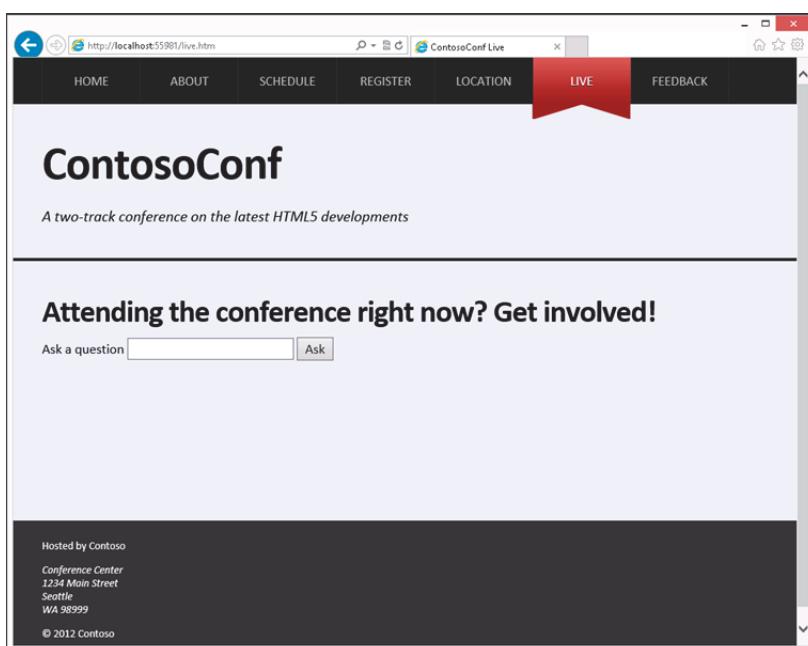
4. Test the application.

► **Task 1: Review the Live page**

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution from the **E:\Mod13\Labfiles\Starter\Exercise 1** folder.
4. Start the application and view the **Live** page. Ignore the error that occurs (this error happens because the JavaScript code for the page is not yet complete).

Using this page, an attendee can type a question and click **Ask** to send it to the presenter. All questions asked by all attendees will appear on this page, underneath the **Ask a question** box.

The **Live** page should look like this:



**FIGURE 13.1:THE LIVE PAGE**

5. Close Internet Explorer and return to Visual Studio.
6. In the **ContosoConf** project, open the **live.htm** file.
7. Review the HTML markup for this page and note the **<form>** element used to ask questions, together with the empty **<ul>** element that will display questions:

```

<form action="#">
 <label for="ask-question-text">Ask a question</label>
 <input id="ask-question-text" type="text" />
 <button type="submit">Ask</button>
</form>

 <!-- Questions will be displayed here when received by the web socket. -->


```

8. Also, note the **<script>** element referencing the **live.js** file in the **/scripts/pages** folder:

```

<script src="/scripts/pages/live.js" type="text/javascript"></script>

```

9. Open the **live.js** file that is in the **/scripts/pages** folder and review the code. The JavaScript code in this file defines a **LivePage** object, which controls the page. The code to manipulate the user interface and respond to DOM events has already been written:

```
var LivePage = Object.inherit({
 ...
})
```

### ► Task 2: Receive messages by using a web socket

1. In the **live.js** file near the end of the file (just before the call to the **create()** method), find the following comment:

```
// TODO: Create a web socket connection to ws://localhost:55981/live/socket.ashx
```

- The web server that handles requests sent to the web socket is located at the URL **localhost:55981/live/socket.ashx**.
2. After this comment, add code to create a new **WebSocket** object that connects to the socket server at **localhost:55981/live/socket.ashx**.



**Note:** The socket is stored in the **LivePage** object as **this.socket**, so methods of **LivePage** can use it.

3. Find the **initializeSocket()** function in the **LivePage** object.
4. In the **initializeSocket()** function, after the comment **// TODO: Assign a callback to handle messages from the socket**, assign a callback function to the **onmessage** property of the socket. As follows:

```
this.socket.onmessage = this.handleSocketMessage.bind(this);
```



**Note:** The JavaScript **bind()** function associates (or binds) an object to a function. It returns a new function that runs as a method of the object, as shown in the following example:

```
var o = {data : 1}; // object
...
function f(a) = {
 return this.data + b;
}; // function
...
var g = f.bind(o); // Calling g() now runs function f() as a method of object o
When you invoke the new function (g()) in the example above), it calls the f()
function as a method of o, so any references to this in f() refer to the object o.
var result = g(2); // evaluates as o.f(2) and returns 3 (f.data + 2)
In the case of the lab code, the handleSocketMessage() function looks like this:
handleSocketMessage: function (event) {
 // TODO: Parse the event data into message object.
 // var message = ... ;
 // TODO: Check if message has a `questions` property, before calling
 handleQuestionsMessage
 this.handleQuestionsMessage(message);
}
```

Assigning the **onmessage** callback of the **socket** variable of the **LivePage** object by using the **bind()** function ensures that the call to the **this.handleQuestionsMessage()** correctly invokes the method of the **LivePage** object rather than attempting (and failing) to invoke a method with the same name on the **socket** variable.



**Note:** You will implement the commented code for the **handleSocketMessage** method in the next step.

### ► Task 3: Display questions received as messages

1. The **handleSocketMessage** method receives incoming messages from the web socket. The message data is stored in the **event** parameter, but it is serialized as a JSON string. In this method, add JavaScript code to perform the following tasks:
  - After the comment // TODO: Parse the event data into message object, parse the JSON message into a JavaScript object named **message**.
  - Modify the statement that calls the **handlQuestionsMessage()** function and only run this statement if the **message** variable contains a **questions** property.
2. Review the **displayQuestion** method:

```
displayQuestion: function (question) {
 var item = this.createQuestionItem(question);
 //item.appendChild(this.createReportLink());
 this.questionListElement.appendChild(item);
}
```

This method adds the questions specified as the parameter to the list below the **Ask a question** box on the page (ignore the statement that is commented out.)

3. In the **handleQuestionsMessage** method, after the comment, add code to display each question by using the **displayQuestion()** function as follows:

```
if (message.questions) {
 message.questions.forEach(this.displayQuestion, this);
}
```



**Note:** The JavaScript **forEach()** function provides a shorthand way of writing a **for** loop. It iterates through an array and invokes a function specified as the first parameter on items in the array. The second parameter specifies the object to use as **this** in the function being invoked.

### ► Task 4: Test the application

1. Run the application and view the **Live** page. The server-side web socket will send a series of test questions.
  2. Verify that the following four questions appear on the page over a period of a few seconds.
    - What are some good resources for getting started with HTML5?
    - Can you explain more about the Web Socket API, please?
    - This is an #&!%!\* inappropriate message!!
    - How much of CSS3 can I use right now?
  3. Close Internet Explorer.

**Results:** After completing this exercise, you will have added JavaScript code to the **Live** web page to receive questions from a web socket and to display them.

## Exercise 2: Sending Messages to a Web Socket

### Scenario

In this exercise, you will create a message object that contains a question to ask. You will serialize this message and send it to the server by using the web socket. Then you will run the application, view two concurrent instances of the **Live** page, and verify that asking questions results in them being displayed on the page in both instances.

The main tasks for this exercise are as follows:

1. Format a question as a message.
2. Send the message to the web socket.
3. Test the application.

#### ► Task 1: Format a question as a message

1. Open the **ContosoConf** solution in the **E:\Mod13\Labfiles\Starter\Exercise 2** folder.
2. Open the **live.js** file in the **scripts/pages** folder. In this file, the **LivePage** object has methods that handle the **Ask a question** form submission. The question text is passed to the **askQuestion** method, which looks like this:

```
askQuestion: function (text) {
 // TODO: Create a message object with the format { ask: text }
 // TODO: Convert the message object into a JSON string
 // TODO: Send the message to the socket
 // Clear the input ready for another question.
 this.questionInput.value = "";
}
```

3. In the **askQuestion** method, after the comment **// TODO: Create a message object with the format { ask: text }**, create a variable named **message** that contains an object with the following format (this is the message structure expected by the socket server):

```
{ ask: text }
```

4. After the comment **// TODO: Convert the message object into a JSON string**, serialize the **message** variable as a JSON string.
  - Use the **JSON.stringify()** function.

#### ► Task 2: Send the message to the web socket

1. In the **askQuestion** method, after the comment **// TODO: Send the message to the socket**, send the JSON serialized message to the web socket.
  - Use the **send()** function of the **socket** variable.

#### ► Task 3: Test the application

1. Run the application and view the **Live** page.
2. Open a second tab in Internet Explorer and view the **Live** page again.
3. Use the **Ask a question** form to submit questions.
4. Verify that the questions are displayed on the pages in both tabs.



**Note:** The message **This is an #&!%!\* inappropriate message** might disappear from the second tab.

5. Close Internet Explorer.

**Results:** After completing this exercise, you will have modified the **Live** question page to enable users to ask questions by sending messages to the server by using the web socket.

## Exercise 3: Handling Different Web Socket Message Types

### Scenario

In this exercise, you will add a link next to each question to enable a student to report the question as inappropriate. Then you will handle messages from the server instructing the page to remove a question; this process will involve handling different types of messages. Finally, you will run the application, view the **Live** page, and verify that clicking the link causes the question to be removed.

The main tasks for this exercise are as follows:

1. Display report links.
2. Send the report message.
3. Handle Remove Question messages.
4. Test the application.

#### ► Task 1: Display report links

1. Open the **ContosoConf** solution in the **E:\Mod13\Labfiles\Starter\Exercise 3** folder.
2. Open the **live.js** file in the **scripts/pages** folder, and review the **createReportLink** method:

```
createReportLink: function () {
 var report = document.createElement("a");
 report.textContent = "Report";
 report.setAttribute("href", "#");
 report.setAttribute("class", "report");
 return report;
}
```

This method creates a new link element that enables a user to report a question to the moderator, and adds attributes that cause the link to be rendered appropriately.

3. Uncomment the following line of the **displayQuestion** method:

```
//item.appendChild(this.createReportLink());
```

- This statement calls the **createReportLink()** function to add the **Report** link next to each question.

#### ► Task 2: Send the report message

1. When a **Report** link is clicked, the **handleQuestionsClick** event handler changes the text of the link to **Reported**, and then calls the **reportQuestion** method:

```
handleQuestionsClick: function (event) {
 event.preventDefault();
 var clickedElement = event.srcElement || event.target;
 if (this.isReportLink(clickedElement)) {
 var questionId = clickedElement.parentNode.questionId;
 this.reportQuestion(questionId);
 clickedElement.textContent = "Reported";
 }
}
```

```
}
```

In the **reportQuestion()** function, after the comment // TODO: Send socket message { report: questionId }, add code to send the socket a message with the following format (this message specifies the ID of the question being reported):

```
{ report: questionId }
```

- Use the **JSON.stringify()** function to serialize the message.

### ► Task 3: Handle Remove Question messages

1. Find the **handleRemoveMessage()** function:

```
handleRemoveMessage: function (message) {
 var listItems = this.questionListElement.querySelectorAll("li");
 for (var i = 0; i < listItems.length; i++) {
 if (listItems[i].questionId === message.remove) {
 this.questionListElement.removeChild(listItems[i]);
 break;
 }
 }
}
```

This function removes all messages that contain the **remove** property from the displayed list of questions; the socket server attaches this property to all messages that should no longer be displayed.

2. Update the **handleSocketMessage()** function to call the **handleRemoveMessage()** function for all messages that have the **remove** property.

### ► Task 4: Test the application

1. Run the application and view the **Live** page.
2. Verify that **Report** links are displayed next to questions.
3. Click the **Report** link next to the question **What are some good resources for getting started with HTML5?**
4. Verify that the question disappears from the page (there will be a short delay while the question is assessed).
5. Close Internet Explorer.

**Results:** After completing this exercise, you will have added a feature to the **Live** page that enables users to report inappropriate questions and causes the application to remove them.

# Module Review and Takeaways

In this module, you have learned how web sockets provide a fast, robust and efficient solution for real-time communication between a web page and its server. You have also learned how to connect to a web socket from a client application by using the WebSocket API, and how to send and receive messages over a web socket.

## Review Question(s)

Verify the correctness of the statement by placing a mark in the column to the right.

| Statement                                                                        | Answer |
|----------------------------------------------------------------------------------|--------|
| Web socket clients send and receive data over an HTTP connection. True or False? |        |

## Test Your Knowledge

| Question                                                             |                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>How does a client create a connection to a web socket server?</b> |                                                                                                                                                                                                                                                                           |
| Select the correct answer.                                           |                                                                                                                                                                                                                                                                           |
|                                                                      | The client uses the open() function of the WebSocket object and specifies the URL of the server.                                                                                                                                                                          |
|                                                                      | The client uses the connect() function of the WebSocket object and specifies the URL of the server.                                                                                                                                                                       |
|                                                                      | Web sockets use a stateless protocol similar to HTTP. A client application simply specifies the address of the server as a parameter of the send() function of the WebSocket object. A connection is automatically established while the message is sent and then closed. |
|                                                                      | The client creates a new WebSocket object and specifies the URL of the server.                                                                                                                                                                                            |
|                                                                      | The client has to wait until the server responds to send a message that grants it permission to connect.                                                                                                                                                                  |



# Module 14

## Performing Background Processing by Using Web Workers

### Contents:

|                                                                   |       |
|-------------------------------------------------------------------|-------|
| Module Overview                                                   | 14-1  |
| Lesson 1: Understanding Web Workers                               | 14-2  |
| Lesson 2: Performing Asynchronous Processing by Using Web Workers | 14-5  |
| Lab: Creating a Web Worker Process                                | 14-11 |
| Module Review and Takeaways                                       | 14-16 |
| Course Evaluation                                                 | 14-17 |

## Module Overview

JavaScript code is a powerful tool for implementing functionality in a web page, but you need to remember that this code runs either when a web page loads or in response to user actions while the web page is being displayed. The code is run by the browser, and if the code performs operations that take a significant time to complete, the browser can become unresponsive and degrade the user's experience. HTML5 introduces web workers, which enable you to offload processing to separate background threads and thus enable the browser to remain responsive. This module describes how web workers operate and how you can use them in your web applications.



**Note:** The F12 Developer Tools in Internet Explorer 10 provides support for debugging web workers, enabling you to test and verify the code running in multiple scripts.

### Objectives

After completing this module, you will be able to:

- Explain how web workers can be used to implement multithreading and improve the responsiveness of a web application.
- Perform processing by using a web worker, communicate with a web worker, and control a web worker.

## Lesson 1

# Understanding Web Workers

Web workers enable you to perform long-running tasks asynchronously, enabling the browser to remain responsive. Each web worker runs as a separate thread in its own isolated environment. An application can initiate a web worker and communicate with it by passing it messages. The application can also terminate the web workers that it creates. In this lesson, you will learn how web workers operate and see the different types of web workers that HTML5 provides.

### Lesson Objectives

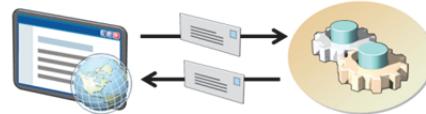
After completing this lesson, you will be able to:

- Describe the purpose of web workers.
- List some common scenarios for using web workers.
- Explain how web workers run in an isolated environment.
- Describe the different types of web workers that HTML5 supports.

### What is a Web Worker?

JavaScript code running in a browser is executed by using a single thread. What this means is that while the browser is running your JavaScript code it cannot do anything else, such as respond to the user clicking a button on a form on a web page displayed by the browser, or even display text that the user might be typing. This is not a problem in many cases: The JavaScript used by a web page usually consists of a number of discrete functions, each of which runs quickly in response to an event, and the user does not notice the minor delay in response while the code is running. However, if you write JavaScript code that has to perform more complex or resource-intensive tasks, then the browser can become noticeably less responsive, or even appear to stop altogether. In this situation, users frequently assume that the application has crashed. They may try and close the browser before trying a task again—and getting the same frustrating result.

- JavaScript code running in a web page is single-threaded
  - Long-running functions can cause the browser to become unresponsive
- Web workers enable a web page to move code to a parallel execution environment, enabling the browser to remain responsive
  - Code in the web page communicates with the web worker by passing messages



A web worker enables you to offload long-running and data-intensive tasks to a separate execution environment, distinct from that of the web page, leaving the browser free to handle the user interface. This keeps the web page responsive to the user while the web worker performs the data processing behind the scenes.

 **Note:** While it is a helpful analogy to consider web workers as the client-side equivalent of multithreading on a server, it is important to note that JavaScript web workers are in fact considerably simpler than their server-side counterparts, and do not feature semaphores, mutexes, or thread-blocking features.

A web worker is a piece of JavaScript code that runs in parallel to the web page. It is initiated by code running in a web page, but has no access to any resources on that page. Instead, the code in the web page communicates with the web worker by sending it messages containing the data that the web worker

needs. The web worker can respond by sending messages back to the code in the web page. The code in the web page has full control over the web worker, and can terminate it at any time.

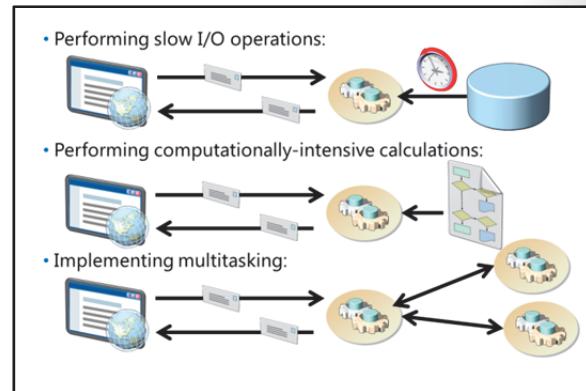
 **Note:** Browser vendors are free to implement web workers in whatever way is most appropriate for the platform on which the browser is running. Internet Explorer uses threads, while other browsers may actually create completely separate host processes. If they are available, some implementations make use of multiple processor cores to implement true multiprocessing.

 **Additional Reading:** For information about the web worker specification, visit <http://go.microsoft.com/fwlink/?LinkId=267757>.

## Why Use a Web Worker?

Web workers are ideally suited to a variety of scenarios, including:

- Performing long-running or slow I/O operations. A web worker could be used to send data to a web service and await a response, while the web page continues running. Alternatively, a web worker could use the File API to read data from the local file system for processing, and then upload this data to a web service or pass it to the web page for display.
- Performing lengthy calculations. A web worker could be used to implement a complex algorithm that implements a computation-intensive calculation. The results can be returned to the web page when the calculation is complete.
- Dividing work between concurrent threads. Operations that involve processing a large amount of data, such as the information held in a large array or a file on disk, could be delegated to a collection of concurrent web workers. A master web-worker initiated by the web page can act as a controller that creates subordinate web workers and delegates work to them. The master web worker aggregates the results and sends them back to the web page. If the computer running the browser has a processor with multiple cores, and depending on how the browser implements web workers, this model provides a good way to exploit the parallel processing capabilities of the processor.



MCT USE ONLY  
STUDENT USE PROHIBITED

## Web Worker Isolation

A web worker runs in isolation from the web page and from any other web workers that the page creates.

Web workers are hosted in the browser, but they do not have access to the document of the web page that creates them, or to the data and JavaScript objects for that page, outside of the confines of its own code. Web workers run in a restricted environment and **only have access to** a limited subset of the features provided by JavaScript, such as the **navigator** object, the **location** object, and the application cache. A web worker can **perform I/O operations** by using the File API, and can send and receive requests to remote servers by using the **XMLHttpRequest** object. Web workers can also create other web workers.

Given that a web worker has no access to the data on a web page, when a web page creates a new web worker it must provide it with the information that it needs by passing messages. The web page can use the **postMessage()** function to send a message, and it **can catch messages sent back from the web worker** by handling the **message** event. The web worker operates in a similar manner, receiving messages by handling the **message** event and sending replies by using the **postMessage()** function.

- A web worker runs isolated from the web page and other web workers
  - It cannot access the document of the web page
  - It cannot access data or JavaScript code in the web page
- A web worker has access to a limit subset of JavaScript functionality
- A web page communicates with a web worker by sending and receiving messages:
  - Send messages by using the **postMessage()** function
  - Receive messages by handling the **message** event

## Dedicated and Shared Web Workers

There are two types of web worker: dedicated workers and shared workers.

**A dedicated worker is the exclusive property of the page that created it.** It runs asynchronously from the page, but can be controlled by the page. Only the page that created the worker can post messages to it and receive messages back from it. A dedicated web worker can be terminated by the page that created it. Note that if the web page is closed (if the browser terminates, or the user navigates to a different page), then any dedicated web workers created by that page will stop. A web worker can also forcibly terminate itself. A dedicated worker is ideal for performing a long-running task on behalf of a web page, such as uploading a large file or processing a large amount of data.

A shared worker is created by one page, but other pages running as part of the same web application can post messages to, and receive messages from, the same shared worker. **Shared workers can be controlled by any page in the web application.** Shared workers stop if the user navigates to a different website.

A shared worker provides a mechanism for **implementing centralized application processes**, and can also be used to **implement cross-page communication**. For example, a shopping cart on a catalog site could be implemented by using a shared worker that uses AJAX methods to quietly download and share very detailed product information for items in the cart. All the catalog pages that the user launches have access to this shared worker, keeping the cart in easy reach and scope no matter how many pages the user opens while casually browsing the store, including clicking the back button in the browser.

- Dedicated web workers:
  - Belong to a single page
  - Can only communicate with that page
  - Stop when the page is closed
- Shared web workers:
  - Can be accessed by all pages in a web application
  - Can communicate with any page in the web application
  - Stop when the web application finishes

## Lesson 2

# Performing Asynchronous Processing by Using Web Workers

HTML5 provides a JavaScript API for creating and managing web workers. In this lesson, you will learn how to create web workers, communicate with them, and handle errors that may occur while a web worker is running.

## Lesson Objectives

After completing this lesson, you will be able to:

- Create and terminate a dedicated web worker.
- Send messages to a web worker, and receive messages from a dedicated web worker.
- Explain how to implement the structure of a web worker.
- Explain how to create a shared web worker.

## Creating and Terminating a Dedicated Web Worker

To create a dedicated web worker, you first create a new **Worker** object. The **Worker** constructor expects the URL of a JavaScript file as a parameter. This JavaScript file contains the code that the web worker runs. The following code example checks that the browser supports web workers, and then creates a new web worker that runs the JavaScript code in the *processScript.js* file:

```
var webWorker;
if(typeof(Worker)!== "undefined") {
 webWorker = new
 Worker("processScript.js");
}
```

• Starting a web worker:

```
var webWorker;
if(typeof(Worker)!== "undefined") {
 webWorker = new Worker("processScript.js");
}
```

• Terminating a web worker from the web page:

```
webWorker.terminate();
```

• Terminating a web worker from inside the web worker:

```
self.close();
```

It is important to understand that the URL of the script is a web-hosted file that **must be part of the same web application** as the HTML5 page running the script; you cannot use this mechanism to run JavaScript code that is located at a different site.

Web workers terminate automatically when the containing page is closed, and the resources used by the web worker are released. However, the web page that creates a dedicated web worker can end the web worker at any time by using the **terminate()** function:

```
webWorker.terminate();
```

Web workers may also terminate themselves:

```
self.close();
```

 **Note:** The **self** variable is an alias for **this**; it references the **Worker** object from code being run by the object. All web workers create the **self** alias when they start running, and it is recommended that you use it rather than **this**; it is possible that the code being run by the web

worker manipulates **this** after the web worker has been started, so performing **this.close()** might not have the expected effect.

You can also create *inline* web workers. The JavaScript code for an inline web worker is defined as part of the web page that starts the worker, rather than being held in a separate JavaScript file. This enables your code to be more self-contained, and reduces the number of code files that you need to track and maintain in a complex application.

You can create an inline web worker in several different ways. The following code example creates a **Blob** containing the JavaScript code for the web worker, and then uses the **createObjectURL()** function of the **URL** object to create a URL that references this object. The code passes this URL to the **Worker** constructor.

```
var workerBlob = new Blob(["{ /* Code for web worker goes here */ }"]);
var workerURL = URL.createObjectURL(workerBlob);
var webWorker = new Worker(workerURL);
...
```

Another technique is to define the code for the web worker in a **<script>** element, as shown in the following example:

```
<!DOCTYPE html>
<html>
 <head>
 <meta charset="utf-8" />
 </head>
 <body>
 ...
 <script id="web-worker" type="javascript/worker">
 // Code for web worker goes here
 </script>
 <script type="text/javascript">
 ...
 var workerBlob = new Blob([document.querySelector('#web-worker').textContent]);
 var workerURL = URL.createObjectURL(workerBlob);
 var webWorker = new Worker(workerURL);
 ...
 </script>
 </body>
</html>
```

The code for the web worker has the type **javascript/worker**. The JavaScript engine in the browser will not recognize this type, so it will not attempt to parse and run the code, but will instead treat it as a block of data (you could use almost any convenient string for the type). The JavaScript code for this page creates a **Blob** object that references this script, and then creates a URL for this object, which in turn is used to create a **Worker** object.

 **Note:** Inline web workers are not widely supported, and they are not currently supported by Internet Explorer.

 **Additional Reading:** For more information about implementing web workers with Internet Explorer 10, visit <http://go.microsoft.com/fwlink/?LinkId=267759>.

## Communicating With A Dedicated Web Worker

Web worker scripts are completely isolated from the web page, meaning that they have no access to objects in that page or in the DOM for that page. Instead, a web page and a web worker communicate by passing messages to each other.

To send a message to a web worker, the web page can use the **postMessage()** function of the web worker object. This function takes a parameter containing the data to send to the web worker. This parameter can be a string or a JSON object. To receive a message, the web worker handles the **message** event. The information sent by the web page is available in the **data** property of the event object passed to the handler. To send a message back from a web worker to the page, the process is reversed; the web worker uses the **postMessage()** function, and the web page receives the message by handling the **message event** of the web worker object.

The following example shows the code for a simple web worker that echoes the data that it is sent back to the web page:

```
// processScript.js
function messageHandler(event) {
 self.postMessage("Received: " + event.data);
}
self.addEventListener("message", messageHandler, false);
```

The code in the web page looks like this:

```
function replyHandler(event) {
 alert("Reply: " + event.data); // Display the reply in an alert
}
var webWorker;
...
webWorker = new Worker("processScript.js"); // The web worker code is in this file
webWorker.addEventListener("message", replyHandler, false);
webWorker.postMessage("Here is some data");
...
```

If an exception occurs while the web worker is running, it will raise the **error event** and then terminate. The web page should be prepared to catch this event. The following code shows an example:

```
function errorHandler(event) {
 console.log(event.message);
}
...
webWorker.addEventListener("error", errorHandler, false);
...
```

The event object passed to the error handler contains the following properties:

- **message**. A human-readable error message.
- **filename**. The name of the script file (as a URL) in which the error occurred.
- **lineno**. The line number of the script file where the error occurred.

MCT ICE ONLY STUDENT LICENSE PROHIBITED

## The Structure of a Web Worker

A typical web worker uses the **message** event to receive messages, and then performs processing based on the data in the message before waiting for the next message. A common idiom is to implement the **Command pattern**, in which each message contains a field that indicates the action that the web worker should perform. Other fields can contain the information that the web worker should process by performing the action. For example, JavaScript code for a web page might construct a message that contains the action "DOWORK", together with the data that the web worker should process:

```
var msg = {
 "command": "DOWORK",
 "data": ...
};
webWorker.postMessage(msg);
```

The message handler in the web worker would look similar to this:

```
function messageHandler(event) {
 var data = event.data; // Input data is expected in JSON format
 switch (data.command) {
 case "DOWORK": // process the DOWORK command
 ...
 break;
 case "DOMOREWORK": // process the DOMOREWORK command
 ...
 break;
 case "FINISH": // tidy up and shut down
 ...
 self.postMessage("Shutting down");
 self.close();
 }
}
```

Remember that a web worker is itself single-threaded, so if the processing that it performs in response to a message takes a lengthy period of time, it will not be able respond to other messages immediately; the messages will be queued and handled when processing completes. In these situations, the web worker could itself delegate work to other web workers that it creates and manages, leaving it free to handle messages as they arrive.

A web worker may require access to functions and utilities defined in other JavaScript files. For example, a web worker may use the jQuery library to send requests to an external web service. [A web worker does not have a DOM](#), so you cannot reference scripts by using **<script>** elements (also, many of the jQuery functions that access the DOM or the Window object will cause errors in a web worker script). Instead, a web worker can use the **importScripts()** function, as shown in the following example:

```
importScripts("myfunctions.js");
```



**Note:** You can add multiple JavaScript files by using the **importScripts()** function.

- Web workers often implement a message loop and the Command pattern:

```
function messageHandler(event) {
 var data = event.data;
 switch (data.command) {
 case "DOWORK": // process the DOWORK command
 ...
 break;
 case "DOMOREWORK": // process the DOMOREWORK command
 ...
 break;
 case "FINISH": // tidy up and shut down
 ...
 self.postMessage("Shutting down");
 self.close();
 }
}
```

- Web workers can import scripts and access some global objects and functions

It is common to use the **importScripts()** function near the start of the web worker code to ensure that the objects and functions defined by these scripts are in scope.

Web workers have access to the **Global** object that defines a set of global functions, in much the same way that a web page does. However, the **Global** object for a web worker is scoped to that web worker, and it does not contain any of the data items or functions defined by the web page. However, this does mean that many of the built-in JavaScript global functions are available to web workers.

Web workers have access to the **navigator** object. This object contains information that the web worker can use identify the browser, including **appName**, **appVersion**, **platform**, and **userAgent**. Web workers also have read-only access to the **location** object, which provides information about the URL of the current page, such as the **hostname**, **pathname**, and **port** properties.

## Creating a Shared Web Worker

A dedicated web worker is accessible only from the page that declared it. To create a web worker that is accessible from all pages in a web application, you can create a shared web worker.

You create a shared web worker by using the **SharedWorker** constructor. However, to enable multiple pages to send messages to the web worker, each page has its own *port* that it uses. A web page sends messages to the web worker through its port by using the **postMessage()** function. Any replies are received by using the **message** event on the same port. The following

code example shows how to create a shared web worker to send and receive messages. Notice that a web page must use the **start()** function of the port before sending the first message. This function alerts the web worker that a new connection is being established and enables it to prepare to receive messages on this port:

```
function replyHandler(event) {
 ...
}
var sharedWebWorker;
...
sharedWebWorker = new SharedWorker("sharedProcessScript.js");
sharedWebWorker.port.addEventListener("message", replyHandler, false);
sharedWebWorker.port.start();
...
var data = ...;
sharedWebWorker.port.postMessage(data);
```

- Use the **SharedWorker** constructor to create a shared web worker
- Each web page communicates with a shared web worker by using its own port

```
function replyHandler(event) { ... }
var sharedWebWorker = new SharedWorker("sharedProcessScript.js");
sharedWebWorker.port.addEventListener("message", replyHandler, false);
sharedWebWorker.port.start();
...
var data = ...;
sharedWebWorker.port.postMessage(data);
```

- The **connect** event in a shared web worker fires when a new port is opened

A shared web worker has a **connect** event that fires each time a web page opens a new port by using the **start()** function. This event receives information about the port that has been opened, and the handler typically uses this information to add a **message** event handler to the port, and then start the port, so that the web worker can receive messages on it. The code below shows an example:

```
function messageHandler(event) {
 // Handle messages received on a port
 ...
}
function connectHandler(event) {
 var port = event.ports[0];
 port.addEventListener("message" messageHandler, false);
```

```
 port.start();
}
self.addEventListener("connect", connectHandler, false);
```

A shared web worker sends a reply back to a web page by using the **postMessage()** function of the port that the web page opened. At the present time, the shared web worker event-handling mechanism does not provide any built-in way to identify which port belongs to which web page, so the web worker must track this information itself. One technique is for a web page to include an identifier in each message that it sends, and for the web worker to associate each identifier with the corresponding port.

 **Best Practice:** It is worth noting that shared web workers have not been widely taken up because of security considerations. Internet Explorer does not currently support them.

## Demonstration: Creating a Web Worker Process

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

# Lab: Creating a Web Worker Process

## Scenario

When a speaker creates a conference badge, the speaker drags and drops an image containing a photograph onto the web page. This photograph may be a color image. However, the conference speaker badges will be printed in grayscale. Therefore, the web page that creates the badges should render the speaker photograph in grayscale in order to give an accurate representation of the printed output.

An image file may be many megabytes in size. To avoid uploading large files to a server for processing, you have decided to convert the photos to grayscale by using JavaScript code running in the web browser.

However, processing large images will cause the web browser to become unresponsive while it performs this processing. You therefore decide to use a web worker to move the image conversion to a background process, enabling the web browser to remain responsive.

## Objectives

After completing this lab, you will be able to:

1. Create a web worker and implement a web worker script.
2. Send messages to and receive messages from a web worker.
  - Estimated Time: 60 minutes
  - Virtual Machines: 20480B-SEA-DEV11, MSL-TMG1
  - User Name: **Student**
  - Password: **Pa\$\$w0rd**

## Exercise 1: Improving Responsiveness by Using a Web Worker

### Scenario

In this exercise, you will move slow-running image processing code into a web worker.

First, you will review the HTML markup and JavaScript code for the **Speaker Badge** page. You will then update the code so that it converts the speaker photo to grayscale. Next, you will run the application and verify that the browser becomes unresponsive while processing a large image. You will then create a web worker and move the CPU-intensive image processing code into the script for the web worker. You will use messages to communicate with the worker. Finally, you will run the application, view the **Speaker Badge** page, and verify that the web browser remains responsive while the image is being processed.

The main tasks for this exercise are as follows:

1. Review the Speaker Badge page.
2. Convert the speaker badge image to grayscale.
3. Create a web worker to perform image processing.
4. Move image processing code into the web worker.
5. Return image data from the web worker.
6. Test the application.

### ► Task 1: Review the Speaker Badge page

1. Start the **MSL-TMG1** virtual machine if it is not already running.

2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.
3. Start Visual Studio and open the **ContosoConf.sln** solution from the **E:\Mod14\Labfiles\Starter\Exercise 1** folder.
4. In the **ContosoConf** project, review the **Speaker Badge** page. Note the script reference to the **grayscale.js** JavaScript file in the **/scripts** folder:

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

5. Open the **grayscale.js** JavaScript file in the **scripts** folder, and review the code. Notice that this file contains a function named **conference.grayscaleImage**, which converts an image to grayscale. This function converts the image one pixel at a time, and can take a long time to convert a large image:

```
conference.grayscaleImage = function (image) {
 // Converts a colour image into gray scale.
 var deferred = $.Deferred();
 var canvas = createCanvas(image);
 var context = canvas.getContext("2d");
 var imageData = getImageData(context, image);
 // TODO: Create a Worker that runs /scripts/grayscale-worker.js
 var pixels = imageData.data;
 // 4 array items per pixel => Red, Green, Blue, Alpha
 for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
 }
 // Update the canvas with the gray scaled image data.
 context.clearRect(0, 0, canvas.width, canvas.height);
 context.putImageData(imageData, 0, 0);
 // Returning a jQuery Deferred makes this function easy to chain together with
 other deferred operations.
 // The canvas object is returned as this can be used like an image.
 deferred.resolveWith(this, [canvas]);
 return deferred;
};
```

-  **Note:** For the purposes of this exercise, you can ignore the references to the jQuery **.Deferred()** function and the **deferred** object; this object simply enables the **grayscaleImage()** function to be used in a pipeline by using the **pipe()** function, as you will see in the next task.

6. Also note that the **/scripts** folder contains another JavaScript file called **grayscale-worker.js**, which currently contains no code.

#### ► Task 2: Convert the speaker badge image to grayscale

1. Open the **speaker-badge.js** JavaScript file in the **scripts/pages** folder. The JavaScript code in this page contains the functions that enable a user to drop an image onto the canvas and draw the badge (you added this functionality in a previous lab). Notice that the JavaScript code also contains a reference to the **grayscaleImage()** function in a variable, also called **grayscaleImage**.
2. In the **handleDrop** method, after the comment **// TODO: Add grayscaleImage into the processing pipeline**, modify the JavaScript code to insert the **grayscaleImage()** function into the processing pipeline for an image by using the jQuery **pipe()** function, as follows in bold:

```
this.readFile(file)
 .pipe(this.loadImage)
 .pipe(grayscaleImage)
```

```
.done(this.drawBadge, this.notBusy);
```



**Note:** The **pipe()** function enables you to chain function calls together into a pipeline. The functions can operate asynchronously, but the **pipe()** function ensures that each function call is made only after the previous call has completed.

3. Run the application and view the **speaker-badge.htm** page.
4. Drag the file **E:\Mod14\Labfiles\Resources\mark-hanson-large.jpg** from File Explorer and drop it onto the speaker badge canvas in Internet Explorer.
5. While the image is being processed, try scrolling the page or moving to a different page. Note that the page is frozen.



**Note:** Internet Explorer may display the message **localhost is not responding due to a long-running script**. If this occurs, allow the script to complete.



**Note:** You can observe the load that Internet Explorer is under by using Task Manager as follows:

6. Right-click the Windows Taskbar and then click **Task Manager**.
7. In Task Manager, click **More details**, and then click the **Performance** tab.

The virtual machine is configured with two virtual CPUs, and the **grayscaleImage()** function will keep one of the CPUs fully occupied, resulting in an overall processor utilization of approximately 50%. When the grayscale image appears, the processor utilization will drop close to 0.

8. Close Internet Explorer.

#### ► **Task 3: Create a web worker to perform image processing**

1. In the **grayscale.js** file in the **grayscaleImage()** function, after the comment `// TODO: Create a Worker that runs /scripts/grayscale-worker.js, add code to create a web worker that runs the /scripts/grayscale-worker.js` JavaScript file.
- The **imageData** variable in this function contains the data to be converted to grayscale. Pass the **imageData** variable to the web worker by using the **postMessage()** function.
2. Open the **grayscale-worker.js** file in the **scripts** folder and add an event listener for the **message** event. Create an anonymous function to handle the event, but leave this function empty (you will write the code for this event handler in a later task, when you implement the functionality to return data from the web worker).
3. Return to the **grayscale.js** file. In the **grayscaleImage()** function, before the call to the **postMessage()** function, add an event listener called **handleMessage** for the **message** event of the web worker, like this:

```
var handleMessage = function (event) {
};
worker.addEventListener("message", handleMessage.bind(this));
```



**Note:** Remember that the **bind()** function ensures that any references to **this** in the anonymous function that handles the event will be resolved to the web page and not the web worker variable.

#### ► Task 4: Move image processing code into the web worker

1. Move the **grayscalePixel** function from **grayscale.js** to **grayscale-worker.js**. The **grayscalePixel** function looks like this:

```
var grayscalePixel = function (pixels, index) {
 /// <summary>Updates the pixel, starting at the given index, to be gray
 scale.</summary>
 var brightness = 0.34 * pixels[index] + 0.5 * pixels[index + 1] + 0.16 *
 pixels[index + 2];
 pixels[index] = brightness; // red
 pixels[index + 1] = brightness; // green
 pixels[index + 2] = brightness; // blue
};
```

2. Find the **pixels** variable and the **for** loop that performs the conversion to grayscale in the **grayscaleImage()** function in **grayscale.js**:

```
var pixels = imageData.data;
// 4 array items per pixel => Red, Green, Blue, Alpha
for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
}
```

3. Move this code into the **message** callback function of **grayscale-worker.js**.
  - In the **message** callback, create and populate the **imageData** variable with the value in the **data** property of the **event** parameter:

```
addEventListener("message", function (event) {
 var imageData = event.data;
 var pixels = imageData.data;
 for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
 }
});
```

#### ► Task 5: Return image data from the web worker

1. In the **grayscale-worker.js** file, in the **message** callback, after the end of the **for** loop, send the updated image data back to the web page by posting a message with the following format:

```
{ done: imageData }
```

- Use the **postMessage()** function to send the message.
- 2. In the **grayscale.js** file, near the end of the **grayscaleImage()** function, find the following lines of code that update a temporary canvas with the processed image data and then resolve the deferred object:

```
// Update the canvas with the gray scaled image data.
context.clearRect(0, 0, canvas.width, canvas.height);
context.putImageData(imageData, 0, 0);
// Returning a jQuery Deferred makes this function easy to chain together with other
deferred operations.
// The canvas object is returned as this can be used like an image.
deferred.resolveWith(this, [canvas]);
```

3. Move this block of code into the **message** event callback function (referenced by the **handleMessage** variable).
  - Retrieve the image data from the **data.done** property of the **event** parameter in the callback function and store it in a variable called **updatedImageData**.
  - Pass **updatedImageData** as the first parameter to the **putImageData()** function (replace the reference to **imageData**).



**Note:** Make sure that you leave the following statement in place at the end of the **grayscaleImage()** function.

```
return deferred;
```

#### ► Task 6: Test the application

1. Run the application and view the **Speaker Badge** page.



**Note:** You may need to clear the browser cache and **restart** Internet Explorer before testing changes to the worker script. To do this, press F12 to show the F12 Developer Tools window, and on the **Cache** menu click **Clear browser cache**.

2. Drag and drop the file **E:\Mod14\Labfiles\Resources\mark-hanson-large.jpg** onto the canvas.
3. Verify that the page is still responsive while the image is being processed.
4. Close Internet Explorer.

**Results:** After completing this exercise, you will have created a web page that remains responsive while slow image processing code runs in a web worker.

MCT USE ONLY. STUDENT USE PROHIBITED

## Module Review and Takeaways

In this module, you have learned how to use web workers to implement parallel processing in a web application. A web worker runs a piece of JavaScript code in parallel to the web page; they may be implemented as separate threads or processes. A web page communicates with a web worker by sending and receiving messages.

There are two forms of web worker:

- A dedicated web worker can only be referenced by the page that created it, and its lifetime is tied to that of the web page.
- A shared web worker can be accessed by any web page in the web application that created it.

### Review Question(s)

Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
Shared web workers can share data in the DOM of a web page, but dedicated web workers cannot. True or false?	

**Question:** How does a web page communicate with a web worker?

## Course Evaluation

Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

MCT USE ONLY. STUDENT USE PROHIBITED



# Module 1: Overview of HTML and CSS

## Lab: Exploring the Contoso Conference Application

### Exercise 1: Exploring the Contoso Conference Application

► **Task 1: Start the web application and view the Home page**

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.

 **Note:** If necessary, click **Switch User** to display the list of users.

3. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
4. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
5. In the **Open Project** dialog box, browse to **E:\Mod01\Labfiles\Starter**, click **ContosoConf.sln**, and then click **Open**.
6. On the **Debug** menu, click **Start Without Debugging**.
7. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
8. Verify that the **Home** page appears in Internet Explorer.

The **Home** page is displayed in Internet Explorer, like this:



**FIGURE 1.1:THE HOME PAGE**

9. Scroll to the bottom of the **Home** page.
10. Click **Play** and verify that the video starts.

11. Click **Pause**.
12. Scroll to the top of the **Home** page.
13. Hover the mouse over the **Register Free** icon and verify that the icon rotates and expands.
14. At the very top of the page, move the mouse over the items in the menu bar but do not click any of them.
15. Verify that each item is highlighted as the mouse traverses the item.

### ► Task 2: View the About and Schedule pages

1. On the menu bar, click **About**.
2. Verify that the **About** page appears, and that the style of the **About** item in the menu bar changes to indicate that the **About** page has been selected.

The **About** page looks like this:

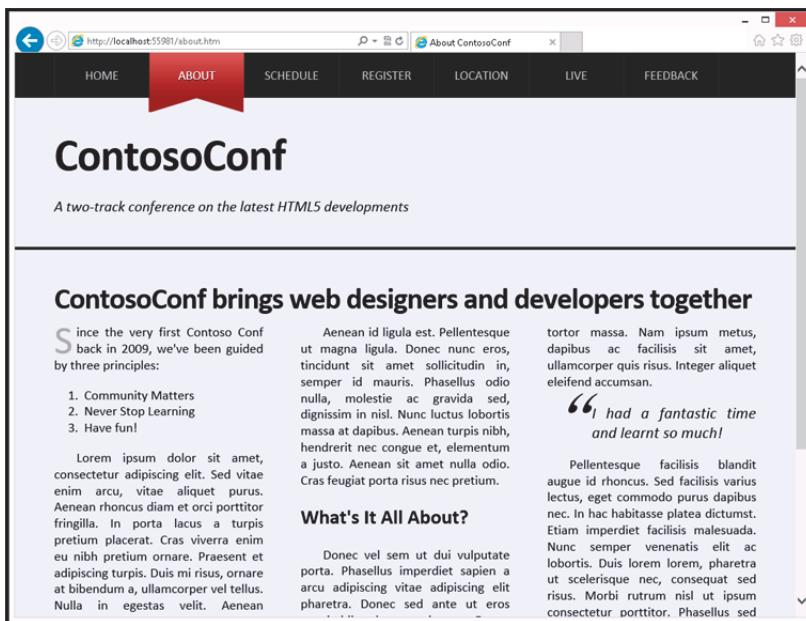
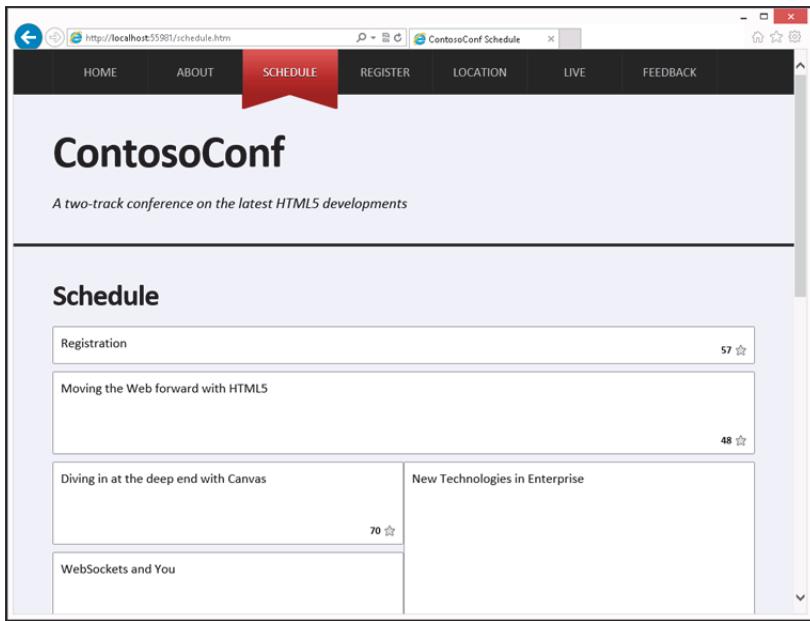


FIGURE 1.2:THE ABOUT PAGE

3. On the menu bar, click **Schedule**.
4. Verify that the **Schedule** page appears.

The **Schedule** page looks like this:



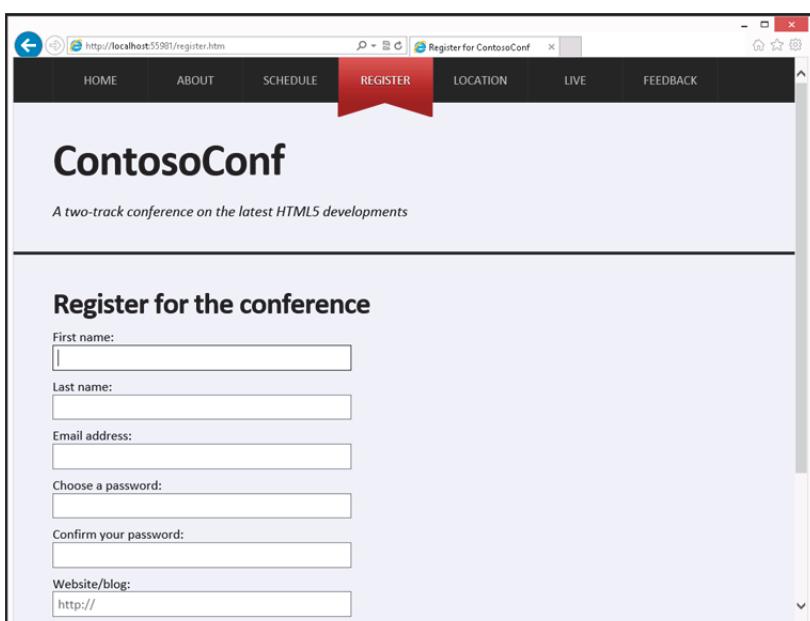
**FIGURE 1.3:THE SCHEDULE PAGE**

5. In the box for the session named **Moving the Web forward with HTML5**, click the star icon.
6. Verify that the star turns yellow, and that the number of interested attendees increases by 1.
7. Click the star icon again.
8. Verify that the star turns white, and that the number of interested attendees decreases by 1.

► **Task 3: View the Register page and register as a new attendee**

1. On the menu bar, click **Register**.
2. Verify that the **Register** page appears.

The **Register** page looks like this:



3. Click **Register** and verify that all fields except **Website/blog** are highlighted and that the message **This is a required field** appears below the **First name** field.
4. In the **First name** field, type **Eric**.
5. In the **Last name** field, type **Gruber**.
6. In the **Email address** field, type **dummy data**, and then click **Register**.
7. Verify that the message **You must enter a valid email address** appears below the **Email address** field.
8. In the **Email address** field, type **grubere@contoso.com**.
9. In the **Choose a password** field, type **abcd**, and then click **Register**.
10. Verify that the message **You must use this format: At least 5 letters and numbers** appears below the **Choose a password** field.
11. In the **Choose a password** field, type **abc1234**.
12. In the **Confirm your password** field, type **wxyz9999**, and then click **Register**.
13. Verify that the message **Your passwords don't match. Please type the same password again** appears below the **Confirm your password** field.
14. In the **Confirm your password** field, type **abc1234**, and then click **Register**.
15. Verify that the **Thanks for registering** confirmation page appears.

The confirmation page looks like this:

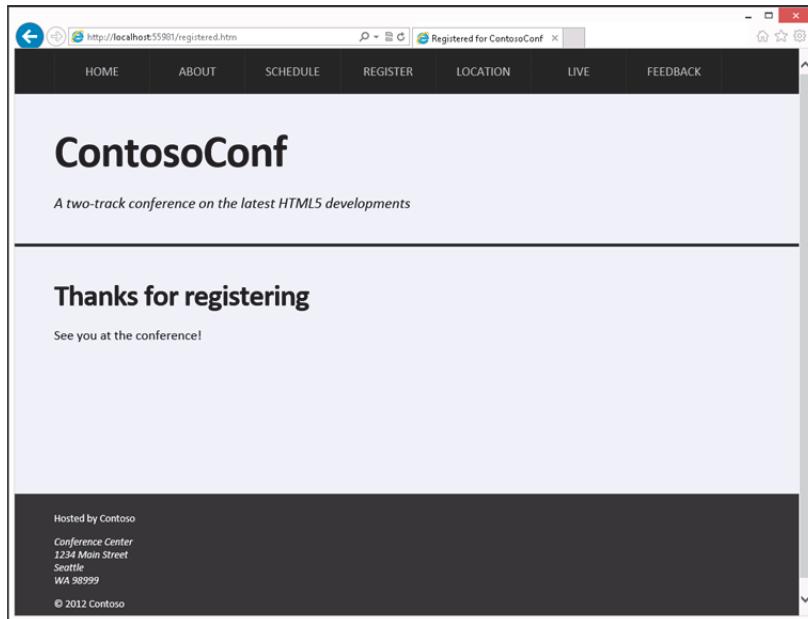


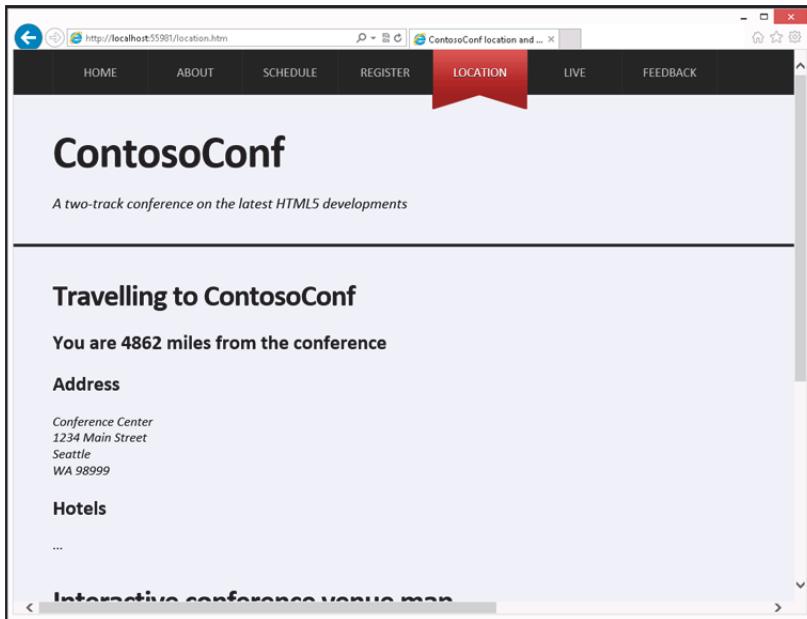
FIGURE 1.5:THE REGISTRATION CONFIRMATION PAGE

► **Task 4: View the Location page**

1. On the menu bar, click **Location**.
2. Verify that the **Location** page appears.
3. If the message **localhost wants to track your physical location** appears in the Internet Explorer message bar, click **Allow once**.

4. In the **Enable Location Services** dialog box, click **Yes**.
5. Verify that your distance from the conference venue is displayed.

The **Location** page looks like this:



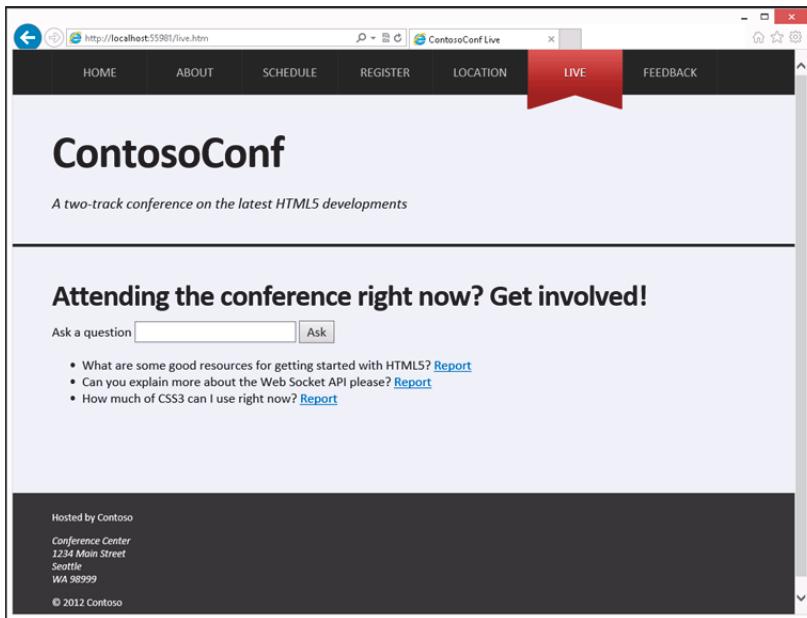
**FIGURE 1.6:THE LOCATION PAGE**

6. Scroll to the bottom of the page.
7. Verify that the conference venue map is displayed.

► **Task 5: Submit a question and provide conference feedback**

1. On the menu bar, click **Live**.
2. Verify that the **Live** page appears.

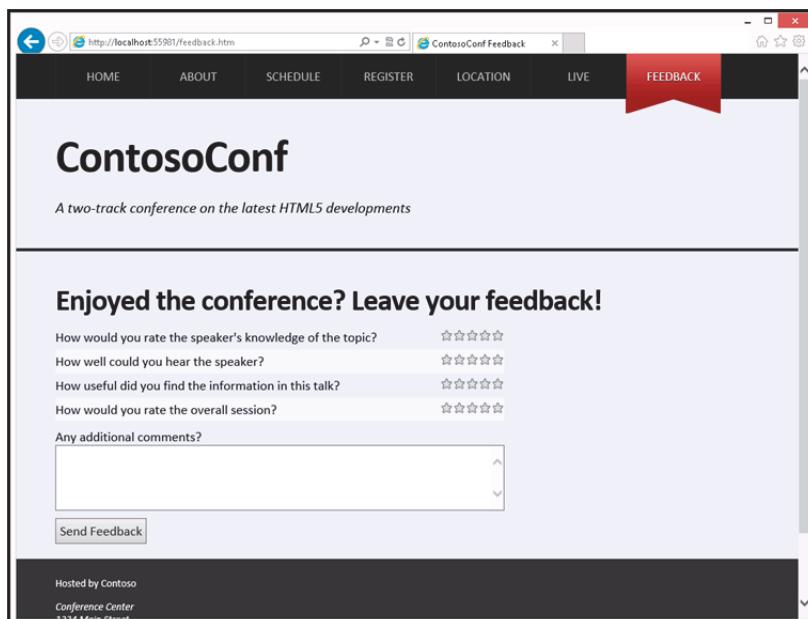
The **Live** page looks like this:



**FIGURE 1.7:THE LIVE PAGE**

3. In the **Ask a question** box, type **What is the best way to learn HTML5?** and then click **Ask**.
4. Verify that the question appears in the list below the box.
5. Click **Report** adjacent to the new question.
6. Verify that the question disappears.
7. On the menu bar, click **Feedback**.
8. Verify that the **Feedback** page appears.

The **Feedback** page looks like this:

**FIGURE 1.8:THE FEEDBACK PAGE.**

9. By the first question, click the third star, and verify that the first three stars are highlighted.
10. By the second question, click the fifth star and verify that all five stars are highlighted.
11. By the remaining two questions click the fourth star and verify that four stars are highlighted.
12. In the **Any additional comments** box, type **Good conference**, and then click **Send Feedback**.
13. Verify that the form flies off the top of the page.
14. Close Internet Explorer.

**Results:** After completing this exercise, you will be able to describe the features of the Contoso Conference web application and list the technologies that are used to implement them.

## Exercise 2: Examining and Modifying the Contoso Conference Application

► **Task 1: Explore the web pages for the application by using Visual Studio 2012**

1. In Visual Studio 2012, in Solution Explorer, expand the **ContosoConf** project.
2. Double-click **index.htm**.
3. In the **index.htm** file, at the start of the **<body>** section, find the **<nav>** element.
4. Verify that the **<nav>** section contains the following HTML markup:

```
<nav class="page-nav">
 <div class="container">
 Home
 About
 Schedule
 Register
 Location
 Live
 Feedback
 </div>
</nav>
```

5. Find the **<section>** element with the **video** class, and verify that this section contains the following HTML markup:

```
<section class="video">
 <h2>Video from last year</h2>
 <video src="http://ak.channel9.msdn.com/ch9/265b/9a76fccd-941e-4285-ad00-9ea200aa265b/MIX09KEY01_high_ch9.mp4"></video>
 <div class="video-controls" style="display: none">
 <button class="video-play">Play</button>
 <button class="video-pause">Pause</button>

 </div>
</section>
```

6. In the **<head>** section near the top of the file, find the **<link>** elements.
7. Verify that the **<head>** section contains the following links:

```
<head>
 ...
 <link href="/styles/site.css" rel="stylesheet" type="text/css" />
 <link href="/styles/nav.css" rel="stylesheet" type="text/css" />
 <link href="/styles/header.css" rel="stylesheet" type="text/css" />
 <link href="/styles/footer.css" rel="stylesheet" type="text/css" />
 <link href="/styles/pages/index.css" rel="stylesheet" type="text/css" />
 <link href="/styles/mobile.css" rel="stylesheet" type="text/css" />
 <link href="/styles/print.css" media="print" rel="stylesheet" type="text/css" />
</head>
```

8. Near the end of the file, find the **<script>** elements.
9. Verify that the following scripts are referenced:

```
<script src="/scripts/offline.js" type="text/javascript"></script>
<script src="/scripts/_namespace.js" type="text/javascript"></script>
<script src="/scripts/datetime.js" type="text/javascript"></script>
<script src="/scripts/pages/video.js" type="text/javascript"></script>
```

10. In Solution Explorer, double-click **about.htm**.
11. In the **about.htm** file, at the start of the **<body>** section, find the **<nav>** section.

12. Verify that the **<nav>** section contains the following HTML markup:

```
<nav class="page-nav">
 <div class="container">
 Home
 About
 Schedule
 Register
 Location
 Live
 Feedback
 </div>
</nav>
```

13. In the **<head>** section near the top of the file, find the **<link>** elements.

14. Verify that the **<head>** section includes the following link:

```
<head>
 ...
 <link href="/styles/pages/about.css" rel="stylesheet" type="text/css" />
 ...
</head>
```

15. In Solution Explorer, double-click **schedule.htm**.

16. In the schedule.htm file, find the **<section class="page-section schedule">** element.

17. Verify that this element contains the following HTML markup:

```
<section class="page-section schedule">
 <div class="container">
 <h1>Schedule</h1>
 <ul id="schedule">
 </div>
</section>
```

18. Verify that the schedule.htm file contains the **<script src="/scripts/pages/schedule.js" type="text/javascript">** element near the end.

19. In Solution Explorer, double-click **register.htm**.

20. In the register.htm file, find the **<section class="page-section register">** element.

21. Verify that this element contains the following HTML markup:

```
<section class="page-section register">
 <div class="container">
 <h1>Register for the conference</h1>
 <form method="post" action="/registration/new" id="registration-form">
 <div class="field">
 <label for="first-name">First name:</label>
 <input type="text" id="first-name" name="FirstName" required="required" autofocus="autofocus"/>
 </div>
 <div class="field">
 <label for="last-name">Last name:</label>
 <input type="text" id="last-name" name="LastName" required="required"/>
 </div>
 <div class="field">
 <label for="email-address">Email address:</label>
 <input type="email" id="email-address" name="EmailAddress" required="required"/>
 </div>
 <div class="field">
 </div>
 </div>
```

```

<label for="password">Choose a password:</label>
<input type="password" id="password" name="Password" required="required"
pattern="[a-zA-Z0-9]{5,}" title="At least 5 letters and numbers"/>
</div>
<div class="field">
 <label for="confirm-password">Confirm your password:</label>
 <input type="password" id="confirm-password" name="ConfirmPassword"
required="required"/>
</div>
<div class="field">
 <label for="website">Website/blog:</label>
 <input type="url" id="website" name="WebsiteUrl" placeholder="http://"/>
</div>
<div>
 <button type="submit">Register</button>
</div>
</form>
</div>
</section>

```

22. In Solution Explorer, double-click **location.htm**.
23. Verify that the location.htm file contains the **<script src="/scripts/pages/location.js" type="text/javascript">** element near the end.
24. Find the **<section class="travel">** element.
25. Verify that this element contains the following HTML markup:

```

<section class="travel">
 <h1>Travelling to ContosoConf</h1>
 <h2 id="distance"></h2>
 ...
</section>

```

26. Find the **<section class="venue">** element.
27. Verify that this element contains the following HTML markup:

```

<section class="venue">
 <h1>Interactive conference venue map</h1>
 <div id="venue-map">
 <svg viewBox="-1 -1 302 102" width="100%" height="230">
 <!-- Room A -->
 <g id="room-a" class="room">
 <rect fill="#ffff" x="0" y="0" width="100" height="100"/>
 <text x="13" y="55" font-weight="bold" font-size="20">ROOM A</text>
 </g>
 <!-- Room B -->
 <g id="room-b" class="room">
 <rect fill="#ffff" x="200" y="0" width="100" height="100"/>
 <text x="213" y="55" font-weight="bold" font-size="20">ROOM B</text>
 </g>
 <!-- The outline of the building -->
 <polyline fill="none" stroke="#000" points="135,95 140,100 0,100 0,0 100,0
100,80 130,80 130,70 110,70 110,30 190,30 190,70 170,70 170,80 200,80 200,0 300,0
300,100 160,100 165,95"/>
 <text x="150" y="55" font-size="12" style="text-anchor:
middle">Registration</text>
 </svg>
 ...
 </div>
</section>

```

28. In Solution Explorer, double-click **live.htm**.

29. Find the **<section class="page-section live">** element.
30. Verify that this section contains the following HTML markup:

```
<section class="page-section live">
 <div class="container">
 <h1>Attending the conference right now? Get involved!</h1>
 <form action="#">
 <label for="ask-question-text">Ask a question</label>
 <input id="ask-question-text" type="text" />
 <button type="submit">Ask</button>
 </form>

 <!-- Questions will be displayed here when received by the web socket. -->

 </div>
</section>
```

31. Verify that the live.htm file contains the following **<script>** element near the end:

```
<script src="/scripts/pages/live.js" type="text/javascript">
```

32. In Solution Explorer, double-click **feedback.htm**.
33. Find the **<section class="page-section feedback">** element.
34. Verify that this section contains the following HTML form:

```
<form method="post" action="/send-feedback">
 <div class="field feedback-question">
 <label>How would you rate the speaker's knowledge of the topic?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How well could you hear the speaker?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How useful did you find the information in this talk?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How would you rate the overall session?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field comments">
 <label>Any additional comments?</label>
 <textarea name="comments" cols="30" rows="5"></textarea>
 </div>
 <div class="field actions">
 <button type="submit">Send Feedback</button>
 </div>
</form>
```

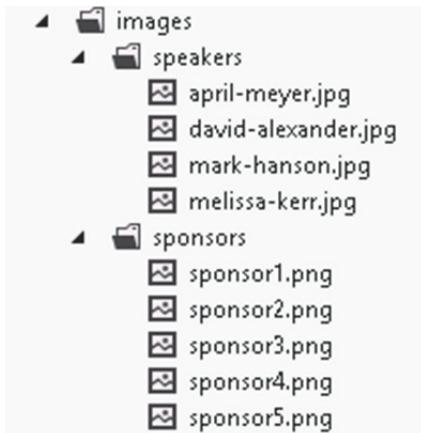
35. Verify that the feedback.htm file contains the following **<script>** elements near the end:

```
<script src="/scripts/StarRatingView.js" type="text/javascript">
<script src="/scripts/pages/feedback.js" type="text/javascript">
```

### ► Task 2: Explore the structure of the project

1. In Solution Explorer, expand the **images** folder and verify that it contains the following folders and files.

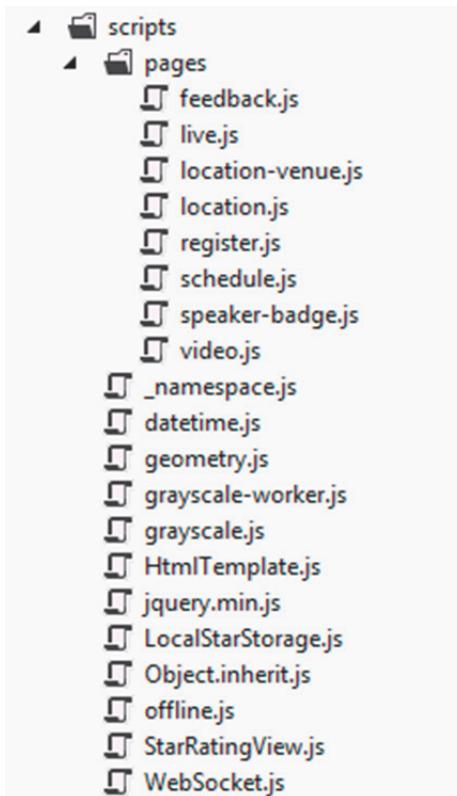
The **images** folder should contain these folders and files:



**FIGURE 1.9:FILES IN THE IMAGES FOLDER**

2. Expand the **scripts** folder.

Verify that this folder contains the following folders and files:



**FIGURE 1.10:FILES IN THE SCRIPTS FOLDER**

3. Expand the **styles** folder.

Verify that this folder contains the following folders and files:

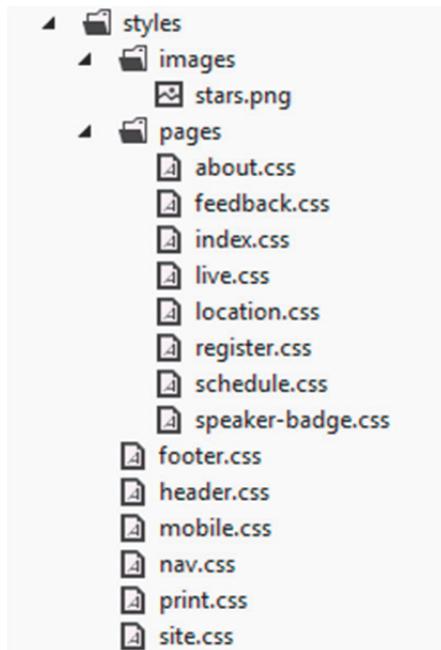


FIGURE 1.11:FILES IN THE STYLES FOLDER

► **Task 3: Run the application and make live modifications**

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Switch to Visual Studio 2012.
4. In Solution Explorer, find the following markup on the **index.htm** page:

```
· Free ·
```

5. Change the word **Free** to **Now**, as shown below in bold:

```
· Now ·
```

6. In Solution Explorer, in the **styles** folder, double-click **nav.css**.
7. Find the following style:

```
nav.page-nav {
 background-color: #1d1d1d;
 line-height: 6rem;
 font-size: 1.7rem;
}
```

8. Change the value of the **background-color** property to **blue**, as shown below in bold:

```
nav.page-nav {
 background-color: blue;
 line-height: 6rem;
 font-size: 1.7rem;
}
```

9. On the **File** menu, click **Save All**.
10. Return to Internet Explorer and press F5 on the keyboard.

11. Verify that the Register button now contains the text **Register Now** and that the background color of the menu bar is blue.
12. Close Internet Explorer.

**Results:** After completing this exercise, you will be able to describe how the Contoso Conference application is structured as a Visual Studio 2012 project.



# Module 2: Creating and Styling HTML Pages

## Lab: Creating and Styling HTML5 Pages

### Exercise 1: Creating HTML5 Pages

#### ► Task 1: Create a new ASP.NET web application

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.

 **Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
6. In the **New Project** dialog box, in the left pane, under **Templates** expand the **Visual C#** node, and then click the **Web** node.
7. In the middle pane, click **ASP.NET Empty Web Application**.
8. In the **Name** box, type **ContosoConf**.
9. In the **Location** box, type **E:\Mod02\Labfiles\Starter\Exercise 1**, and then click **OK**.

#### ► Task 2: Add the Home page

1. In Visual Studio, on the **Project** menu, click **Add New Item**.
2. In the **Add New Item – ContosoConf** dialog box, click **HTML Page**.
3. In the **Name** box, type **index.htm**.
4. Click **Add**.
5. Using Notepad, open **index.txt** file located in the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder.
6. In Visual Studio, return to the Design View window displaying the **Source** view for the index.htm file.
7. Inside the **<title>** element, add the following text from the header section of the index.txt file:

ContosoConf

8. Inside the **<body>** element, add the following HTML markup, with the text also from the header section of the index.txt file:

```
<header>
 <h1>ContosoConf</h1>
 <p>A two-track conference on the latest HTML5 developments</p>
</header>
```

9. After the **</header>** element and before the **</body>** element, add the following HTML markup with the text taken from the content section of the index.txt file:

```
<section>
 <p>
```

```
 The web keeps evolving. There's a wealth of new technologies to keep up
with! ContosoConf is an in-depth exploration of HTML5, CSS3 and JavaScript, in the
heart of Seattle, WA.
 </p>
 <p>
 Don't miss the best speakers in the business, talking about the future of
the web.
 </p>
</section>
```

10. After the body text that you have just added but before the **</body>** element, add the following HTML markup with the text also taken from the content section of the index.txt file:

```
<section>
 <h2>Featuring these excellent speakers</h2>

 Melissa Kerr
 David Alexander
 Mark Hanson
 April Meyer

</section>
```

11. After the body text that you have just added but before the **</body>** element, add the following HTML markup with the text also taken from the content section of the index.txt file:

```
<section>
 <h2>Thanks to our sponsors</h2>

 Trey Research
 Litware, Inc
 Fourth Coffee
 Graphic Design Institute
 Southridge Video

</section>
```

12. After the body text that you have just added but before the **</body>** element, add the following HTML markup with the text taken from the footer section of the index.txt file:

```
<footer>
 <p>
 Hosted by Contoso
 </p>
 <address>
 Conference Center

 3 Somewhere Street

 Seattle

 WA 98999
 </address>
 <p>
 © 2012 Contoso
 </p>
</footer>
```

-  **Note:** Be sure to replace the © symbol with the value ©;

13. Close Notepad.

### ► Task 3: Add images to the Home Page

1. On the Windows Taskbar, click **File Explorer**.
2. In Internet Explorer, browse to the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder.
3. Drag and drop the **images** folder onto the ContosoConf project in the Visual Studio Solution Explorer window.
4. In the Design View window displaying the **Source** view for the index.htm file, replace the text **Melissa Kerr** with the following HTML markup:

```

Melissa Kerr
```

5. Replace the text **David Alexander** with the following HTML markup:

```

David Alexander
```

6. Replace the text **Mark Hanson** with the following HTML markup:

```

Mark Hanson
```

7. Replace the text **April Meyer** with the following HTML markup:

```

April Meyer
```

8. Replace the text **Trey Research** with the following HTML markup:

```

```

9. Replace the text **Litware, Inc** with the following HTML markup:

```

```

10. Replace the text **Fourth Coffee** with the following HTML markup:

```

```

11. Replace the text **Graphic Design Institute** with the following HTML markup:

```

```

12. Replace the text **Southridge Video** with the following HTML markup:

```

```

### ► Task 4: Add the About page

1. In Visual Studio, on the **Project** menu, click **Add New Item**.
2. In the **Add New Item – ContosoConf** dialog box, click **HTML Page**.
3. In the **Name** box, type **about.htm**.
4. Click **Add**.

5. Using Notepad, open **about.txt** file located in the **E:\Mod02\Labfiles\Starter\Exercise 1\Resources** folder.
6. In Visual Studio, return to the Design View window displaying the **Source** view for the about.htm file.
7. Inside the **<title>** element, add the following text from the header section of the about.txt file:

```
About ContosoConf
```

8. Inside the **<body>** element, add the following HTML markup copied from the index.htm page:

```
<header>
 <h1>ContosoConf</h1>
 <p>A two-track conference on the latest HTML5 developments</p>
</header>
```

9. After the **</header>** element and before the **</body>** element, add the following HTML markup with the text taken from the **about.txt** file:

```
<article>
 <h1>ContosoConf brings web designers and developers together</h1>
 <section>
 <p>
 Since the very first Contoso Conf back in 2009, we've been guided by
 three principles:
 </p>

 Community Matters
 Never Stop Learning
 Have fun!

 <p>
 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae enim
 arcu, vitae aliquet purus.
 Aenean rhoncus diam et orci porttitor fringilla. In porta lacus a turpis
 pretium placerat. Cras viverra
 enim eu nibh pretium ornare. Praesent et adipiscing turpis. Duis mi
 risus, ornare at bibendum a,
 ullamcorper vel tellus. Nulla in egestas velit. Aenean consequat mi sed
 tellus iaculis laoreet. Donec et
 odio vel felis commodo porttitor.
 </p>
 <p>
 Aenean id ligula est. Pellentesque ut magna ligula. Donec nunc eros,
 tincidunt sit amet sollicitudin
 in, semper id mauris. Phasellus odio nulla, molestie ac gravida sed,
 dignissim in nisl. Nunc luctus
 lobortis massa at dapibus. Aenean turpis nibh, hendrerit nec congue et,
 elementum a justo. Aenean sit
 amet nulla odio. Cras feugiat porta risus nec pretium.
 </p>
 <h2>What's It All About?</h2>
 <p>
 Donec vel sem ut dui vulputate porta. Phasellus imperdiet sapien a arcu
 adipiscing vitae adipiscing
 elit pharetra. Donec sed ante ut eros mattis bibendum non in erat. Donec
 sagittis, massa eu accumsan
 eleifend, eros justo cursus justo, id consequat mauris diam id magna.
 Vivamus quis tortor massa. Nam ipsum metus,
 dapibus ac facilisis sit amet, ullamcorper quis risus. Integer aliquet
 eleifend accumsan.
 </p>
 <blockquote>I had a fantastic time and learnt so much!</blockquote>
 <p>
```

```

Pellentesque facilisis blandit augue id rhoncus. Sed facilisis varius
lectus, eget commodo purus dapibus
nec. In hac habitasse platea dictumst. Etiam imperdiet facilisis
malesuada. Nunc semper venenatis elit ac
lobortis. Duis lorem lorem, pharetra ut scelerisque nec, consequat sed
risus. Morbi rutrum nisl ut ipsum
consectetur porttitor. Phasellus sed nunc id diam tempus congue in a
leo.
</p>
<p>
 Proin feugiat, turpis id tempor tempor, lorem libero malesuada.
</p>
</section>
</article>
```

10. Inside the **<body>** element, add the following HTML markup copied from the index.htm page:

```

<footer>
 <p>
 Hosted by Contoso
 </p>
 <address>
 Conference Center

 3 Somewhere Street

 Seattle

 WA 98343
 </address>
 <p>
 © 2012 Contoso
 </p>
</footer>
```

11. Close Notepad.

#### ► Task 5: Add navigation links

1. In Solution Explorer, double-click **index.htm**.
2. In the **<body>** element, before the **<header>** element, insert the following HTML markup:

```

<nav>
 Home
 About
</nav>
```

3. In Solution Explorer, double-click **about.htm**.
4. In the **<body>** element, before the **<header>** element, insert the following HTML markup:

```

<nav>
 Home
 About
</nav>
```

#### ► Task 6: Run the web application

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
4. Verify that the Home page is displayed.

5. Click the **About** link.
6. Verify that the About page is displayed.
7. Click the **Home** link.
8. Verify that the Home page is displayed.
9. Close Internet Explorer and return to Visual Studio.

**Results:** After completing this exercise, you will have built a simple HTML5 web application with a Home page and an About page.

## Exercise 2: Styling HTML pages

### ► Task 1: Create a new style sheet

1. In the Solution Explorer, click the **ContosoConf** project node.
2. On the **Project** menu, click **New Folder**.
3. In Solution Explorer, change the name of the folder to **styles**.
4. On the **Project** menu, click **Add New Item**.
5. In the **Add New Item – ContosoConf** dialog box, click **Style Sheet**.
6. In the **Name** box, type **site.css**.
7. Click **Add**.
8. In Solution Explorer, double-click **index.htm**.
9. After the title but before the **</head>** element, insert the following HTML markup:

```
<link href="/styles/site.css" rel="stylesheet" type="text/css" />
```

10. In Solution Explorer, double-click **about.htm**.
11. After the title but before the **</head>** element, insert the following HTML markup:

```
<link href="/styles/site.css" rel="stylesheet" type="text/css" />
```

### ► Task 2: Add CSS rules to style the pages

1. In the Visual Studio Solution Explorer, double-click **site.css**.
2. Delete all existing CSS content.
3. Insert the following CSS code:

```
html {
 /* font-size 62.5% makes 1rem equal 10px for easy size calculations. */
 font-size: 62.5%;
 font-family: Calibri, Arial, sans-serif;
 background-color: #EAEFFA;
}
body {
 margin: 0;
 /* Default font-size to about 18px */
 font-size: 1.8rem;
}
.container {
 padding: 0 1rem;
 max-width: 94rem;
 /* Horizontally center containers */
 margin: 0 auto;
}
nav {
 background-color: #1d1d1d;
 line-height: 6rem;
 font-size: 1.7rem;
}
nav a {
 color: #fff;
 padding: 1rem;
}
h1 {
 font-size: 4rem;
 letter-spacing: -1px;
```

```
 margin: 1em 0 .25em 0;
}
```

4. In Solution Explorer, double-click **index.htm**.
5. Replace the contents of the **<nav>** element with the following HTML markup:

```
<div class="container">
 Home
 About
</div>
```

6. After the **</nav>** element and before the **<header>** element, insert the following HTML markup:

```
<div class="container">
```

7. Before the **</body>** element, insert the following HTML markup to close the **<div>** element:

```
</div>
```

8. In Solution Explorer, double-click **about.htm**.
9. Replace the contents of the **<nav>** element with the following HTML markup:

```
<div class="container">
 Home
 About
</div>
```

10. After the **</nav>** element and before the **<header>** element, insert the following HTML markup:

```
<div class="container">
```

11. Before the **</body>** element, insert the following HTML markup to close the **<div>** element:

```
</div>
```

### ► Task 3: Run the web application

1. On the **Debug** menu, click **Start Without Debugging**.
2. In Internet Explorer, verify that the About page matches the styling shown by the image in the previous task.
3. Move to the Home page, and verify that the styling is consistent with the About page.
4. Close Internet Explorer.

**Results:** After completing this exercise, you will have used CSS rules to style the Home and About pages.

## Module 3: Introduction to JavaScript

# Lab: Displaying Data and Handling Events by Using JavaScript.

### Exercise 1: Displaying Data Programmatically

► **Task 1:** Review the existing code for the Schedule page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod03\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project node, and then double-click **schedule.htm**.
8. Review the contents of this file and verify that the **schedule** page section contains the following markup:

```
<section class="page-section schedule">
 <div class="container">
 <h1>Schedule</h1>
 <ul id="schedule">
 </div>
</section>
```

9. Verify that the file contains the following HTML markup towards the end:

```
<script src="/scripts/pages/schedule.js" type="text/javascript"></script>
```

10. In Solution Explorer, expand the **scripts** folder.
11. Expand the **pages** sub-folder.
12. Double-click **schedule.js**.
13. Review the contents of this file and verify that the first few lines contain the following code:

```
var schedule = [
 {
 "id": "session-1",
 "title": "Registration",
 "tracks": [1, 2]
 },
 {
 "id": "session-2",
 "title": "Moving the Web forward with HTML5",
 "tracks": [1, 2]
 },
 {
```

```
 "id": "session-3",
 "title": "Diving in at the deep end with Canvas",
 "tracks": [1]
 },
 {
 "id": "session-4",
 "title": "New Technologies in Enterprise",
 "tracks": [2]
 },
 ...
];
```

► **Task 2: Write code to get the schedule list element on the Schedule page**

1. In Solution Explorer, double-click **schedule.js**.
2. After the line containing the **TODO: Task 2** comment, add the following line of JavaScript:

```
var list = document.getElementById("schedule");
```

► **Task 3: Implement the createSessionElement function that creates the list item for a session**

1. In **schedule.js**, after the line containing the **TODO: Task 3** comment, add the following JavaScript code:

```
var li = document.createElement("li");
li.textContent = session.title;
return li;
```

► **Task 4: Implement the displaySchedule function that adds session items to the list for display**

1. In **schedule.js**, after the line containing the **TODO: Task 4** comment, add the following JavaScript code:

```
for (var i = 0; i < schedule.length; i++) {
 var li = createSessionElement(schedule[i]);
 list.appendChild(li);
}
```

► **Task 5: Run the web application and view the Schedule page**

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
4. Verify that the list of sessions is displayed.
5. Close Internet Explorer.

**Results:** After completing this exercise, you will have added a Schedule page to the ContosoConf application that displays the details of conference sessions.

## Exercise 2: Handling Events

► **Task 1: Add checkbox HTML elements**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod03\Labfiles\Start\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project and then double-click **schedule.htm**.
4. Find the line containing the following HTML markup:

```
<ul id="schedule">
```

5. Before this line, insert the following HTML markup:

```
Show:
<input type="checkbox" id="show-track-1" checked="checked"/><label for="show-track-1">Track 1</label>
<input type="checkbox" id="show-track-2" checked="checked"/><label for="show-track-2">Track 2</label>
```

► **Task 2: Write code to get the checkbox elements from the Schedule page**

1. In Solution Explorer, expand the **scripts** folder, expand the **pages** sub-folder, and then double-click **schedule.js**
2. In **schedule.js**, find the line containing the following JavaScript code:

```
var list = document.getElementById("schedule");
```

3. After this line, add the following JavaScript code:

```
var track1CheckBox = document.getElementById("show-track-1");
var track2CheckBox = document.getElementById("show-track-2");
```

► **Task 3: Add click event listeners for each checkbox**

1. In **schedule.js**, add the following JavaScript code to the end of the file:

```
track1CheckBox.addEventListener("click", displaySchedule, false);
track2CheckBox.addEventListener("click", displaySchedule, false);
```

► **Task 4: Update the displaySchedule function to display the sessions for selected tracks**

1. In the **schedule.js** file, in the **displaySchedule** function, replace the body of the **for** loop with the following JavaScript code:

```
var tracks = schedule[i].tracks;
var isCurrentTrack = (track1CheckBox.checked && tracks.indexOf(1) >= 0) ||
 (track2CheckBox.checked && tracks.indexOf(2) >= 0);
if (isCurrentTrack) {
 var li = createSessionElement(schedule[i]);
 list.appendChild(li);
}
```

► **Task 5: Run the web application and view the Schedule page**

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.

3. In Internet Explorer, verify that both check boxes are selected and that the complete list of sessions is displayed.
4. Clear **Track 1** and verify that only the sessions for Track 2 appear.
5. Select **Track 1**, clear **Track 2**, and verify that only the sessions for Track 1 appear.
6. Clear **Track 1** and verify that no sessions appear.
7. Close Internet Explorer.

**Results:** After completing this exercise, you will have updated the **Schedule** page to filter sessions based on which tracks have been selected.

# Module 4: Creating Forms to Collect and Validate User Input

## Lab: Creating a Form and Validating User Input

### Exercise 1: Creating a Form and Validating User Input by Using HTML5 Attributes

#### ► Task 1: Modify the Register page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If it is necessary, click **Switch User** to display the list of users

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project, and then double-click **register.htm**.
8. Find and review the comment that starts with the following text:

TODO: Add form inputs

9. In the **<head>** element, verify that the following HTML markup is present:

```
<link href="/styles/pages/register.css" rel="stylesheet" type="text/css" />
```

10. Before the **</body>** tag near the end of the file, verify that the following HTML markup is present:

```
<script src="/scripts/pages/register.js" type="text/javascript"></script>
```

#### ► Task 2: Add form inputs to the Register page

1. In the **register.htm** file, delete the following HTML markup:

```
<!-- Use the following template for the inputs -->
<div class="field">
 <label for="{input-id}">label:</label>
 <input type="{type}" id="{input-id}" name="{input-name}" />
</div>
```

2. Add the following HTML markup in place of the code that you deleted in step 1:

```
<div class="field">
 <label for="first-name">First name:</label>
 <input type="text" id="first-name" name="FirstName" />
</div>
<div class="field">
 <label for="last-name">Last name:</label>
 <input type="text" id="last-name" name="LastName" />
```

```
</div>
<div class="field">
 <label for="email-address">Email address:</label>
 <input type="email" id="email-address" name="EmailAddress" />
</div>
<div class="field">
 <label for="password">Choose a password:</label>
 <input type="password" id="password" name="Password" />
</div>
<div class="field">
 <label for="confirm-password">Confirm your password:</label>
 <input type="password" id="confirm-password" name="ConfirmPassword" />
</div>
<div class="field">
 <label for="website">Website/blog:</label>
 <input type="url" id="website" name="WebsiteUrl" />
</div>
```

3. On the **Debug** menu, click **Start Without Debugging**.
4. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
5. In the **First name** box, type **Josh**.
6. In the **Last name** box, type **Bailey**.
7. In the **Email address** box, type **josh.bailey@adatum.com**.
8. In the **Choose a password** box, type **Passw0rd**.
9. In the **Confirm your password** box, type **Passw0rd**.
10. In the **Website/blog** box, type **http://adatum.com/**.
11. Click **Register**.
12. Verify that the **Thanks for registering** page is displayed.
13. In the page header, click the **Register** link to return to the **Register** page.
14. Leave the fields empty, and then click **Register**.
15. Verify that the following messages are displayed:

Invalid registration information  
The FirstName field is required.  
The LastName field is required.  
The EmailAddress field is required.  
The Password field is required.  
The ConfirmPassword field is required.

16. Close Internet Explorer.

#### ► Task 3: Make the form more user friendly

1. In Visual Studio, in Solution Explorer, double-click **register.htm**.
2. Find the line containing the following HTML markup:

```
<input type="text" id="first-name" name="FirstName" />
```

3. Add the **autofocus** attribute shown below in bold to this line:

```
<input type="text" id="first-name" name="FirstName" autofocus="autofocus" />
```

4. Find the line containing the following HTML markup:

```
<input type="url" id="website" name="WebsiteUrl" />
```

5. Add the **placeholder** attribute shown below in bold to this line:

```
<input type="url" id="website" name="WebsiteUrl" placeholder="http://" />
```

6. On the **Debug** menu, click **Start Without Debugging**.

7. In Internet Explorer, verify that the cursor is placed in the **First name** box.

8. Verify that the **Website/blog** box has the placeholder text **http://**.

9. Close Internet Explorer.

#### ► Task 4: Check for missing mandatory data

1. In Solution Explorer, double-click **register.htm**.

2. Find the line containing the following HTML markup:

```
<input type="text" id="first-name" name="FirstName" autofocus="autofocus" />
```

3. Add the **required** attribute shown below in bold to this line:

```
<input type="text" id="first-name" name="FirstName" required="required" autofocus="autofocus" />
```

4. Find the line containing the following HTML markup:

```
<input type="text" id="last-name" name="LastName" />
```

5. Add the **required** attribute shown below in bold to this line:

```
<input type="text" id="last-name" name="LastName" required="required" />
```

6. Find the line containing the following HTML markup:

```
<input type="email" id="email-address" name="EmailAddress" />
```

7. Add the **required** attribute shown below in bold to this line:

```
<input type="email" id="email-address" name="EmailAddress" required="required" />
```

8. Find the line containing the following HTML markup:

```
<input type="password" id="password" name="Password" />
```

9. Add the **required** attribute shown below in bold to this line:

```
<input type="password" id="password" name="Password" required="required" />
```

10. Find the line containing the following HTML markup:

```
<input type="password" id="confirm-password" name="ConfirmPassword" />
```

11. Add the **required** attribute shown below in bold to this line:

```
<input type="password" id="confirm-password" name="ConfirmPassword" required="required" />
```

12. On the **Debug** menu, click **Start Without Debugging**.
13. In Internet Explorer, click **Register** without entering any data.
14. Verify that each mandatory field is highlighted and that the form displays the following error message:

This is a required field

15. In the **First name** box, type **Josh**.
16. In the **Last name** box, type **Bailey**.
17. In the **Email address** box, type **josh.bailey@adatum.com**.
18. In the **Choose a password** box, type **Passw0rd**.
19. In the **Confirm your password** box, type **Passw0rd**.
20. In the **Website/blog** box, type **http://adatum.com/**.
21. Click **Register**, and verify that the **Thanks for registering** page appears.
22. Close Internet Explorer.

#### ► Task 5: Add password complexity validation

1. In Solution Explorer, double-click **register.htm**.
2. Find the line containing the following HTML markup:

```
<input type="password" id="password" name="Password" required="required" />
```

3. Add the **pattern** and **title** attributes shown below in bold to this line:

```
<input type="password" id="password" name="Password" required="required" pattern="[a-zA-Z0-9]{5,}" title="At least 5 letters and numbers" />
```

4. On the **Debug** menu, click **Start Without Debugging**.
5. In Internet Explorer, in the **First name** box, type **Josh**.
6. In the **Last name** box, type **Bailey**.
7. In the **Email address** box, type **josh.bailey@adatum.com**.
8. In the **Choose a password** box, type **abc**.
9. In the **Confirm your password** box, type **abc**.
10. Click **Register**.
11. Verify that the **Choose a password** box displays the following error message:

You must use this format: At least 5 letters and numbers

12. In the **Choose a password** box, type **Passw0rd**.
13. In the **Confirm your password** box, type **Passw0rd**.
14. In the **Website/blog** box, type **http://adatum.com/**.
15. Click **Register**, and verify that the **Thanks for registering** page appears.
16. Close Internet Explorer.

**Results:** After completing this exercise, you will have modified the attendee registration page to validate the data entered by attendees.

## Exercise 2: Validating User Input by Using JavaScript

► **Task 1: Write code to get the contents of the password input elements**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Start\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, and then expand the **pages** folder.
4. Double-click **register.js**.
5. Find the following comment:

```
// TODO: Task 1 - Get the password <input> elements from the DOM by ID
```

6. After the comment, add the following JavaScript code:

```
var passwordInput = document.getElementById("password");
var confirmPasswordInput = document.getElementById("confirm-password");
```

► **Task 2: Write code to compare the password and confirm-password inputs**

1. In the **register.js** file, find the line containing the following comment:

```
// TODO: Task 2 - Compare passwordInput value to confirmPasswordInput value
```

2. After the comment, add the following JavaScript code:

```
var passwordsMatch = passwordInput.value === confirmPasswordInput.value;
```

► **Task 3: Write code to display a custom error message if the passwords differ**

1. In the **register.js** file, find the comment that starts with the following text:

```
// TODO: Task 3
```

2. After this comment, add the following JavaScript code:

```
if (passwordsMatch) {
 // Clear any previous error message.
 confirmPasswordInput.setCustomValidity("");
} else {
 // Setting this error message will prevent the form from being submitted.
 confirmPasswordInput.setCustomValidity("Your passwords don't match. Please type
the same password again.");
}
```

► **Task 4: Add input event listeners to the inputs to call the checkPasswords method**

1. In the **register.js** file, find the comment that starts with the following text:

```
// TODO: Task 4
```

2. After the comment, add the following JavaScript code:

```
passwordInput.addEventListener("input", checkPasswords, false);
confirmPasswordInput.addEventListener("input", checkPasswords, false);
```

3. In Solution Explorer, double-click **register.htm**.

4. On the **Debug** menu, click **Start Without Debugging**.
5. In Internet Explorer, in the **First name** box, type **Josh**.
6. In the **Last name** box, type **Bailey**.
7. In the **Email address** box, type **josh.bailey@adatum.com**.
8. In the **Choose a password** box, type **abc123**.
9. In the **Confirm your password** box, type **abc456**.
10. Click **Register**.

11. Verify that the **Confirm your password** box displays the following error message:

Your passwords don't match. Please type the same password again.

12. In the **Confirm your password** box, type **abc123**.
13. Click **Register**, and verify that the **Thanks for registering** page appears.
14. Close Internet Explorer.

#### ► Task 5: Style elements that are not valid

1. In Solution Explorer, in the **ContosoConf** project, expand the **styles** folder, and then expand the **pages** folder.
2. Double-click **register.css**.
3. Find the comment that starts with the following text:

/\* TODO: Task 5

4. Below the comment, add the following style:

```
.register form.submission-attempted input:invalid {
 background-color: #f9b2b2;
 outline: none;
}
```

5. In Solution Explorer, double-click **register.htm**.
6. On the **Debug** menu, click **Start Without Debugging**.
7. In Internet Explorer, click **Register**.
8. Verify that the **First name**, **Last name**, **Email address**, **Choose a password**, and **Confirm your password** boxes are highlighted with colored backgrounds.
9. Close Internet Explorer.

**Results:** After completing this exercise, you will have modified the registration page to validate password inputs.



# Module 5: Communicating with a Remote Server

## Lab: Communicating with a Remote Data Source

### Exercise 1: Retrieving Data

#### ► Task 1: Review the Schedule page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.

 **Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, and then expand the **pages** folder.
8. Double-click **schedule.js**.
9. Verify that the first line of the file now contains the following JavaScript code:

```
var schedule = [];
```

10. Verify that the **createSessionElement** function contains the following code:

```
function createSessionElement(session) {
 var li = document.createElement("li");
 li.sessionId = session.id;
 var star = document.createElement("a");
 star.setAttribute("href", "#");
 star.setAttribute("class", "star");
 li.appendChild(star);
 var title = document.createElement("span");
 title.textContent = session.title;
 li.appendChild(title);
 return li;
}
```

#### ► Task 2: Create the downloadSchedule function

1. In the **schedule.js** file, find the comment that starts with the following text:

```
// TODO: Create a function called downloadSchedule
```

2. After the last line of this comment, add the following JavaScript code:

```
function downloadSchedule() {
 var request = new XMLHttpRequest();
}
```

3. In the **schedule.js** file, after the previous statement in the **downloadSchedule** function, add the following JavaScript code:

```
request.open("GET", "/schedule/list", true);
request.onreadystatechange = function () {
 if (request.readyState === 4) {
 var response = JSON.parse(request.responseText);
 schedule = response.schedule;
 displaySchedule();
 }
};
request.send();
```

4. Add the following statement to the end of the **schedule.js** file:

```
downloadSchedule();
```

#### ► Task 3: Add error handling to the **downloadSchedule** function

1. In the **schedule.js** file, add the following code shown in bold to the **downloadSchedule** function:

```
function downloadSchedule() {
 var request = new XMLHttpRequest();
 request.open("GET", "/schedule/list", true);
 request.onreadystatechange = function () {
 if (request.readyState === 4) {
 try {
 var response = JSON.parse(request.responseText);
 if (request.status === 200) {
 schedule = response.schedule;
 displaySchedule();
 } else {
 alert(response.message);
 }
 } catch (exception) {
 alert("Schedule list not available.");
 }
 }
 };
 request.send();
}
```

#### ► Task 4: Test the Schedule page

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
4. Verify that the page displays a list of conference sessions.
5. Close Internet Explorer.
6. In Visual Studio, double-click the **schedule.js** file.
7. In the **downloadSchedule** function, find the following statement:

```
request.open("GET", "/schedule/list", true);
```

8. Change the URL in this statement as shown in bold in the following code:

```
request.open("GET", "/schedule/list?fail", true);
```

9. On the **Debug** menu, click **Start Without Debugging**.
10. In Internet Explorer, in the navigation bar click **Schedule**.
11. Verify that the message **Service currently unavailable** appears.
12. In the **Message from webpage** dialog box, click **OK**.
13. Close Internet Explorer.
14. In Visual Studio, in Solution Explorer, double-click the **schedule.js** file.
15. In the **downloadSchedule** function, change the string **/schedule/list?fail** back to **/schedule/list**.
16. On the **File** menu, click **Save All**.

**Results:** After completing this exercise, you will have modified the code for the Schedule page to displays the list of sessions retrieved from a web service, and to handle errors that can occur when communicating with a remote service.

## Exercise 2: Serializing and Transmitting Data

► **Task 1: Send the request to indicate the session that an attendee has selected**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, and then expand the **pages** folder.
4. Double-click **schedule.js**.
5. Find the following function:

```
function saveStar(sessionId, isStarred) {
```

6. After the last line of the **TODO** comment in the body of this function, add the following JavaScript code:

```
var request = new XMLHttpRequest();
request.open("POST", "/schedule/star/" + sessionId, true);
```
7. Add the following JavaScript code to the end of the **saveStar** function:

```
request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
var data = "starred=" + isStarred;
request.send(data);
```

► **Task 2: Handle the web service response**

1. In the **schedule.js** file, find the following line of code in the **saveStar** function:

```
request.open("POST", "/schedule/star/" + sessionId, true);
```

2. After this statement, add the following JavaScript code:

```
if (isStarred) {
 request.onreadystatechange = function() {
 if (request.readyState === 4 && request.status === 200) {
 var response = JSON.parse(request.responseText);
 if (response.starCount > 50) {
 alert("This session is very popular! Be sure to arrive early to get
a seat.");
 }
 }
 };
}
```

► **Task 3: Test the Schedule page**

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, click the star next to **New Technologies in Enterprise**.
4. Verify that the following message alert is displayed:

```
This session is very popular! Be sure to arrive early to get a seat.
```

5. In the **Message from webpage** dialog box, click **OK**.

6. Click the star next to **Diving in at the deep end with Canvas**.
7. Verify that no message alert is displayed.
8. Close Internet Explorer.

**Results:** After completing this exercise, you will have updated the Schedule page to send attendee selections to a web service, and to display a message when a session is popular.

## Exercise 3: Refactoring the Code by Using the jQuery ajax Method

### ► Task 1: Refactor the downloadSchedule function

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 3**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, and then double-click **schedule.htm**.
4. Find the following HTML markup near the end of the file:

```
<script src="/scripts/pages/schedule.js" type="text/javascript"></script>
```
5. Immediately above this line, add the following HTML markup:

```
<script src="/scripts/jquery.min.js" type="text/javascript"></script>
```
6. In Solution Explorer, expand the **scripts** folder, expand the **pages** folder, and then double-click **schedule.js**.
7. In the **schedule.js** file, replace the **downloadSchedule** function with the following JavaScript code:

```
function downloadSchedule() {
 $.ajax({
 type: "GET",
 url: "/schedule/list"
 }).done(function(response) {
 schedule = response.schedule;
 displaySchedule();
 }).fail(function() {
 alert("Schedule list not available.");
 });
}
```

### ► Task 2: Refactor the saveStar function

1. In the **schedule.js** file, replace the **saveStar** function with the following JavaScript code:

```
function saveStar(sessionId, isStarred) {
 $.ajax({
 type: "POST",
 url: "/schedule/star/" + sessionId,
 data: { starred: isStarred }
 }).done(function (response) {
 if (isStarred && response.starCount > 50) {
 alert("This session is very popular! Be sure to arrive early to get a
seat.");
 }
 });
}
```

### ► Task 3: Test the Schedule page

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, verify that the page displays a list of Conference sessions.
4. Click the star icon next to **New Technologies in Enterprise**.
5. Verify that the following message alert is displayed:

This session is very popular! Be sure to arrive early to get a seat.

6. In the **Message from webpage** dialog box, click **OK**.
7. Click the star icon next to **Diving in at the deep end with Canvas**.
8. Verify that no message alert is displayed.
9. Close Internet Explorer.
10. In Visual Studio, in Solution Explorer, double-click **schedule.js**.
11. In the **downloadSchedule** function, find the string **/schedule/list** and replace with **/schedule/list?fail**.
12. In Solution Explorer, double-click **schedule.htm**.
13. On the **Debug** menu, click **Start Without Debugging**.
14. Verify that the error message **Schedule list not available** is displayed.
15. In the **Message from webpage** dialog box, click **OK**.
16. Close Internet Explorer.

**Results:** After completing this exercise, you will have refactored the JavaScript code that sends and receives data to use the jQuery **ajax** method.



# Module 6: Styling HTML5 by Using CSS3

## Lab: Styling Text and Block Elements by Using CSS3

### Exercise 1: Styling the Navigation Bar

#### ► Task 1: Review the HTML structure

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the ContosoConf project and then double-click **index.htm**.
8. Verify that the **<head>** element contains the following HTML markup:

```
<link href="/styles/nav.css" rel="stylesheet" type="text/css" />
```

9. Find the lines containing the following HTML markup:

```
<nav class="page-nav">
 <div class="container">
 Home
```

10. On the **Debug** menu, click **Start Without Debugging**.
11. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
12. Verify that the navigation bar at the top of the page contains a row of unstyled links.
13. Close Internet Explorer.

#### ► Task 2: Style the navigation bar and links

1. In Visual Studio, in Solution Explorer, expand the **styles** folder, and then double-click **nav.css**.
2. Find the following comment:

```
/* TODO: nav.page-nav */
```

3. After this comment, insert the following CSS:

```
nav.page-nav {
 background-color: #1d1d1d;
 line-height: 6rem;
 font-size: 1.7rem;
```

```
}
```

4. Find the following CSS comment:

```
/* TODO: nav.page-nav .container */
```

5. After this comment, insert the following CSS:

```
nav.page-nav .container {
 display: -ms-flexbox;
 display: flexbox;
}
```

6. Find the following comment:

```
/* TODO: nav.page-nav a */
```

7. After the comment, insert the following CSS:

```
nav.page-nav a {
 display: block;
 min-width: 9rem;
 padding: 0 1.8rem;
 border-right: 1px dotted #3d3d3d;
 text-decoration: none;
 text-transform: uppercase;
 text-align: center;
 color: #c3c3c3;
 text-shadow: 0 1px 0 #000;
}
```

8. Find the following comment:

```
/* TODO: nav.page-nav a:first-child */
```

9. After the comment, insert the following CSS:

```
nav.page-nav a:first-child {
 border-left: 1px dotted #3d3d3d;
}
```

10. Find the following comment:

```
/* TODO: nav.page-nav a:hover */
```

11. After the comment, insert the following CSS:

```
nav.page-nav a:hover {
 color: #e4e4e4;
 background-color: black;
}
```

12. Find the following comment:

```
/* TODO: nav.page-nav .active */
```

13. After the comment, insert the following CSS:

```
nav.page-nav .active {
 color: #fff;
 background: -ms-linear-gradient(#c95656, #8d0606);
```

```
 background: linear-gradient(#c95656, #8d0606);
 }
```

14. Find the following comment:

```
/* TODO: nav.page-nav .active:hover */
```

15. After the comment, insert the following CSS:

```
nav.page-nav .active:hover {
 /* Override hover effect for active page link */
 color: #fff;
}
```

#### ► Task 3: Create graphics by using pseudo elements

1. Find the following comment:

```
/* TODO: nav.page-nav .active:before */
```

2. After the comment, insert the following CSS:

```
nav.page-nav .active:before {
 position: absolute;
 top: 6rem;
 display: block;
 height: 0;
 width: 0;
 margin-left: -1.9rem;
 border-top: 2rem solid #8d0606;
 border-right: 6.5rem solid transparent;
 content: "";
}
```

3. Find the following comment:

```
/* TODO: nav.page-nav .active:after */
```

4. After the comment, insert the following CSS:

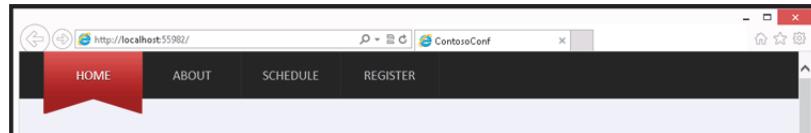
```
nav.page-nav .active:after {
 position: absolute;
 display: block;
 height: 0;
 width: 0;
 margin-left: 4.3rem;
 border-top: 2rem solid #8d0606;
 border-left: 6.5rem solid transparent;
 content: "";
}
```

#### ► Task 4: Test the navigation bar

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Verify that the navigation bar looks similar to the following image:

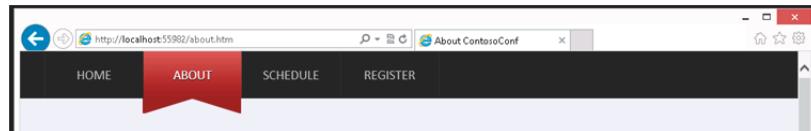
The navigation bar should look like this:

MCT USE ONLY. STUDENT USE PROHIBITED



4. In the navigation bar, click **About**.
5. Verify that the About item is styled correctly.

The About item in the navigation bar should be displayed like this:



**FIGURE 6.2:THE ABOUT ITEM IN THE NAVIGATION BAR**

6. Close Internet Explorer.

**Results:** After completing this exercise, you will have styled the navigation bar to match the design mockup.

## Exercise 2: Styling the Register Link

### ► Task 1: Review the HTML and CSS

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Open Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project and then double-click **index.htm**.
4. Verify that the **<head>** element contains the following HTML markup:

```
<link href="/styles/header.css" rel="stylesheet" type="text/css" />
```

5. Verify the **<header>** element contains the following HTML markup:

```
<header class="page-header">
 <div class="container">
 <h1>ContosoConf</h1>
 <p class="tag-line">A two-track conference on the latest HTML5
 developments</p>

 Register

 Free ·

 </div>
</header>
```

6. On the **Debug** menu, click **Start Without Debugging**.
7. In Internet Explorer, verify that the **Register** link in the page header appears as an ordinary link and is not styled.
8. Close Internet Explorer.

### ► Task 2: Position the Register link and set the text styling

1. In Solution Explorer, expand the **styles** folder, and then double-click **header.css**.
2. Find the following comment:

```
/* TODO: header.page-header .register */
```

3. Modify the CSS rule following this comment, and add the CSS properties shown below in bold:

```
header.page-header .register {
 display: block;
 position: absolute;
 top: 2rem;
 right: 3.5rem;
 width: 16rem;
 height: 10rem;
 padding-top: 6rem;
}
```

4. Add the following CSS properties shown in bold to the **header.page-header .register** CSS rule:

```
header.page-header .register {
 display: block;
 position: absolute;
 top: 2rem;
 right: 3.5rem;
 width: 16rem;
 height: 10rem;
}
```

```
padding-top: 6rem;
font-size: 2.7rem;
text-align: center;
text-decoration: none;
text-transform: uppercase;
text-shadow: 0 1px 0 #000;
color: #fff;
}
```

► Task 3: Style the Register link background, shape, and rotation properties

1. Add the following CSS properties shown in bold to the **header.page-header .register** CSS rule:

```
header.page-header .register {
 display: block;
 position: absolute;
 top: 2rem;
 right: 3.5rem;
 width: 16rem;
 height: 10rem;
 padding-top: 6rem;
 font-size: 2.7rem;
 text-align: center;
 text-decoration: none;
 text-transform: uppercase;
 text-shadow: 0 1px 0 #000;
 color: #fff;
 background: -ms-linear-gradient(#a80000, #740404);
 background: linear-gradient(#a80000, #740404);
 -ms-border-radius: 100%;
 border-radius: 100%;
 -ms-transform: rotate(6deg);
 transform: rotate(6deg);
}
```

2. Find the following CSS rule:

```
header.page-header .register:hover {
```

3. Add the following CSS properties shown below in bold to this rule:

```
header.page-header .register:hover {
 background: -ms-linear-gradient(#bc0101, #8c0909);
 background: linear-gradient(#bc0101, #8c0909);
}
```

4. Find the following CSS rule:

```
header.page-header .register:before {
```

5. Add the following CSS properties shown in bold to this rule:

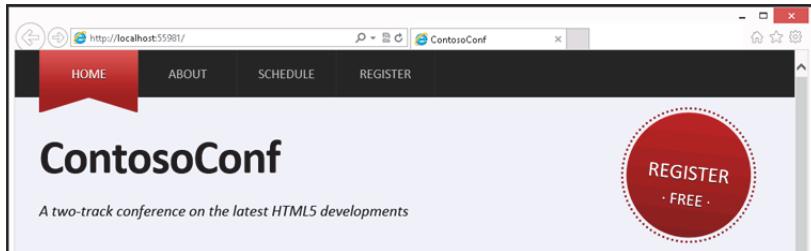
```
header.page-header .register:before {
 display: block;
 position: absolute;
 top: -.7rem;
 right: -.7rem;
 height: 16.8rem;
 width: 16.8rem;
 content: "";
 border: 3px dotted #740404;
 -ms-border-radius: 100%;
```

```
border-radius: 100%;
}
```

► **Task 4: Test the Register link**

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, verify that the **Register** link in the page header looks like the following image:

The Register link should look like this:



**FIGURE 6.3:THE STYLED REGISTER LINK**

4. Move the mouse over the **Register** link and verify that it changes color.
5. Close Internet Explorer.

**Results:** After completing this exercise, you will have styled the **Register** link in the header of the Home page.

## Exercise 3: Styling the About Page

### ► Task 1: Review the HTML and CSS

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Open Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 3**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project and then double-click **about.htm**.
4. Verify that the **<head>** element contains the following HTML markup:

```
<link href="/styles/pages/about.css" rel="stylesheet" type="text/css" />
```

5. Verify that the **<body>** element contains the following HTML markup:

```
<section class="page-section about">
 <article class="container">
 ...
 </article>
</section>
```

### ► Task 2: Define text columns

1. In Solution Explorer, expand the **styles** folder, then expand the **pages** folder, and then double-click **about.css**.
2. In **about.css**, find the following CSS comment:

```
/* TODO: .about > article > section */
```
3. After this comment, add the following CSS:

```
.about > article > section {
 column-count: 3;
 column-gap: 5rem;
 text-align: justify;
}
```

### ► Task 3: Add a drop cap at the start of the text

1. Find the following comment:

```
/* TODO: Add drop cap styling */
```
2. After this comment, add the following CSS:

```
.about p:first-child:first-letter {
 font-size: 300%;
 float: left;
 margin: 0 0.5rem 0 0;
 line-height: .8;
 color: #aaa;
}
```

### ► Task 4: Indent paragraphs

1. Find the following comment:

```
/* TODO: Indent paragraphs */
```
2. After this comment, add the following CSS:

```
.about p {
 text-indent: 3rem;
}
.about p:first-child {
 /* Prevents text indenting after drop cap */
 text-indent: 0;
 margin-top: 0;
}
```

### ► Task 5: Style the block quote

1. Find the following comment:

```
/* TODO: Blockquote */
```

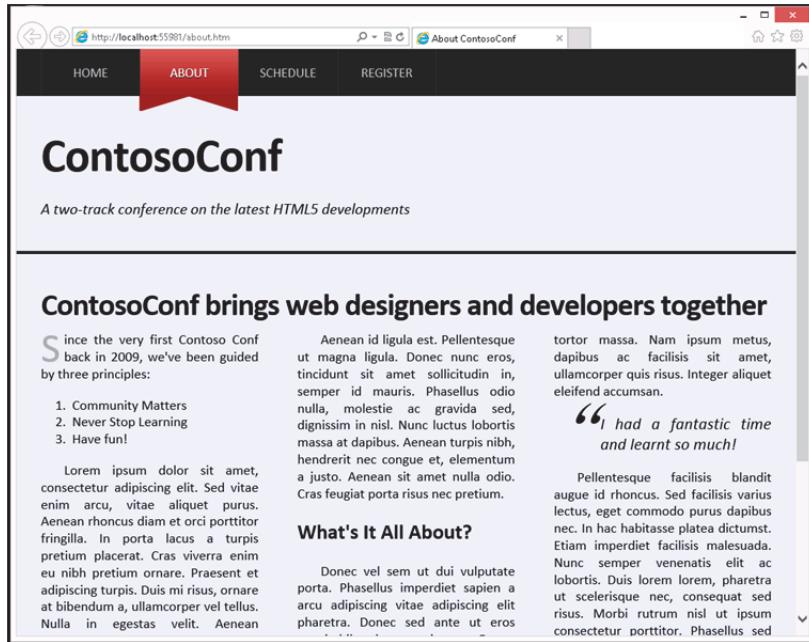
2. After the comment, add the following CSS:

```
.about blockquote {
 font-size: 1.2em;
 padding: 0 0 0 6rem;
 margin: 0;
 font-style: italic;
 position: relative;
}
.about blockquote:before {
 content: '\201C';
 position: absolute;
 font-size: 10rem;
 font-family: serif;
 left: 0;
 top: -1rem;
 line-height: 1;
}
```

### ► Task 6: Test the About page

1. In Solution Explorer, double-click **about.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, verify that the page looks similar to the following image:

The About page should look like this:



**FIGURE 6.4:THE ABOUT PAGE**

4. Close Internet Explorer.

**Results:** After completing this exercise, you will have styled the text on the **About** page.

## Module 7: Creating Objects and Methods by Using JavaScript

# Lab: Refining Code for Maintainability and Extensibility

### Exercise 1: Object Inheritance

- ▶ Task 1: Review the **Object.inherit.js** JavaScript file

  1. Start the **MSL-TMG1** virtual machine if it is not already running.
  2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
  3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod07\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project node, and then expand the **scripts** folder.
8. Double-click **Object.inherit.js**.
9. Verify that the file contains the following JavaScript code:

```
var copyOwnProperties = function (from, to) {
 for (var propertyName in from) {
 if (from.hasOwnProperty(propertyName)) {
 to[propertyName] = from[propertyName];
 }
 }
};
var inherit = function (additionalProperties) {
 ...
};
```

- ▶ Task 2: Implement the **Inherit** function

1. In **Object.inherit.js**, find the following comment:

```
// TODO: Create a variable named `factory`, assign it a new object who's prototype
// is `this`.
```

2. After the comment, add the following JavaScript code:

```
var factory = Object.create(this);
```

3. Find the following comment:

```
// TODO: Add a method called `create` to `factory`, that does the following
```

4. After the comment, add the following JavaScript code:

- ```
factory.create = function() {
```
5. Find the following comment:
- ```
// TODO: return `instance`.
```
6. After the comment, add a blank line followed by the following JavaScript code:
- ```
};
```
7. Find the following comment:
- ```
// TODO: Copy properties of `additionalProperties` onto `factory` (using copyOwnProperties).
```
8. After the comment, add the following JavaScript code:
- ```
copyOwnProperties(additionalProperties, factory);
```
9. Find the following comment:
- ```
// TODO: Return the `factory` object.
```
10. After the comment, add the following JavaScript code:
- ```
return factory;
```
11. In the **inherit** function, find the comment that starts with the following line:
- ```
// TODO: Define a variable named `instance`.
```
12. After the comment, add the following JavaScript code:
- ```
var instance = Object.create(factory);
```
13. Find the following comment:
- ```
// TODO: If `instance` has a function named "initialize",
```
14. After the comment, add the following JavaScript code:
- ```
if (typeof instance.initialize === "function") {  
    instance.initialize.apply(instance, arguments);  
}
```
15. Find the following comment:
- ```
// TODO: return `instance`.
```
16. After the comment, add the following JavaScript code:
- ```
return instance;
```
17. Find the following comment:
- ```
// TODO: Add the inherit function to the built-in `Object` object.
```
18. After the comment, add the following of JavaScript code:

```
Object.inherit = inherit;
```

► **Task 3: Scope the JavaScript code inside an immediately invoked function expression**

1. In **Object.inherit.js**, add the following JavaScript code at the top of the file:

```
(function () {
```

2. Add the following JavaScript code at the end of the file:

```
}());
```

► **Task 4: Use strict mode**

1. In **Object.inherit.js**, find the following comment:

```
// TODO: Strict mode
```

2. After the comment, add the following line (including the quotes):

```
"use strict";
```

3. On the **File** menu, click **Save All**.

**Results:** After completing this exercise, you will have written the **Object.inherit** utility function that you will use in later exercises to structure the code for the web application. You will also have modified the scope for this function, and specified that the code should run by using strict mode.

## Exercise 2: Refactoring JavaScript Code to Use Objects

### ► Task 1: Define the ScheduleList factory

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod07\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. Expand the **ContosoConf** project, expand the **scripts** folder, and then expand the **pages** folder.
4. Double-click **schedule.js**.
5. Find the following comment:

```
// TODO: Create a ScheduleList factory object using the Object.inherit helper method.
```

6. After the comment, add the following JavaScript code:

```
var ScheduleList = Object.inherit({
 initialize: function (element, localStarStorage) {
 this.element = element;
 this.localStarStorage = localStarStorage;
 }
});
```

7. Find the comment that starts with the following line:

```
// TODO: Refactor these variables into properties of the ScheduleList object.
```

8. Delete the following line of JavaScript code after this comment:

```
var element, localStarStorage;
```

### ► Task 2: Convert functions into methods of the ScheduleList object

1. In **schedule.js**, find the following comment:

```
// TODO: Refactor these functions into methods of the ScheduleList object.
```

2. Delete the **startDownload**, **downloadDone**, **downloadFailed**, **addAll**, and **add** functions that follow this comment.
3. Add the following JavaScript code shown in bold to the **ScheduleList** object (note that there is a comma after the closing brace of the **initialize** function):

```
var ScheduleList = Object.inherit({
 initialize: function (element, localStarStorage) {
 this.element = element;
 this.localStarStorage = localStarStorage;
 },
 startDownload: function () {
 var request = $.ajax({
 url: "/schedule/list",
 context: this
 });
 request.done(this.downloadDone)
 .fail(this.downloadFailed);
 },
 downloadDone: function (responseData) {
 this.addAll(responseData.schedule);
 },
 downloadFailed: function () {
```

```
 alert("Could not retrieve schedule data at this time. Please try again later.");
 },
 addAll: function (itemsArray) {
 itemsArray.forEach(this.add, this);
 },
 add: function (itemData) {
 var item = ScheduleItem.create(itemData, this.localStarStorage);
 this.element.appendChild(item.element);
 }
});
```

► **Task 3: Create and use a ScheduleList object**

1. In **schedule.js**, find the following JavaScript code:

```
// TODO: Replace the following code by creating a ScheduleList object
// and calling the startDownload method.
element = document.getElementById("schedule");
localStarStorage = LocalStarStorage.create(localStorage);
startDownload();
```

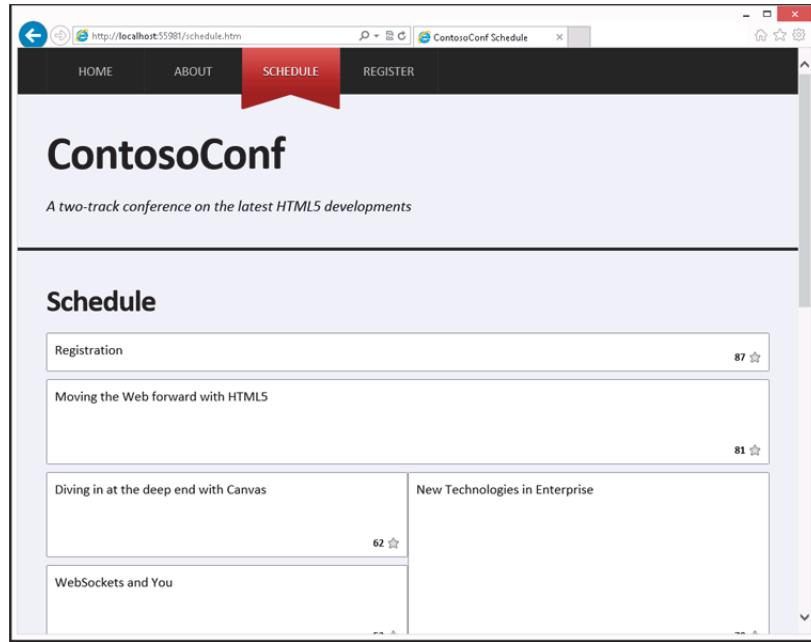
2. Delete this block of JavaScript code, and then replace it with the following code:

```
var scheduleList = ScheduleList.create(
 document.getElementById("schedule"),
 LocalStarStorage.create(localStorage)
);
scheduleList.startDownload();
```

► **Task 4: Test the application**

1. In Solution Explorer, double-click **schedule.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
4. Verify that the page looks similar to the image below:

The Schedule page should still display the list of sessions for the conference, like this:



**FIGURE 7.1:THE SCHEDULE PAGE**

5. Close Internet Explorer.

**Results:** After completing this exercise, you will have refactored the JavaScript code for the Schedule page to be more maintainable by using objects.

# Module 8: Creating Interactive Pages by Using HTML5 APIs

## Lab: Creating Interactive Pages with HTML5 APIs

### Exercise 1: Dragging and Dropping Images

► **Task 1:** Review the HTML markup and JavaScript code for the Speaker Badge page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.

 **Note:** If necessary, click **Switch User** to display the list of users

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project node.
8. In Solution Explorer, double-click **speaker-badge.htm**.
9. Verify that the HTML markup contains the following **<img>** element:

```

```

10. Verify that the HTML markup contains the following **<script>** elements:

```
<script src="/scripts/jquery.min.js" type="text/javascript"></script>
<script src="/scripts/pages/speaker-badge.js" type="text/javascript"></script>
```

11. In Solution Explorer, expand the **scripts** folder and then expand the **pages** folder.
12. Double-click **speaker-badge.js**.
13. Verify that the JavaScript file contains the following line of code:

```
var SpeakerBadgePage = Object.inherit{
```

► **Task 2: Add drag-and-drop event listeners**

1. In **speaker-badge.js**, find the following comment:

```
// TODO: Add event listeners for element "dragover" and "drop" events.
// handle with this.handleDragOver.bind(this) and this.handleDrop.bind(this)
```

2. After the comment, add the following JavaScript code:

```
element.addEventListener("dragover", this.handleDragOver.bind(this), false);
element.addEventListener("drop", this.handleDrop.bind(this), false);
```

► **Task 3: Handle the drop event**

1. In **speaker-badge.js**, find the following comment:

```
// TODO: Get the files from the event
```

2. After the comment, add the following JavaScript code:

```
var files = event.dataTransfer.files;
```

3. Find the following comment:

```
// TODO: Read the first file in the array
```

4. After the comment, add the following JavaScript code:

```
var file = files[0];
```

5. Find the following comment:

```
// Check the file type is an image
```

6. After the comment, add the following JavaScript code:

```
if (this.isImageType(file.type)) {
```

7. Find the following comment:

```
// Use this.readFile to read the file, then display the image
```

8. After the comment, add the following JavaScript code:

```
 this.readFile(file).done(this.displayImage);
} else {
 alert("Please drop an image file.");
}
```

► **Task 4: Read the image by using a FileReader**

1. In **speaker-badge.js**, find the following comment:

```
// TODO: Create a new FileReader
```

2. After the comment, insert the following JavaScript code:

```
var reader = new FileReader();
```

3. Find the following comments:

```
// TODO: Assign a callback function for reader.onload
// TODO: In the callback use reading.resolveWith(context, [fileDataUrl]); to return
the file data URL
```

4. After these comments, insert the following JavaScript code:

```
reader.onload = function (loadEvent) {
 var fileDataUrl = loadEvent.target.result;
 reading.resolveWith(context, [fileDataUrl]);
};
```

5. Find the following comment:

```
// TODO: Start reading the file as a DataURL
```

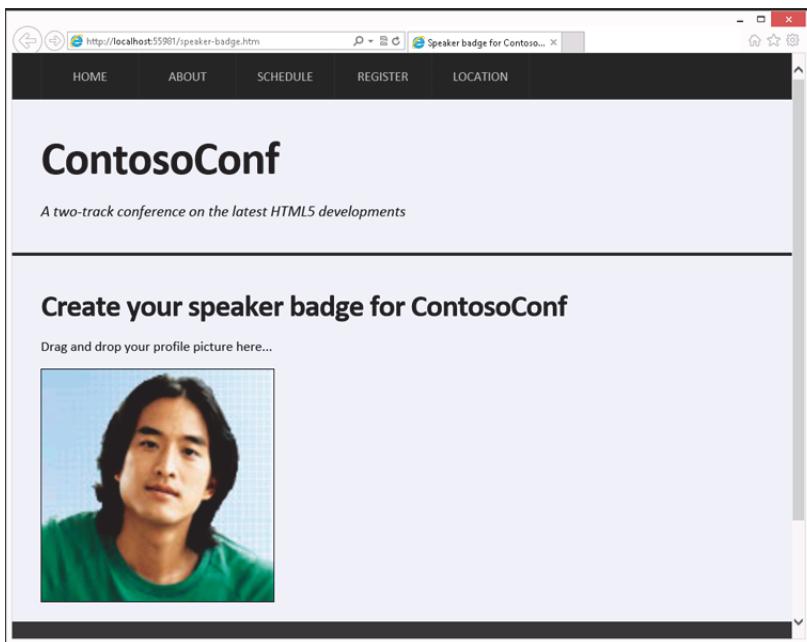
6. After the comment, insert the following JavaScript:

```
reader.readAsDataURL(file);
```

► **Task 5: Test the Speaker Badge page**

1. In Solution Explorer, double-click **speaker-badge.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
4. On the Windows taskbar, click **File Explorer** and browse to **E:\Mod08\Labfiles\Resources**.
5. Drag-and-drop **mark-hanson.jpg** from File Explorer, onto the empty rectangle in Internet Explorer.

The page should look like this:



**FIGURE 8.1:THE SPEAKER BADGE PAGE WITH THE SPEAKER'S PHOTO**

6. Close Internet Explorer and File Explorer.

**Results:** After completing this exercise, you will have implemented functionality that enables the user to drag-and-drop an image from File Explorer onto the web page.

## Exercise 2: Incorporating Video

### ► Task 1: Add a video player to the Home page

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project node.
4. Double-click **index.htm**.
5. Find the comment that starts with the line:

```
<!-- TODO: Add video tag here -->
```

6. After the second line of the comment, add the following HTML markup:

```
<video src="http://ak.channel9.msdn.com/ch9/265b/9a76fccd-941e-4285-ad00-9ea200aa265b/MIX09KEY01_high_ch9.mp4"></video>
```

### ► Task 2: Add controls to the video player

1. In **index.htm**, after the **</video>** element, insert the following HTML markup:

```
<div class="video-controls" style="display: none">
 <button class="video-play">Play</button>
 <button class="video-pause">Pause</button>

</div>
```

2. Find the following comment and select it with the mouse:

```
<!--<script src="/scripts/pages/video.js" type="text/javascript"></script>-->
```

3. In the toolbar, click the **Uncomment the selected lines** button.
4. Verify that the HTML markup now looks like this:

```
<script src="/scripts/pages/video.js" type="text/javascript"></script>
```

### ► Task 3: Control the video by using JavaScript code

1. In Solution Explorer, expand the **scripts** folder, and then expand the **pages** folder.
2. Double-click **video.js**.
3. Find the following comment:

```
// TODO: display the video controls
```

4. After this comment add the following JavaScript code:

```
controls.style.display = "block";
```

5. Find the comment that starts with the following lines:

```
// TODO: Add event listeners for:
// video loaddata
```

6. After the second line of comment, add the following JavaScript code:

```
video.addEventListener("loadeddata", ready, false);
```

7. In **video.js**, find the following comment:

```
// TODO: play the video
```

8. After the comment, add the following JavaScript code:

```
video.play();
playButton.style.display = "none";
pauseButton.style.display = "";
```

9. Find the following comment:

```
// TODO: pause the video
```

10. After the comment add the following JavaScript code:

```
video.pause();
pauseButton.style.display = "none";
playButton.style.display = "";
```

11. Find the following comments:

```
// play click
// pause click
```

12. After the comments, add the following JavaScript code:

```
playButton.addEventListener("click", play, false);
pauseButton.addEventListener("click", pause, false);
```

#### ► Task 4: Display the video elapsed time

1. In **video.js**, find the following comment:

```
// TODO: Set time.textContent using video.currentTime.
// Use the formatTime function to convert raw seconds into HH:MM:SS format.
```

2. After the comment, add the following JavaScript code:

```
time.textContent = formatTime(video.currentTime);
```

3. Find the following comment:

```
// video timeupdate
```

4. After the comment, add the following JavaScript code:

```
video.addEventListener("timeupdate", updateTime, false);
```

#### ► Task 5: Test the video player

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, wait for the **Play** button to appear underneath the video.
4. Click **Play**.

5. Verify that the video begins to play and that the current video time is displayed.
6. Click **Pause**.
7. Verify that the video pauses.
8. Close Internet Explorer.

**Results:** After completing this exercise, you will have added a video player to the Home page.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 3: Using the Geolocation API to Report the User's Current Location

### ► Task 1: Review the HTML markup and JavaScript code

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 3**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project node.
4. In Solution Explorer, double-click **location.htm**.
5. Verify that the HTML markup contains the following **<h2>** element:

```
<h2 id="distance"></h2>
```

6. In Solution Explorer, expand the **scripts** folder, and then expand the **pages** folder.
7. Double-click **location.js**.
8. Verify that the file contains the following JavaScript code:

```
var conferenceLocation = {
 latitude: 47.6097, // decimal degrees
 longitude: 122.3331 // decimal degrees
};
```

### ► Task 2: Get the current location of the user viewing the page

1. In **location.js**, find the following comment:

```
// TODO: Get current position from the geolocation API.
```

2. After the comment, add the following JavaScript code:

```
navigator.geolocation.getCurrentPosition(updateUIForPosition, error);
```

### ► Task 3: Display the distance to the conference venue

1. In **location.js**, find the following comment:

```
// TODO: Calculate the distance from the conference
```

2. After the comment, add the following JavaScript code:

```
var distance = distanceFromConference(position.coords);
```

### ► Task 4: Test the Location page

1. In Solution Explorer, double-click **location.htm**.
2. On the **Debug** menu, double-click **Start Without Debugging**.
3. In Internet Explorer, if the message **localhost wants to track your location** appears, click **Allow once**.
4. In the **Enable Location Service** dialog box, click **Yes**.
5. Verify that the page displays the distance to the conference venue, in miles (the actual value will vary, depending on your distance from the conference venue).

**Results:** After completing this exercise, you will have a Location page that displays the distance of the user from the conference venue.

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 9: Adding Offline Support to Web Applications

## Lab: Adding Offline Support to Web Applications

### Exercise 1: Caching Offline Data by Using the Application Cache API

- ▶ Task 1: Configure the application cache manifest
  1. Start the **MSL-TMG1** virtual machine if it is not already running.
  2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running, and log on as **Student** with the password **Pa\$\$w0rd**.

 **Note:** If necessary, click **Switch User** to display the list of users.

3. On the Windows 8 **Start** screen, click the **Desktop** tile.
4. On the Windows taskbar, click **Internet Explorer**.
5. In the Internet Explorer, press F10 to display the menu bar.
6. On the **Tools** menu, click **Internet options**.
7. In the **Internet Options** dialog box, click **Settings**.
8. In the **Website Data Settings** dialog box, click the **Caches and databases** tab.
9. Select the **Allow website caches and databases** check box, and then click **OK**.
10. In the **Internet Options** dialog box, click **OK**.
11. Close Internet Explorer.
12. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
13. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
14. In the **Open Project** dialog box, browse to **E:\Mod09\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
15. In Solution Explorer, expand the **ContosoConf** project node, and then double-click **appcache.manifest**.
16. Find the following comment:

```
TODO: Add index, about, schedule and location pages
```
17. After the comment, type the following URLs:

```
/index.htm
/about.htm
/location.htm
/schedule.htm
```
18. In Solution Explorer, double-click **index.htm**.
19. Add the **manifest** attribute to the **<html>** element, as shown in bold below:

```
<html lang="en" manifest="/appcache.manifest">
```

20. In Solution Explorer, double-click **about.htm**.
21. Add the manifest attribute to the **<html>** element, as shown in bold below:

```
<html lang="en" manifest="/appcache.manifest">
```

22. In Solution Explorer, double-click **schedule.htm**.
23. Add the manifest attribute to the **<html>** element, as shown in bold below:

```
<html lang="en" manifest="/appcache.manifest">
```

24. In Solution Explorer, double-click **location.htm**.
25. Add the manifest attribute to the **<html>** element, as shown in bold below:

```
<html lang="en" manifest="/appcache.manifest">
```

#### ► Task 2: Detect offline mode by using JavaScript code

1. In Solution Explorer, expand the **scripts** folder, and then double-click **offline.js**.
2. Find the following comment:

```
// TODO: if currently offline, hide navigation links that require online
```

3. After this comment, add the following JavaScript code:

```
if (!navigator.onLine) {
 hideLinksThatRequireOnline();
}
```

4. Find the following comment:

```
// TODO: add onoffline and ononline events to document.body
```

5. After the second line of this comment, add the following JavaScript code:

```
document.body.onoffline = hideLinksThatRequireOnline;
document.body.ononline = showLinks;
```

6. Find the following comment:

```
// TODO: also handle the applicationCache error event to hide links
```

7. After this comment, add the following JavaScript code:

```
applicationCache.addEventListener("error", hideLinksThatRequireOnline, false);
```

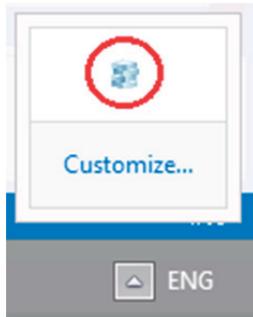
#### ► Task 3: Test the application

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
4. Verify that Internet Explorer displays the **Home** page.
5. Expand the Windows notification area, right-click **IIS Express**, and then click **Exit**.



**Note:** When IIS Express first starts running, the IIS Express icon may appear in the Windows task bar rather than the notification area. If this occurs, right-click the **IIS Express** icon and then click **Exit**.

The following image shows the IIS Express icon.



**FIGURE 9.1:THE IIS EXPRESS ICON IN THE WINDOWS NOTIFICATION AREA**

6. In the **Confirmation** dialog box, click **Yes**.
7. In Internet Explorer, click **Schedule**.
8. Verify that the page loads and displays the schedule information.
9. Wait five seconds, and then verify that the web site navigation bar no longer displays the **Register** link.
10. Click **About**.
11. Verify that the page loads and displays the information describing the conference.
12. Verify that the web site navigation bar no longer displays the **Register** link.
13. Close Internet Explorer.

**Results:** After completing this exercise, you will have modified the web application and made the **Home**, **About**, **Schedule**, and **Location** pages available offline.

## Exercise 2: Persisting User Data by Using the Local Storage API

### ► Task 1: Observe the current behavior of the Schedule page

1. In Visual Studio, on the **File** menu, click point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod09\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, and then double-click **schedule.htm**.
4. On the **Debug** menu, click **Start Without Debugging**.
5. Expand the Windows notification area, right-click **IIS Express**, and then click **Exit**.
6. In the **Confirmation** dialog box, click **Yes**.
7. In Internet Explorer, click the star icon in the **Registration** box, and verify that the icon is now colored yellow.
8. Click **Refresh**.
9. Verify that the star icon for **Registration** is now colored white.
10. Close Internet Explorer.

### ► Task 2: Save information about starred session to local storage

1. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, and then double-click **LocalStarStorage.js**.
2. Find the following comments:

```
// TODO: convert this.sessions into a JSON string
// TODO: save this JSON string into local storage as "stars"
```

3. After the second comment, insert the following JavaScript code:

```
this.localStorage.setItem("stars", JSON.stringify(this.sessions));
```

### ► Task 3: Load information about starred session from local storage

1. In **LocalStarStorage.js**, find the following comment:

```
// TODO: get the "stars" from local storage
```

2. After this comment, add the following JavaScript code:

```
var json = this.localStorage.getItem("stars");
```

3. Find the following comments:

```
// TODO: parse the JSON string into this.sessions
// TODO: handle failures due to missing data etc
```

4. After the second comment, add the following JavaScript code:

```
if (json) {
 try {
 this.sessions = JSON.parse(json) || [];
 } catch (exception) {
 this.sessions = [];
 }
} else {
```

```
 this.sessions = [];
}
```

► **Task 4: Use the local storage wrapper to save and load data in the Schedule page**

1. In Solution Explorer, expand the **scripts** folder, then expand the **pages** folder, and then double-click **schedule.js**.
2. Find the following comment:

```
// TODO: Check if item is starred
```

3. After this comment, add the following JavaScript code:

```
if (localStarStorage.isStarred(this.id)) {
 this.element.classList.add(this.starredClass);
}
```

4. Find the following comment:

```
// TODO: remove the star from the item
```

5. After this the comment, add the following JavaScript code:

```
this.localStarStorage.removeStar(this.id);
```

6. Find the following comment:

```
// TODO: add a star to the item
```

7. After this comment, add the following JavaScript code:

```
this.localStarStorage.addStar(this.id);
```

► **Task 5: Test the application**

1. In Solution Explorer, double-click **appcache.manifest**.
2. Find the following line:  

```
CACHE MANIFEST
```
3. After the line, insert the following line:  

```
version 2
```
4. In Solution Explorer, double-click **schedule.htm**.
5. On the **Debug** menu, click **Start Without Debugging**.
6. In Internet Explorer, press F5 to refresh the page.
7. Expand the Windows notification area, right-click **IIS Express**, and then click **Exit**.
8. In the **Confirmation** dialog box, click **Yes**.
9. In Internet Explorer, click the star icon in the **Registration** box, and verify that the icon is now colored yellow.
10. Press F5.
11. Verify that the star icon for **Registration** is still colored yellow.

12. Close Internet Explorer.

► **Task 6: Reset Internet Explorer caching**

1. On the Windows taskbar, click **Internet Explorer**.
2. In the Internet Explorer, press F10 to display the menu bar.
3. On the **Tools** menu, click **Internet options**.
4. In the **Internet Options** dialog box, click **Settings**.
5. In the **Website Data Settings** dialog box, click the **Caches and databases** tab.
6. Clear the **Allow website caches and databases** check box, and then click **OK**.
7. In the **Internet Options** dialog box, click **OK**.
8. Close Internet Explorer.

**Results:** After completing this exercise, you will have updated the Schedule page to locally record starred sessions.

# Module 10: Implementing an Adaptive User Interface

## Lab: Implementing an Adaptive User Interface

### Exercise 1: Creating a Print-Friendly Style Sheet

#### ► Task 1: Review the existing application

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If it is necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod10\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Visual Studio, in Solution Explorer, expand the **ContosoConf** project, and then double-click **about.htm**.
8. On the **Debug** menu, click **Start Without Debugging**.
9. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
10. Press F10 to display the menu bar
11. On the **File** menu, click **Print preview**.
12. In the **Print Preview** window, verify that the preview looks like the following image:

The Print Preview version of the **About** page looks like this:



13. Close the **Print Preview** window.
  14. Close Internet Explorer.
  15. In Visual Studio, in the Solution Explorer window, double-click **about.htm**.
  16. Verify that the file contains the following HTML markup:

```
<nav class="page-nav">
<header class="page-header">
<footer class="page-footer">
```

#### ► Task 2: Create a style sheet for printing web pages

1. In Visual Studio, in Solution Explorer, click the **styles** folder.
  2. On the **Project** menu, click **Add New Item**.
  3. In the **Add New Item - ContosoConf** dialog box, in the left pane, expand the **Visual C#** node, and then click **Web**.
  4. In the middle pane, click **Style Sheet**.

5. In the **Name** box, type **print.css**.
6. Click **Add**.
7. In **print.css**, delete the existing file contents.
8. Add the following CSS:

```
nav.page-nav,
header.page-header,
footer.page-footer {
 display: none;
}
.container {
 padding: 0;
 max-width: none;
}
```

9. In **print.css**, at the end of the file, add the following CSS:

```
.about > article > section {
 column-count: 1;
}
```

► **Task 3: Link the print style sheet to the About page**

1. In Solution Explorer, double-click **about.htm**.
2. Find the following comment:

```
<!-- TODO: Add print.css <link> here -->
```

3. After the comment, add the following HTML:

```
<link href="/styles/print.css" media="print" rel="stylesheet" type="text/css" />
```

► **Task 4: Test the application**

1. On the **Debug** menu, click **Start Without Debugging**.
2. In Internet Explorer, press F5 to refresh the page.
3. Press F10 to display the menu bar.
4. On the **File** menu, click **Print preview**.
5. Verify that the **Print Preview** window displays the following image:

The Print Preview version of the **About** page should look like this:

About ContosoConf

Page 1 of 2

## ContosoConf brings web designers and developers together

Since the very first Contoso Conf back in 2009, we've been guided by three principles:

1. Community Matters
2. Never Stop Learning
3. Have fun!

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed vitae enim arcu, vitae aliquet purus. Aenean rhoncus diam et orci porttitor fringilla. In porta lacus a turpis pretium placerat. Cras viverra enim eu nibh pretium ornare. Praesent et adipiscing turpis. Duis mi risus, ornare at bibendum a, ullamcorper vel tellus. Nulla in egestas velit. Aenean consequat mi sed tellus iaculis laoreet. Donec et odio vel felis commodo porttitor.

Aenean id ligula est. Pellentesque ut magna ligula. Donec nunc eros, tincidunt sit amet sollicitudin in, semper id mauris. Phasellus odio nulla, molestie ac gravida sed, dignissim in nisl. Nunc luctus lobortis massa at dapibus. Aenean turpis nibh, hendrerit nec congue et, elementum a justo. Aenean sit amet nulla odio. Cras feugiat porta risus nec pretium.

### What's It All About?

Donec vel sem ut dui vulputate porta. Phasellus imperdiet sapien a arcu adipiscing vitae adipiscing elit pharetra. Donec sed ante ut eros mattis bibendum non in erat. Donec sagittis, massa eu accumsan eleifend, eros justo cursus justo, id consequat mauris diam id magna. Vivamus quis tortor massa. Nam ipsum metus, dapibus ac facilisis sit amet, ullamcorper quis risus. Integer aliquet eleifend accumsan.

*“I had a fantastic time and learnt so much!”*

Pellentesque facilisis blandit augue id rhoncus. Sed facilisis varius lectus, eget commodo purus dapibus nec. In hac habitasse platea dictumst. Etiam imperdiet facilisis malesuada. Nunc semper venenatis elit ac lobortis. Duis lorem lorem, pharetra ut scelerisque nec, consequat sed risus. Morbi rutrum nisl ut ipsum consectetur porttitor. Phasellus sed nunc id diam tempus congue in a leo.

Proin feugiat, turpis id tempor tempor, lorem libero malesuada.

<http://localhost:56789/about.htm>

8/16/2012

6. Close the **Print Preview** window.
7. Close Internet Explorer.

**Results:** After completing this exercise, you will have added a style sheet that implements a print-friendly format for web pages.

## Exercise 2: Adapting Page Layout to Fit Different Form Factors

► **Task 1: Simulate the web application running on a small device**

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod10\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, and then double-click **index.htm**.
4. On the **Debug** menu, click **Start Without Debugging**.
5. In Internet Explorer, press F12.
6. In the **ContosoConf – F12** window, on the **Tools** menu, point to **Resize**, and then click **480x800**.
7. Press F12.
8. Verify that the **Home** page looks similar to the following image:

The **Home** page should look like this:



FIGURE 10.3:THE HOME PAGE

9. Close Internet Explorer.
- **Task 2: Implement styles for hand-held devices and smartphones**
1. In Solution Explorer, expand the **styles** folder, and then double-click **mobile.css**.
  2. Add the following CSS to the file:

```
@media screen and (max-width: 480px) {
}
```

3. Add the CSS rules shown below in bold to the file:

```
@media screen and (max-width: 480px) {
 nav.page-nav .container {
 display: -ms-flexbox;
 -ms-flex-wrap: wrap;
 -ms-flex-pack: center;
 }
 nav.page-nav .active:before,
 nav.page-nav .active:after {
 display: none;
 }
 nav.page-nav a {
 border: 1px dotted #3d3d3d;
 margin: .5rem;
 }
}
```

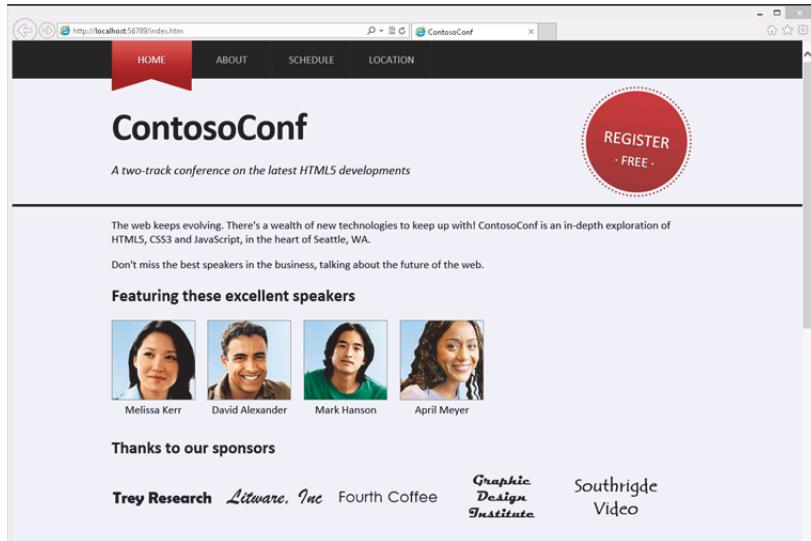
4. Add the following CSS to the end of the file:

```
@media screen and (max-width: 720px) {
 header.page-header {
 height: auto;
 }
 header.page-header .register {
 display: none;
 }
 header.page-header h1 {
 font-size: 3rem;
 }
}
```

### ► Task 3: Test the application

1. In Solution Explorer, double-click **index.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, press F12.
4. In the **ContosoConf – F12** window, on the **Tools** menu, point to **Resize**, and then click **1280×1024**.
5. Press F12.
6. Verify that the **Home** page is displayed correctly on the desktop.

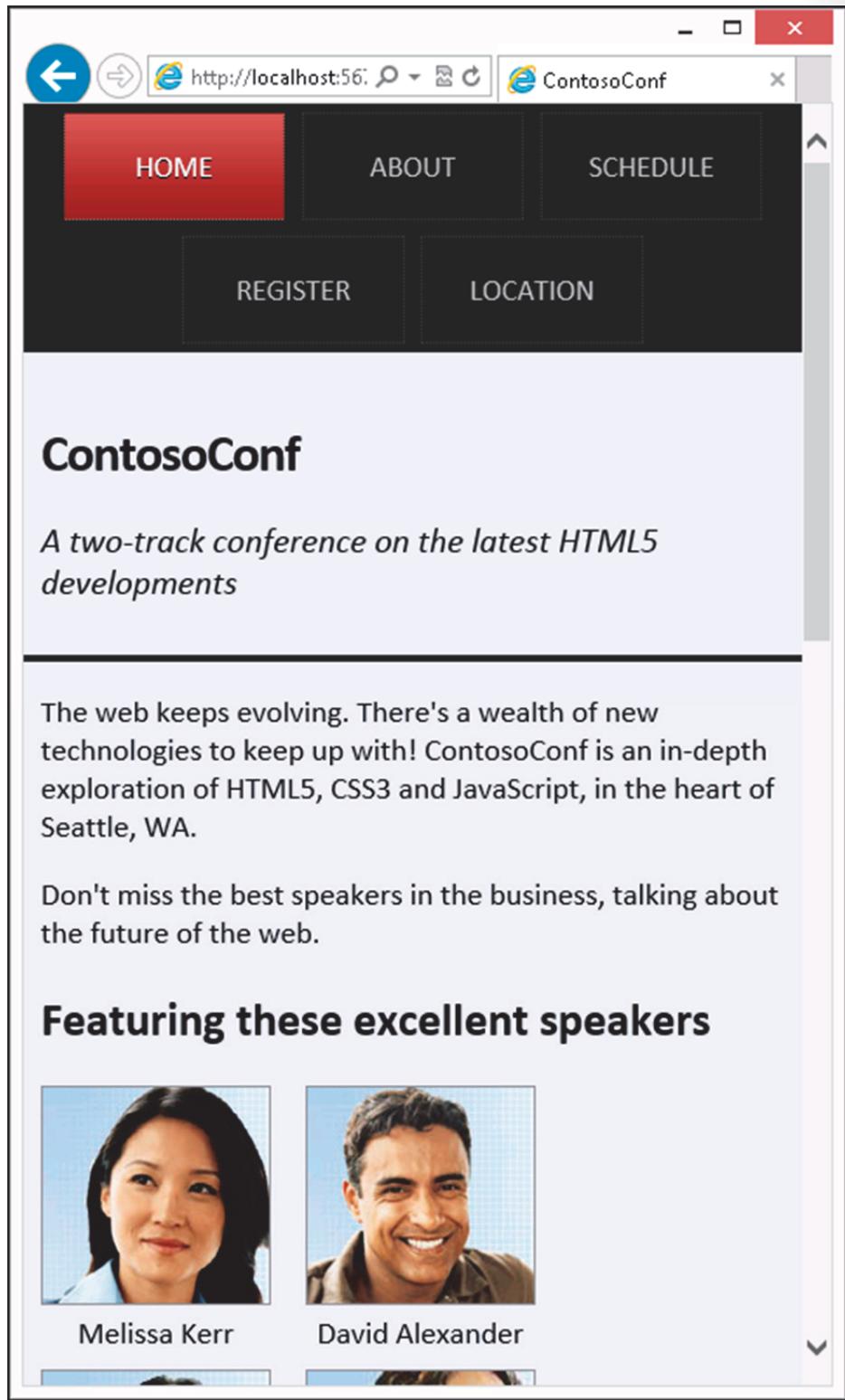
The **Home** page should look like this on the desktop:



**FIGURE 10.4:THE HOME PAGE DISPLAYED ON THE DESKTOP**

7. Press F12.
8. In the **ContosoConf – F12** window, on the **Tools** menu, point to **Resize**, and then click **480x800**.
9. Press F12.
10. Verify that the navigation bar is displayed correctly in the reduced form factor (it wraps), and that the **Register** link does not appear in the header of the web page.

The **Home** page should look like this in the reduced size window:



11. Close Internet Explorer.

**Results:** After completing this exercise, you will have a website that adapts to different screen sizes.



# Module 11: Creating Advanced Graphics

## Lab: Creating Advanced Graphics

### Exercise 1: Creating an Interactive Venue Map by Using SVG

- ▶ Task 1: Review the incomplete HTML markup for the venue map
1. Start the **MSL-TMG1** virtual machine if it is not already running.
  2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
  3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project node, and then double-click **location.htm**.
8. Verify that the page contains the following HTML markup:

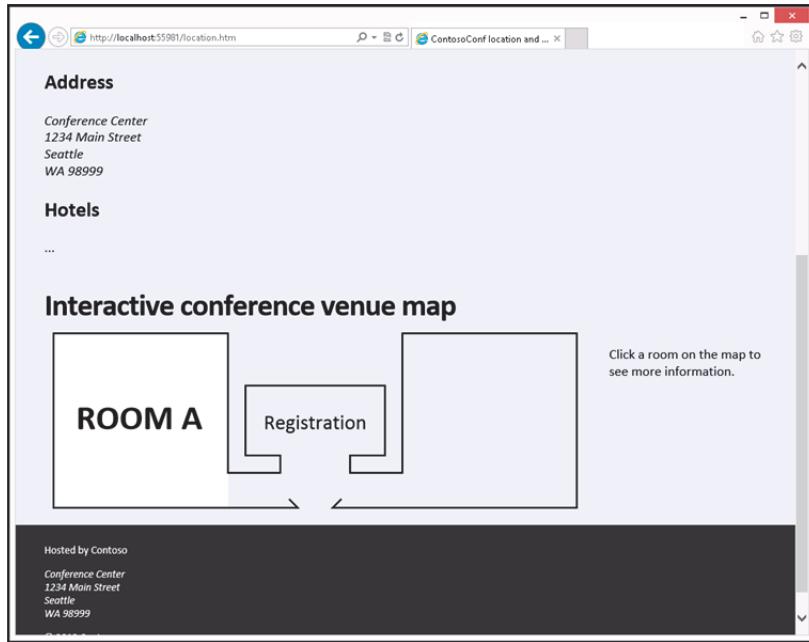
```
<svg viewBox="-1 -1 302 102" width="100%" height="230">
 <!-- Room A -->
 <g id="room-a" class="room">
 <rect fill="#ffff" x="0" y="0" width="100" height="100"/>
 <text x="13" y="55" font-weight="bold" font-size="20">ROOM A</text>
 </g>
 <!-- Room B -->
 <!-- The outline of the building -->
 <polyline fill="none" stroke="#000" points="135,95 140,100 0,100 0,0 100,0
100,80 130,80 130,70 110,70 110,30 190,30 190,70 170,70 170,80 200,80 200,0 300,0
300,100 160,100 165,95"/>
 <text x="150" y="55" font-size="12" style="text-anchor: middle">Registration</text>
</svg>
```

9. Verify that the HTML contains the following markup:

```
<script src="/scripts/pages/location-venue.js" type="text/javascript"></script>
```

10. On the **Debug** menu, click **Start Without Debugging**.
11. Click **Location**.
12. If the message **localhost wants to track your location** appears, click **Allow once**.
13. In the **Enable Location Services** dialog box, click **Yes**.
14. If the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
15. Scroll down and view the venue map.

The venue map should look like this:



**FIGURE 11.1:THE INCOMPLETE VENUE MAP**

16. Close Internet Explorer.

► **Task 2: Complete the SVG venue map**

1. In Visual Studio, in **location.htm**, find the following comment:

```
<!-- Room B -->
```

2. After the comment, add the following markup:

```
<g id="room-b" class="room">
 <rect fill="#ffff" x="200" y="0" width="100" height="100"/>
 <text x="213" y="55" font-weight="bold" font-size="20">ROOM B</text>
</g>
```

► **Task 3: Add interactivity to the venue map**

1. In Solution Explorer, double-click **location.htm**.
2. Find the following HTML markup:

```
<div id="room-a-info" style="display: none">
 <h2>Room A</h2>
 <p>Capacity: 250</p>
 <p>Popular sessions in here:</p>

 Diving in at the deep end with Canvas
 Real-world Applications of HTML5 APIs
 Transforms and Animations

</div>
<div id="room-b-info" style="display: none">
 <h2>Room B</h2>
 <p>Capacity: 230</p>
 <p>Popular sessions in here:</p>

 Building Responsive UIs
 Getting to Grips with JavaScript
 A Fresh Look at Layouts

</div>
```

```

</div>
```

3. In Solution Explorer, expand the **styles** folder, expand the **pages** folder, and then double-click **location.css**.
4. At the end of the file, add the following CSS:

```
.room:hover rect {
 fill: #b1f8b0;
}
```

5. In Solution Explorer, expand the **scripts** folder, expand the **pages** folder, and then double-click **location-venue.js**.
6. Find the following comment:

```
// TODO: Get the room elements in the svg element.
```

7. After this comment, add the following JavaScript code:

```
var rooms = document.querySelectorAll(".room");
```

8. Find the comment that starts with the following text:

```
// TODO: Add a click event listener for each room element.
```

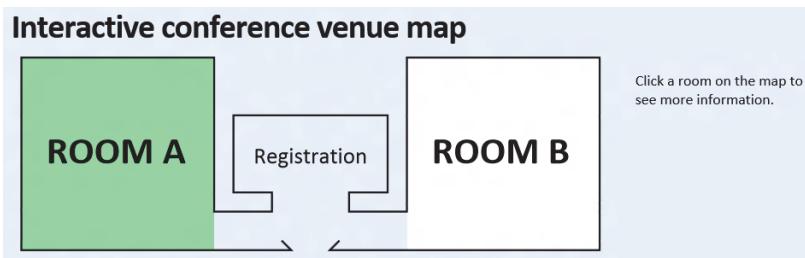
9. After the second line of this comment, add the following JavaScript code:

```
for (var i = 0; i < rooms.length; i++) {
 var room = rooms[i];
 room.addEventListener("click", function () {
 showRoomInfo(this.id);
 });
}
```

#### ► Task 4: Test the application

1. In Solution Explorer, double-click **location.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, if the message **localhost wants to track your physical location** appears, click **Allow once**.
4. Scroll down to the venue map and hover the mouse over **Room A**.
5. Verify that the venue map looks similar to the following image:

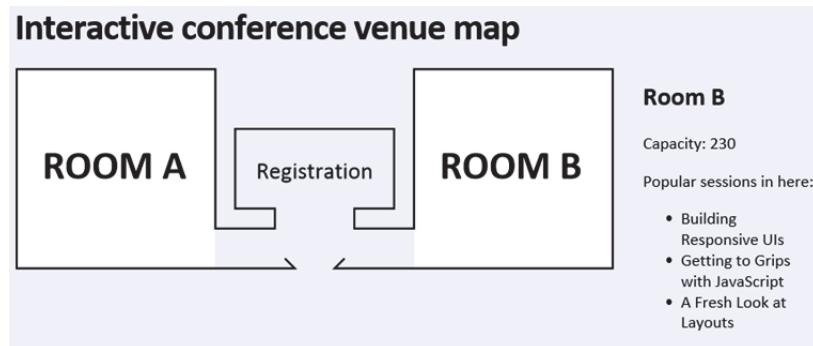
The venue map should look like this:



**FIGURE 11.2:THE VENUE MAP WITH ROOM A HIGHLIGHTED**

6. Click **Room B**, verify that the room information is displayed, and that it is similar to the following image:

The information displayed for Room B should look like this:



7. Close Internet Explorer.

**Results:** After completing this exercise, you will have a venue map that displays extra information when clicked.

## Exercise 2: Creating a Speaker Badge by Using the Canvas API

### ► Task 1: Create the canvas element

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 2** click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, and then double-click **speaker-badge.htm**.
4. Find the following comment:

```
<!-- TODO: Add canvas here -->
```

5. After this comment, add the following HTML markup:

```
<canvas
 width="500"
 height="200"
 style="border: 1px solid "#888"
 data-speaker-id="234724"
 data-speaker-name="Mark Hanson">
</canvas>
```

### ► Task 2: Draw the details for the badge

1. In Solution Explorer, expand the **scripts** folder, expand the **pages** folder, and then double-click **speaker-badge.js**.
2. Verify that the file contains the following JavaScript code:

```
this.canvas = element.querySelector("canvas");
```

3. Find the following comment:

```
// TODO: Get the canvas's (this.canvas) context and assign to this.context
```

4. After this comment, add the following JavaScript code:

```
this.context = this.canvas.getContext("2d");
```

5. Find the comment that starts with the following line:

```
// TODO: Draw the following by calling the helper methods of `this`
```

6. After the last line of this comment, add the following JavaScript code:

```
this.drawBackground();
this.drawTopText();
this.drawSpeakerName();
if (image) {
 this.drawSpeakerImage(image);
} else {
 this.drawImagePlaceholder();
}
this.drawBarcode(this.speakerId);
```

7. Find the following comment:

```
// TODO: Fill the canvas with a white rectangle
```

8. After this comment, add the following JavaScript code:

```
this.context.fillStyle = "white";
this.context.fillRect(0, 0, this.canvas.width, this.canvas.height);
```

9. Find the comment that starts with the following line:

```
// TODO: Draw the image on the canvas
```

10. After the last line of this comment, add the following JavaScript code:

```
var size = Math.min(image.width, image.height);
var sourceX = image.width / 2 - size / 2;
var sourceY = image.height / 2 - size / 2;
this.context.drawImage(image, sourceX, sourceY, size, size, 20, 20, 160, 160);
```

11. Find the comment that starts with the following line:

```
// TODO: Draw this.speakerName on the canvas
```

12. After the last line of this comment, add the following JavaScript code:

```
this.context.font = "40px sans-serif";
this.context.fillStyle = "black";
this.context.textBaseline = "top";
this.context.textAlign = "left";
this.context.fillText(this.speakerName, 200, 60);
```

### ► Task 3: Test the application

1. In Solution Explorer, double-click **speaker-badge.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. On the taskbar, click **File Explorer** and then browse to the **E:\Mod11\Labfiles\Resources** folder.
4. Drag and drop **mark-hanson.jpg** from File Explorer onto the speaker badge in Internet Explorer.
5. Verify that the speaker badge looks similar to the following image:

The speaker badge should look like this:



**FIGURE 11.4:THE SPEAKER BADGE FOR MARK HANSON**

6. Close Internet Explorer.

**Results:** After completing this exercise, you will have a Speaker Badge page that enables a conference speaker to create their badge.

# Module 12: Animating the User Interface

## Lab: Animating the User Interface

### Exercise 1: Applying CSS Transitions

#### ► Task 1: Review the Feedback page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project node, and then double-click **feedback.htm**.
8. Verify that this page contains the following HTML markup:

```
<form method="post" action="/send-feedback">
 <div class="field feedback-question">
 <label>How would you rate the speaker's knowledge of the topic?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How well could you hear the speaker?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How useful did you find the information in this talk?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field feedback-question">
 <label>How would you rate the overall session?</label>
 <input name="question" type="range" min="1" max="5"/>
 </div>
 <div class="field comments">
 <label>Any additional comments?</label>
 <textarea name="comments" cols="30" rows="5"></textarea>
 </div>
 <div class="field actions">
 <button type="submit">Send Feedback</button>
 </div>
</form>
```

9. Verify that the HTML contains the following:

```
<link href="/styles/pages/feedback.css" rel="stylesheet" type="text/css" />
```

10. Verify that the HTML contains the following:

```
<script src="/scripts/pages/feedback.js" type="text/javascript"></script>
```

11. On the **Debug** menu, click **Start Without Debugging**.
12. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
13. Verify that the **Feedback** form is displayed and has five star icons next to each question.
14. Close Internet Explorer.

► **Task 2: Animate the stars on the Feedback form**

1. In Solution Explorer, expand the **styles** folder, then expand the **pages**, and then double-click **feedback.css**.
2. Update the following CSS rule, and add the code shown below in bold:

```
.star:hover,
.star.hover {
 background-position: 0 -15px;
 /* TODO: Scale transform by 1.3 */
transform: scale(1.3, 1.3);
 /* TODO: Transition the transform property over 0.5 seconds */
transition: transform .5s;
}
```

3. Update the following CSS rule, and add the code shown below in bold:

```
.star {
 background-image: url(..../images/stars.png);
 width: 15px;
 height: 15px;
 cursor: pointer;
 margin: .1rem;
 -ms-flex: 0 0 15px;
 flex: 0 0 15px;
 /* TODO: Transition the transform property over 0.5 seconds */
transition: transform .5s;
}
```

4. Update the following CSS rule, and add the code shown below in bold:

```
.star.selected {
 background-position: 0 -30px;
 /* TODO: Scale transform by 1.3 */
transform: scale(1.3, 1.3);
}
```

► **Task 3: Animate the Register link on the Home page**

1. In Solution Explorer, in the **styles** folder, double-click **header.css**.
2. Update the following CSS rule, and add the code shown below in bold:

```
header.page-header .register:hover {
 color: #fff;
 background: -ms-linear-gradient(#bc0101, #8c0909);
 background: linear-gradient(#bc0101, #8c0909);
 /* TODO: rotate to 16 degrees and scale by 1.1 */
transform: rotate(16deg) scale(1.1,1.1);
 /* TODO: Transition the transform property over 1 second */
transition: transform 1s;
}
```

3. Update the following CSS rule, and add the code shown below in bold:

```
header.page-header .register {
 display: block;
 position: absolute;
 top: 20px;
 right: 35px;
 width: 160px;
 height: 100px;
 padding-top: 60px;
 opacity: 0.8;
 font-size: 2.7rem;
 color: #fff;
 text-align: center;
 text-decoration: none;
 text-transform: uppercase;
 -ms-border-radius: 100%;
 border-radius: 100%;
 -ms-text-shadow: 0 1px 0 #000;
 text-shadow: 0 1px 0 #000;
 -ms-transform: rotate(6deg);
 transform: rotate(6deg);
 background: -ms-linear-gradient(#a80000, #740404);
 background: linear-gradient(#a80000, #740404);
 /* TODO: Transition the transform property over 1 second */
 transition: transform 1s;
}
```

#### ► Task 4: Test the application

1. In Solution Explorer, double-click **feedback.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, move the mouse over the third star of **How would you rate the speaker's knowledge of the topic?**.
4. Verify that the first three stars animate to a larger size.
5. Click the third star, and then move the mouse away from the stars.
6. Verify that the first three stars remain a larger size.
7. Click **Home**.
8. Move the mouse over the **Register Free** link and verify that it rotates and enlarges.
9. Close Internet Explorer.

**Results:** After completing this exercise, the **Register** button will rotate and the feedback stars will animate when the mouse moves over them.

## Exercise 2: Applying Keyframe Animations

### ► Task 1: Define a keyframe animation

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 2**, click **ContosoCon.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, expand the **styles** folder, expand the **pages** folder, and then double-click **feedback.css**.
4. Find the following comment:

```
/* TODO: Add key frame animation named "send"
 At 0% scale(1)
 At 50% scale(.8)
 At 100% translate(0, -1000px)
*/
```

5. After this comment, add the following CSS code:

```
@keyframes send {
 0% {
 transform: scale(1);
 }
 50% {
 transform: scale(.8);
 }
 100% {
 transform: translate(0, -1000px);
 }
}
```

6. Find the following comment:

```
/* TODO: Use the "send" animation
 iteration-count: 1
 fill-mode: forwards
*/
```

7. After this comment, add the following set of properties and values:

```
animation-name: send;
animation-duration: 1s;
animation-iteration-count: 1;
animation-fill-mode: forwards;
```

### ► Task 2: Trigger the animation

1. In Solution Explorer, expand the **scripts** folder, expand the **pages** folder, and then double-click **feedback.js**.
2. Find the following comment:

```
// TODO: Trigger the animation by adding the "sending" CSS class to the form
```
3. After this comment, add the following JavaScript code:

```
form.classList.add("sending");
```
4. find the following comment:

```
// TODO: Add listener for the animationend event,
// calling the animationEnded function.
```

5. After this comment, add the following JavaScript code:

```
form.addEventListener("animationend", animationEnded, false);
```

► **Task 3: Test the application**

1. In Solution Explorer, double-click **feedback.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, click **Send Feedback**.
4. Verify that the form shrinks and flies off the top of the page.
5. Close Internet Explorer.

**Results:** After completing this exercise, submitting the conference feedback form will trigger an animation that makes the form fly off the page.



## Module 13: Implementing Real-time Communication by Using Web Sockets

# Lab: Performing Real-time Communication by Using Web Sockets

### Exercise 1: Receiving Messages from a Web Socket

#### ► Task 1: Review the Live page

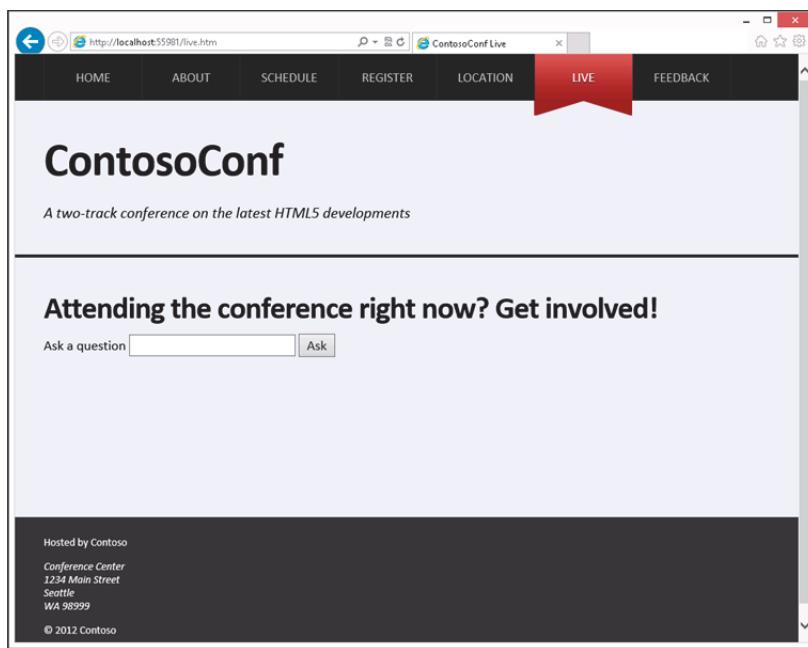
1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod13\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. On the **Debug** menu, click **Start Without Debugging**.
8. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message** again.
9. Click **Live**.
10. In the **Webpage Error** dialog box, click **No**.
11. Verify that the **Live** page looks like the following image:

The **Live** page should look like this:



**FIGURE 13.1:THE LIVE PAGE**

12. Close Internet Explorer and return to Visual Studio.
13. In Solution Explorer, expand the **ContosoConf** project, and then double-click **live.htm**.
14. Verify that this page contains the following HTML markup:

```
<form action="#">
 <label for="ask-question-text">Ask a question</label>
 <input id="ask-question-text" type="text" />
 <button type="submit">Ask</button>
</form>

 <!-- Questions will be displayed here when received by the web socket. -->

```

15. Verify that the page contains the following HTML markup:

```
<script src="/scripts/pages/live.js" type="text/javascript"></script>
```

16. In Solution Explorer, expand the **scripts** folder, expand the **pages** folder, and then double-click **live.js**.
17. Verify that the file contains the following JavaScript code:

```
var LivePage = Object.inherit({
```

#### ► Task 2: Receive messages by using a web socket

1. In **live.js**, find the following comment near the end of the file:

```
// TODO: Create a web socket connection to ws://localhost:55981/live/socket.ashx
```

2. After this comment, add the following JavaScript code:

```
var socket = new WebSocket("ws://localhost:55981/live/socket.ashx");
```

3. Scroll to the top of the file and find the following comment:

```
// TODO: Assign a callback to handle messages from the socket.
```

4. After this comment, add the following JavaScript code:

```
this.socket.onmessage = this.handleSocketMessage.bind(this);
```

► **Task 3: Display questions received as messages**

1. In **live.js**, find the following comment:

```
// TODO: Parse the event data into message object.
```

2. After this comment, add the following JavaScript code:

```
var message = JSON.parse(event.data);
```

3. Find the following comment:

```
// TODO: Check if message has a `questions` property, before calling
handleQuestionsMessage
```

4. Modify the statement following this comment, as shown below in bold:

```
if (message.questions) {
 this.handleQuestionsMessage(message);
}
```

5. Find the following code:

```
displayQuestion: function (question) {
 var item = this.createQuestionItem(question);
 //item.appendChild(this.createReportLink());
 this.questionListElement.appendChild(item);
}
```

6. Find the following comment in the **handleQuestionsMessage()** function:

```
// TODO: Display each question in the page, using the displayQuestion function.
```

7. After this comment, add the following JavaScript code:

```
if (message.questions) {
 message.questions.forEach(this.displayQuestion, this);
}
```

► **Task 4: Test the application**

1. In Solution Explorer, double-click **live.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, once the page has loaded, wait five seconds.
4. Verify that the page displays the following four questions:
  - What are some good resources for getting started with HTML5?
  - Can you explain more about the Web Socket API, please?
  - This is an #&!%!\* inappropriate message!!
  - How much of CSS3 can I use right now?

5. Close Internet Explorer.

**Results:** After completing this exercise, you will have added JavaScript code to the **Live** web page to receive questions from a web socket and to display them.

MCT USE ONLY. STUDENT USE PROHIBITED

## Exercise 2: Sending Messages to a Web Socket

### ► Task 1: Format a question as a message

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod13\Labfiles\Starter\Exercise 2**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, expand the **pages** folder, and then double-click **live.js**.
4. Verify that the **live.js** file contains the following function:

```
askQuestion: function (text) {
 // TODO: Create a message object with the format { ask: text }
 // TODO: Convert the message object into a JSON string
 // TODO: Send the message to the socket
 // Clear the input ready for another question.
 this.questionInput.value = "";
}
```

5. In the **askQuestion()** function, find the following comment:

```
// TODO: Create a message object with the format { ask: text }
```

6. After this comment, add the following JavaScript code:

```
var message = {
 ask: text
};
```

7. Find the following comment:

```
// TODO: Convert the message object into a JSON string
```

8. After this comment, add the following JavaScript code:

```
var json = JSON.stringify(message);
```

### ► Task 2: Send the message to the web socket

1. In **live.js**, find the following comment:

```
// TODO: Send the message to the socket
```

2. After this comment, add the following JavaScript code:

```
this.socket.send(json);
```

### ► Task 3: Test the application

1. In Solution Explorer, double-click **live.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, press Ctrl + T to open a new tab.
4. In the **Address bar**, type **http://localhost:55981/live.htm**, and then press **ENTER**.
5. In the **Ask a question** box, type **This is a test.**, and then click **Ask**.
6. Verify that the list of questions displays **This is a test.**.

7. Click the first Internet Explorer tab.
8. Verify that the list of questions includes **This is a test**.
9. Close Internet Explorer.

**Results:** After completing this exercise, you will have modified the **Live** question page to enable users to ask questions by sending messages to the server by using the web socket.

## Exercise 3: Handling Different Web Socket Message Types

### ► Task 1: Display report links

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod13\Labfiles\Starter\Exercise 3**, click **ContosoConf.sln**, and then click **Open**.
3. In Solution Explorer, expand the **ContosoConf** project, expand the **scripts** folder, expand the **pages** folder, and then double-click **live.js**.
4. In **live.js**, find the following code:

```
createReportLink: function () {
 var report = document.createElement("a");
 report.textContent = "Report";
 report.setAttribute("href", "#");
 report.setAttribute("class", "report");
 return report;
}
```

5. In the **displayQuestion()** function, find the following comment:

```
//item.appendChild(this.createReportLink());
```

6. Modify this statement and remove the **//** characters, as shown in the following JavaScript code:

```
item.appendChild(this.createReportLink());
```

### ► Task 2: Send the report message

1. In **live.js**, find the following function:

```
handleQuestionsClick: function (event) {
 event.preventDefault();
 var clickedElement = event.srcElement || event.target;
 if (this.isReportLink(clickedElement)) {
 var questionId = clickedElement.parentNode.questionId;
 this.reportQuestion(questionId);
 clickedElement.textContent = "Reported";
 }
}
```

2. In the **reportQuestion()** function, find the following comment:

```
// TODO: Send socket message { report: questionId }
```

3. After the comment, add the following code:

```
this.socket.send(JSON.stringify({ report: questionId }));
```

### ► Task 3: Handle Remove Question messages

1. In **live.js**, find the **handleRemoveMessage()** function:

```
handleRemoveMessage: function (message) {
 var listItems = this.questionListElement.querySelectorAll("li");
 for (var i = 0; i < listItems.length; i++) {
 if (listItems[i].questionId === message.remove) {
 this.questionListElement.removeChild(listItems[i]);
 break;
 }
 }
}
```

```
 }
}
```

2. Find the **handleSocketMessage** method:

```
handleSocketMessage: function (event) {
 // TODO: Parse the event data into message object.
 var message = JSON.parse(event.data);
 // TODO: Check if message has a `questions` property, before calling
 handleQuestionsMessage
 if (message.questions) {
 this.handleQuestionsMessage(message);
 }
}
```

3. Modify the JavaScript code in the **handleSocketMessage** method as shown below in bold:

```
handleSocketMessage: function (event) {
 // TODO: Parse the event data into message object.
 var message = JSON.parse(event.data);
 if (message.questions) {
 this.handleQuestionsMessage(message);
 } else if (message.remove) {
 this.handleRemoveMessage(message);
 }
}
```

#### ► Task 4: Test the application

1. In Solution Explorer, double-click **live.htm**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In Internet Explorer, after the page loads, wait five seconds.
4. Next to **What are some good resources for getting started with HTML5?**, click **Report**.
5. Wait for several seconds.
6. Verify that the **What are some good resources for getting started with HTML5?** question is removed from the page.
7. Close Internet Explorer.

**Results:** After completing this exercise, you will have added a feature to the **Live** page that enables users to report inappropriate questions and causes the application to remove them.

## Module 14: Performing Background Processing by Using Web Workers

# Lab: Creating a Web Worker Process

### Exercise 1: Improving Responsiveness by Using a Web Worker

► Task 1: Review the Speaker Badge page

1. Start the **MSL-TMG1** virtual machine if it is not already running.
2. Start the **20480B-SEA-DEV11** virtual machine if it is not already running.
3. Log on to the **20480B-SEA-DEV11** virtual machine as **Student** with the password **Pa\$\$w0rd**.



**Note:** If necessary, click **Switch User** to display the list of users.

4. On the Windows 8 **Start** screen, click the **Visual Studio 2012** tile.
5. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, browse to **E:\Mod14\Labfiles\Starter\Exercise 1**, click **ContosoConf.sln**, and then click **Open**.
7. In Solution Explorer, expand the **ContosoConf** project, and then double-click **speaker-badge.htm**.
8. Verify that the page contains the following HTML markup:

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

9. In Solution Explorer, expand the **scripts** folder, and then double-click **grayscale.js**.
10. Verify that the file contains the following JavaScript function:

```
conference.grayscaleImage = function (image) {
 // Converts a colour image into gray scale.
 var deferred = $.Deferred();
 var canvas = createCanvas(image);
 var context = canvas.getContext("2d");
 var imageData = getImageData(context, image);
 // TODO: Create a Worker that runs /scripts/grayscale-worker.js
 var pixels = imageData.data;
 // 4 array items per pixel => Red, Green, Blue, Alpha
 for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
 }
 // Update the canvas with the gray scaled image data.
 context.clearRect(0, 0, canvas.width, canvas.height);
 context.putImageData(imageData, 0, 0);
 // Returning a jQuery Deferred makes this function easy to chain together with
 // other deferred operations.
 // The canvas object is returned as this can be used like an image.
 deferred.resolveWith(this, [canvas]);
 return deferred;
};
```

► Task 2: Convert the speaker badge image to grayscale

1. In Solution Explorer, in the **scripts** folder, expand the **pages** folder, and then double-click **speaker-badge.js**.

2. In **speaker-badge.js**, find the following comment:

```
// TODO: Add grayscaleImage into the processing pipeline.
```

3. Modify the code following this comment, as shown below in bold:

```
this.readFile(file)
 .pipe(this.loadImage)
.pipe(grayscaleImage)
 .done(this.drawBadge, this.notBusy);
```

4. In Solution Explorer, double-click **speaker-badge.htm**.
5. On the **Debug** menu, click **Start Without Debugging**.
6. In Internet Explorer, if the message **Intranet settings are turned off by default** appears, click **Don't show this message again**.
7. On the Windows taskbar, click **File Explorer**.
8. Browse to the folder **E:\Mod14\Labfiles\Resources**.
9. Drag **mark-hanson-large.jpg** from Windows Explorer into Internet Explorer and drop it onto the canvas with the label **Drag your profile photo here**.
10. Verify that you cannot scroll the page or move to another page until after the image has been displayed. This may take several seconds.
11. If Internet Explorer displays the message **localhost is not responding due to a long-running script**, wait for the message to disappear.
12. Close Internet Explorer.

#### ► Task 3: Create a web worker to perform image processing

1. In Visual Studio, in Solution Explorer, in the **scripts** folder, double-click **grayscale.js**

2. Find the following comment:

```
// TODO: Create a Worker that runs /scripts/grayscale-worker.js
```

3. After this comment, add the following JavaScript code:

```
var worker = new Worker("/scripts/grayscale-worker.js");
worker.postMessage(imageData);
```

4. In Solution Explorer, in the **scripts** folder, double-click **grayscale-worker.js**.

5. Add the following JavaScript code to this file, after the comment at the top:

```
addEventListener("message", function (event) {
});
```

6. In Solution Explorer, double-click **grayscale.js**.

7. Find the following JavaScript statements:

```
var worker = new Worker("/scripts/grayscale-worker.js");
worker.postMessage(imageData);
```

8. Insert the following code shown in bold between these two statements:

```
var worker = new Worker("/scripts/grayscale-worker.js");
```

```
var handleMessage = function (event) {
};
worker.addEventListener("message", handleMessage.bind(this));
worker.postMessage(imageData);
```

► Task 4: Move image processing code into the web worker

1. In **grayscale.js**, cut the following code that defines the **grayscalePixel** function to the clipboard:

```
var grayscalePixel = function (pixels, index) {
 // Updates the pixel, starting at the given index, to be gray
 var brightness = 0.34 * pixels[index] + 0.5 * pixels[index + 1] + 0.16 *
 pixels[index + 2];
 pixels[index] = brightness; // red
 pixels[index + 1] = brightness; // green
 pixels[index + 2] = brightness; // blue
};
```

2. In **grayscale-worker.js**, underneath the existing code, paste the JavaScript code for the **grayscalePixel** function from the clipboard.
3. In **grayscale.js**, in the **grayscaleImage()** function, delete the following code:

```
var pixels = imageData.data;
// 4 array items per pixel => Red, Green, Blue, Alpha
for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
}
```

4. In **grayscale-worker.js**, add the following code shown in bold to the **addEventListener** statement:

```
addEventListener("message", function (event) {
 var imageData = event.data;
 var pixels = imageData.data;
 for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
 }
});
```

► Task 5: Return image data from the web worker

1. In **grayscale-worker.js**, after the **for** loop in the **addEventListener** statement, add the following code shown in bold:

```
for (var i = 0; i < pixels.length; i += 4) {
 grayscalePixel(pixels, i);
}
postMessage({ done: imageData });
```

2. In **grayscale.js**, in the **grayscaleImage()** function, delete the following code:

```
// Update the canvas with the gray scaled image data.
context.clearRect(0, 0, canvas.width, canvas.height);
context.putImageData(imageData, 0, 0);
// Returning a jQuery Deferred makes this function easy to chain together with other
// deferred operations.
// The canvas object is returned as this can be used like an image.
deferred.resolveWith(this, [canvas]);
```

3. Find the following JavaScript statement:

```
var handleMessage = function (event) { };
```

4. Add following JavaScript code shown in bold to this statement:

```
var handleMessage = function (event) {
 var message = event.data;
 var updatedImageData = message.done;
 // Update the canvas with the gray scaled image data.
 context.clearRect(0, 0, canvas.width, canvas.height);
 context.putImageData(updatedImageData, 0, 0);
 deferred.resolveWith(this, [canvas]);
};
```

► **Task 6: Test the application**

1. On the Windows taskbar, click the Internet Explorer icon.
2. In Internet Explorer, press F12.
3. On the **Cache** menu, click **Clear browser cache**.
4. In the **Clear Browser Cache** message box, click **Yes**.
5. Press F12, and then close Internet Explorer.
6. In Visual Studio, in Solution Explorer, double-click **speaker-badge.htm**.
7. On the **Debug** menu, click **Start Without Debugging**.
8. On the taskbar, click **File Explorer**.
9. Browse to the **E:\Mod14\Labfiles\Resources** folder.
10. Drag **mark-hanson-large.jpg** from File Explorer into Internet Explorer and drop it onto the canvas with the label **Drag your profile photo here**.
11. Verify that you can scroll the page up and down while and image is being processed.
12. Close Internet Explorer.

**Results:** After completing this exercise, you will have created a web page that remains responsive while slow image processing code runs in a web worker.