



O F F I C I A L M I C R O S O F T L E A R N I N G P R O D U C T

20480C

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.

© 2018 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at <https://www.microsoft.com/en-us/legal/intellectualproperty/trademarks/en-us.aspx> are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners

Product Number: 20480C

Part Number: X21-94599

Released: 11/2018

MICROSOFT LICENSE TERMS
MICROSOFT INSTRUCTOR-LED COURSEWARE

These license terms are an agreement between Microsoft Corporation (or based on where you live, one of its affiliates) and you. Please read them. They apply to your use of the content accompanying this agreement which includes the media on which you received it, if any. These license terms also apply to Trainer Content and any updates and supplements for the Licensed Content unless other terms accompany those items. If so, those terms apply.

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

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

1. DEFINITIONS.

- a. "Authorized Learning Center" means a Microsoft IT Academy Program Member, Microsoft Learning Competency Member, or such other entity as Microsoft may designate from time to time.
- b. "Authorized Training Session" means the instructor-led training class using Microsoft Instructor-Led Courseware conducted by a Trainer at or through an Authorized Learning Center.
- c. "Classroom Device" means one (1) dedicated, secure computer that an Authorized Learning Center owns or controls that is located at an Authorized Learning Center's training facilities that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
- d. "End User" means an individual who is (i) duly enrolled in and attending 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 content accompanying this agreement which may include the Microsoft Instructor-Led Courseware or Trainer Content.
- 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, and (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program.
- g. "Microsoft Instructor-Led Courseware" means the Microsoft-branded instructor-led training course that educates IT professionals and developers on Microsoft technologies. A Microsoft Instructor-Led Courseware title may be branded as MOC, Microsoft Dynamics or Microsoft Business Group courseware.
- h. "Microsoft IT Academy Program Member" means an active member of the Microsoft IT Academy Program.
- i. "Microsoft Learning Competency Member" means an active member of the Microsoft Partner Network program in good standing that currently holds the Learning Competency status.
- j. "MOC" means the "Official Microsoft Learning Product" instructor-led courseware known as Microsoft Official Course that educates IT professionals and developers on Microsoft technologies.
- k. "MPN Member" means an active Microsoft Partner Network program member in good standing.

- I. "Personal Device" means one (1) personal computer, device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular Microsoft Instructor-Led Courseware.
 - m. "Private Training Session" means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective using Microsoft Instructor-Led Courseware. 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.
 - n. "Trainer" means (i) an academically accredited educator engaged by a Microsoft IT Academy Program Member to teach an Authorized Training Session, and/or (ii) a MCT.
 - o. "Trainer Content" means the trainer version of the Microsoft Instructor-Led Courseware and additional supplemental content designated solely for Trainers' use to teach a training session using the Microsoft Instructor-Led Courseware. Trainer Content may include Microsoft PowerPoint presentations, trainer preparation guide, train the trainer materials, Microsoft One Note packs, classroom setup guide and Pre-release course feedback form. To clarify, Trainer Content does not include any software, virtual hard disks or virtual machines.
- 2. 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 five separate sets of use rights. Only one set of rights apply to you.

a. **If you are a Microsoft IT Academy Program Member:**

- i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
- ii. For each license you acquire on behalf of an End User or Trainer, you may either:
 - 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User who is enrolled in the Authorized Training Session, and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
 - 2. provide one (1) End User with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 - 3. provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content,

provided you comply with the following:

- iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
- iv. you will ensure each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
- v. you will ensure that each End User provided with the hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
- vi. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,

- vii. you will only use qualified Trainers who have in-depth knowledge of and experience with the Microsoft technology that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Authorized Training Sessions,
- viii. you will only deliver a maximum of 15 hours of training per week for each Authorized Training Session that uses a MOC title, and
- ix. you acknowledge that Trainers that are not MCTs will not have access to all of the trainer resources for the Microsoft Instructor-Led Courseware.

b. **If you are a Microsoft Learning Competency Member:**

- i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
- ii. For each license you acquire on behalf of an End User or Trainer, you may either:
 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Authorized Training Session and only immediately prior to the commencement of the Authorized Training Session that is the subject matter of the Microsoft Instructor-Led Courseware provided, **or**
 2. provide one (1) End User attending the Authorized Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 3. you will provide one (1) Trainer with the unique redemption code and instructions on how they can access one (1) Trainer Content,

provided you comply with the following:

- iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
- iv. you will ensure that each End User attending an Authorized Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Authorized Training Session,
- v. you will ensure that each End User provided with a hard-copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
- vi. you will ensure that each Trainer teaching an Authorized Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Authorized Training Session,
- vii. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for your Authorized Training Sessions,
- viii. you will only use qualified MCTs who also hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Authorized Training Sessions using MOC,
- ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
- x. you will only provide access to the Trainer Content to Trainers.

c. **If you are a MPN Member:**

- i. Each license acquired on behalf of yourself may only be used to review one (1) copy of the Microsoft Instructor-Led Courseware in the form provided to you. If the Microsoft Instructor-Led Courseware is in digital format, you may install one (1) copy on up to three (3) Personal Devices. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.
 - ii. For each license you acquire on behalf of an End User or Trainer, you may either:
 1. distribute one (1) hard copy version of the Microsoft Instructor-Led Courseware to one (1) End User attending the Private Training Session, and only immediately prior to the commencement of the Private Training Session that is the subject matter of the Microsoft Instructor-Led Courseware being provided, **or**
 2. provide one (1) End User who is attending the Private Training Session with the unique redemption code and instructions on how they can access one (1) digital version of the Microsoft Instructor-Led Courseware, **or**
 3. you will provide one (1) Trainer who is teaching the Private Training Session with the unique redemption code and instructions on how they can access one (1) Trainer Content,
- provided you comply with the following:**
- iii. you will only provide access to the Licensed Content to those individuals who have acquired a valid license to the Licensed Content,
 - iv. you will ensure that each End User attending an Private Training Session has their own valid licensed copy of the Microsoft Instructor-Led Courseware that is the subject of the Private Training Session,
 - v. you will ensure that each End User provided with a hard copy version of the Microsoft Instructor-Led Courseware will be presented with a copy of this agreement and each End User will agree that their use of the Microsoft Instructor-Led Courseware will be subject to the terms in this agreement prior to providing them with the Microsoft Instructor-Led Courseware. Each individual will be required to denote their acceptance of this agreement in a manner that is enforceable under local law prior to their accessing the Microsoft Instructor-Led Courseware,
 - vi. you will ensure that each Trainer teaching an Private Training Session has their own valid licensed copy of the Trainer Content that is the subject of the Private Training Session,
 - vii. you will only use qualified Trainers who hold the applicable Microsoft Certification credential that is the subject of the Microsoft Instructor-Led Courseware being taught for all your Private Training Sessions,
 - viii. you will only use qualified MCTs who hold the applicable Microsoft Certification credential that is the subject of the MOC title being taught for all your Private Training Sessions using MOC,
 - ix. you will only provide access to the Microsoft Instructor-Led Courseware to End Users, and
 - x. you will only provide access to the Trainer Content to Trainers.

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

For each license you acquire, you may use the Microsoft Instructor-Led Courseware solely for your personal training use. If the Microsoft Instructor-Led Courseware is in digital format, you may access the Microsoft Instructor-Led Courseware online using the unique redemption code provided to you by the training provider and install and use one (1) copy of the Microsoft Instructor-Led Courseware on up to three (3) Personal Devices. You may also print one (1) copy of the Microsoft Instructor-Led Courseware. You may not install the Microsoft Instructor-Led Courseware on a device you do not own or control.

e. **If you are a Trainer.**

- i. For each license you acquire, you may install and use one (1) copy of the Trainer Content in the form provided to you on one (1) Personal Device solely to prepare and deliver an Authorized Training Session or Private Training Session, and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Trainer Content. You may not install or use a copy of the Trainer Content on a device you do not own or control. You may also print one (1) copy of the Trainer Content solely to prepare for and deliver an Authorized Training Session or Private Training Session.

- ii. You may customize the written portions of the Trainer Content that are logically associated with instruction of a training session in accordance with the most recent version of the MCT agreement. If you elect to exercise the foregoing rights, you agree to comply with the following: (i) customizations may only be used for teaching Authorized Training Sessions and Private Training Sessions, and (ii) all customizations will comply 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 is licensed as a single unit and you may not separate their components and install them on different devices.

2.3 Redistribution of Licensed Content. Except as expressly provided in the use rights above, you may not distribute any 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 Notices. The Licensed Content may include third party code tent that Microsoft, not the third party, licenses to you under this agreement. Notices, if any, for the third party code ntent are included for your information only.

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 your use of that respective component and supplements the terms described in this agreement.

3. LICENSED CONTENT BASED ON PRE-RELEASE TECHNOLOGY. If the Licensed Content's subject matter is based on a pre-release version of Microsoft technology ("Pre-release"), then in addition to the other provisions in this agreement, these terms also apply:

- a. **Pre-Release Licensed Content.** This Licensed Content subject matter is on the Pre-release version of the Microsoft technology. The technology may not work the way a final version of the technology will and we may change the technology for the final version. We also may not release a final version. Licensed Content based on the final version of the technology may not contain the same information as the Licensed Content based on the Pre-release version. Microsoft is under no obligation to provide you with any further content, including any Licensed Content based on the final version of the technology.
- 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 technology, 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 technology, technologies, or products to third parties because we include your feedback in them. These rights survive this agreement.
- c. **Pre-release Term.** If you are an Microsoft IT Academy Program Member, Microsoft Learning Competency Member, MPN Member or Trainer, you will cease using all copies of the Licensed Content on the Pre-release technology upon (i) the date which Microsoft informs you is the end date for using the Licensed Content on the Pre-release technology, or (ii) sixty (60) days after the commercial release of the technology that is the subject of the Licensed Content, whichever is earliest ("Pre-release term"). Upon expiration or termination of the Pre-release term, you will irretrievably delete and destroy all copies of the Licensed Content in your possession or under your control.

- 4. 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:
- access or allow any individual to access the Licensed Content if they have not acquired a valid license for the Licensed Content,
 - alter, remove or obscure any copyright or other protective notices (including watermarks), branding or identifications contained in the Licensed Content,
 - modify or create a derivative work of any Licensed Content,
 - publicly display, or make the Licensed Content available for others to access or use,
 - copy, print, install, sell, publish, transmit, lend, adapt, reuse, link to or post, make available or distribute the Licensed Content to any third party,
 - work around any technical limitations in the Licensed Content, or
 - 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.
- 5. 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.
- 6. 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.
- 7. SUPPORT SERVICES.** Because the Licensed Content is "as is", we may not provide support services for it.
- 8. 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 termination of this agreement for any reason, you will immediately stop all use of and delete and destroy all copies of the Licensed Content in your possession or under your control.
- 9. 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.
- 10. ENTIRE AGREEMENT.** This agreement, and any additional terms for the Trainer Content, updates and supplements are the entire agreement for the Licensed Content, updates and supplements.
- 11. 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.
- 12. 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.
- 13. DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS" AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT AND ITS RESPECTIVE AFFILIATES GIVES NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT AND ITS RESPECTIVE AFFILIATES EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**
- 14. LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT, ITS RESPECTIVE AFFILIATES AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO US\$5.00. YOU CANNOT RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES.**

This limitation applies to

- anything related to the Licensed Content, services, 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.

Revised July 2013

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

Acknowledgements

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.

Ishai Ram – Content Development Lead

Ishai is the Vice President of SELA Group. He has over 20 years of experience as a professional trainer and consultant on computer software and electronics.

Baruch Toledano – Senior Content Developer

Baruch is a senior project manager at SELA Group. He has extensive experience in producing Microsoft Official Courses and managing software development projects. Baruch is also a lecturer at SELA College delivering a variety of development courses.

Sasha Goldshtain – Subject Matter Expert

Sasha Goldshtain is the CTO at Sela Group, a Microsoft C# MVP and Regional Director, a Pluralsight and O'Reilly author, and an international consultant and trainer. Sasha is the author of "Introducing Windows 7 for Developers" (Microsoft Press, 2009) and "Pro .NET Performance" (Apress, 2012). His is also a prolific blogger and open source contributor, and author of numerous training courses including .NET Debugging, .NET Performance, Android Application Development, and Modern C++. His consulting work revolves mainly around distributed architecture, production debugging and performance diagnostics, and mobile application development.

Yonatan Horovitz - Subject Matter Expert

Yonatan is a consultant at SELA Group. Yonatan has many years of experience developing both client and server sides application in .Net. Specializing in client-side and XAML, as well as developing UWP applications. Yonatan also has a deep interest in .Net internals and performance optimizations.

Roi Godelman- Subject Matter Expert

Roi is a senior developer and lecturer at SELA Group. Roi has over five years' experience in developing desktop, web and mobile applications. Roi is a full stack developer, specializing in both front-end and back-end development. Roi delivers many courses in the IT industry.

Viacheslav Brekel - Subject Matter Expert

Viacheslav is a senior developer and lecturer at SELA Group. Viacheslav has six years' experience in developing and maintaining large-scale solutions in a variety of technologies. Viacheslav is a proficient problem solver and content developer. Viacheslav's main technology interests vary between web and desktop development.

Shalev Zahavi- Subject Matter Expert

Shalev is a senior developer and lecturer at SELA Group. Shalev has over five years' experience in software development and a proven track record in development of large-scale hybrid applications. Shalev's main interest is in back-end solutions development. Shalev delivers many training sessions in the industry. Among his other fields of interest: Azure development, Web development and Mobile development.

Apposite Learning & SELA Teams – Content Contributors

Shelly Aharoni, Naor Michelsohn, Amith Vincent, Kavitha Ravipati, Vinay Antony, Sugato Deb, Dhananjaya Punugoti and the Enfec Team.

Contents

Module 1: Overview of HTML and CSS

Module Overview	1-1
Lesson 1: Overview of HTML	1-2
Lesson 2: Overview of CSS	1-13
Lesson 3: Creating a Web Application by Using Visual Studio 2017	1-20
Lab: Exploring the Contoso Conference Application	1-24
Module Review and Takeaways	1-26

Module 2: Creating and Styling HTML Pages

Module Overview	2-1
Lesson 1: Creating an HTML5 Page	2-2
Lesson 2: Styling an HTML5 Page	2-7
Lab: Creating and Styling HTML5 Pages	2-14
Module Review and Takeaways	2-15

Module 3: Introduction to JavaScript

Module Overview	3-1
Lesson 1: Overview of JavaScript	3-2
Lesson 2: Introduction to the Document Object Model	3-14
Lab: Displaying Data and Handling Events by Using JavaScript.	3-22
Module Review and Takeaways	3-23

Module 4: Creating Forms to Collect and Validate User Input

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	4-16

Module 5: Communicating with a Remote Server

Module Overview	5-1
Lesson 1: Async Programming in JavaScript	5-2
Lesson 2: Sending and Receiving Data by Using the XMLHttpRequest Object	5-8
Lesson 3: Sending and Receiving Data by Using the Fetch API	5-14
Lab: Communicating with a Remote Data Source	5-18
Module Review and Takeaways	5-20

Module 6: Styling HTML5 by Using CSS3

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	6-11
Lesson 4: Enhancing Graphical Effects by Using CSS3	6-15
Lab: Styling Text and Block Elements by Using CSS3	6-22
Module Review and Takeaways	6-24

Module 7: Creating Objects and Methods by Using JavaScript

Module Overview	7-1
Lesson 1: Writing Well-Structured JavaScript Code	7-2
Lesson 2: Creating Custom Objects	7-9
Lesson 3: Extending Objects	7-16
Lab: Refining Code for Maintainability and Extensibility	7-22
Module Review and Takeaways	7-23

Module 8: Creating Interactive Pages by Using HTML5 APIs

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	8-19
Module Review and Takeaways	8-21

Module 9: Adding Offline Support to Web Applications

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	9-16
Module Review and Takeaways	9-18

Module 10: Implementing an Adaptive User Interface

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	10-14
Module Review and Takeaways	10-15

Module 11: Creating Advanced Graphics

Module Overview	11-1
Lesson 1: Creating Interactive Graphics by Using SVG	11-2
Lesson 2: Drawing Graphics by Using the Canvas API	11-18
Lab: Creating Advanced Graphics	11-25
Module Review and Takeaways	11-26

Module 12: Animating the User Interface

Module Overview	12-1
Lesson 1: Applying CSS Transitions	12-2
Lesson 2: Transforming Elements	12-7
Lesson 3: Applying CSS Keyframe Animations	12-15
Lab: Animating the User Interface	12-20
Module Review and Takeaways	12-21

Module 13: Implementing Real-time Communication by Using WebSockets

Module Overview	13-1
Lesson 1: Introduction to WebSockets	13-2
Lesson 2: Using the WebSocket API	13-4
Lab: Performing Real-time Communication by Using WebSockets	13-10
Module Review and Takeaways	13-12

Module 14: Performing Background Processing by Using Web Workers

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	14-12

Module 15: Packaging JavaScript for Production Deployment

Module Overview	15-1
Lesson 1: Understanding Transpilers And Module Bundling	15-2
Lesson 2: Creating Separate Packages for Cross Browser Support	15-8
Lab: Setting up Webpack Bundle for Production	15-12

About This Course

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

Course Description

This course introduces HTML5, CSS3, and JavaScript to the students. 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 twelve months of programming experience and who are interested in developing web applications by using HTML5 with JavaScript and CSS3.

Additionally, the students can 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 Microsoft Visual Studio 2015 or Visual Studio 2017

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 webpage:
 - 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 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 2017 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 Fetch API.
- Style HTML5 pages by using CSS3.
- Create well-structured and easily-maintainable JavaScript code.
- Write modern JavaScript code and use Babel to make it compatible with all browsers.
- Use common HTML5 APIs in interactive web applications.
- Create web applications that support offline operations.
- Create HTML5 webpages that can adapt to different devices and form factors.
- Add advanced graphics to an HTML5 page by using elements from Canvas API and Scalable Vector Graphics (SVG).
- Enhance the user experience by adding animations to an HTML5 page.
- Use WebSockets 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.

Use webpack to package web applications for production.

Course Outline

The course outline is as follows:

Module 1. Overview of HTML and CSS

This module provides an overview of HTML and CSS. Much of the material on HTML will be a review, but it ensures that all the students are up to speed and understand the terminology used throughout the course.

Module 2. Creating and Styling HTML Pages

This module covers the new features in HTML5 and CSS3. It describes the new HTML5 elements, such as using the `<nav>`, `<article>`, `<section>`, `<aside>`, and `<footer>` tags to add semantic structure to an HTML document. It also covers markup for images and for text elements, such as `<hgroup>`, `<time>`, `<mark>`, `<small>`, `<figure>`, and `<figcaption>`.

Module 3. Introduction to JavaScript

This module describes how to use JavaScript to add dynamic functionality to a webpage.

Module 4. Creating Forms to Collect and Validate User Input

This module covers the new forms input features provided by HTML5.

Module 5. Communicating with a Remote Server

This module covers the creation of webpages that can communicate with web services.

Module 6. Styling HTML5 by Using CSS3

This module goes into more detail on using CSS to style text and graphics.

Module 7. Creating Objects and Methods by Using JavaScript

This module provides more information on JavaScript, concentrating on techniques and best practices for writing well-structured, maintainable code.

Module 8. Creating Interactive Pages by Using HTML5 APIs

This module shows some of the new features that have been added to HTML5 to enable webpages to interact with the user and with the operating system running the browser.

Module 9. Adding Offline Support to Web Applications

This module describes how to enable a web application to support offline operations.

Module 10. Implementing an Adaptive User Interface

This module describes how to use CSS styles to implement an adaptive user interface that can detect the form factor of the device running the browser, and to modify the layout of the content accordingly.

Module 11. Creating Advanced Graphics

This module covers implementing high-resolution interactive graphics. This module features two technologies, SVG and Canvas API.

Module 12. Animating the User Interface

This module covers how to make a webpage interesting by using CSS animations. Students learn how to implement transitions to modify the appearance of an element over time, and how to transform elements in 2D and 3D.

Module 13. Implementing Real-time Communication by Using WebSockets

This module describes how to implement peer-to-peer communication between a web application and a web server by using WebSockets.

Module 14. Performing Background Processing by Using Web Workers

This module covers how to offload long-running operations to a web worker. This strategy enables a webpage to remain responsive while a separate task performs the processing. The module also describes how to create and control a web worker, and how to pass messages between a web worker and a webpage.

Module 15. Packaging JavaScript for Production Deployment

This module describes how to package JavaScript code for production deployment with tools such as webpack and Babel.

Course Materials

The following materials are included with your kit:

Course Handbook is 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.

You may be accessing either a printed course handbook or digital courseware material via the Skillpipe reader by Arvato. Your Microsoft Certified Trainer will provide specific details, but both printed and digital versions contain the following:

- 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 sections provide on-the-job reference material to boost knowledge and skills retention.
- Lab Answer Keys provide step-by-step lab solution guidance.

To run the labs and demos in this course, the code and instruction files are available in GitHub:

- Instruction files: <https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/tree/master/Instructions>
- Code files: <https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/tree/master/Allfiles>

Make sure to clone the repository to your local machine. Cloning the repository before the course ensures that you have all the required files without depending on the connectivity in the classroom.

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, or to report a problem with course resources, visit the Training Support site at <https://trainingsupport.microsoft.com/en-us>. To inquire about the Microsoft Certification Program, send an e-mail to certify@microsoft.com.

Module 1

Overview of HTML and CSS

Contents:

Module Overview	1-1
Lesson 1: Overview of HTML	1-2
Lesson 2: Overview of CSS	1-13
Lesson 3: Creating a Web Application by Using Visual Studio 2017	1-20
Lab: Exploring the Contoso Conference Application	1-24
Module Review and Takeaways	1-26

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.

- All HTML pages have the same structure
 - DOCTYPE declaration
 - HTML section containing:
 - Header
 - Body
- Each version of HTML has its own DOCTYPE
 - The browser uses the DOCTYPE declaration to determine how to interpret the HTML markup
 - For HTML5 pages, specify a DOCTYPE of **html**

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/htm14/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/htm14/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/htm14/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

- 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

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

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 element, do not need content however
    and consist of a
    single, self-closing element. These are known as empty elements.
  </p>
</body>
```

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 of HTML global attributes on the W3C website, at <https://aka.ms/moc-20480c-m1-pg1>.

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.

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

- In lists

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

- **<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.

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 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.
  </p>
  <h3>Browser Wars</h3>
  <p>The openness of WorldWideWeb meant many different web browsers were created early
  on, including Netscape Navigator and NCSA Mosaic, which later became Microsoft Edge.</p>
</body>
```

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 Internet Hall of Fame and what they did</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>
```

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 2017</li>
  </ol>
</body>
```

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 <https://aka.ms/moc-20480c-m1-pg3> 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,

- Use the **** tag to display an image
 - The **src** attribute specifies the URL of the image source:

```

```

- Use the **<a>** tag to define a link
 - The **href** attribute specifies the target of the link:

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

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
- 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 **<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 **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 **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>
```

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

You will find the steps in the "Demonstration: Creating a Simple Contact Form" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD01_DEMO.md

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.

```
<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.

- 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.

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;
}
```

- 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';
}
```

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.

- 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:

h2.blue	Returns any <h2> elements of class "blue"
h2#blue	Returns any <h2> elements with id "blue"
section, h2	Returns any <h2> and any <section> elements
section h2	Returns any <h2> elements nested within a <section> element at any level.
section > h2	Returns any <h2> elements nested immediately under a <section> 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
<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 <https://aka.ms/moc-20480c-m1-pg4>.

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">
```

- Use an element's style attribute to define styles specific to that element:

```
<p style="color:blue;">
some text </p>
```

- Use the `<style>` element in the `<head>` to include styles specific to a page:

```
<style type="text/css">
  p { color: blue; }
</style>
```

- Use the `<link>` element to reference an external style sheet:

```
<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.

Lesson 3

Creating a Web Application by Using Visual Studio 2017

Visual Studio 2017 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. Microsoft Edge 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 2017 and the support it includes for building web applications. This lesson also examines the F12 Developer Tools for Microsoft Edge.

Lesson Objectives

After completing this lesson, you will be able to:

- Open, inspect and run a web application by using Visual Studio 2017
- Explain how to use the F12 Developer Tools for Microsoft Edge.

Developing Web Applications by Using Visual Studio 2017

Visual Studio 2017 is the current version of Microsoft's primary suite of tools for developers. With Visual Studio 2017, you can perform the following tasks:

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

Visual Studio 2017 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 2017 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 Microsoft Azure.

Visual Studio 2017 provides wizards that enable you to quickly deploy a web application to a web server. If you have the 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 Community 2017 is free to download and install. It targets hobbyists. At the other end, Visual Studio Enterprise 2017 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.

- Visual Studio 2017 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 2017 features include:
 - Full support for HTML5
 - IntelliSense for JavaScript code
 - Support for CSS3 properties and values
 - CSS color picker

All versions of Visual Studio 2017 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 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 Website by Using Visual Studio 2017

Visual Studio 2017 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

You will find the steps in the "Demonstration: Creating a Web Site by Using Visual Studio 2017" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD01_DEMO.md.

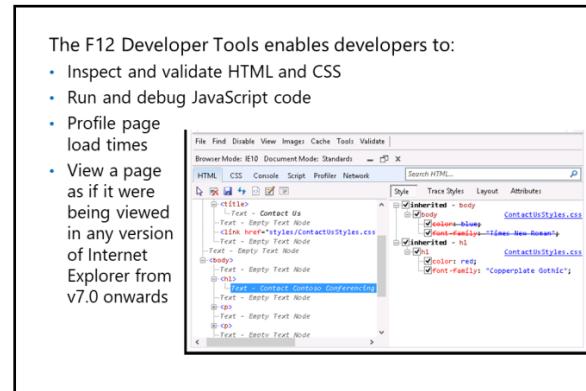
Using the Microsoft Edge F12 Developer Tools

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 Microsoft Edge. These tools enable a developer to quickly perform the following tasks while the application is running in Microsoft Edge:

- 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 7, Internet Explorer 8, Internet Explorer 9, or Internet Explorer 10

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 Microsoft Edge, press **F12** to display the F12 Developer Tools while a web application is running:

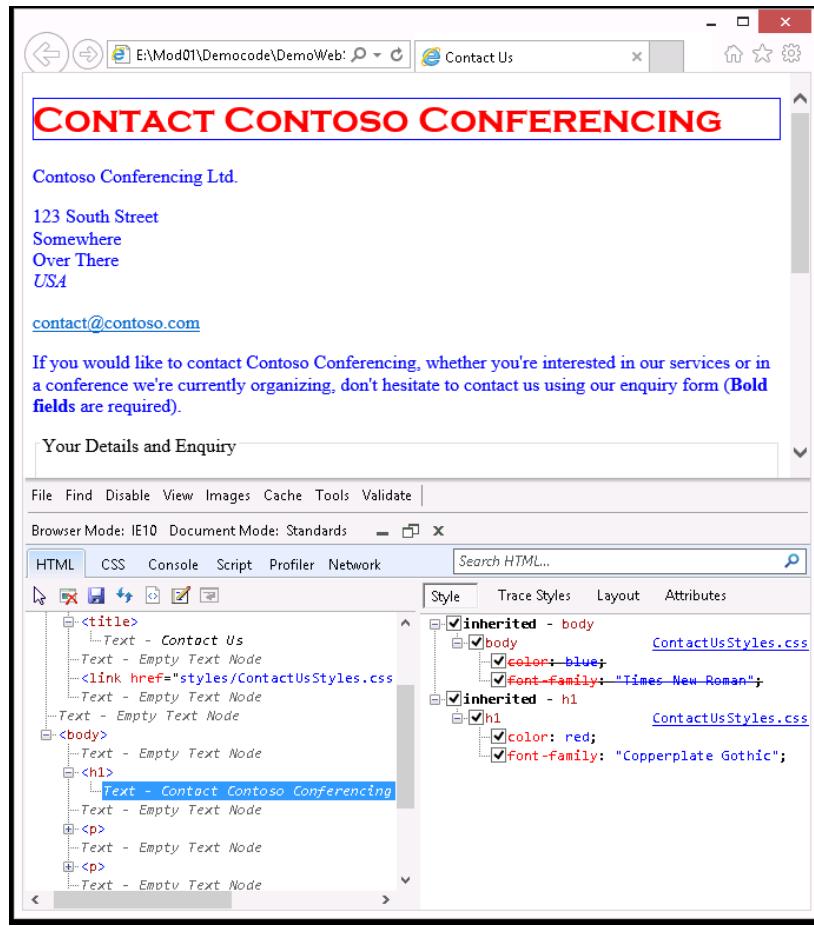


FIGURE 1.1: THE F12 DEVELOPER TOOLS

The menu bar across the top of the **Tools** window gives you quick access to some Microsoft Edge 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 Microsoft Edge'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 Microsoft Edge's support for displaying images along with the display of information about the images in the page.
- The **Cache** menu enables you to change Microsoft Edge'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 Microsoft Edge window changes with each new version, so the **Browser Mode** and **Document Mode** drop-down lists allow you to select which version of Microsoft Edge 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 Microsoft Edge 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.

Demonstration Steps

You will find the steps in the "Demonstration: Exploring the Contoso Conference Application" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD01_DEMO.md.

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 2017 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 2017 to examine the structure of a web application, run a web application, and modify a web application.

Lab Setup

Estimated Time: **30 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD01_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD01_LAK.md.

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.

Exercise 2: Examining and Modifying the Contoso Conference Application

Scenario

In this exercise, you will examine the Visual Studio 2017 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.

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 2017 to create and run a web application, and how to use the F12 Developer Tools in Microsoft Edge to examine a live application.

Review Questions

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

Check 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	2-7
Lab: Creating and Styling HTML5 Pages	2-14
Module Review and Takeaways	2-15

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 10 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 the 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 <https://aka.ms/moc-20480c-m2-pg1>.

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 <https://aka.ms/moc-20480c-m2-pg2>.

You can find a version of the specification for web developers (minus the browser interoperability instructions) at <https://aka.ms/moc-20480c-m2-pg3>.

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>
```

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>**.

HTML5 defines new text elements, including:

- <hgroup>**

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

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

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

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

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

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

- 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 <https://aka.ms/moc-20480c-m2-pg4>.

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

You will find the steps in the "Demonstration: Using HTML5 Features in a Simple Contact Form" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD02_DEMO.md.

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

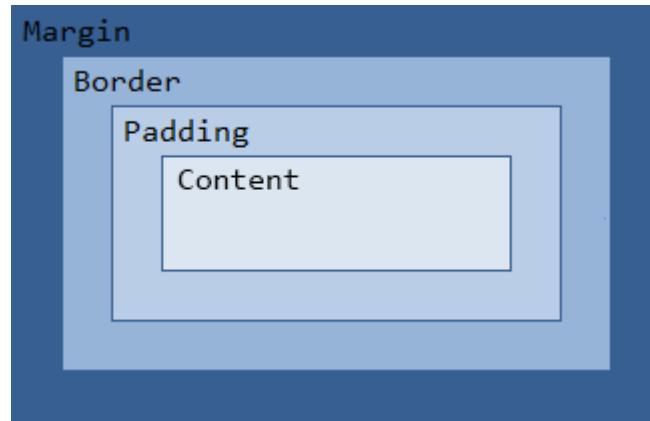


FIGURE 2.1: THE CSS BOX MODEL

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 Microsoft Edge, 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 Microsoft Edge 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

You will find the steps in the “Demonstration: Adding CSS Styles to an HTML Page” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD02_DEMO.md.

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.

Demonstration Steps

You will find the steps in the “Demonstration: Creating and Styling an HTML5 Page” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD02_DEMO.md.

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 pages that allow attendees to register for the conference and provide information about the conference sessions.

Objectives

After completing this lab, you will be able to:

- Create HTML5 pages
- Style HTML5 elements

Lab Setup

Estimated Time: **45 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD02_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD02_LAK.md.

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.

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.

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 Questions

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

Check 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	3-14
Lab: Displaying Data and Handling Events by Using JavaScript.	3-22
Module Review and Takeaways	3-23

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 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 DOM.

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.

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.

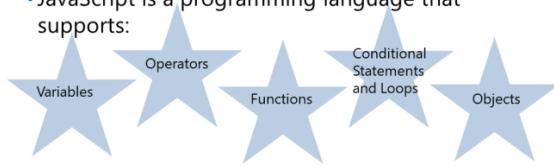
What is JavaScript?

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.

• JavaScript is a programming language that supports:



- Use JavaScript with the Document Object Model and Browser Object Model to make web pages dynamic.
- Use the Fetch API to make asynchronous requests to a web server.

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:

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

- A JavaScript statement represents a line of code to be run
- Terminate statements with a semicolon

```
let 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 */
```

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");
```

 **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 four 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 **let** or **var** keywords. Until a variable is given a value, JavaScript will return its value as undefined.

```
let mystery;
```

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

```
let code = "Spang";
```

4. Declare a read-only reference to a value by using the **const** keyword:

```
const constant = "foo";
```

- Use **let** to declare variables

```
let answer = 3;
let actuallyAsString = "42";
```

- JavaScript has three simple types

- String, Number, and Boolean
- Variables can also be undefined or null

```
let noValue; // noValue has the value undefined
```

```
let nullValue = null; // null is different to undefined
```

- JavaScript supports many operators

- Arithmetic, assignment, comparison, Boolean, conditional, and string.

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**.

 **Note:** The **let** and **const** keywords are part of ECMA-262 6th edition also referred to as ECMAScript 2015, which means they are supported only by browsers with JavaScript engines that implements this version of the standard.

The difference between variables defined by using the **let** and **const** keywords and variables defined by using the **var** keyword will be discussed in further detail in module 7.

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 **let**, **const** or **var** keywords, 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 single or 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.

```
let simple = "Green Eggs and Ham";
let escaped = "\"Green Eggs \& Ham \\"";
let 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.

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

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

```
let 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:

```
let variableWithValueNullValue = 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:

```
let 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:

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

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

2. **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)

3. **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)

4. **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.

5. 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.

6. 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.

```
let zero = 0;
let emptyString = "";
let 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:

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

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

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  
const 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:

JavaScript provides two conditional constructs

- if: if (TotalAmountPaid > AdvancePaid){

 GenerateNewInvoice();

 } else {

 WishGuestAPleasantJourney();

 }

- switch: let RoomRate;

 switch (typeOfRoom) {

 case "Suite":

 RoomRate = 500;

 break;

 case "King":

 RoomRate = 400;

 break;

 default:

 RoomRate = 300;

 }

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

 **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.

```
let 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 (let 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 (let 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.

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

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

- String, Date, Array, RegExp

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

```
let 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.

```
let today = new Date(1346454000); // Number of milliseconds since 01/01/1970
let today = new Date("September 1, 2012");
let 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.

```
let emptyThreeItemArray = new Array(3);
let seasonsArray = new Array("Spring", "Summer", "Autumn", "Winter");
let 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:

```
let 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):

```
let 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.

```
let 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()**; **let 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 <https://aka.ms/moc-20480c-m3-pg1>.

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:

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

- JSON is a format for serializing objects:

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

- JavaScript provides APIs for serializing and parsing JSON data

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

```
let 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.

```
let 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://aka.ms/moc-20480c-m3-pg2>.

Demonstration: Creating a Simple JavaScript File that Defines Variables, Array and Functions.

In this demonstration you will see how to create a JavaScript file that includes reusable functions that can filter an array of persons over a certain age and print the array to the browser's console.

Demonstration Steps

You will find the steps in the "Demonstration: Creating a simple JavaScript file that defines Variables, array and functions" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD03_DEMO.md.

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.

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
```

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")`

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:

- **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:

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

Querying the DOM for Elements

There are situations where we need to find elements based on multiple attribute values.

For instance, we would like to find a **div** element whose ID attribute value is **highlight**. Although this can be done by using common methods such **getElementById** and **getElementsByName**, the **document** object provides more powerful and convenient methods that enable us to perform complex queries on the DOM by using css selectors.

- **document.querySelector(cssSelectorString)**, which returns a single element whose attribute values match the css selector specified by the **cssSelectorString**.
- **document.querySelectorAll(cssSelectoreString)**, which returns a collection of elements whose attribute values match the css selector specified by the **cssSelectorString**.

The following example shows how to obtain **reference** for an input element whose class attribute is **valid**.

```
let validInput = document.querySelector("input.valid");
let validInput = validInput.value;
```

Similarly, we can also query for all elements that match a css selector.

The following example shows how to obtain **reference** for all input elements that have the **checked** attribute.

```
let checkedInputs = document.querySelectorAll("input[checked]");
let checkedInputsCount = checkedInputs.length;
```

The **querySelector** and **querySelesctorAll** methods also exist in the **Element** object which is the base class from which all objects in **Document** are inherited.

This enables us to only query against the contents of a specific **Element** object.

Given the following form:

```
<form name="contactForm">
<input type="text" class="valid" name="nameBox" />
<input type="email" name="emailBox" />
<input type="number" class="valid" name="numberBox" />
</form>
```

- You can reference the email field by using

```
let emailInput = document.querySelector("input[type=email]");
```

- You can reference all the input elements with "valid" class attribute

```
let validInputs = document.querySelectorAll("input.valid");
```

The following example shows how to obtain **reference** for all the **span** elements where the **class** attribute has the value **text-bold**, and are contained inside **div** where the **ID** attribute has the value **main**.

```
let mainDV = document.querySelector("div#main");
let boldSpans = mainDV.querySelectorAll("span.text-bold");
let boldSpansCount = boldSpans.length;
```

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
const list = document.getElementById("VenueList");

// Create a new venue
const 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.

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

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

```
const 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

- 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:

```
let 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);
```

example, you can handle the **mouseover** event for an **** element by using the **onmouseover** callback, as follows:

```

```

 **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, useCapture)**, which adds the listener function to the element for the given **eventName**. You can pass the **listenerFunction** by name or as an anonymous function.

```
let 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, useCapture)**, 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.

Previous browsers implemented two opposing event models, capturing and bubbling. These models define the order in which event handlers are triggered inside a hierarchy of DOM nodes. In the capturing event model, events handlers are triggered from the root element downwards until they reach the target element. In the bubbling event model, event handlers are triggered from the target element upwards

until they reach the root element. For backward compatibility reasons, modern browsers implement both models and the optional third argument, **useCapture**, allows developers to choose if they want to trigger the event handler in the capturing phase by setting it to **true** or in the bubbling phase by setting it to **false**. The default value of this argument is **false**.

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

```
<!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() {
                let userName = document.getElementById("NameBox").value;

                let thankYou = document.createElement("p");
                thankYou.textContent = "Thank you " + userName;
                document.getElementById("thankYouArea").appendChild(thankYou);
            }

            document.getElementById("submit").addEventListener("click", sayThankYou);
        </script>
    </body>
</html>
```

The DOM is constructed by the browser while parsing the HTML markup from top to down. Some elements may block the construction of the DOM; the **script** element is one of these elements.

When the browser hits a **script** element, the construction of the DOM is blocked until the script file is loaded and executed. Consider the following example:

```
<script type="text/javascript">
    const main = document.getElementById("mainContent");
    console.log(main) //will output null
</script>
<div id="mainContent" >Some content</div>
```

In the code example above, **document.getElementById** will return **null** because the element where the **id** attribute has the value **mainContent** does not exist in the DOM when the script is run.

The above situation can be avoided by listening to the document's **DOMContentLoaded** event.

The **DOMContentLoaded** event is fired by the browser as soon as the DOM hierarchy has been fully constructed.

```
<script type="text/javascript">
function init(){
    const main = document.getElementById("mainContent");
    console.log(main) //will output the div element reference object.
}
document.addEventListener("DOMContentLoaded",init);
</script>
<div id="mainContent" >Some content</div>
```

Demonstration: Manipulating the DOM with JavaScript

Demonstration Steps

You will find the steps in the “Demonstration: Manipulating the DOM with JavaScript” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD03_DEMO.md.

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 add a Schedule page that lists the tracks and sessions for the conference to the ContosoConf website.

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.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD03_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD03_LAK.md.

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.

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.

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.

Review Questions

Question: 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?

Check Your Knowledge

Question
Which DOM event indicates that the contents for a page have been loaded into the browser?
Select the correct answer.
loaded
available
DOMReady
DOMContentLoaded
ready

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	4-16

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.

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>
```

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>
```

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.

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 **datalist** 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:

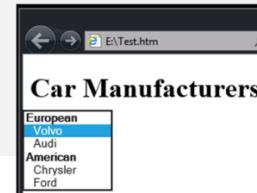
```
<textarea id="carDescription" name="carDescription" cols="80" rows="5"
placeholder="Enter a short description of your car" maxlength="399"/>
```

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">
```

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>
```



```
<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>
```

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

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.

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:

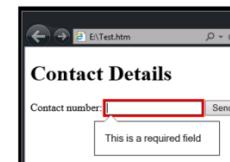
```
<input id="contactNo" name="contactNo"
      type="tel" placeholder="Enter your
      mobile number" required="required" />
```

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; Microsoft Edge 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" />
```

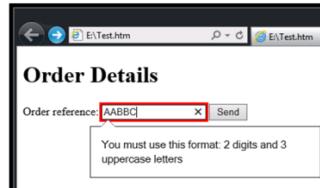


Validating Text Input

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.

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" />
```



```
<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.

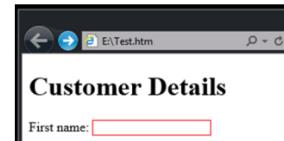
Styling Fields to Provide Feedback

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:

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;
}
```



```
<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>
```

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 Microsoft Edge 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>
```

- Catch the **submit** event to validate an entire form
 - Return true if the data is valid, false otherwise
 - The form is only submitted if the **submit** event handler returns true
- Catch the **input** event to validate individual fields on a character-by-character basis
 - If the data is not valid, display an error message by using the **setCustomValidity** function
 - If the data is valid, reset the error message to an empty string

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
    let ageValid = ...;
    if (!ageValid) {
        ageInput.setCustomValidity("Age is invalid. Please specify a value between 18 and 120");
    } else {
        ageInput.setCustomValidity("");
    }
}
...
const 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**:

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 isInteger( text ){
    const intTestRegex = /\^$|(\+|-)?\d+\s*\$/;
    return String(text).search(intTestRegex) != -1;
}

function validateForm()
{
    if( !isInteger(document.getElementById('score').value))
        return false; /* No, it's not a number! Form validation fails */

    return true;
}
```

```
<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>
```

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 ){
    const 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:

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 ){
    const 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;
}
```

```
<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>
```

The following JavaScript code performs the validation:

```
function isSignificant( text ){
    const 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() {
    const 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() {
    const textbox = document.getElementById("penalties");
    if( ! isSignificant(textBox.value) ) {
        textbox.className = "validationError";
        return false; /* No! Form validation fails */
    }
    textbox.className = "validatedFine";
    return true;
}
```

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.

Demonstration Steps

You will find the steps in the “Demonstration: Creating a Form and Validating User Input” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD04_DEMO.md.

Lab: Creating a Form and Validating User Input

Scenario

The delegates who want to attend ContosoConf will need 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 minimal validation and 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.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD04_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD04_LAK.md.

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. You will then 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 the submission of incomplete or invalid data. Finally, you will run the application, view the Register page, and then verify that form validation happens correctly.

Exercise 2: Validating User Input by Using JavaScript

Scenario

The conference registration form requires that the values in the Choose a Password and Confirm your Password boxes 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 Microsoft Edge already highlight them with a red border, but other browsers might not implement this feature by default). You will run the application, open the Register page, and then verify that the application highlights the invalid elements.

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 Questions

Question: 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?

Question: 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: Async Programming in JavaScript	5-2
Lesson 2: Sending and Receiving Data by Using the XMLHttpRequest Object	5-8
Lesson 3: Sending and Receiving Data by Using the Fetch API	5-14
Lab: Communicating with a Remote Data Source	5-18
Module Review and Takeaways	5-20

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 **Fetch API**, which simplifies many of the tasks involved in sending requests and receiving data. Because the **Fetch API** and **XMLHttpRequest** objects are asynchronous API, you will first learn about how to handle asynchronous tasks with the **Promise** object, **arrow functions** and the new **async/await** syntax that lets you handle asynchronous request as if they were synchronous.

Objectives

After completing this module, you will be able to:

- Handle asynchronous JavaScript tasks by using the new **async** programming technologies.
- 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 **Fetch API** object.

Lesson 1

Async Programming in JavaScript

Almost every web application needs to communicate with a server to fetch, update, create or delete data on or from the server. Some of them make dozens of requests to the server during the application lifecycle. Because web requests are asynchronous by default, it is challenging to do it in a way that improves user experience and application performance. ECMAScript 6 comes with new features to handle this challenge and help us do it right.

In this lesson, you will learn how to use arrow functions to shorten the callback functions that are used for handling asynchronous calls. Then you will learn about **Promise**, a new feature that flattens the code for complex asynchronous call flows and makes it easier to read and maintain them. You will learn how to create a new **Promise** object and use it to perform asynchronous calls to the server to get data, and how to handle the data received.

Finally, you will learn how to use **async/await**, a new JavaScript syntax that gives an asynchronous code a synchronous-like behavior.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain arrow functions and how to use them to shorten JavaScript code.
- Describe **Promise** and how it helps to deal with asynchronous tasks.
- Create a new **Promise** object and use its methods to resolve or reject it.
- Use the **async/await** syntax to get synchronous-like behavior for asynchronous code.

Arrow Functions

Arrow functions are a new ECMAScript 6 syntax to write JavaScript functions. They are called arrow functions because they use the `=>` sign. Arrow functions are anonymous, one-line mini functions, and are more concise than old-style function expressions. With arrow functions, we can decrease the use of the **function** and **return** keywords and curly braces, in the declaration of our function.

A basic construct of an arrow function looks like this:

`(parameters) => { statements }`

- Arrow functions are a concise way to write JavaScript functions.
- They help to shorten code length and save typing long code.
- They are useful for small anonymous functions.
- They use the **this** value of their enclosing execution context.

Let's see the difference between an old-style function expression and an arrow function expression:

The following code example shows an old-style function expression.

An old-style function expression

```
const combineText = function(text){  
    return "I love to " + text;
```

```
};

combineText("eat"); // will return: "I love to eat".
```

Now, this is what it looks like when you use an arrow function:

The following code example shows an expression that uses an arrow function.

An arrow function style of a function expression.

```
let combineText = (text) => {
    return "I love to " + text;
};

combineText("eat");
```

As you can see we omitted the **function** keyword and added the **=>** token.

Because we had just one parameter, we can omit the parenthesis around the parameter:

The following code example shows how you can omit parenthesis from a one-parameter function.

Omit parenthesis from one-parameter function:

```
let combineText = text => {
    return "I love to " + text;
};

combineText("eat");
```

Since we have in the function body only one **return** statement, we can also omit the **return** keyword and the curly braces around the statement. In this case, the **return** keyword is added implicitly:

The following code example shows how you can omit the **return** keyword and the curly braces.

Omit the return keyword and the curly brackets:

```
let combineText = text => "I love to " + text;
combineText("eat");
```

When we have no parameters for the function, empty parentheses are required:

The following code example shows how you can write an arrow function without parameters.

Parameterless arrow function:

```
let logToConsole = () => console.log("logging..");
logToConsole();
```

Arrow functions are useful to shorten code length when we want to manipulate arrays or to chain promises. We will demonstrate chaining of promises later in this lesson.

Arrow functions also change the context of the **this** keyword. They have no **this** of their own. Instead, they get the **this** value from their enclosing execution context. We will discuss more about the **this** keyword in lesson 7.

What is a Promise?

The **Promise** object helps you deal with asynchronous tasks in your JavaScript code. It helps you to check the success or failure of an asynchronous operation, to return the needed value in each case, and to react accordingly.

The **Promise** object represents the future result of an asynchronous operation. This result's value doesn't exist yet, but it will be resolved in the future. Until then, the **Promise** object is used as a proxy for that value. This value might be typically one of two: a resolved value or a rejected value if some error occurred along with the reason why it was rejected. The type of both kinds of values can be any legal JavaScript value: a string, an object, an integer or even nothing.

- Promises help to handle asynchronous tasks.
- A **Promise** object is a proxy for a future value.
- It can be in one of four states:
 - Fulfilled
 - Rejected
 - Pending
 - Settled

The **Promise** object can be in one of the following states:

- Resolved. The task related to the **Promise** succeeded.
- Rejected. The task related to the **Promise** failed.
- Pending. The task related to the **Promise** is not yet fulfilled or rejected.
- Settled. The task related to the **Promise** is fulfilled or rejected.

A **Promise** object can be used for any asynchronous task. For example, to send a request to a remote server by using **XMLHttpRequest**, or to call the **setTimeout** and **setInterval** JavaScript functions.

The following example shows how to wrap a function call with a **Promise**:

Wrapping setTimeout

```
let getPromise = timeToWait => new Promise(resolve => setTimeout(resolve, timeToWait, "bar"));
getPromise(5000).then(data => console.log(data));
```

In the first section of the code, we created a function that returned a **Promise** object. Then, in the second section, we ran the function, got the promise, and when it was resolved, we wrote the resolved value to the console. In this case, the string 'bar'. We will explain more about the creation of the **Promise** object and its properties and methods in the next topic.

The biggest advantage of promises is chaining. We will talk about chaining in the next topic.

 **Note:** Promises are new in ECMAScript 6, and supported by all modern browsers, except Internet Explorer.

The Promise Object

A **Promise** object can be created from scratch by using its constructor. Basically, the **Promise** constructor takes a function as a parameter that lets us resolve or reject the **Promise** object manually.

The following example shows how to create a **Promise**:

- A **promise** object is initiated by its constructor and supplied by an executor.
- A **Promise** object's **resolve()** and **reject()** methods indicate success or failure of the promise.
- A **Promise** object's **then()** method is used for reactions when a promise is fulfilled or rejected, and for chaining promises.
- A **Promise** object's **all()** method is used to resolve multiple promises at the same time, regardless to their order.

Initiate new Promise using its constructor:

```
let newPromise = new Promise((resolve, reject) => {
});
```

We initiate a new promise by using the **new** keyword, and pass it a callback function, which we call the executor. The executor gets two functions as parameters, **resolve()** and **reject()**. We write our asynchronous code inside the executor. When our asynchronous task succeeded, we call **resolve()** to indicate that the promise is fulfilled, and typically we return the results of the task as a value. When an error occurs, we call **reject()** to indicate that the task failed, typically with an **Error** object that describes the reason for the failure. Let's see how this reflected, by using a **Promise** object together with **XMLHttpRequest**:

 **Note:** We will talk about **XMLHttpRequest** and its features later in this module. For now, all we have to know is that **XMLHttpRequest** is a JavaScript object used to communicate with a server to transmit and receive data for a webpage.

This example shows how to wrap **XMLHttpRequest** functionality with a **Promise** object:

Using promises with XMLHttpRequest

```
let getRemoteData = url => new Promise((resolve, reject) => {
    let request = new XMLHttpRequest();

    request.onload = () => {
        if (request.status === 200){
            resolve(request.response);
        }
        else {
            reject(Error(request.statusText));
        }
    };

    // Handle network errors
    request.onerror = () => {
        reject(Error("Network error occurred"));
    };

    request.open("GET", url);
    request.send();
});
```

This example shows how to handle the promise returned from the function.

Handling promises

```
getRemoteData("http://contoso.com/resources/...")  
.then(data => console.log("success!", data))  
.catch(error => console.log("failed!", error));
```

The `then()` method

To react to the fulfillment or rejection of a promise, we use the `then()` method of the **Promise** object. This method gets two callback functions as parameters: a success callback and a failure callback. When the promise is settled, one of the two functions fires based on the state of the **Promise** object. The success callback function when the **Promise** fulfilled, the failure callback function when it is rejected. The `then()` function returns a new promise, which is different from the original, and this gives us the ability to chain promises together.

Chaining promises

Sometimes we want to do some asynchronous tasks where the second task depends on the result of the first task, and so on. With promises, we can accomplish this by attaching our success and failure callbacks to the new **Promise** object returned by the `then()` method. This creates a promise chain:

The following code example shows how to chain promises.

Chaining promises

```
getRemoteData("http://contoso.com/resources/...")  
.then(result => getMoreRemoteData("http://fake.com/resources/...", result))  
.then(newResult => getOtherKindOfRemoteData("http://dummy.com/resources/...", newResult))  
.then(finalResult => console.log('Final result arrived', finalResult));
```

With this kind of chaining, the promises are ordered in a sequence because each one depends on the success of the previous one. In case there is no dependency between the promises and we want all of them to proceed regardless to their order, we can resolve them all at the same time by using the `all()` method of the **Promise** object:

This code example shows how to resolve many promises at the same time:

Using `Promise.all()`

```
Promise.all([  
    getForecast(/* here will come the first url */),  
    getFootballScores(/* here will come the second url */),  
    getShirtsPrices(/* here will come the third url */)  
]).then(results => console.log('All results arrived!', results));
```

Async/await

The **Async/await** statements were built on top of promises. They let us write our asynchronous code for the promises in the same way as we write synchronous code, but without blocking the normal execution of main thread code when waiting for a promise to be resolved. This includes:

- Using **try/catch** blocks.
- The ability to debug the code by putting a breakpoint at the beginning of the asynchronous code statements and move to the next statements as if they were synchronous calls.

- **Async/await** statements were built on top of promises. They allow us to write asynchronous code as if it were synchronous.
- Using **async/await** syntax make our code cleaner, and lets us debug and handle errors the same way we do in synchronous code.
- The **Await** statement is allowed only within an **async** function. An **async** function is defined by using the **async** keyword.
- **Await** statements are run in serial, one at a time. For running a few promises concurrently, you should use the **Promise.all** method.

Let's see how we can improve the chained promises code from the previous topic to work with **Async/await**:

Async/await example

```
let myAsyncFunc = async function(url1, url2, url3){
  try{
    let result = await getRemoteData(url1);
    let newResult = await getMoreRemoteData(url2, result);
    let finalResult = await getOtherKindOfRmoteData(url3, newResult);
    console.log("Final result arrived", finalResult);
  }
  catch(error)
  {
    //handle errors here
  }
}

let url1 = "http://contoso.com/resources/...";
let url2 = "http://fake.com/resources/...";
let url3 = "http://dummy.com/resources/...";
myAsyncFunc(url1, url2, url3);
```

As you can see, the code is now nicer and cleaner than the previous one and looks almost like regular synchronous code. However, you should notice to the following:

- We use the **await** keyword to wait for a promise to resolve. Use of this keyword is only allowed within an **async** function. So, we must declare our **myAsyncFunc** function with the **async** keyword.
- If one of awaited function's result is not a promise, it will automatically be casted into a promise. No error will be thrown.
- You can mix promises and **async** functions. The **Async** function returns a promise. Its value will be the return value of the **async** function if it is resolved, and an error reason if it is rejected because of an unhandled exception. This allow us to use the **then()** function as with a regular promise.

Lesson 2

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.

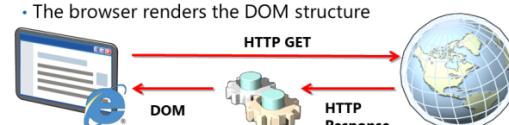
How a Browser Retrieves Web Pages

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.

- A web browser issues HTTP **GET** requests to fetch a web page to display

- The response is parsed into a DOM structure
- The browser renders the DOM structure



- Elements with a **src** attribute can initiate further HTTP **GET** requests

- JavaScript code can trigger HTTP **GET** requests

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:

```
const 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:

```
const 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.

- 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

```
function tryMyLuck() {
    const 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);
    }
    ...
}
```

 **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 <https://aka.ms/moc-20480c-m5-pg1>

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:

- 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;
}

```

```

function getResponse(request) {
    const type = request.getResponseHeader("Content-Type");
    switch (type) {
        case "text/xml":
            return request.responseXML;
        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) {
  const 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:

- 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);
    ...
  }
};
```

- 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.

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) {
        const 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 3

Sending and Receiving Data by Using the Fetch API

Fetch API is a new JavaScript interface to make network requests from web applications. It is a simpler and cleaner API to make common actions such as getting or sending different types of data from and to a server, handling errors that occur during transmission, handle cross origin access for resources, and customize requests and responses according to the developer's needs. Fetch API uses promises as a return value , to avoid using callback functions or registering to request events to handle the server response, like we use to do with **XmIHttpRequest**.

In this lesson, you will learn how use the **fetch()** method to get and send data from and to a server. Then you will learn how **Fetch API** uses promises to get a **Response** object from a server, and how to extract the data and parse it in to its relevant format. You will also learn how to handle errors that occur during this process. Finally, you will learn how to combine **Fetch API** with the **async/await** technique, to improve the fetch code and get a synchronous-like look and feel for the code.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use **Fetch API** to transmit and receive data from a server.
- Combine **Fetch API**, arrow functions and **async/await** to get a clean and readable code.
- Describe how **Fetch API** uses promises to handle server response.

Using the Fetch API to Send Asynchronous Requests

To send a request to a server, we use the **fetch()** method. This method exists on the global **window** scope of the browser, and it takes two arguments: the first one is the **url**, which is required. The second one is an **options** object that let you configure the request's method, body, headers etc, which is optional.

 **Note:** Most browsers (except Internet Explorer) support the basic usage of **Fetch API**, but not all of its features. Before using it, please check the browser support.

Use **fetch()** method to get a resource from server:

- For a **GET** request, supply a URL as the first argument.
- For a **POST** request, supply the URL and an **options** object with the initial settings. These may include values for: **headers**, **body** and **credentials**.

```
fetch(url, {
  body: JSON.stringify(data),
  credentials: "include",
  headers: {
    "Content-Type": "application/json"
  },
  method: "POST"
});
```

Here is an example how to make a simple request using Fetch API:

A GET request by using the **fetch()** method

```
fetch("http://contoso.com/resources/...");
```

If you don't configure the request method in the **options** object, the **fetch()** method uses **GET** as its default request method.

We can also send data with **fetch()**. We just need to use the optional **options** object, and set at least three of its options:

- The request's **method**. The optional values are **POST**, **PUT** or **DEL**.
- The request's **headers**. The most important header is the **Content-Type**.
- The request's **body**. The actual content to be sent to the server. The **body** content must be in accordance to the **Content-Type** value.

You can pass the **options** object to **fetch()** as an external JavaScript object, or by using literal object notation, as demonstrated in the example below:

Sending data to server with **Fetch API**:

Using options object to configure a request.

```
fetch("http://contoso.com/resources/...", {  
    method: "POST",  
    headers: {  
        "Content-Type": "application/json"  
    },  
    body: JSON.stringify({name: "James"})  
});
```

If you want to send form data, which is typically data that a user inputs when he fills a form on the page, we can use the **FormData** object.

Using **FormData** to transmit user data to server:

```
fetch("http://contoso.com/resources/...", {  
    method: "POST",  
    body: new FormData(document.getElementById("UserDetailsForm"))  
});
```

By default, **fetch()** does not send cookies together with the request. If you have to send cookies, you should set it explicitly in the **credentials** property of the **options** object.

The following example shows how to include cookies in the request with **Fetch API**:

Using the options object to include cookies in the request.

```
fetch("http://contoso.com/resources/...", {  
    credentials: "include"  
});
```

Handling Promises

As mentioned earlier, **Fetch API** was built to use promises. When calling the **fetch()** method, a **Promise** object is returned. When it resolves, it will contain a **Response** object, which is the HTTP response to the request. We can access this **Response** object and fetch the data returned by the server, by using the known **then()** method of the **Promise** object:

The following example shows how to handle the returned promise from the **fetch()** method.

- When calling **fetch()**, it returns a **Promise** object.
- This object can be used within a **then()** call, to get access to the **Response** object returned by the server.
- A **Promise** object resolved to server data is available by calling **response.json()**.

Handling the **fetch()** promise.

```
fetch("http://contoso.com/resources/...")  
  .then(response => {/* get server response and do something... */});
```

The response keeps the data in the **body** property. The data is wrapped by an object of type: **ReadableStream**. To fetch the data from this stream, you use one of the following methods:

- **json()**. For data formatted as JSON.
- **blob()**. For data of type: image, etc.
- **text()**. For data formatted as string, such as XML or HTML data.

Each one of them returns a **Promise** object that resolves to data with the appropriate format.

Here is an example how to extract JSON data from the response:

Using **response.json()** to get response data:

```
fetch("http://contoso.com/resources/...")  
  .then(response => response.json())  
  .then(data => console.log(data));
```

Handling errors with Fetch API

Errors in **Fetch API** are handled as they are with promises. We use the **catch()** method of the **Promise** object:

The following code example shows how to handle errors within a Fetch API response.

Fetch API Error Handling

```
fetch("http://contoso.com/resources/...")  
  .then(response => response.json())  
  .then(data => console.log("received data: ", data))  
  .catch(error => console.log("error occurred: ", error));
```

Using async functions

The **Fetch API** and **async** functions can work together perfectly. With them, we can make the code cleaner. Let's see an example:

The following example shows how to combine **Fetch API** with **async/await**:

Fetch API and **async/await** techniques can combine together to make cleaner code:

```
let getResponse = async () => {
    let response = await fetch("http://contoso.com/resources/");
    let json = await response.json();
    console.log(json);
};
```

Combine Fetch API and async/await.

```
async function getResponse(){
    let response = await fetch("http://contoso.com/resources/");
    let json = await response.json();
    console.log(json);
};

getResponse();
```

Both **fetch()** and **response.json()** return a **Promise** object, so we can use the **await** code execution until the **Promise** object is resolved, to get its value.

Finally, let's improve the previous code by combining Fetch API, **async/await** and arrow functions together:

The following example shows how to combine **Fetch API** with **async/await** and an arrow function:

Combine Fetch API, async/await and arrow functions

```
let getResponse = async () => {
    let response = await fetch("http://contoso.com/resources/");
    let json = await response.json();
    console.log(json);
};
```

Demonstration: Communicating with a Remote Data Source

Demonstration Steps

You will find the steps in the "Demonstration: Communicating with a Remote Data Source" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD05_DEMO.md.

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 stored as a hard-coded array 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 asked whether it was 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.

For popular sessions, 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:

- Write JavaScript code to retrieve data from a remote data source.
- Write JavaScript code to send data to a remote data source.
- Use the `async fetch` method to simplify code that performs remote communications.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD05_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD05_LAK.md.

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 must be parsed 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 that it also handles errors correctly.

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. You will also 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.

Exercise 3: Refactoring the Code by Using the `async fetch` Method

Scenario

The existing code that uses 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 `fetch()` function.

First, you will refactor the **downloadSchedule** function by replacing the use of an **XMLHttpRequest** object with a call to the **fetch** method. Then you will refactor the **saveStar()** function in a similar manner. Using the **fetch()** 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.

Module Review and Takeaways

In this module, you have learned how to write better asynchronous code using **promises**, **arrow functions** and **async/await**.

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 **Fetch API** to simplify many of the operations associated with sending and receiving data, and in particular you have learned how to use the **fetch()** function to perform these tasks in a concise manner.

Review Question

Check Your Knowledge

Question
In the <code>onreadystatechange</code> 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="radio"/> The readyState property should be set to 0.
<input type="radio"/> The responseText property should be set to a non-null value.
<input type="radio"/> The readyState property should be set to 4.
<input type="radio"/> The status property should be set to 200 (HTTP OK).
<input type="radio"/> The HTTPResponse property should be set to 0.

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	6-11
Lesson 4: Enhancing Graphical Effects by Using CSS3	6-15
Lab: Styling Text and Block Elements by Using CSS3	6-22
Module Review and Takeaways	6-24

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, 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: 300px;
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 <https://aka.ms/moc-20480c-m6-pg1>.

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:

- | | |
|---|--|
| <ul style="list-style-type: none"> • Outlines | outline: 2px solid green;
outline-offset: 5rem; |
| <ul style="list-style-type: none"> • Presentation | border-radius: 50% / 30%;
overflow: hidden;
resize: horizontal; |
| <ul style="list-style-type: none"> • 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.

 **Reference Links:** You can find the latest draft of the CSS3 Multi-Column Layout Module at <https://aka.ms/moc-20480c-m6-pg2>.

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	display: block;
• Inline	display: inline; display: inline-block;
• Table	display: table;
• Positioned	position: relative; position: absolute; position: fixed;
• Flexbox	display: flexbox;

- **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 Cascading Style Sheets (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

You will find the steps in the "Demonstration: Switching Between Cascading Style Sheets (CSS) Layout Models" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD06_DEMO.md.

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.

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:

• Link classes

a:link
a:visited
a:focus
a:hover
a:active

• Form classes

input:enabled
input:disabled
input:checked

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:

- | | |
|---|---|
| <ul style="list-style-type: none"> • Position • Position and type • Document structure | <p>p:first-child
p:nth-child(2)</p> <p>p:last-of-type
p:nth-last-of-type(4)</p> <p>:empty
:root
:not(p, h1)
:target</p> |
|---|---|

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.

CSS3 defines several different sets of color values:

- **Keywords**
color: red;
color: transparent;
color: currentColor;
- **RGB \ RGBA model values**
color: #ff0000;
color: rgb(255,0,0);
color: rgba(100%,0,0,0.5);
- **HSL \ HSLA model values**
color: hsl(240, 100%, 50%);
color: hsl(120, 100%, 50%, 0.5);

```
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: #ff0;           /* #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).

```
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 <https://aka.ms/moc-20480c-m6-pg3>.

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:

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);
```



```
article {
    background-image: url('bluearrow.png'), url('greenarrow.png');
    background-repeat: repeat-x, repeat-y;
}
```

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:

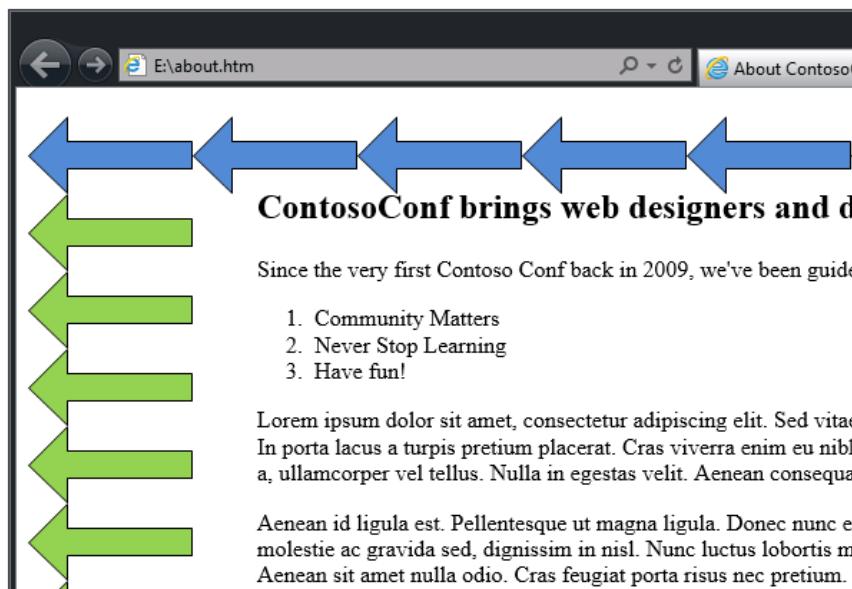


FIGURE 6.1: THE ABOUT PAGE STYLED WITH A REPEATED MULTI-IMAGE BACKGROUND

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.

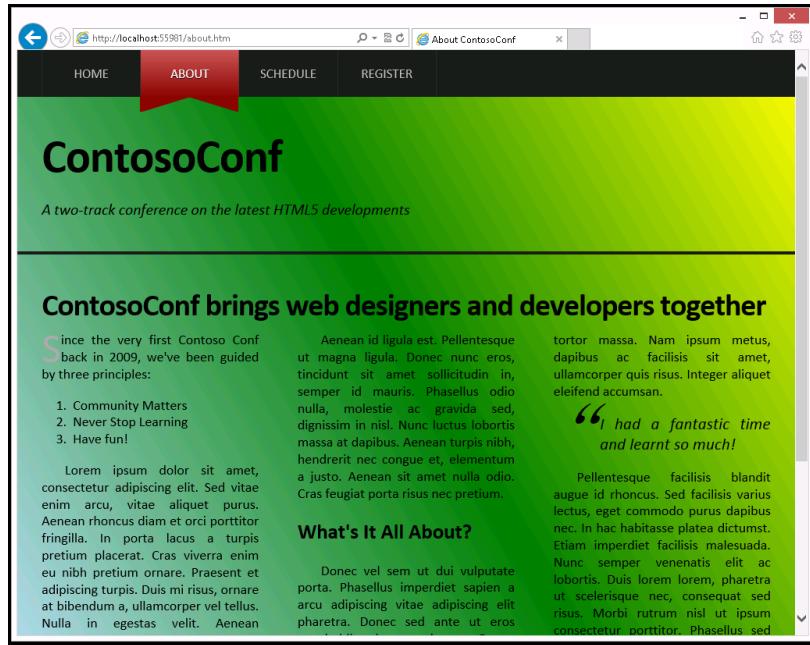


FIGURE 6.2: THE ABOUT PAGE STYLED WITH A LINEAR-GRADIENT BACKGROUND

- 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 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:

- **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 support transforms, but only with vendor-prefixes attached.

```
transform: rotate(30deg);
-ms-transform: rotate(30deg); // IE10
```

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%;
}
```

The screenshot shows a web page with the title "Confbring". The page contains several lines of text in Latin, some of which are rotated or skewed. There are also some circular shapes. The browser window has a standard toolbar at the top.



Additional Reading: For an in-depth discussion of 3D transforms in IE10, see <https://aka.ms/moc-20480c-m6-pg4>.

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:

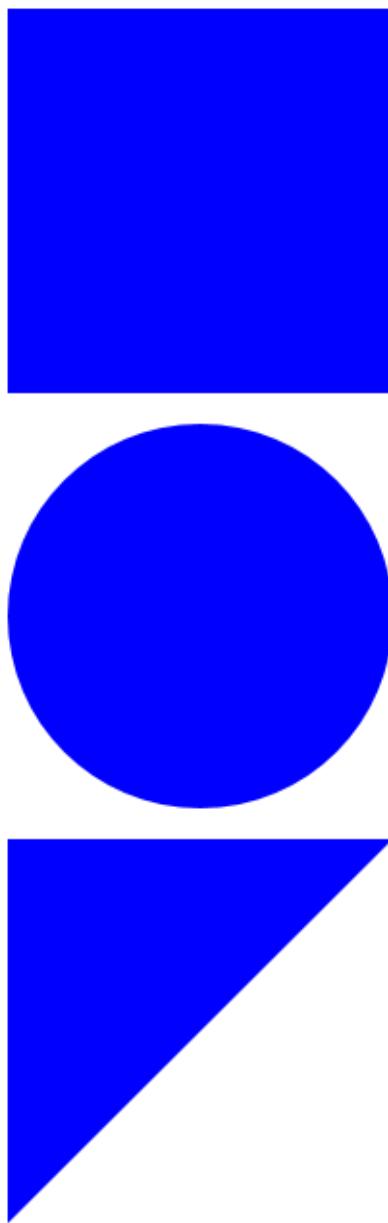


FIGURE 6.3: SHAPES DRAWN BY USING CSS STYLES

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.

Demonstration Steps

You will find the steps in the “Demonstration: Styling Text and Block Elements by Using CSS3” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD06_DEMO.md.

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:

- Implement advanced styling for text elements by using CSS
- Style block elements by using CSS.
- Create graphical elements by using CSS.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD06_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD06_LAK.md.

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:

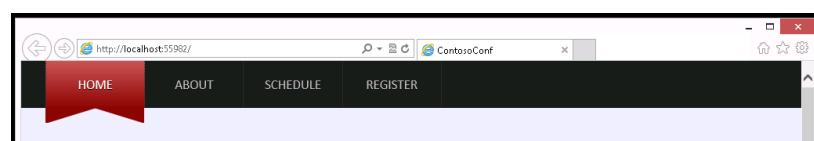


FIGURE 6.4: THE NAVIGATION BAR

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.

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:

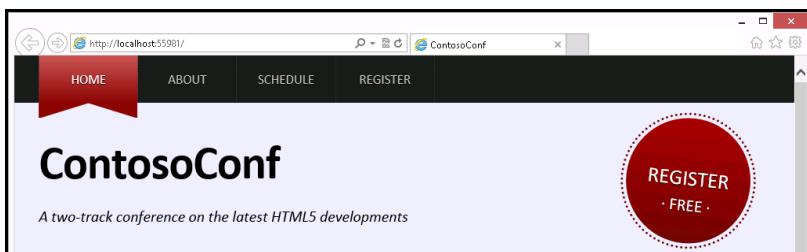


FIGURE 6.5: THE REGISTER LINK IN THE HEADER OF THE HOME PAGE

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.

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:

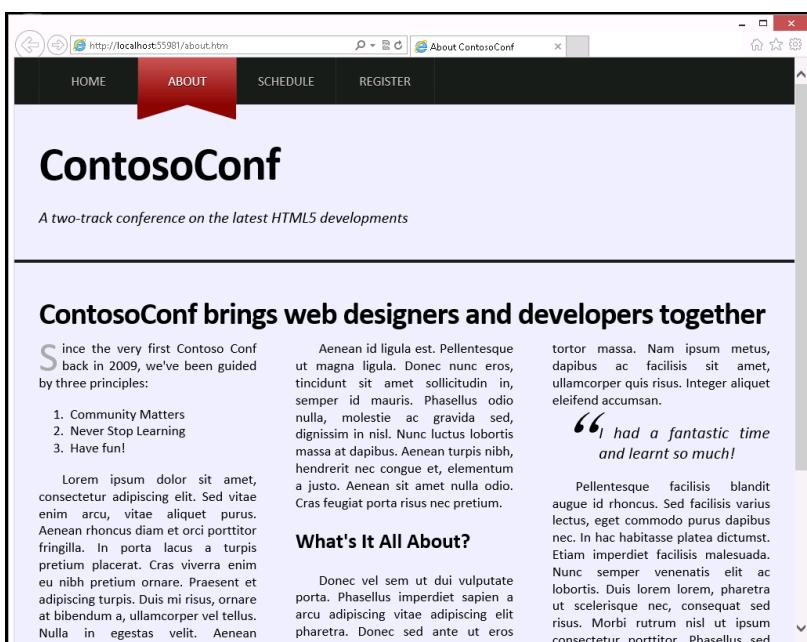


FIGURE 6.6: THE ABOUT PAGE

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 Questions

Check 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?

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	7-9
Lesson 3: Extending Objects	7-16
Lab: Refining Code for Maintainability and Extensibility	7-22
Module Review and Takeaways	7-23

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.
- Describe ES2015 Modules
- 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 with the **var** keyword 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.

ECMA-262 6th edition, also known as ECMAScript 2015, introduced block-scoped variables.

Block-scoped variables are scoped to the **block** statement they were defined in as well as in any contained sub-blocks.

Block-scoped variables are defined with the **let** or **const** keywords.

- JavaScript variables declared with **var** have one of two scopes:
 - Global scope
 - Local scope within a function
- JavaScript variables declared with **let** or **const** are block scoped.
- Prefer **let** and **const**.

 **Note:** A **block** statement is a group of statements delimited by a pair of braces which may optionally be labeled.

When a function is invoked in JavaScript, all functions and variables declared by using the **var** keyword within the invoked function are defined first, before the function code is run. This mechanism is also referred to as hoisting, because it is as if the function and variable declarations were “hoisted” to the top of the invoked function.

The following example demonstrates how scoping and hoisting work in JavaScript. Note the following points:

- The example declares a global variable named **num1** and assigns it the value 7.
- The **demonstrateScopingAndHoisting()** function declares a **var** local variable named **num1** and another local variable named **num2** by using the **let** keyword inside an **if** statement block, and assigns them the values 42 and 43. This variable declared with the **var** keyword is hoisted automatically to function scope, as if it was declared at the start of the **demonstrateScopingAndHoisting()** function.
- After the **if** statement, the local **num1** variable is still in scope. Therefore the first **alert()** function displays the local **num1** variable (42), not the global **num1** variable (7).
- The local variable **num2** is block scoped to block the **if** statement, so after the **if** statement **num2** is no longer in scope and therefore trying to reference it will result in an exception that will be handled inside the **catch** block and the second **alert()** function will display the **num2 is not defined** message.

```
<script>
  const num1 = 7;
  function demonstrateScopingAndHoisting() {
    if (true) {
      const num1 = 42;
      let num2 = 43;
    }
    alert("The value of num1 is " + num1);
    try {
      alert("The value of num2 is " + num2);
    }
    catch(err){
      alert("num2 is not defined");
    }
  }
  demonstrateScopingAndHoisting();
</script>
```

 **Best Practice:** When possible in terms of browser compatibility, always prefer declaring variables with the **let** and **const** keywords.

The **var** keyword behavior is a common source of errors since many developers are unaware of the hoisting mechanism explained above. Working with block scope variables is more intuitive and less error prone.

Furthermore, the **let** and **const** keywords can help make your code easier to understand, as they help developers signal their intent. The **const** keyword indicates that the variable won't be reassigned, while **let** indicates that the variable might get reassigned inside its containing block.

```
const elem = document.querySelector("input");
let count = 1;
if( ... ){
  count+=1;
}
else{
  elem = null // this will throw an exception
}
```

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

ECMA-262 6th edition (ES2015) introduced two more features that can help manage the **Global** namespace better:

- Block-scope variables – discussed in the previous lesson.
- Module – discussed in the next topic.

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
    let localVar = ... ;
    let 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.

- 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
 - Block scoped variables (ES2015 only)
 - Modules (ES2015 only)

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:

```
const 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() {
    let 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.

ES2015 Modules

An important part of building a well-structured JavaScript code is breaking it down into modules. Modules are a self-contained and implement a distinct functionality allowing them to be updated, added or removed as necessary, without disrupting the entire system. Breaking your code into modules increases the maintainability and reusability of the code. In earlier versions of JavaScript, it was possible to emulate modules by using immediately invoked functions and namespaces discussed in the previous topic. ECMAScript-262 6th edition (ES2015) has introduced a standard format for modules in JavaScript. ES2015 modules are written in files. One module per file and one file per module.

To load a JavaScript file as a **module**, we need to add a **script** tag with a **type** attribute whose value is **module**.

```
<script src="mymodule.js" type="module" ></script>
```

Modules have a few key differences from regular script files

	Scripts	Modules
Default mode	Non-strict	Strict
Top-level variables and functions	Global	Local to the module
Top-level value of this	window	undefined
Run	Synchronously	Asynchronously

ES2015 modules are self-contained, so all top-level variables and functions are scoped locally to the module. Exposing an API from the module is done explicitly by using the **export** declaration, whereas consuming the module API's will be done by using the **import** declaration.

There are two types of exports:

- **Named exports**

A **module** can export multiple functionalities by prefixing them with the **export** declaration.

```
//---- calc.js ----
export function sum(x,y) {
    return x + y;
}
export function multiply(x, y) {
    return x * y;
}
//---- main.js ----
import { sum, multiply } from 'calc.js';
console.log(sum(4,4)); // 8
console.log(multiply(5,2)); // 10
```

The **calc** module is exporting the **sum** and **multiply** functions and they are imported by using their names inside the **main.js** file.

- ES2015 modules are written in files. One module per file and one file per module.
- To load a JavaScript file as a **module** we need to add a **script** tag with **type** attribute whose value is **module**
- Top-level variables and functions are scoped locally to the module.
- Exposing an API from the module is done explicitly by using the **export** declaration, whereas consuming the module API's will be done by using the **import** declaration.

A module can also be imported as a whole and its named exports will be referred by using the **property** notation.

```
//---- main.js ----
import * as calc from 'calc.js';
console.log(calc.sum(4,4)); // 8
console.log(calc.multiply(5,2)); // 10
```

- **Default exports**

A module can define a single default export, the main exported value.

```
//---- sum.js ----
export default function sum(x,y) {
    return x + y;
}
//---- main.js ----
import sumFunc from 'sum.js';
console.log(sumFunc(4,4)); // 8
```

The sum module defines the **sum** function as the default export, and it's imported inside the **main.js** file into the **sumFunc** variable.

 **Note: Imports and exports must be declared at top level.**

The structure of ES2015 modules is static. For example, we can't conditionally import or export. Not using **import** or **export** at the top level will result in a syntax error.

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 defines several singleton objects, such as:

- **Math**
- **JSON**

JavaScript also defines global functions, such as:

- **parseInt()**
- **parseFloat()**
- **isNaN()**

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:

```
let radius = 100 * Math.random();
let circumference = 2 * Math.PI * radius;
let 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:

```
let anObject;
...
let anObjectAsJsonString = JSON.stringify(anObject);
let 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:

```
let ageEnteredByUser;
let heightEnteredByUser;
...
let age = parseInt(ageEnteredByUser);
let 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**.

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.
- Describe and use ES2015 Classes

Creating Simple Objects

JavaScript defines a standard object named **Object**. You can create a new object by using the following simple syntax:

```
let 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:

```
let employee1 = new Object();
let employee2 = {};
```

- You can define properties and methods on an object:

```
let 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:

```
let 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:

```
let 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:

```
let 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:

```
const objectName = {
    property1: value1,
    property2: value2,
    ...
};
```

The following example shows how to create a new object and define its properties:

Object literal notation provides a shorthand way to create new objects and assign properties and methods:

```
const 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);
    }
};
```

```
const 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:

```
const 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);
    }
};
```

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:

```
const Account = function (id, name) {
    this.id = id;
    this.name = name;
    this.balance = 0;
    this.numTransactions = 0;
};
```

- 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

```
const 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:

```
let acc1 = new Account(1, "John");
let acc2 = new Account(2, "Mary");
```

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.

```
let acc1 = new Account(1, "John");
let 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:

```
let acc1 = new Account(1, "John");
let 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

```
let 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:

```
let 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.
const Account = function (id, name) { ... };
// Account prototype, same as before.
Account.prototype = { ... };
acc1 = new Account(...);
// Create an object by using the Account prototype.
let 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.

Using ES2015 Classes

ECMA-262 6th edition (ES2015) introduced classes for creating and extending objects. ES2015 classes are similar in syntax to a class definition in languages such as C# and Java but under the hood they are substantially different and implement the same JavaScript object-oriented model explained in the previous topics.

The following example shows how to define a class by using the **class** keyword:

- Classes, introduced in ES2015, are merely a shorthand syntax over the existing prototype-based object-oriented model.
 - Use the **class** keyword to create new class
 - Define constructor, methods and members inside the **class body**
- ```
class Account {
 constructor (id, name) {
 this.id = id;
 this.name = name;
 }
 deposit(amount) {
 this.balance += amount;
 this.numTransactions++;
 }
}
```

```
class Account {
 constructor (id, name) {
 this.id = id;
 this.name = name;
 this.balance = 0;
 this.numTransactions = 0;
 }
}
```

The **Account** class defined above is merely a shorthand syntax over the **Account** constructor function defined in the previous lessons. Just as with constructor functions we can now instantiate an **Account** instance by using the **new** keyword.

```
let acc1 = new Account(1, "John");
let acc2 = new Account(2, "Mary");
```

The body of the class is the part inside the braces, and this is where we define the constructor function as well as methods and members. Methods are actually functions that are defined in the **prototype** property of the object, as described in earlier lessons.

The following example shows how to add methods to our **Account** class:

```
class Account {
 constructor (id, name) {
 this.id = id;
 this.name = name;
 this.balance = 0;
 this.numTransactions = 0;
 }
 deposit(amount) {
 this.balance += amount;
 this.numTransactions++;
 }
 withdraw(amount) {
 this.balance -= amount;
 this.numTransactions++;
 }
 displayDetails() {
 alert(this.id + ", " +
 this.name + " balance $" +
 this.balance + "(" +
 this.numTransactions + " transactions)");
 }
}
```

**deposit**, **withdraw** and **displayDetails** are eventually assigned as properties to the **Account** prototype object, and we can invoke them as we did earlier.

```
let acc1 = new Account(1, "John");
let acc2 = new Account(2, "Mary");
acc1.deposit(100);
acc1.displayDetails();
acc2.withdraw(50);
acc2.displayDetails();
```

 **Best Practice:** ES2015 classes offer a more concise and intuitive syntax for defining an object and for inheritance (discussed in detail in the next lesson). If browser compatibility is not an issue, use classes instead of the constructor functions and directly manipulating the prototype object.

## 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.
- Implement inheritance in ES2015.
- 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.

- 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;
 }
 ...
}
```

## Using Closures to Achieve Encapsulation

```
const Person = function(name, age)
{
 // Private properties.
 let _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.
const 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 <https://aka.ms/moc-20480c-m7-pg1>.

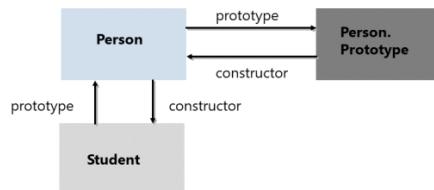
## Implementing Inheritance by Chaining Prototypes

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.
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.

- 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



**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.
const Person = function(name, age) {
 this.name = name;
 this.age = age;
}

// Base prototype.
Person.prototype = {
 haveBirthday: function() {
 this.age++;
 }
};

// Derived constructor.
const 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.
let aStudent = new Student("Jim", 20, "Physics");
aStudent.subject = "BioChemistry";
aStudent.haveBirthday();
alert(aStudent.age);
```

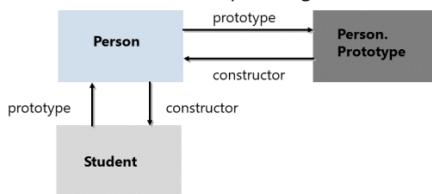
## Implementing Inheritance in ES2015 Classes

ECMA262-6<sup>th</sup> edition (ES2015) has introduced a much simpler and intuitive syntax for inheriting classes.

In ES2015, when defining a class with the **class** keyword we can define inheritance with the **extends** keyword followed by the base **object**. An important addition in ES2015 class is the **super** keyword, that is used to call the base class constructor or corresponding methods.

The following example shows how to implement inheritance in ES2015 classes.

- Define the base class with the **class** keyword
- Define the derived class with the **extends** keyword followed by the base class.
- Use the **super** keyword to call the base class constructor and corresponding methods



### Implementing inheritance in ES2015 classes

```
// Base class.
class Person {
 constructor(name, age) {
 this.name = name;
 this.age = age;
 }

 haveBirthday() {
 this.age++;
 }
}

// Derived class.
class Student extends Person {
 constructor(name, age, subject) {
 super(name, age);
 this.subject = subject;
 }
}

// 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.
let 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

```
const 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;
}

const p1 = new Point(100, 200);
p1.moveBy(10, 20);

const 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:

```
const 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.

### Demonstration Steps

You will find the steps in the "Demonstration: Refining Code for Maintainability and Extensibility" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD07\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD07_DEMO.md).

# 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 JavaScript code to enhance the website, you decide to refactor the existing code for better organizational practices. The resulting code will be more maintainable and provide a good pattern for implementing features in the future.

## Objectives

After completing this lab, you will be able to:

- Implement good practices for writing JavaScript code.
- Refactor JavaScript code to use object-oriented principles.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD07\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD07_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD07\\_LAK.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD07_LAK.md).

## Exercise 1: Refactoring JavaScript Code to Use Classes and 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 create a new class **ScheduleList**, and then you will move the existing functions and variables relating to the schedule list into this new class.

## 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 and ES2015 modules. Next, you saw how to create custom objects by using object literal syntax, constructors, prototypes and ES2015 classes. Finally, you saw how to implement encapsulation and inheritance in JavaScript.

### Review Questions

**Question:** How can you guard against name clashes in JavaScript?

**Question:** 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? |        |

### Check 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  |
| <b>Lesson 1:</b> Interacting with Files                    | 8-2  |
| <b>Lesson 2:</b> Incorporating Multimedia                  | 8-7  |
| <b>Lesson 3:</b> Reacting to Browser Location and Context  | 8-12 |
| <b>Lesson 4:</b> Debugging and Profiling a Web Application | 8-17 |
| <b>Lab:</b> Creating Interactive Pages with HTML5 APIs     | 8-19 |
| Module Review and Takeaways                                | 8-21 |

## 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 Microsoft Edge 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 <https://aka.ms/moc-20480c-m8-pg1>.

- 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:

- |                                                                                            |                                                                                          |                                                                    |
|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>• <b>progress</b></li> <li>• <b>load</b></li> </ul> | <ul style="list-style-type: none"> <li>• <b>abort</b></li> <li>• <b>error</b></li> </ul> | <ul style="list-style-type: none"> <li>• <b>loadend</b></li> </ul> |
|--------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|--------------------------------------------------------------------|

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

```

<input type="file" id="theTextFile"
onchange="onLoadTextFile()" />
<textarea id="theMessageArea" rows="30" cols="40"></textarea>

<script type="text/javascript">
 function onLoadTextFile() {
 const 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.
 const reader = new FileReader();
 reader.onload = function(e){
 const 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>

```

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.

## 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 **<img>** element on the web page.

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.

## Reading a Binary File

```
<input type="file" id="theBinaryFile" onchange="onLoadBinaryFile()" />

<script type="text/javascript">
 function onLoadBinaryFile() {
 const 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.
 const reader = new FileReader();
 reader.onload = function(e){
 const 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>
```

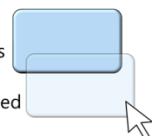
## Implementing Drag-and-Drop

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.
- **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)">
 Some content 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.

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.

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

• 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>
```

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 here
</video>
```

- A **<video>** element can support multiple video formats:

```
<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' />
</video>
```

- You can embed Silverlight or Flash content in a **<video>** tag as a fall-back

## 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.

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);
```

### Creating a Video Player by Using JavaScript Code

```
function createVideoElement(nameOfFile) {
 // Create a video object and set its properties.
 const 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.
 const hostElem = document.getElementById("videoDir");
 hostElem.appendChild(newVideo);
}
```

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**.

## 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 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



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.
- 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.

```
const 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) {
 const latitude = position.coords.latitude;
 const longitude = position.coords.longitude;
 const accuracy = position.coords.accuracy;
 const heading = position.coords.heading;
 const speed = position.coords.speed;
 const altitude = position.coords.altitude;
 const 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) {
 const errorMessage = error.message;
 const 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 2017 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 Microsoft Edge. 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

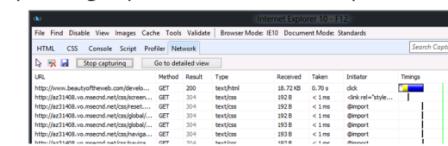
After completing this lesson, you will be able to describe how to use the HTML5 APIs and the F12 Developer Tools to debug Java Script code and to profile the performance of web pages.

## Overview of the F12 Developer Tools in Microsoft Edge

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



 **Note:** For more information about the timing data available in the **window.performance.timing** object, see <https://aka.ms/moc-20480c-m8-pg2>.

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

You will find the steps in the "Demonstration: Using the F12 Developer Tools to Debug JavaScript Code" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD08\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD08_DEMO.md).

## 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

You will find the steps in the "Demonstration: Using the F12 Developer Tools to Profile a Web Application" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD08\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD08_DEMO.md).

## 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.

### Demonstration Steps

You will find the steps in the "Demonstration: Creating Interactive Pages with HTML5 APIs" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD08\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD08_DEMO.md).

# Lab: Creating Interactive Pages with HTML5 APIs

## Scenario

The ContosoConf organizers want to highlight the latest HTML5 technologies to create an interactive experience for the visitors to 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. Add a web page that enables a speaker to drag-and-drop a profile picture to start creating their badge.
- A video from a previous conference is available. Make this video available on the Home page.
- Customize the Location page to display information about the visitor's current physical location.

## Objectives

After completing this lab, you will be able to:

- Add a video to an HTML page.
- Create interactive pages by using a drag-and-drop operation.
- Read files by using the File API.
- Get the location of the user by using the Geolocation API.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD08\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD08_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD08\\_LAK.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD08_LAK.md).

## 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.

## 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.

## 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.

# 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 Questions

**Question:** What methods are provided by the **FileReader** interface for reading files on the local file system?

**Question:** 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?

**Question:** 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?	



# 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	9-16
Module Review and Takeaways	9-18

## Module Overview

Web applications have a dependency on being able to connect to a network to fetch webpages 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 webpages, 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.

- 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

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.

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 Microsoft Edge, 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){
 ...
}
```

- 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

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**.

```
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 += "
" + 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 <https://aka.ms/moc-20480c-m9-pg1>.

## 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 webpages, although it is only available to webpages 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 += "
" + 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.getItem("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 <https://aka.ms/moc-20480c-m9-pg2>.

## 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 webpage of changes to data held in the store; it fires whenever data is modified.

The following example shows how to subscribe to this event:

- 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

```
function myStorageCallback(e) {
 alert("Key:" + e.key + " changed to " + e.newValue);
}
window.addEventListener("storage", myStorageCallback, true);
```

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 webpages to listen for storage events and to update themselves when data they use from the store changes state. For example, in Microsoft Edge, if you have the same webpage 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.



**Reader Aid:** For more information about handling storage events, visit <https://aka.ms/moc-20480c-m9-pg1>.

## 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");
};
```

- IndexedDB provides an efficient means for storing structured data on the user's computer
- The API is asynchronous, and includes the following features:
  - Multiple object stores
  - **add()**, **put()**, **get()**, and **delete()** operations on data
  - Transactions
  - Queries by using cursors
  - Indexes to speed up common queries

 **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$$wOrd"
};
var attendeeStore = db.createObjectStore("attendees", { keyPath: "id" });
```

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.



**Additional Reading:** For more information about using the IndexedDB API, visit <https://aka.ms/moc-20480c-m9-pg2>.

## 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 webpages 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 webpage or other resource is cached is transparent to the JavaScript code running on a webpage, 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

NETWORK:  
/login

# alternatives paths  
FALLBACK:  
/ajax/account/ /noCode.htm

<html manifest="appcache.manifest">

```
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 webpage 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 webpage must reference the manifest file that lists the resources to cache. A website can contain multiple manifest files, and different webpages can reference different manifest files. You specify the name of the manifest file to use in a webpage by adding the **manifest** attribute to the **<html>** element.

```
<html manifest="appcache.manifest">
```

## 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 webpage 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 webpage.
- **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);
```

- Use the **applicationCache** object to monitor the cache:

```
applicationCache.addEventListener("error", function() {
 alert("Error while downloading resources to the application cache");
}, true);
```

- Examine the **status** property to determine cache state:

0	UNCACHED	No resources have been downloaded.
1	IDLE	All cached resources have been downloaded.
2	CHECKING	The cache is being checked for updates.
3	DOWNLOADING	Resources are being downloaded to the cache.
4	UPDATEREADY	The cache has been updated with new resources.
5	OBsolete	The manifest is missing and no cache is available.



**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 <b>cached</b> event has been fired.
2	CHECKING	The cache is being checked for updates to download. The <b>checking</b> event has been fired. If no updates are found, the <b>noupdate</b> event is fired.
3	DOWNLOADING	Resources are being downloaded to the cache. The <b>downloading</b> event has been fired.
4	UPDATEREADY	The cache has been updated with new resources, and all resources have been downloaded. The <b>updateready</b> event has been fired.
5	OBSOLETE	The manifest is missing and no cache is available. The <b>obsolete</b> event has been fired.

 **Additional Reading:** For more information about using the Application Cache API to monitor the application cache, visit <https://aka.ms/moc-20480c-m9-pg3>.

## 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 webpage 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
```

- 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();
}
```

You can also use the **update()** function of the **applicationCache** object to initiate a check for updates, similar to the one performed when a webpage 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 webpage to use the new resources.

```
applicationCache.update();
...
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 webpage 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 webpage.

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.

- 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

```
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);
}
```

## 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.

### Demonstration Steps

You will find the steps in the "Demonstration: Adding Offline Support to Web Applications" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD09\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD09_DEMO.md).

# 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:

- Use the Application Cache API to make webpages available offline.
- Use the Local Storage API to persist user data locally between browser sessions.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD09\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD09_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD09\\_LAK.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD09_LAK.md).

## 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.

## 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.

## 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 webpages and to enable them to cache resources locally on the user's device.

### Review Questions

**Question:** What are the primary differences between data retained on a user's device by using session storage and by using local storage?

**Question:** Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
You configure a webpage 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	10-14
Module Review and Takeaways	10-15

## 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 webpages 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 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.

- 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



**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 <https://aka.ms/moc-20480c-m10-pg1>.

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 webpage, 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.

The `<wbr>` tag is similar to the `<br>` 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 webpage 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 Microsoft Edge 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, Microsoft Edge 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 that have their own levels of compliance.

The website at <https://aka.ms/moc-20480c-m10-pg2> 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 webpage. Alternatively, you can use a feature-detection library such as Modernizr.

## Printers

Users may want to print copies of selected parts of a webpage, 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 browser version that is Microsoft Edge being used to view a webpage, and to apply appropriate style sheet rules.
- Create a style sheet suitable for printing the content displayed by a webpage.

### 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 <https://aka.ms/moc-20480c-m10-pg3>.

## 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 webpages.

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 <https://aka.ms/moc-20480c-m10-pg4>.

## Detecting an Older Version of Internet Explorer by Using Conditional Comments

One of the main issues in developing a website front end in the past 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:

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]-->
```

```
<!--[if IE 9]>
<p>Welcome to Internet Explorer 9.</p>
<![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 webpage, 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 <https://aka.ms/moc-20480c-m10-pg5>.

## 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.

- 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

```

<link rel="stylesheet" type="text/css" href="print.css" media="print" />
@media print {
 .. print styling rules go here ..
}

```

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, refer to <https://aka.ms/moc-20480c-m10-pg1>.

## 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.

### Demonstration Steps

You will find the steps in the “Demonstration: Implementing an Adaptive User Interface” section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD10\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD10_DEMO.md).

# 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 webpage.
- Use CSS media queries to enable a webpage to adapt to different device form factors.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD10\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD10_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD10\\_LAK.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD10_LAK.md).

## Exercise 1: Creating a Print-Friendly Style Sheet

### Scenario

In this exercise, you will add a style sheet for formatting webpages 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.

## 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.

## 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 Questions

**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	11-18
Lab: Creating Advanced Graphics	11-25
Module Review and Takeaways	11-26

## 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 webpage. 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 webpages. 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 webpage.

## 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 webpage, the elements are kept in the DOM tree for the webpage, 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 webpage, 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 webpage, add an **<svg>** element and specify an XML namespace as follows:

```
<svg
 xmlns="http://www.w3.org/2000/svg">
 ...
</svg>
```

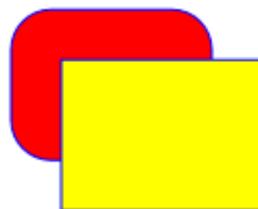
The **<svg>** element can contain any number of SVG-related elements, such as **<rect>** or **<ellipse>**. Each of these elements has a set of properties that enable you to configure its appearance within the **<svg>** element. The following list describes some of the common properties that you can set 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:



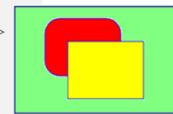
**FIGURE 11.1: RECTANGLES DRAWN BY USING AN <SVG> ELEMENT**

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>
```

- 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>
```



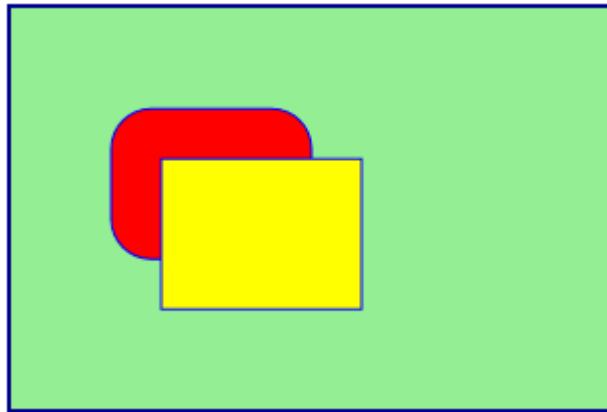
- Style SVG elements by using CSS:

```
<style type="text/css">
 svg {
 border: 2px solid darkblue;
 background-color: lightgreen;
 width: 300px;
 height: 200px;
 }
</style>
```

```

width: 300px;
height: 200px;
}
</style>
```

The `<svg>` element from the previous example looks like this after the styling shown above is applied:



## 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:

```

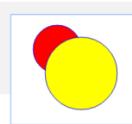
<svg xmlns="http://www.w3.org/2000/svg">
 <circle cx="120" cy="80" r="40" stroke="blue" fill="red" />
 <circle cx="160" cy="120" r="60" stroke="blue" fill="yellow" />
</svg>
```

The circles generated by this code look like this:

- To draw circles:

```

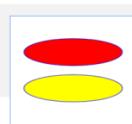
<circle cx="120" cy="80" r="40"
stroke="blue" fill="red" />
<circle cx="160" cy="120" r="60"
stroke="blue" fill="yellow" />
```

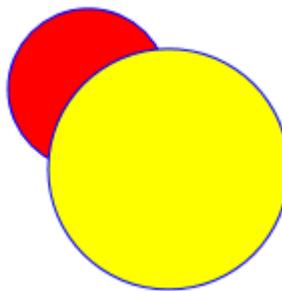


- To draw ellipses:

```

<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**

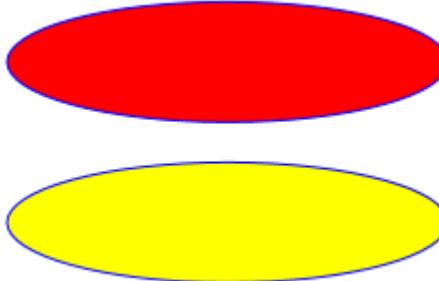
### Ellipse

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>**

- To draw a polyline:

```
<polyline points="105 100, 120 100, 125 90, 135 110, 145 90, ...
fill="none" stroke="blue" />
```



- To draw a polygon:

```
<polygon points="110 70, 150 40, 190 70, 190 160, 150 130, 110 160"
fill="yellow" stroke="blue" />
```



- To draw a path:

```
<path d="M 150 50 L 250 150 L 50 150 Z"
fill="red" stroke="blue" />
```



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**

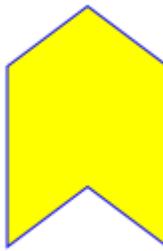
### Polygon

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:

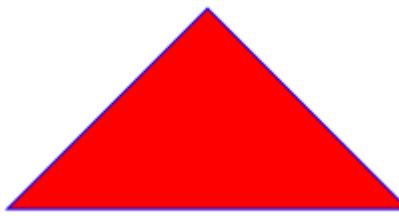
- **M:** Move to a new location, without drawing a line.
- **L:** Draw a line from the current location to a new location.

- **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 <https://aka.ms/moc-20480c-m11-pg1>.

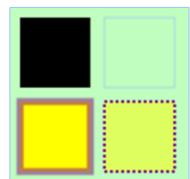
## Specifying Fill Styles and Stroke Styles

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.

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"
```



- **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**.



**Additional Reading:** For more information about fill rules, visit <https://aka.ms/moc-20480c-m11-pg2>.

- **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 11.8: 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 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:

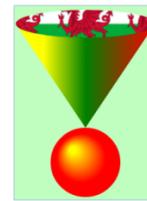
- **offset**: Specifies the location of the color stop along the linear gradient.
- **stop-color**: Specifies the color to apply at this location along the linear gradient.

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>
```

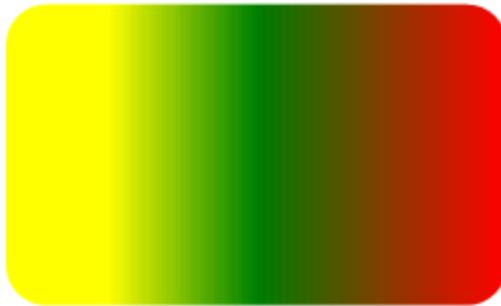
- You can define a linear or radial gradient for shapes
- Patterns can specify image files

```
<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)" />
```



 **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 11.9: A LINEAR GRADIENT APPLIED TO A RECTANGLE**

### The **<radialGradient>** Element

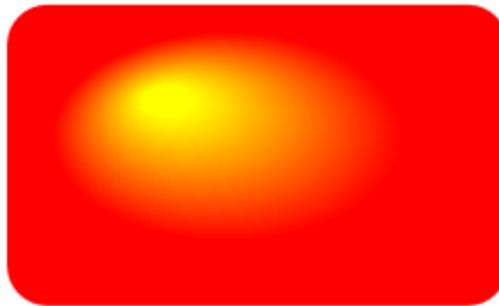
The **<radialGradient>** element creates a radial gradient of colors that can be applied to a shape. The **<radialGradient>** element has two attributes named **fx** and  that define the focal point of the radial gradient on the target shape.

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 11.10: 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 11.11: A PATTERN APPLIED TO A RECTANGLE**

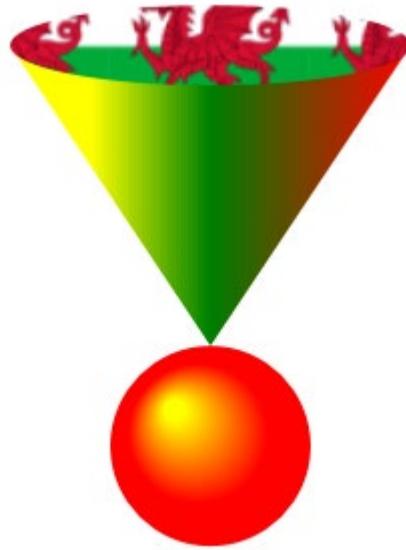
### Creating Reusable Gradients and Patterns

A webpage 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 11.12: 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>
```

<ul style="list-style-type: none"> <li>• To draw text:</li> </ul> <pre>&lt;text x="20" y="100"&gt;   Simple text &lt;/text&gt;</pre>	<ul style="list-style-type: none"> <li>• Use the following elements to achieve additional text effects:</li> </ul> <ul style="list-style-type: none"> <li>• <code>&lt;linearGradient&gt;</code>, <code>&lt;radialGradient&gt;</code>, <code>&lt;pattern&gt;</code></li> <li>• <code>&lt;textPath&gt;</code></li> <li>• <code>&lt;tspan&gt;</code></li> </ul> 
<p><b>Normal text</b></p> <p><b>Text with line through</b></p> <p><b>Underlined text</b></p> <p><b>Underlined text with line through</b></p>	 

The resulting text looks like this:

# Styled text

**FIGURE 11.13: 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 11.14: TEXT DRAWN AND DECORATED BY USING THE <TEXT> ELEMENT OF AN <SVG> ELEMENT**

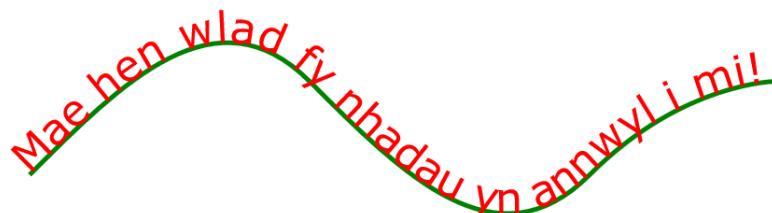
### Text Paths

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 webpage. 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>
```

The text and path drawn by this code looks like this:



**FIGURE 11.15: TEXT DRAWN BY USING TEXT PATH**

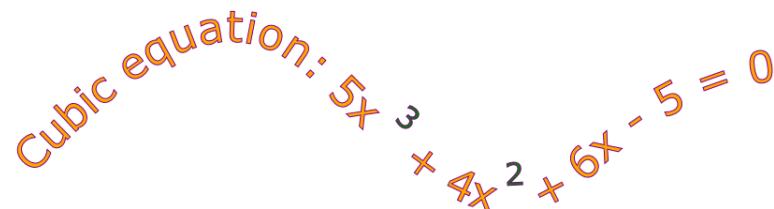
## Text Spans

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 11.16: TEXT CONSISTING OF SEVERAL `<TSPAN>` ELEMENTS**

## 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.

- To transform SVG elements, set the **transform** attribute to a transformation function

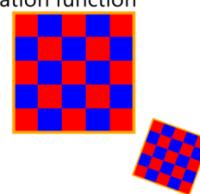
`rotate(angle, cx, cy)`

`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>`

- **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.

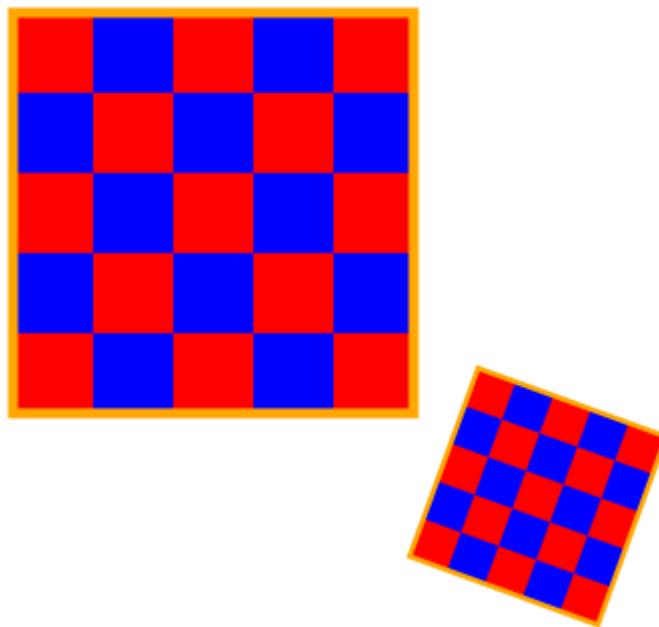


**Additional Reading:** For more information about SVG transformations, visit <https://aka.ms/moc-20480c-m11-pg3>.

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>
```

The rectangles drawn by this code look like this:



**FIGURE 11.17: A RECTANGLE ROTATED, TRANSFORMED, AND SCALED BY USING AN <SVG> ELEMENT**

## Demonstration: Using Scalable Vector Graphics (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

You will find the steps in the “Demonstration: Using Scalable Vector Graphics (SVG) Transformations and Events” section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD11\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD11_DEMO.md)

## 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 webpage. 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 webpage 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 webpage. 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"

## Using the Canvas API

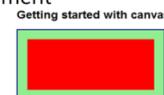
To use the Canvas API, the first step is to add a `<canvas>` element to your webpage. You can also define a CSS style for `<canvas>` elements, if required.

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.

- 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);
```

To draw graphics on a canvas, you must write JavaScript code. Follow these steps:

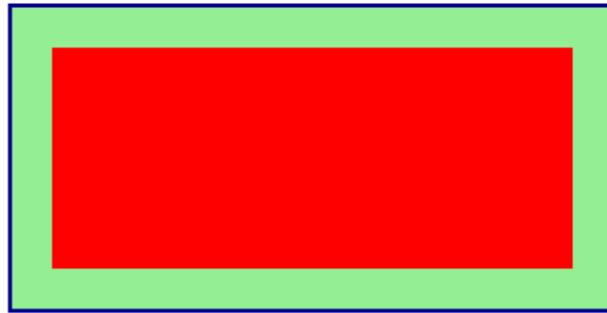
1. Get a reference to the `<canvas>` element by calling a DOM function such as `document.getElementById()`.
2. Call `getContext('2d')` on the canvas object, to get the two-dimensional drawing context for the canvas.
3. Invoke methods in the context object, to draw shapes on the canvas surface.

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>
 const canvas = document.getElementById('myCanvas');
 const 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:

# Getting started with canvas



**FIGURE 11.18: 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, refer to <https://aka.ms/moc-20480c-m11-pg1>.

## 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)**.

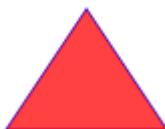
```
<canvas id="myCanvas">
 Sorry, your browser does not support canvas.
</canvas>
```

```

</canvas>
<script>
 // Get the canvas element and its drawing context.
 const canvas = document.getElementById('myCanvas');
 const 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 11.19: 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.

- To fill a shape with a gradient:

```
var grad = ctx.createLinearGradient(x1, y1, x2, y2);
var grad = ctx.createRadialGradient(startCx, startY, 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("animImageElement");
var pattern = ctx.createPattern(image, "repeat");
ctx.fillStyle = pattern;
```



## 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.

## The **createPattern()** Function

The **createPattern()** function creates a pattern, typically based on an image or some other content on the webpage. 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.
 const canvas = document.getElementById('myCanvas');
 const context = canvas.getContext('2d');
 context.lineWidth = 5;

 demoLinearGradient();
 demoRadialGradient();
 demoPattern();

 function demoLinearGradient()
 {
 const 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()
 {
 const 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()
 {
 const image = document.getElementById("wales");
 const 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)";
 }
</script>
```

```

 context.beginPath();
 context.moveTo(70, 30);
 context.lineTo(130, 140);
 context.lineTo(10, 140);
 context.closePath();
 context.fill();
 context.stroke();

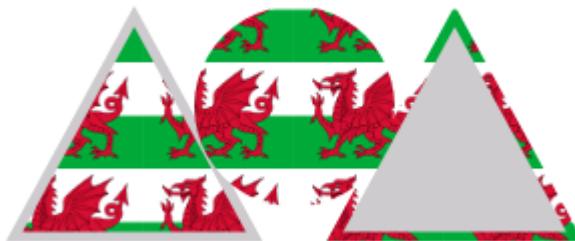
 // Draw a stroked circle, still using the specified style.
 context.beginPath();
 context.arc(canvas.width/2, canvas.height/2, 50, 0, 2*Math.PI);
 context.fill();

 // 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 11.20: 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.

- 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);
```

- **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.



**Additional Reading:** For more information about Canvas transformations, visit <https://aka.ms/moc-20480c-m11-pg2>.

## 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

You will find the steps in the "Demonstration: Performing Transformations by Using the Canvas API" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD11\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD11_DEMO.md).

## Demonstration: Creating Advanced Graphics

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

### Demonstration Steps

You will find the steps in the "Demonstration: Creating Advanced Graphics" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD11\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD11_DEMO.md).

# 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 webpage that enables a speaker to create a badge.

## Objectives

After completing this lab, you will be able to:

- Create graphics by using Scalable Vector Graphics (SVG), interactively style SVG graphics, and handle SVG graphics events.
- Draw graphics by using the Canvas API.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD11\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD11_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD11\\_LAK.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD11_LAK.md).

## 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 Cascading Style Sheets (CSS). Then, you will handle SVG element click events to display extra information about the conference rooms. Finally, you will run the application, view the Location page, and then verify that the venue map is interactive.

## 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 the JavaScript code to implement methods that draw parts of the badge. Finally, you will run the application and test the speaker badge page.

## Module Review and Takeaways

In this module, you have seen how to use SVG and the Canvas API to draw graphical content in a webpage.

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 webpage.

### Review Questions

#### Check Your Knowledge

Question
Which of the following statements about SVG is false?
Select the correct answer.
<input type="checkbox"/> You can use SVG to draw complex shapes, and fill them with gradients and patterns.
<input type="checkbox"/> SVG elements are parsed by the browser when the page is first loaded, and they are then discarded from memory.
<input type="checkbox"/> You can create SVG elements dynamically by using DOM functions such as <code>document.createElement()</code> .
<input type="checkbox"/> You can handle events on SVG elements.
<input type="checkbox"/> SVG elements must be enclosed in an <code>&lt;svg&gt;</code> container element on a webpage.

**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	12-15
Lab: Animating the User Interface	12-20
Module Review and Takeaways	12-21

## 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 webpage and provide useful visual feedback on user actions.

This module describes how to enhance webpages 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:

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;
}
```



```
div {
 width: 300px;
}
div:hover {
 width: 600px;
}
```

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 <https://aka.ms/moc-20480c-m12-pg1>.

## 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:

<b>transition-property</b>	Target property of the transition
<b>transition-duration</b>	Duration of the transition
<b>transition-timing-function</b>	Defines how the transition speed varies
<b>transition-delay</b>	Delay before the transition starts
<b>transition</b>	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 <https://aka.ms/moc-20480c-m12-pg1>.

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 propertyName = e.propertyName;
 var elapsedTime = e.elapsedTime;
}
```

- **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.

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() {
 const divElements = document.querySelectorAll("div");
 for (let i = 0; i < divElements.length; i++) {
 divElements[i].addEventListener("transitionend", onTransitionend, true);
 }
 }
 const 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:

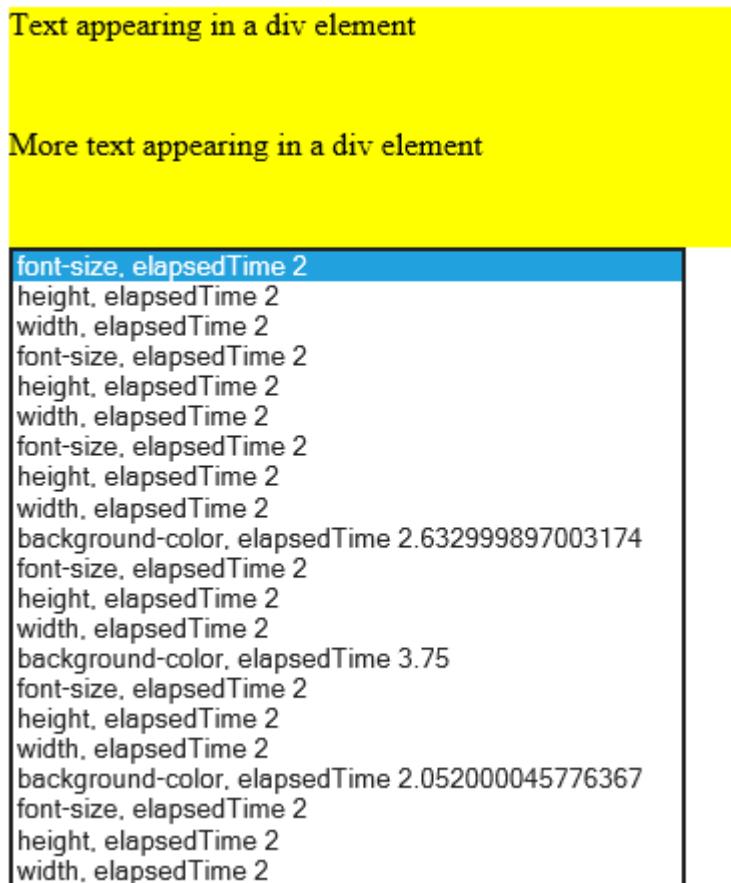


FIGURE 12.1: MESSAGES GENERATED BY HANDLING THE TRANSITIONEND EVENT

## 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

You will find the steps in the “Demonstration: Using CSS Transitions” section on the following page:

[https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md)

[CSS3/blob/master/Instructions/20480C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md).

## 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.

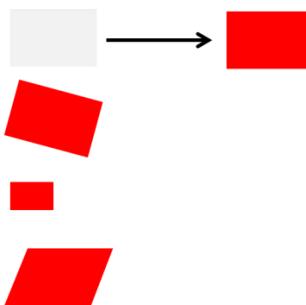
### Applying Transformations by Using CSS

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 webpage. 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 webpage 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;
 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 <https://aka.ms/moc-20480c-m12-pg3>.

## 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

### 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)**

- 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)`

#### Example

```
div.rotate1 {
 transform: rotate(10deg);
 transform-origin: left top;
}
```

- To perform a 2D skew transformation:  
`skew(anglex, [angley])`   `skewX(anglex)`   `skewY(angley)`

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);
}
```

## 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

You will find the steps in the "Demonstration: Performing 2D Transformations" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md).

## 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

- 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)`  
  
 This image shows two 3D perspective-transformed orange rectangles. The left rectangle is slightly larger and has the text 'This is child!' written on it. The right rectangle is slightly smaller and has the text 'This is child.' written on it. They are positioned as if they are children of a parent element.
- You must also specify a perspective:
  - Use the `perspective()` function, or
  - Specify `perspective` and `perspective-origin` properties

### 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

## 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:

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>
```

```
<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

You will find the steps in the "Demonstration: Performing 3D Transformations" section on the following page: [https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md).

## 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 webpage, 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:

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 ...
 }
}
```

```
@keyframes ballmovement {
 0% {
 left: 0px;
 top: 0px;
 background-color: yellow;
 }

 33% {
 left: 100px;
 top: 160px;
 }

 66% {
 left: 200px;
 top: 0px;
 }

 100% {
 left: 250px;
 top: 160px;
 background-color: red;
 }
}
```

```

}
100% {
 left: 300px;
 top: 160px;
 background-color: purple;
}
}

```

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 <https://aka.ms/moc-20480c-m12-pg4>.

## 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.
- **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.

- Apply the animation to a target element in a CSS rule

```

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**

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;
}
```

## 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 webpage 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() {
        const ball = document.getElementById("ball");
        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>
  ...
  const 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>
```

 **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

You will find the steps in the “Demonstration: Implementing KeyFrame Animations” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md.

- Keyframe animations raise the following events:

- **animationstart**
- **animationiteration**
- **animationend**

- The event-handler function receives an event argument with the following properties:

- **animationName**
- **elapsedTime**

Demonstration: Animating the User Interface

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

Demonstration Steps

You will find the steps in the “Demonstration: Animating the User Interface” section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_DEMO.md.

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:

- Animate HTML elements by using CSS transitions.
- Animate HTML elements using CSS keyframes, and trigger animations and handle animation events by using JavaScript code.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD12_LAK.md.

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.

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.

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 Questions

Question: What happens if you do not set the **transition-duration** property of a CSS transition?

Check 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 WebSockets

Contents:

Module Overview	13-1
Lesson 1: Introduction to WebSockets	13-2
Lesson 2: Using the WebSocket API	13-4
Lab: Performing Real-time Communication by Using WebSockets	13-10
Module Review and Takeaways	13-12

Module Overview

Webpages 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 WebSockets are useful. The WebSockets API provides a mechanism for implementing real-time, two-way communication between web server and browser.

This module introduces WebSockets, describes how they work, and explains how to create a WebSocket connection that can be used to transmit data in real time between a webpage and a web server.

Objectives

After completing this module, you will be able to:

- Describe how using WebSockets helps to enable real-time communications between a webpage and a web server.
- Use the WebSockets API to connect to a web server from a webpage, and exchange messages between the webpage and the web server.

Lesson 1

Introduction to WebSockets

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 WebSockets API enables webpages and web servers to exploit the socket protocol. In this lesson, you will learn why the WebSockets 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 WebSocket.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the problems that WebSockets are intended to solve.
- Describe how a client application connects to a server and exchanges data by using a WebSocket.

The Problem of Web-based Real-time Communications

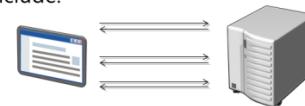
When creating real-time web applications, the need for continuous communication between the webpage in a browser and the web server is paramount. As soon as a user views a webpage, the data it displays might already be obsolete; stock values may have fallen or risen, or the tickets for a concert may 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

- Common techniques for implementing real-time communications include:

- Continuous polling
- Long polling



- Real-time communication solutions between webpage and server have two main issues

- Additional HTTP headers increase message size
- HTTP is not full-duplex

number of concurrent network connections, and a webpage may not be able to connect to a server when this limit is reached.

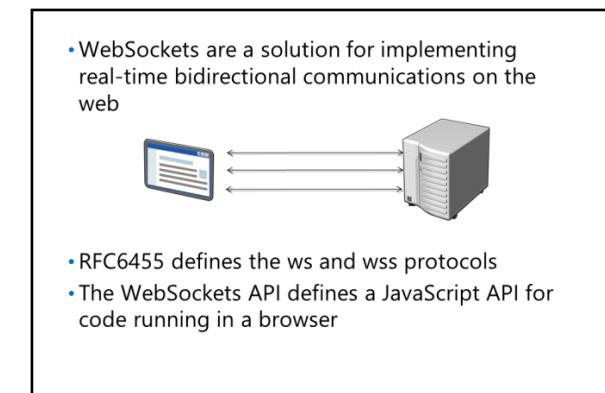
Both of these methods are well understood, but they share a common set of disadvantages:

- 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 WebSocket?

WebSockets 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 WebSockets 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.
- **WebSockets API**, which is a World Wide Web Consortium (W3C) specification that details a JavaScript API enabling webpages to use WebSockets. 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).



Modern browsers support the WebSockets API. Internet Explorer 10 is the first version of Internet Explorer to support WebSockets.

How Do WebSockets Work?

The WebSockets 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 WebSockets 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 WebSockets API.



Additional Reading: You can read RFC6455 at <https://aka.ms/moc-20480c-m13-pg1> and the latest copy of the WebSockets API at <https://aka.ms/moc-20480c-m13-pg2>.

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 webpage that:

- Connects to and disconnects from a web server by using a WebSocket.
- Sends messages to a server by using a WebSocket.
- Receives messages from a server by using a WebSocket.

Connecting to a Server by Using a WebSocket

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 webpage supports WebSockets 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 WebSocket

- Establish a connection by creating a new WebSocket:

```
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 WebSocket server is the opening handshake over HTTP between the client code running in a webpage 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 WebSocket address:

```
const 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:

```
const socket = new WebSocket("ws://localhost:55981/bookings");
```

If you need to establish an encrypted connection, you can use the secure WebSockets protocol **wss://**. This protocol uses port 443 by default.

```
const 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 WebSocket 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() {
    // WebSocket 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 WebSocket

After you have established a connection to a server through a **WebSocket**, you can send a message to the server by using the **send()** function of the **WebSocket** object, like this:

```
const 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) {
    const 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) {
    const 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) {
    const image = canvas.getImageData(0,0,50,50);
    const binArray = new Uint8Array(image.data.length);
    for (let 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 WebSocket

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 **WebSocket** 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) {
    const 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") {
        const binArray = new Uint8Array(data);
        ...
    } else {
        const canvas = document.getElementById("serverCanvas");
        const image = canvas.getImageData(0,0,50,50);
        for (let 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 WebSockets

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

Demonstration Steps

You will find the steps in the "Demonstration: Performing Real-time Communication by Using WebSockets" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD13_DEMO.md.

Lab: Performing Real-time Communication by Using WebSockets

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 WebSocket 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 WebSocket 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 WebSocket server will transmit a message to all connected clients, and the webpage must be updated to remove the question.

Objectives

After completing this lab, you will be able to:

- Create a WebSocket that connects to a server and receives messages.
- Serialize and send messages to a WebSocket.
- Send and receive different types of messages by using a WebSocket.

Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD13_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD13_LAK.md.

Exercise 1: Receiving Messages from a WebSocket

Scenario

In this exercise, you will review the new **Live** page and JavaScript. You will write JavaScript that creates a WebSocket and connect the socket to the server. Then you will handle messages received from the WebSocket. 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.

Exercise 2: Sending Messages to a WebSocket

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 WebSocket. 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.

Exercise 3: Handling Different WebSocket 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.

Module Review and Takeaways

In this module, you have learned how WebSockets provide a fast, robust and efficient solution for real-time communication between a webpage and its server. You have also learned how to connect to a WebSocket from a client application by using the WebSocket API, and how to send and receive messages over a WebSocket.

Review Questions

Question: Verify the correctness of the statement by placing a mark in the column to the right.

Statement	Answer
WebSocket clients send and receive data over an HTTP connection. True or False?	

Check Your Knowledge

Question
How does a client create a connection to a WebSocket server?
Select the correct answer.
<input type="checkbox"/> The client uses the open() function of the WebSocket object and specifies the URL of the server.
<input type="checkbox"/> The client uses the connect() function of the WebSocket object and specifies the URL of the server.
<input type="checkbox"/> WebSockets 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.
<input type="checkbox"/> The client creates a new WebSocket object and specifies the URL of the server.
<input type="checkbox"/> 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	14-12

Module Overview

JavaScript code is a powerful tool for implementing functionality in a webpage, but you need to remember that this code runs either when a webpage loads or in response to user actions while the webpage 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 Microsoft Edge 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

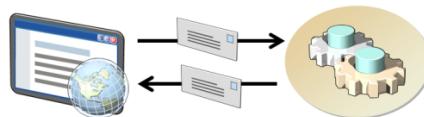
After completing this lesson, you will be able to:

- 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 webpage 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 webpage 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.

- 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



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.

A web worker enables you to offload long-running and data-intensive tasks to a separate execution environment, distinct from that of the webpage, leaving the browser free to handle the user interface. This keeps the webpage 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 webpage. It is initiated by code running in a webpage, but has no access to any resources on that page. Instead, the code in the webpage

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 webpage. The code in the webpage has full control over the web worker, and can terminate it at any time.

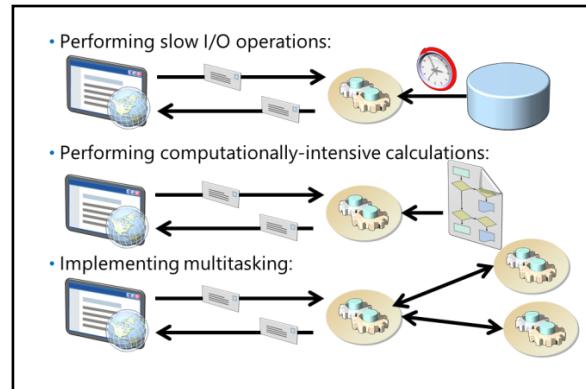


Additional Reading: For information about the web worker specification, visit <https://aka.ms/moc-20480c-m14-pg1>.

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 webpage 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 webpage 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 webpage 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 webpage 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 webpage. 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.



Web Worker Isolation

A web worker runs in isolation from the webpage 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 webpage 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.

- 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

Given that a web worker has no access to the data on a webpage, when a webpage creates a new web worker it must provide it with the information that it needs by passing messages. The webpage 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.

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 webpage 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 webpage, 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:

- 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();
```

```
const webWorker;
if( typeof(Worker)!== "undefined" ) {
    webWorker = new Worker("processScript.js");
}
```

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 webpage 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 webpage 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.

```
const workerBlob = new Blob(["{ /* Code for web worker goes here */ }"]);
const workerURL = URL.createObjectURL(workerBlob);
const 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">
      ...
      const workerBlob = new Blob([document.querySelector('#web-
worker').textContent]);
      const workerURL = URL.createObjectURL(workerBlob);
      const 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.

Communicating With A Dedicated Web Worker

Web worker scripts are completely isolated from the webpage, meaning that they have no access to objects in that page or in the DOM for that page. Instead, a webpage and a web worker communicate by passing messages to each other.

To send a message to a web worker, the webpage 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

webpage 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 webpage 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 webpage:

```
// processScript.js
function messageHandler(event) {
    self.postMessage("Received: " + event.data);
}
self.addEventListener("message", messageHandler, false);
```

The code in the webpage looks like this:

```
function replyHandler(event) {
    alert("Reply: " + event.data); // Display the reply in an alert
}
const 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 webpage 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.

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 webpage might construct a message that contains the action "DOWORK", together with the data that the web worker should process:

```
const msg = {
    "command": "DOWORK",
    "data": ...
};

webWorker.postMessage(msg);
```

- 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

The message handler in the web worker would look similar to this:

```
function messageHandler(event) {
    const 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.

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 webpage 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 webpage. 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 webpage 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 webpage 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) {
  ...
}
const sharedWebWorker;
...
sharedWebWorker = new SharedWorker("sharedProcessScript.js");
sharedWebWorker.port.addEventListener("message", replyHandler, false);
sharedWebWorker.port.start();
...
const 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 webpage 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) {
  const 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 webpage by using the **postMessage()** function of the port that the webpage 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 webpage, so the web worker must track this information itself. One technique is for a webpage to include an identifier in each message that it sends, and for the web worker to associate each identifier with the corresponding port.

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.

Demonstration Steps

You will find the steps in the "Demonstration: Creating a Web Worker Process" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD14_DEMO.md.

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 webpage. This photograph may be a color image. However, the conference speaker badges will be printed in grayscale. Therefore, the webpage 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:

- Create a web worker and implement a web worker script.
- Send messages to and receive messages from a web worker.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD14_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD14_LAK.md.

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.

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 webpage; they may be implemented as separate threads or processes. A webpage 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 webpage.
- A shared web worker can be accessed by any webpage in the web application that created it.

Review Questions

Question: 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 webpage, but dedicated web workers cannot. True or false?	

Question: How does a webpage communicate with a web worker?

Module 15

Packaging JavaScript for Production Deployment

Contents:

Module Overview	15-1
Lesson 1: Understanding Transpilers And Module Bundling	15-2
Lesson 2: Creating Separate Packages for Cross Browser Support	15-8
Lab: Setting up Webpack Bundle for Production	15-12

Module Overview

Modules help with managing and maintaining large and complex applications. Although ECMA262-6th edition (ES 2015) had introduced a new module format for JavaScript, browsers are just starting to support it, and using them natively may also cause some performance issues.

Tools such as webpack and Babel help optimize and turn modern modular JavaScript applications to be compatible with all browsers including older ones.

In this module, we will introduce the theory behind these tools, when to use them, and how to configure them. At the end of the module, we will see how to use these tools to write ECMA262-6th edition (ES 2015) code that is supported in all browsers.

Objectives

After completing this module, you will be able to:

- Describe the reasons for using transpilers and module bundling.
- Create separate packages for cross-browser support.

Lesson 1

Understanding Transpilers And Module Bundling

Modules, dependency management, and dynamic loading are basic requirements of any modern programming language, these are some of the most important features added to JavaScript in the latest versions. In this lesson, you will learn about the techniques and tools that enable the use of this technology in browser development.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe modules strategy in JavaScript.
- Describe the role of NodeJS and NPM and their importance in the compilation process.
- Describe the role of module bundlers and transpilers.

Modules strategy in JavaScript

Programmers use module distribution just as authors divide their books into chapters and sections.

Why use modules?

- Maintainability. Using a better division of code allows multiple people to work on the same code simultaneously without one person affecting another person's code.
- Scoping. Encapsulating variables and functions inside a private scope allows you to avoid defining them in the global scope, which might lead to their values getting overwritten.
- Reusability. Defining a collection of objects and functions under a module allows you to precisely define the role of those objects. This ensures that the module is wrapped and externalized only by its defined interface, which enables the accurate use of the capabilities provided by the module.

- Why use Modules?
 - Maintainability
 - Namespacing
 - Reusability
- The Module pattern is like a class in OOP
- Modules in JavaScript in past used Closure and Revealing module
- This method has several downsides:
 - Manage dependencies
 - Namespace collisions
- CommonJs and AMD

Module pattern

The idea behind the definition of a module is essentially similar to the concept of a class in object-oriented programming. As in a class, we define a public area intended for external use and a private area intended for internal implementation. Thus, a module is a collection of objects and functions where the internal implementation is encapsulated within the module and may be explicitly exposed for other modules.

Modules in JavaScript

JavaScript has existed since 1995 and to this day no browser supports modules natively. At the time of writing this module, native JavaScript modules introduced in the ECMA262-6th edition (ES 2015) are not supported in all browsers.

The various ways to produce JavaScript modules were based on one principle: use of a single global variable to wrap code in a function, thereby creating a private namespace for itself using a closure scope.

This method has several downsides:

- Manage dependencies. A developer who wants to use a module must recognize its dependencies. For example, if you are a developer and want to use a library such as **Backbone.js**, you add the line that loads the script to index.html to use the directory. Suppose this library has a dependency on another library named **Underscore.js**, you should know about this dependency to load the relevant script one line before you load the backbone library.
- Namespace collisions. What if two of your modules have the same name? Or what if you have two versions of a module, and you need both?

CommonJS

Node and CommonJS were created in 2009 and the vast majority of packages in npm use CommonJS modules.

With CommonJS, each JavaScript file gets its own module context, and its functionality outputs when using the **export** command. Other files use the **required** command to load the module.

When you're defining a CommonJS module, it might look something like this:

The following example shows how to define a CommonJS module:

CommonJS module

```
/app/myModule/myModule.js

function myModule() {
  this.sayHello = function() {
    return 'hello!';
  }

  this.foo = function() {
    return '1234!';
  }
}

module.exports = myModule;
```

CommonJS module system performs a transition on .js files and finds export sentences to know what we want to expose. When we want to use **myModule** in other files we need to call it with the **required** command, and then CommonJS knows to load the module.

The following example shows how to load a **CommonJS** module:

Loading a CommonJS Module

```
/app/main.js

var myModule = require('myModule');

var myModuleInstance = new myModule();
myModuleInstance.sayHello(); // 'hello!'
myModuleInstance.foo(); // 1234'
```

As you can see, there are several advantages to using CommonJS. First, there is no use of namespaces and there is no global variable from which everything should start. In addition, CommonJS simplifies dependency resolution of various modules, the code is also simple and understandable.

However, CommonJS is intended for server-side work, so when we load multiple modules they are loaded synchronously one after the other and this may lead to performance issues when working in the browser.

AMD

AMD takes a browser-first approach alongside asynchronous behavior to get the job done. Loading modules using AMD looks something like this:

The following example shows how to define and load the AMD module:

Defining and Loading an AMD Module

```
/app/main.js

define(['myModule', 'myOtherModule'], function(myModule, myOtherModule) {
  console.log(myModule.hello());
});

/app/myModule/myModule.js

define([], function() {

  return {
    sayHello: function() {
      console.log('hello');
    },
    foo: function() {
      console.log('1234!');
    }
  };
});
```

The **define** function takes as its first argument an array of each of the module's dependencies. These dependencies are loaded in the background (in a non-blocking manner), and once loaded, **define** calls the **callback** function it was given. The **callback** function takes the dependencies that were loaded as arguments. Finally, the dependencies themselves must also be defined by using the **define** keyword.

What is the role of module bundlers and transpilers?

Bundlers

Module bundling is the process of stitching a group of modules (and their dependencies) into a single file (or group of files) in the correct order.

When you divide your program into modules, you typically organize those modules into different files and folders. To load the different modules, the HTML page should contain **script** tags for each module. When a user visits the page, the browser loads each module individually from the server depending on the number of **script** tags on the page, which is bad news for page load times.

To get around this problem, you bundle or concatenate all your files into one big file (or a couple of files, as the case may be) to reduce the number of requests. Along with connecting the different modules to one large file, you also perform a "minify" operation. *Minification* is a process that removes unnecessary spaces and rows from the file to reduce its size. This process can achieve significant savings in the final file size and shorter load time. Concatenating and minifying is all you need if your code is written in standard JavaScript code.

- Bundlers
 - Stitch together a group of modules to single file
 - Minify and other operations to improve performance
 - Webpack is a relatively new module bundler
 - Webpack allows splitting code to "chunks"
- Transpilers
 - Compile latest version of JavaScript to an older version
 - Allow cross-browser compatibility

However, if you're adhering to non-native module systems that browsers can't interpret, such as CommonJS, AMD or even the native JavaScript module formats, simply connecting the files in the correct order is not enough. You need a tool that will also convert the modules into a language that the browser will understand. These tools are called *module bundlers* or *module loaders*.

Among module bundlers, webpack is relatively new. In the next lesson, we will learn how to configure and use webpack. It provides some useful features, such as *code splitting*—a way to split your codebase into chunks that are loaded on demand.

For example, if you have a web app where some of the code is not required all the time but only in some cases for special users, the bundle process is not very efficient because the first time you load the JavaScript file, you also load code that is not required in the first step and may not be needed at all. With webpack, you can divide the code that is not required in the initial loading of the page into the chunks and load them at run time, only when they are needed.

Transpilers

ECMA262-6th edition (ES 2015) includes many extras that are not supported in some browsers. To allow developers to write unified code that is supported by all browsers, we use tools that convert the advanced JavaScript code to JavaScript code supported by all browsers. These tools allow us to use all the latest features of the ECMAScript standard and compile it to ES5 so that we can use it in a production environment that is compatible with older browsers.

The following example shows ES6 code before compilation:

ES6 code

```
let foo1 = () => {};
let foo2 = (arg) => arg;

const double = [1,2,3].map((num) => num * 2);
console.log(double); // [2,4,6]

let person = {
  _name: " Dan Drayton",
  _friends: ["Joann Chambers", " Victoria Gray"],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};
console.log(person.printFriends());
```

The following example shows ES6 code after compilation:

ES5 Code

```
var foo1 = function foo1() {};
var foo2 = function foo2(arg) {
  return arg;
};

var double = [1, 2, 3].map(function (num) {
  return num * 2;
});
console.log(double); // [2,4,6]

var person = {
  _name: " Dan Drayton",
  _friends: ["Joann Chambers", "Victoria Gray"]
  printFriends: function printFriends() {
    var _this = this;
```

```

        this._friends.forEach(function (f) {
            return console.log(_this._name + " knows " + f);
        });
    };
    console.log(person.printFriends());
}

```

What is the role of Node.js and npm? Why are they important for bundling?

What is Node.js?

Node.js is a free open source server JavaScript framework that can run on various platforms (Windows, Linux, Unix, Mac OS X, etc.). Unlike JavaScript code in the client that runs inside the browser, in Node.js the JavaScript code runs on the operating system.

Why use Node.js?

Node.js allows JavaScript developers to write the client-side code and the server-side code too.

Additionally, node uses language benefits on the server side as well. For example, setting up a web server that can receive a request from a client to access a file in the file system and return its contents. In other technologies such as ASP or PHP, the workflow is synchronous. Therefore, until the contents of the file are not returned to the client, the system is not available to receive additional requests. In Node.js, however, the work is asynchronous. So, after the system receives the request from the client, an access to the file system does not limit it to accepting additional requests.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

The following example shows the code in Node.js which creates a web server on port 80 and returns "Hello World":

Node.js code

```

var http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);

```

- Node.js
 - Free open source server JavaScript framework which can run on various platforms
 - Single-threaded, non-blocking, asynchronous programming
- npm
 - Package (module) manager for node.js
 - Npm saves the installed modules into a unique folder called **node_modules**
 - Node.js and npm allow you to create a build pipeline to bundle and transpolar code

To execute a Node.js program, open your terminal, navigate to the folder that contains the main program file, and then run the **node** command followed by the file name.

What is NPM?

npm is a package (module) manager for Node.js. <http://www.npmjs.com> hosts thousands of free packages (modules) to download and use.

The npm program is installed on your computer when you install Node.js.

When you want to use a module from npm, you need to install the module with npm. Open the terminal, navigate to your app folder, and then type **npm install <module_name>**.

The installed modules are saved to a unique folder called **node_modules**, the installed package name and version is updated in a file named **package.json**. This file manages the app's metadata including third-party dependencies installed through npm.

Once the package is installed, it is ready to use.

The following example shows how to use the upper-case package in node code:

Node code

```
var http = require('http');
var uc = require('upper-case');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(uc("Hello World!"));
  res.end();
}).listen(8080);
```

How do node and npm help with bundling and transpilation

Tools such as webpack and Babel are based on Node.js and support third-party dependencies installed through npm. Additionally, you can use **package.json** to manage your project's dependencies and run various build scripts to create your application bundle.

Lesson 2

Creating Separate Packages for Cross Browser Support

In this lesson, you will learn how to use webpack and other tools to create packages that are optimized for different browsers.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe webpack concepts and configuration.
- Describe Babel concepts and configuration.
- Describe Babel polyfills.

Configuring webpack

To build a dependency pack, webpack runs over the code recursively. In the next stage, webpack loads every module in the app and packages all them into one or more bundles.

Entry

An entry point indicates which module webpack should use to begin building out its internal dependency graph. After entering the entry point, webpack will figure out the modules and libraries on which the entry point depends (directly and indirectly).

- Webpack runs over the code recursively and builds a dependency graph.
- Webpack loads every module in the app and packages all those modules into one or more bundles.
- The webpack configuration contains the following commands:
 - Entry
 - Output
 - Loaders
 - Plugins

Output

The output property indicates to webpack where to emit the bundles it creates and how to name these files. In the example below, we use the **path** package to publish the file to a new folder called **dist**.

Loaders (rules)

Loaders enable webpack to process more than just JavaScript files (webpack itself only understands JavaScript). They give you the ability to leverage webpack's bundling capabilities for all kinds of files by converting them to valid modules that webpack can process.

Essentially, webpack loaders transform all types of files into modules that can be included in your application's dependency graph (and eventually a bundle).

Loaders have two special keys in the webpack config file:

- The **test** property identifies which file or files should be transformed.
- The **use** property indicates which loader should be used to do the transforming.

The configuration below defines a **rules** property. This tells webpack to search for files with a .txt extension and to use the raw-loader to import them as a string into the bundle. The **rules** property should be defined under the **module.rules** module.

The ability to import any type of module, such as a .css files, is a feature that is specific to webpack and may not be supported by other bundlers or task runners.

Plugins

While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks. Plugins range from bundle optimization and minification all the way to defining environment-like variables. The plugin interface is extremely powerful and can be used to tackle a wide variety of tasks.

To use plugins in webpack, you need to install and require the specific package that you want to use. Inside the plugins array, you need to call the plugin with a new operator. Most plugins are customizable through options, this allows you to use a plugin multiple times in a config for different purposes.

In the example below, we use the **UglifyJS** webpack plugin to minify JavaScript, and the **Html** webpack plugin to find or create the HTML index file and to insert the script tag into the HTML that includes the reference to the bundle file.

Configuring webpack

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins
const path = require('path');

module.exports = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

Babel concepts and configuration

Babel solves the problem of lack of browser and platform (such as Node) support for the most up-to-date features of the ECMAScript standard. Developers always want to use the most modern features in their code but it takes time for browsers to support these features and update their versions.

Babel takes the JavaScript code that is written according to the latest standard and transpiles it (converting from one code language to another) to code that is compatible with older standards.

In many cases, Babel needs to add code snippets that mimic the operation of the code lines written in the new standard. These code snippets that mimic the capabilities of an innovative code are called *polyfills*.

Babel converts new a JavaScript standard to older versions to allow cross browser compatibility.



Demonstration: Using Babel CLI to Compile JavaScript Code

Demonstration Steps

You will find the steps in the "Demonstration: Using Babel CLI to Compile JavaScript Code" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD15_DEMO.md.

Babel Polyfills

Some of the features in ES6 are not enough to convert them to another code. In some cases, it is necessary to create an appropriate object because the object does not exist in the browser version.

If we try to compile code that includes the **promise** object, we can see that the **promise** object does not change with Babel transpiler. This is because Babel assumes that we have a global object called **Promise**.

Babel polyfill

- Allows the use of new built-in features such as Promise or WeakMap, and static methods such as Array.from or Object.assign.
- Install with npm.
- Need to be called before all other code or require statements.
- Load separately for each feature.



Additional Reading: Promises are explained in detail in Module 5 - Communicating with a Remote Server.

The following code example shows how to compile a promise ES6 object with Babel:

Compile Promise ES6 object with Babel

```
ES6 Code
let promise = new Promise((resolve, reject) => {
    let request = new XMLHttpRequest();

    request.open('GET', 'some REST api');
    request.onload = function() {
        if (request.status == 200) {
            resolve(request.response);
        } else {
            reject(Error(request.statusText));
        }
    };
    request.send(); //send the request
});

let success = (data) => {
    console.log('success');
}

let error = (error) => {
    console.log('error');
}

myPromise.then(success,error)

ES5 Code
'use strict';

var promise = new Promise(function (resolve, reject) {
    var request = new XMLHttpRequest();
```

```
request.open('GET', 'some REST api');
request.onload = function () {
    if (request.status == 200) {
        resolve(request.response);
    } else {
        reject(Error(request.statusText));
    }
};

request.send(); //send the request
});

var success = function success(data) {
    console.log('success');
};

var error = function error(_error) {
    console.log('error');
};

myPromise.then(success, error);
```

As can be seen from the code example, the **promise** object does not change. In this case, we need to use a Babel polyfill. A polyfill is simply a code snippet in JavaScript that simulates an inherent function in obsolete browsers. The **Promise** polyfill checks if the **Promise** global object exists. If it does not exist, it creates the **Promise** global object and puts in it all the functionality that should be there. For example, resolve, then and reject methods, all in standard ES5. So, browsers or environments that do not even know ES6 know what to do with it.

Demonstration: Using Webpack and Babel to build a JavaScript App

Demonstration Steps

You will find the steps in the "Demonstration: Using Webpack and Babel to Build a JavaScript App" section on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD15_DEMO.md.

Lab: Setting up Webpack Bundle for Production

Scenario

When you create a website and deploy it to a production server, you would want the site to be light and fast. You would also not want to make too many requests to get the site JavaScript file.

Therefore, you will want to bundle your website files into one file.

Objectives

After completing this lab, you will be able to:

- Configure webpack and Babel and create a deploy package for a web project.

Lab Setup

Estimated Time: **45 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD15_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20480-Programming-in-HTML5-with-JavaScript-and-CSS3/blob/master/Instructions/20480C_MOD15_LAK.md.

Exercise 1: Creating Deploy Package using Webpack

Scenario

In this exercise, you will add webpack and configure it for the ContosoConf project.

First, you will install the webpack package in the solution. You will then configure it to create a bundle file for all your JavaScript files. Then, you will add Babel to your solution to create a cross-platform package.

Next, you will run webpack to create a deploy package.

Finally, you will run the application and verify that the site remains the same with the bundle file.

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.

- 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.

Notes

Notes