



OFFICIAL MICROSOFT LEARNING PRODUCT

20483C

Programming in C#

MCT USE ONLY. STUDENT USE PROHIBITED

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: 20483C

Part Number: X21-88133

Released: 07/2018

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,
 - b. **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 Goldshtein – Subject Matter Expert

Sasha Goldshtein 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, Dhananjaya Punugoti and the Enfec Team.

Contents

Module 1: Review of Visual C# Syntax

Module Overview	1-1
Lesson 1: Overview of Writing Application by Using Visual C#	1-2
Lesson 2: Data Types, Operators, and Expressions	1-8
Lesson 3: Visual C# Programming Language Constructs	1-19
Lab: Developing the Class Enrollment Application	1-28
Module Review and Takeaways	1-30

Module 2: Creating Methods, Handling Exceptions, and Monitoring Applications

Module Overview	2-1
Lesson 1: Creating and Invoking Methods	2-2
Lesson 2: Creating Overloaded Methods and Using Optional and Output Parameters	2-8
Lesson 3: Handling Exceptions	2-13
Lesson 4: Monitoring Applications	2-18
Lab: Extending the Class Enrollment Application Functionality	2-24
Module Review and Takeaways	2-26

Module 3: Basic Types and Constructs of Visual C#

Module Overview	3-1
Lesson 1: Implementing Structs and Enums	3-2
Lesson 2: Organizing Data into Collections	3-9
Lesson 3: Handling Events	3-17
Lab: Writing the Code for the Grades Prototype Application	3-22
Module Review and Takeaways	3-24

Module 4: Creating Classes and Implementing Type-Safe Collections

Module Overview	4-1
Lesson 1: Creating Classes	4-2
Lesson 2: Defining and Implementing Interfaces	4-12
Lesson 3: Implementing Type-Safe Collections	4-21
Lab: Adding Data Validation and Type-Safety to the Application	4-34
Module Review and Takeaways	4-36

Module 5: Creating a Class Hierarchy by Using Inheritance

Module Overview	5-1
Lesson 1: Creating Class Hierarchies	5-2
Lesson 2: Extending .NET Framework Classes	5-11
Lab: Refactoring Common Functionality into the User Class	5-19
Module Review and Takeaways	5-21

Module 6: Reading and Writing Local Data

Module Overview	6-1
Lesson 1: Reading and Writing Files	6-2
Lesson 2: Serializing and Deserializing Data	6-12
Lesson 3: Performing I/O by Using Streams	6-26
Lab: Generating the Grades Report	6-34
Module Review and Takeaways	6-36

Module 7: Accessing a Database

Module Overview	7-1
Lesson 1: Creating and Using Entity Data Models	7-2
Lesson 2: Querying Data by Using LINQ	7-8
Lab: Retrieving and Modifying Grade Data	7-13
Module Review and Takeaways	7-15

Module 8: Accessing Remote Data

Module Overview	8-1
Lesson 1: Accessing Data Across the Web	8-2
Lesson 2: Accessing Data by Using OData Connected Services	8-11
Lab: Retrieving and Modifying Grade Data Remotely	8-21
Module Review and Takeaways	8-23

Module 9: Designing the User Interface for a Graphical Application

Module Overview	9-1
Lesson 1: Using XAML to Design a User Interface	9-2
Lesson 2: Binding Controls to Data	9-12
Lesson 3: Styling a UI	9-19
Lab: Customizing Student Photographs and Styling the Application	9-25
Module Review and Takeaways	9-27

Module 10: Improving Application Performance and Responsiveness

Module Overview	10-1
Lesson 1: Implementing Multitasking	10-2
Lesson 2: Performing Operations Asynchronously	10-14
Lesson 3: Synchronizing Concurrent Access to Data	10-24
Lab: Improving the Responsiveness and Performance of the Application	10-30
Module Review and Takeaways	10-31

Module 11: Integrating with Unmanaged Code

Module Overview	11-1
Lesson 1: Creating and Using Dynamic Objects	11-2
Lesson 2: Managing the Lifetime of Objects and Controlling Unmanaged Resources	11-7
Lab: Upgrading the Grades Report	11-13
Module Review and Takeaways	11-14

Module 12: Creating Reusable Types and Assemblies

Module Overview	12-1
Lesson 1: Examining Object Metadata	12-2
Lesson 2: Creating and Using Custom Attributes	12-10
Lesson 3: Generating Managed Code	12-16
Lesson 4: Versioning, Signing, and Deploying Assemblies	12-23
Lab: Specifying the Data to Include in the Grades Report	12-30
Module Review and Takeaways	12-32

Module 13: Encrypting and Decrypting Data

Module Overview	13-1
Lesson 1: Implementing Symmetric Encryption	13-2
Lesson 2: Implementing Asymmetric Encryption	13-8
Lab: Encrypting and Decrypting the Grades Report	13-16
Module Review and Takeaways	13-18

MCT USE ONLY. STUDENT USE PROHIBITED

About This Course

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

Course Description

This training course teaches developers the programming skills that are required to create Windows applications using the Visual C# language. During their five days in the classroom, students review the basics of Visual C# program structure, language syntax, and implementation details, and then consolidate their knowledge throughout the week as they build an application that incorporates several features of the .NET Framework 4.7.

Audience

This course is intended for experienced developers who already have programming experience in C, C++, JavaScript, Objective-C, Microsoft Visual Basic®, or Java and understand the concepts of object-oriented programming.

The developers targeted by this training are professional developers who have 3-6 months of experience creating software applications for a production environment and who have a basic understanding of Windows client application development. Students should have a minimum of the following experience:

- Three months of experience creating .NET Framework applications.
- One month of experience using Visual Studio 2015 or Visual Studio 2017.

This course is not designed for students who are new to programming. It is targeted at professional developers with at least one month of experience programming in an object-oriented environment.

Student Prerequisites

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

Additionally, developers attending this course should already have gained some limited experience of using Visual C# to complete basic programming tasks. More specifically, students should have hands-on experience using Visual C# that demonstrates their understanding of the following:

- How to name, declare, initialize and assign values to variables within an application.
- 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 create the code syntax for simple programming statements using Visual C# language keywords and recognize syntax errors by using the Visual Studio IDE.
- How to create a simple branching structure by using an if statement.
- How to create a simple looping structure by using a for statement to iterate through a data array.
- How to use the Visual Studio IDE to locate simple logic errors.
- How to create a method that accepts arguments and returns a value of a specified type.
- How to design and build a simple user interface by using standard controls from the Visual Studio toolbox.
- How to connect to an SQL Server database and the basics of how to retrieve and store data.

- How to sort data using loops.
- How to recognize the classes and methods used in a program.

Course Objectives

After completing this course, students will be able to:

- Describe the core syntax and features of Visual C#.
- Create methods, handle exceptions, and describe the monitoring requirements of large-scale applications.
- Implement the basic structure and essential elements of a typical desktop application.
- Create classes, define and implement interfaces, and create and use generic collections.
- Use inheritance to create a class hierarchy and to extend a .NET Framework class.
- Read and write data by using file input/output and streams, and serialize and deserialize data in different formats.
- Create and use an entity data model for accessing a database and use LINQ to query data.
- Access and query remote data by using the types in the System.Net namespace and WCF Data Services.
- Build a graphical user interface by using XAML.
- Improve the throughput and response time of applications by using tasks and asynchronous operations.
- Integrate unmanaged libraries and dynamic components into a Visual C# application.
- Examine the metadata of types by using reflection, create and use custom attributes, generate code at runtime, and manage assembly versions.
- Encrypt and decrypt data by using symmetric and asymmetric encryption.

Course Outline

The course outline is as follows:

- Module 1, "Review of Visual C# Syntax", explains some of the core features provided by the .NET Framework and Microsoft Visual Studio®, along with some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.
- Module 2, "Creating Methods, Handling Exceptions, and Monitoring Applications", explains how to create and use methods and how to handle exceptions, as well as logging and tracing to record the details of any exceptions that occur.
- Module 3, "Developing the Code for a Graphical Application", explains how to create and use structs and enums, organize data into collections, and create and subscribe to events.
- Module 4, "Creating Classes and Implementing Type-Safe Collections", describes how to use interfaces and classes to define and create custom, reusable types. Additionally, it covers creating and using enumerable, type-safe collections of any type.
- Module 5, "Creating a Class Hierarchy by Using Inheritance", describes how to use inheritance to create class hierarchies and to extend .NET Framework types.
- Module 6, "Reading and Writing Local Data", explains how to read and write data by using transactional file system I/O operations, how to serialize and deserialize data to the file system, and how to read and write data to the file system by using streams.

- Module 7, "Accessing a Database", explains how to create and use entity data models (EDMs) and how to query many types of data by using Language-Integrated Query (LINQ).
- Module 8, "Accessing Remote Data", depicts how to use the request and response classes in the System.Net namespace to directly manipulate remote data sources. It further expands on how to use Windows® Communication Foundation (WCF) Data Services to expose and consume an entity data model (EDM) over the web.
- Module 9, "Designing the User Interface for a Graphical Application", describes how to use Extensible Application Markup Language (XAML) and Windows Presentation Foundation (WPF) to create engaging UIs.
- Module 10, "Improving Application Performance and Responsiveness", explains how to improve the performance of applications by distributing operations across multiple threads.
- Module 11, "Integrating with Unmanaged Code", explains how to interoperate unmanaged code in your applications and how to ensure that your code releases any unmanaged resources.
- Module 12, "Creating Reusable Types and Assemblies", explains how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. Also how to generate code at run time by using the Code Document Object Model (CodeDOM) and how to ensure that the application's assemblies are signed and versioned, and available to other applications, by using the global assembly cache (GAC).
- Module 13, "Encrypting and Decrypting Data", explains how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of the data. It further teaches how to create and manage X509 certificates and how to use them in the asymmetric encryption process.

MCT USE ONLY STUDENT USE PROHIBITED

Course Materials

The following materials are included with your kit:

Course Handbook: a succinct classroom learning guide that provides the critical technical information in a crisp, tightly-focused format, which is essential for an effective in-class learning experience.

- *Lessons guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.*
- *Labs provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.*
- *Module Reviews and Takeaways provide on-the-job reference material to boost knowledge and skills retention.*
- *Lab Answer Keys provide step-by-step lab solution guidance.*

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

- Instruction files: <https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Instructions>
- Code files: <https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/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 email to certify@microsoft.com.

Virtual Machine Environment

This revision of the course does not include virtual machines. The minimum hardware and software requirements are detailed in the course's Setup Guide, at: <https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp>.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1

Review of Visual C# Syntax

Contents:

Module Overview	1-1
Lesson 1: Overview of Writing Application by Using Visual C#	1-2
Lesson 2: Data Types, Operators, and Expressions	1-8
Lesson 3: Visual C# Programming Language Constructs	1-19
Lab: Developing the Class Enrollment Application	1-28
Module Review and Takeaways	1-30

Module Overview

The Microsoft® .NET Framework version 4.7 provides a comprehensive development platform that you can use to build, deploy, and manage applications and services. By using the .NET Framework, you can create visually compelling applications, enable seamless communication across technology boundaries, and provide support for a wide range of business processes.

In this module, you will learn about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You will also learn about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

Objectives

After completing this module, you will be able to:

- Describe the architecture of .NET Framework applications and the features that Visual Studio 2017 and Visual C# provide.
- Use basic Visual C# data types, operators, and expressions.
- Use standard Visual C# constructs.

Lesson 1

Overview of Writing Application by Using Visual C#

The .NET Framework 4.7 and Visual Studio provide many features that you can use when developing your applications.

In this lesson, you will learn about the features that Visual Studio 2017 and the .NET Framework 4.7 provide that enable you to create your own applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of the .NET Framework.
- Describe the key features of Visual Studio 2017.
- Describe the project templates provided in Visual Studio 2017.
- Create a .NET Framework application.
- Describe XAML.

What Is the .NET Framework?

The .NET Framework 4.7 provides a comprehensive development platform that offers a fast and efficient way to build applications and services. By using Visual Studio 2017, you can use the .NET Framework 4.7 to create a wide range of solutions that operate across a broad range of computing devices.

The .NET Framework 4.7 provides three principal elements:

- The Common Language Runtime (CLR).
- The .NET Framework class library.
- A collection of development frameworks.

- CLR
 - Robust and secure environment for your managed code
 - Memory management
 - Multithreading
- Class library
 - Foundation of common functionality
 - Extensible
- Development frameworks
 - WPF
 - Universal Windows Platform
 - ASP.NET

The Common Language Runtime

The .NET Framework provides an environment called the CLR. The CLR manages the execution of code and simplifies the development process by providing a robust and highly secure execution environment that includes:

- Memory management.
- Transactions.
- Multithreading.

The .NET Framework Class Library

The .NET Framework provides a library of reusable classes that you can use to build applications. The classes provide a foundation of common functionality and constructs that help to simplify application development by, in part, eliminating the need to constantly reinvent logic. For example, the **System.IO.File** class contains functionality that enables you to manipulate files on the Windows file

system. In addition to using the classes in the .NET Framework class library, you can extend these classes by creating your own libraries of classes.

Development Frameworks

The .NET Framework provides several development frameworks that you can use to build common application types, including:

- Desktop client applications, by using Windows Presentation Foundation (WPF).
- Universal Windows Platform (UWP) applications, by using XAML.
- Server-side web applications, by using Microsoft ASP.NET Web Applications or ASP.NET MVC.
- Service-oriented web applications, by using ASP.NET Web API.
- Long-running applications, by using Windows services.

Each framework provides the necessary components and infrastructure to get you started.



Additional Reading: For more information about the .NET Framework, see the [Overview of the .NET Framework](#) page at <http://go.microsoft.com/fwlink/?LinkId=267639>.

Key Features of Visual Studio 2017

Visual Studio 2017 provides a single development environment that enables you to rapidly design, implement, build, test, and deploy various types of applications and components by using a range of programming languages.

Some of the key features of Visual Studio 2017 are:

- Intuitive integrated development environment (IDE). The Visual Studio 2017 IDE provides all of the features and tools that are necessary to design, implement, build, test, and deploy applications and components.
- Rapid application development. Visual Studio 2017 provides design views for graphical components that enable you to easily build complex user interfaces. Alternatively, you can use the Code Editor views, which provide more control but are not as easy to use. Visual Studio 2017 also provides wizards that help speed up the development of particular components.
- Server and data access. Visual Studio 2017 provides the Server Explorer, which enables you to log on to servers and explore their databases and system services. It also provides a familiar way to create, access, and modify databases that your application uses by using the new table designer.
- Internet Information Services (IIS) Express. Visual Studio 2017 provides a lightweight version of IIS as the default web server for debugging your web applications.
- Debugging features. Visual Studio 2017 provides a debugger that enables you to step through local or remote code, pause at breakpoints, and follow execution paths.
- Error handling. Visual Studio 2017 provides the **Error List** window, which displays any errors, warnings, or messages that are produced as you edit and build your code.

- Intuitive IDE
- Rapid application development
- Server and data access
- IIS Express
- Debugging features
- Error handling
- Help and documentation

- Help and documentation. Visual Studio 2017 provides help and guidance through Microsoft IntelliSense®, code snippets, and the integrated help system, which contains documentation and samples.



Additional Reading: For more information about what is new in Visual Studio 2017, see the **What's New in Visual Studio 2017** page at <https://aka.ms/moc-20483c-m1-pg1>

Templates in Visual Studio 2017

Visual Studio 2017 supports the development of different types of applications such as Windows-based client applications, web-based applications, services, and libraries. To help you get started, Visual Studio 2017 provides application templates that provide a structure for the different types of applications. These templates:

- Provide starter code that you can build on to quickly create functioning applications.
- Include supporting components and controls that are relevant to the project type.
- Configure the Visual Studio 2017 IDE to the type of application that you are developing.
- Add references to any initial assemblies that this type of application usually requires.

- Console Application
- WPF Application
- Universal Windows Platform (UWP)
- Class Library
- ASP.NET Web Application
- ASP.NET MVC 4 Application
- WCF Service Application

Types of Templates

The following table describes some of the common application templates that you might use when you develop .NET Framework applications by using Visual Studio 2017.

Template	Description
Console Application	Provides the environment settings, tools, project references, and starter code to develop an application that runs in a command-line interface. This type of application is considered lightweight because there is no graphical user interface.
WPF Application	Provides the environment settings, tools, project references, and starter code to build a rich graphical Windows application. A WPF application enables you to create the next generation of Windows applications, with much more control over user interface design.
UWP	Provides the environment settings, tools, project references, and starter code to build a rich graphical application targeted at the Windows 10 operating systems. UWP applications enable you to reuse skills obtained from WPF development by using XAML and Visual C#, but also from web development by using HTML 5, CSS 3.0, and JavaScript.
Class Library	Provides the environment settings, tools, and starter code to build a .dll assembly. You can use this type of file to store functionality that you might want to invoke from many other applications.
ASP.NET Web Application	Provides the environment settings, tools, project references, and starter code to create a server-side, compiled ASP.NET web application.

Template	Description
ASP.NET MVC 4 Application	Provides the environment settings, tools, project references, and starter code to create a Model-View-Controller (MVC) Web application. An ASP.NET MVC web application differs from the standard ASP.NET web application in that the application architecture helps you separate the presentation layer, business logic layer, and data access layer.
WCF Service Application	Provides the environment settings, tools, project references, and starter code to build Service Orientated Architecture (SOA) services.

Creating a .NET Framework Application

The application templates provided in Visual Studio 2017 enable you to start creating an application with minimal effort. You can then add your code and customize the project to meet your own requirements.

The following steps describe how to create a console application:

1. Open **Visual Studio 2017**.
2. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
3. In the **New Project** dialog box, do the following:
 - a. Expand **Templates**, **Visual C#**, and then click **Windows**.
 - b. Click the **Console Application** template.
 - c. In the **Name** box, specify a name for the project.
 - d. In the **Location** box, specify the path where you want to save the project.
4. Click **OK**.
5. The **Code Editor** window now shows the default **Program** class, which contains the entry point method for the application.

- ```

1. In Visual Studio, on the File menu, point to New, and then click Project.
2. In the New Project dialog box, choose a template, location, name, and then click OK.

```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
 class Program
 {
 static void Main(string[] args) { }
 }
}

```

The following code example shows the default **Program** class that Visual Studio provides when you use the **Console Application** template.

### Program Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
 class Program
 {
 static void Main(string[] args)
 {
 }
 }
}
```

After you create a project, you can then use the features that Visual Studio provides to create your application.

### Programmer Productivity Features

Visual Studio 2017 provides a host of features that can help you to write code. When writing code, you need to recall information about many program elements. Instead of manually looking up information by searching help files or other source code, the IntelliSense feature in Visual Studio provides the information that you need directly from the editor. IntelliSense provides the following features:

- The Quick Info option displays the complete declaration for any identifier in your code. Move the mouse so that the pointer rests on an identifier to display Quick Info for that identifier, which appears in a yellow pop-up box.
- The Complete Word option enters the rest of a variable, command, or function name after you have typed enough characters to disambiguate the term. Type the first few letters of the name and then press Alt+Right Arrow or Ctrl+Spacebar to complete the word.

## Overview of XAML

Extensible Application Markup Language (XAML) is an XML-based language that you can use to define your .NET application UIs. By declaring your UI in XAML as opposed to writing it in code makes your UI more portable and separates your UI from your application logic.

XAML uses elements and attributes to define controls and their properties in XML syntax. When you design a UI, you can use the toolbox and properties pane in Visual Studio to visually create the UI, you can use the XAML pane to declaratively create the UI, you can use Microsoft Expression Blend, or you can use other third-party tools. Using the XAML pane gives you finer grained control than dragging controls from the toolbox to the window.

- XML-based language for declaring UIs
- Uses elements to define controls
- Uses attributes to define properties of controls

```
<Label Content="Name." />
<TextBox Text="" Height="23" Width="120" />
<Button Content="Click Me!" Width="75" />
```

The following example shows the XAML declaration for a label, textbox, and button:

### Defining Controls in XAML

```
<Label Content="Name:"/>
<TextBox Text="" Height="23" Width="120" />
<Button Content="Click Me!" Width="75" />
```

You can use XAML syntax to produce simple UIs as shown in the previous example or to create much more complex interfaces. The markup syntax provides the functionality to bind data to controls, to use gradients and textures, to use templates for application-wide formatting, and to bind events to controls in the window. The toolbox in Visual Studio also includes container controls that you can use to position and size your controls appropriately regardless of how your users resize their application window.



**Additional Reading:** For more information about XAML, see Module 9 of this course.

## Lesson 2

# Data Types, Operators, and Expressions

All applications use data. This data might be supplied by the user through a user interface, from a database, from a network service, or from some other source. To store and use data in your applications, you must familiarize yourself with how to define and use variables and how to create and use expressions with the variety of operators that Visual C# provides.

In this lesson, you will learn how to use some of the fundamental constructs in Visual C#, such as variables, type members, casting, and string manipulation.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the data types provided by Visual C#.
- Create and use expressions.
- Declare and assign variables.
- Access type members.
- Cast data from one type to another.
- Concatenate and validate strings.

### What are Data Types?

A variable holds data of a specific type. When you declare a variable to store data in an application, you need to choose an appropriate data type for that data. Visual C# is a type-safe language, which means that the compiler guarantees that values stored in variables are always of the appropriate type.

### Commonly Used Data Types

The following table shows the commonly used data types in Visual C#, and their characteristics.

- int – whole numbers
- long – whole numbers (bigger range)
- float – floating-point numbers
- double - double precision
- decimal - monetary values
- char - single character
- bool - Boolean
- DateTime - moments in time
- string - sequence of characters

Type	Description	Size (bytes)	Range
int	Whole numbers	4	-2,147,483,648 to 2,147,483,647
long	Whole numbers (bigger range)	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Floating-point numbers	4	+/-3.4 × 10^38
double	Double precision (more accurate) floating-point numbers	8	+/-1.7 × 10^308

Type	Description	Size (bytes)	Range
decimal	Monetary values	16	28 significant figures
char	Single character	2	N/A
bool	Boolean	1	True or false
DateTime	Moments in time	8	0:00:00 on 01/01/2001 to 23:59:59 on 12/31/9999
string	Sequence of characters	2 per character	N/A

 **Additional Reading:** For more information about data types, see the [Reference Tables for Types \(C# Reference\)](#) page at <http://go.microsoft.com/fwlink/?LinkId=267770>.

## Expressions and Operators in Visual C#

Expressions are a central component of practically every Visual C# application, because expressions are the fundamental constructs that you use to evaluate and manipulate data. Expressions are collections of operands and operators, which you can define as follows:

- Operands are values, for example, numbers and strings. They can be constant (literal) values, variables, properties, or return values from method calls.
- Operators define operations to perform on operands, for example, addition or multiplication. Operators exist for all of the basic mathematical operations, as well as for more advanced operations such as logical comparison or the manipulation of the bits of data that constitute a value.

### Example expressions:

- + operator  
`a + 1`
- / operator  
`5 / 2`
- + and - operators  
`a + b - 2`
- + operator (string concatenation)  
`"ApplicationName: " + appName.ToString()`

All expressions are evaluated to a single value when your application runs. The type of value that an expression produces depends on the types of the operands that you use and the operators that you use. There is no limit to the length of expressions in Visual C# applications, although in practice, you are limited by the memory of your computer and your patience when typing. However, it is usually advisable to use shorter expressions and assemble the results of expression-processing piecemeal. This makes it easier for you to see what your code is doing, as well as making it easier to debug your code.

## Operators in Visual C#

Operators combine operands together into expressions. Visual C# provides a wide range of operators that you can use to perform most fundamental mathematical and logical operations. Operators fall into the following three categories:

- Unary. This type of operator operates on a single operand. For example, you can use the `-` operator as a unary operator. To do this, you place it immediately before a numeric operand, and it converts the value of the operand to its current value multiplied by `-1`.

- Binary. This type of operand operates on two values. This is the most common type of operator, for example, `*`, which multiplies the value of two operands.
- Ternary. There is only one ternary operator in Visual C#. This is the `? :` operator that is used in conditional expressions.

The following table shows the operators that you can use in Visual C#, grouped by type.

Type	Operators
Order of operations	<code>()</code>
Arithmetic	<code>+, -, *, /, %</code>
Increment, decrement	<code>++, --</code>
Comparison	<code>==, !=, &lt;, &gt;, &lt;=, &gt;=, is, ??</code>
String concatenation	<code>+</code>
Logical/bitwise operations	<code>&amp;,  , ^, !, ~, &amp;&amp;,   </code>
Indexing (counting starts from element 0)	<code>[ ]</code>
Casting	<code>( ), as</code>
Assignment	<code>=, +=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=</code>
Bit shift	<code>&lt;&lt;, &gt;&gt;</code>
Type information	<code>sizeof, typeof</code>
Delegate concatenation and removal	<code>+, -</code>
Overflow exception control	<code>checked, unchecked</code>
Indirection and Address (unsafe code only)	<code>*, -&gt;, [ ], &amp;</code>
Conditional (ternary operator)	<code>?:</code>

## Expression Examples

You can combine the basic building blocks of operators and operands to make expressions as simple or as complex as you like.

The following code example shows how to use the `+` operator.

### + Operator

```
a + 1
```

The `+` operator can operate on different data types, and the result of this expression depends on the data types of the operands. For example, if `a` is an integer, the result of the expression is an integer with the value 1 greater than `a`. If `a` is a double, the result is a double with the value 1 greater than `a`. The difference is subtle but important. In the second case (`a` is a double), the Visual C# compiler has to generate code to convert the constant integer value `1` into the constant double value `1` before the expression can be evaluated. The rule is that the type of the expression is the same as the type of the

operands, although one or more of the operands might need to be converted to ensure that they are all compatible.

The following code example shows how to use the `/` operator to divide two **int** values.

### / Operator

```
5 / 2
```

The value of the result is the integer value 2 (not 2.5). If you convert one of the operands to a double, the Visual C# compiler will convert the other operand to a double, and the result will be a double.

The following code example shows how to use the `/` operator to divide a **double** value by an **int** value.

### / Operator

```
5.0 / 2
```

The value of the result now is the double value 2.5. You can continue building up expressions with additional values and operators.

The following code example shows how use the `+` and `-` operators in an expression.

### + and - Operators

```
a + b - 2
```

This expression evaluates to the sum of variables **a** and **b** with the value 2 subtracted from the result.

Some operators, such as `+`, can be used to evaluate expressions that have a range of types.

The following code example shows how to use the `+` operator to concatenate two **string** values.

### + Operator

```
"ApplicationName: " + appName.ToString()
```

The `+` operator uses an operand that is a result of a method call, **ToString()**. The **ToString()** method converts the value of a variable into a string, whatever type it is.

The .NET Framework class library contains many additional methods that you can use to perform mathematical and string operations on data, such as the **System.Math** class.

 **Additional Reading:** For more information about operators, see the **C# Operators** page at <https://aka.ms/moc-20483c-m1-pg2>.

## Declaring and Assigning Variables

Before you can use a variable, you must declare it so that you can specify its name and characteristics. The name of a variable is referred to as an identifier. Visual C# has specific rules concerning the identifiers that you can use:

- An identifier can only contain letters, digits, and underscore characters.
- An identifier must start with a letter or an underscore.
- An identifier for a variable should not be one of the keywords that Visual C# reserves for its own use.

• Declaring variables:

```
int price;
// OR
int price, tax;
```

• Assigning variables:

```
price = 10;
// OR
int price = 10;
```

• Implicitly typed variables:

```
var price = 20;
```

• Instantiating object variables by using the **new** operator

```
ServiceConfiguration config = new ServiceConfiguration();
```

Visual C# is case sensitive. If you use the name **MyData** as the identifier of a variable, this is not the same as **myData**. You can declare two variables at the same time called **MyData** and **myData** and Visual C# will not confuse them, although this is not good coding practice.

When declaring variables you should use meaningful names for your variables, because this can make your code easier to understand. You should also adopt a naming convention and use it!

### Declaring and Assigning Variable

When you declare a variable, you reserve some storage space for that variable in memory and the type of data that it will hold. You can declare multiple variables in a single declaration by using the comma separator; all variables declared in this way have the same type.

The following example shows how to declare a new variable.

#### Declaring a Variable

```
// DataType variableName;
int price;
// OR
// DataType variableName1, variableName2;
int price, tax;
```

After you declare a variable, you can assign a value to it by using an assignment statement. You can change the value in a variable as many times as you want during the running of the application. The assignment operator **=** assigns a value to a variable.

The following code example shows how to use the **=** operator to assign a value to a variable.

#### Assigning a Variable

```
// variableName = value;
price = 10;
```

The value on the right side of the expression is assigned to the variable on the left side of the expression.

You can declare a variable and assign a value to it at the same time.

The following code example declares an **int** named **price** and assigns the value **10**.

## Declaring and Assigning Variables

```
int price = 10;
```

When you declare a variable, it contains a random value until you assign a value to it. This behavior was a rich source of bugs in C and C++ programs that created a variable and accidentally used it as a source of information before giving it a value. Visual C# does not allow you to use an unassigned variable. You must assign a value to a variable before you can use it; otherwise, your program might not compile.

## Implicitly Typed Variables

When you declare variables, you can also use the **var** keyword instead of specifying an explicit data type such as **int** or **string**. When the compiler sees the **var** keyword, it uses the value that is assigned to the variable to determine the type.

In the following example shows how to use the **var** keyword to declare a variable.

### Declaring a Variable by Using the var Keyword

```
var price = 20;
```

In this example, the **price** variable is an implicitly typed variable. However, the **var** keyword does not mean that you can later assign a value of a different type to **price**. The type of **price** is fixed, in much the same way as if you had explicitly declared it to be an **integer** variable.

Implicitly typed variables are useful when you do not know, or it is difficult to establish explicitly, the type of an expression that you want to assign to a variable.

## Object Variables

When you declare an object variable, it is initially unassigned. To use an object variable, you must create an instance of the corresponding class, by using the **new** operator, and assign it to the object variable.

The **new** operator does two things: it causes the CLR to allocate memory for your object, and it then invokes a constructor to initialize the fields in that object. The version of the constructor that runs depends on the parameters that you specify for the **new** operator.

The following code example shows how to create an instance of a class by using the **new** operator.

### The new Operator

```
ServiceConfiguration config = new ServiceConfiguration();
```

 **Additional Reading:** For more information about declaring and assigning variables, see the **Implicitly Typed Local Variables (C# Programming Guide)** page at <http://go.microsoft.com/fwlink/?LinkId=267772>.

## Accessing Type Members

To access a member of an instance of a type, use the name of the instance, followed by a period, followed by the name of the member. This is known as dot notation. Consider the following rules and guidelines when you access a member of an instance:

- To access a method, use parentheses after the name of the method. In the parentheses, pass the values for any parameters that the method requires. If the method does not take any parameters, the parentheses are still required.
- To access a public property, use the property name. You can then get the value of that property or set the value of that property.

The following code example shows how to invoke the members that the **ServiceConfiguration** class exposes.

### Invoking Members

```
var config = new ServiceConfiguration();
// Invoke the LoadConfiguration method.
var loadSuccessful = config.LoadConfiguration();
// Get the value from the ApplicationName property.
var applicationName = config.ApplicationName;
// Set the .DatabaseServerName property.
config.DatabaseServerName = "78.45.81.23";
// Invoke the SaveConfiguration method.
var saveSuccessful = config.SaveConfiguration();
```

- Invoke instance members

```
<instanceName>.<memberName>
```

- Example:

```
var config = new ServiceConfiguration();

// Invoke the LoadConfiguration method.
config.LoadConfiguration();

// Get the value from the ApplicationName property.
var applicationName = config.ApplicationName;

// Set the .DatabaseServerName property.
config.DatabaseServerName = "78.45.81.23";

// Invoke the SaveConfiguration method.
config.SaveConfiguration();
```

 **Additional Reading:** For more information about using properties, see the **Properties (C# Programming Guide)** page at <http://go.microsoft.com/fwlink/?LinkId=267773>.

 **Additional Reading:** For more information about using methods, see the **Methods (C# Programming Guide)** page at <http://go.microsoft.com/fwlink/?LinkId=267774>.

## Casting Between Data Types

When you are developing an application, you will often need to convert data from one type to another type, for example, when a value of one type is assigned to a variable of a different type. Consider the scenario where a user enters a number into a text box. To use this number in a numerical calculation, you will need to convert the string value 99 that you have read from the text box into the integer value 99 so that you can store it in an integer variable. The process of converting a value of one data type to another type is called type conversion or casting.

- Implicit conversion:

```
int a = 4;
long b = 5;
b = a;
```

- Explicit conversion:

```
int a = (int) b;
```

- **System.Convert** conversion:

```
string possibleInt = "1234";
int count = Convert.ToInt32(possibleInt);
```

MCT USE ONLY. STUDENT USE PROHIBITED

There are two types of conversions in the .NET Framework:

- Implicit conversion, which is automatically performed by the CLR on operations that are guaranteed to succeed without losing information.
- Explicit conversion, which requires you to write code to perform a conversion that otherwise could lose information or produce an error.

Explicit conversion reduces the possibility of bugs in your code and makes your code more efficient. Visual C# prohibits implicit conversions that lose precision. However, be aware that some explicit conversions can yield unexpected results.

### Implicit Conversions

An implicit conversion occurs when a value is converted automatically from one data type to another. The conversion does not require any special syntax in the source code. Visual C# only allows safe implicit conversions, such as the widening of an integer.

The following code example shows how data is converted implicitly from an integer to a long, which is termed *widening*.

#### Implicit Conversion

```
int a = 4;
long b;
b = a; // Implicit conversion of int to long.
```

This conversion always succeeds and never results in a loss of information. However, you cannot implicitly convert a **long** value to an **int**, because this conversion risks losing information (the **long** value might be outside the range supported by the **int** type). The following table shows the implicit type conversions that are supported in Visual C#.

From	To
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long, ulong	float, double, decimal
float	double
char	ushort, int, uint, long, ulong, float, double, decimal

### Explicit Conversions

In Visual C#, you can use a cast operator to perform explicit conversions. A cast specifies the type to convert to, in round brackets before the variable name.

The following code example shows how to perform an explicit conversion.

### Explicit Conversion

```
int a;
long b = 5;
a = (int) b; // Explicit conversion of long to int.
```

You can only perform meaningful conversions in this way, such as converting a **long** to an **int**. You cannot use a cast if the format of the data has to physically change, such as if you are converting a **string** to an **integer**. To perform these types of conversions, you can use the methods of the **System.Convert** class.

### Using the **System.Convert** Class

The **System.Convert** class provides methods that can convert a base data type to another base data type. These methods have names such as **ToDouble**, **ToInt32**, **ToString**, and so on. All languages that target the CLR can use this class. You might find this class easier to use for conversions than implicit or explicit conversions because IntelliSense helps you to locate the conversion method that you need.

The following code example converts a string to an int.

### Conversions by Using the **ToInt32** Method

```
string possibleInt = "1234";
int count = Convert.ToInt32(possibleInt);
```

Some of the built-in data types in Visual C# also provide a **TryParse** method, which enables you to determine whether the conversion will succeed before you perform the conversion.

The following code example shows how to convert a **string** to an **int** by using the **int.TryParse()** method.

### TryParse Conversion

```
int number = 0;
string numberString = "1234";

if (int.TryParse(numberString, out number))
{
 // Conversion succeeded, number now equals 1234.
}
else
{
 // Conversion failed, number now equals 0.
}
```



**Additional Reading:** For more information about casting variables, see the **Casting and Type Conversions (C# Programming Guide)** page at <http://go.microsoft.com/fwlink/?LinkId=267775>.

## Manipulating Strings

Strings are a very useful data type that enable you to capture and store alphanumeric data.

### Concatenating Strings

Concatenating multiple strings in Visual C# is simple to achieve by using the + operator. However, this is considered bad coding practice because strings are immutable. This means that every time you concatenate a string, you create a new string in memory and the old string is discarded.

- Concatenating strings

```
StringBuilder address = new StringBuilder();
address.Append("23");
address.Append(", Main Street");
address.Append(", Buffalo");
string concatenatedAddress = address.ToString();
```

- Validating strings

```
var textToTest = "hello w0rld";
var regularExpression = "\\d";

var result = Regex.IsMatch(textToTest, regularExpression,
 RegexOptions.None);

if (result)
{
 // Text matched expression.
}
```



**Note:** In Visual C#, data types are either mutable or immutable. This refers to their ability to change their intrinsic values. Immutable data types cannot be changed, and any change made to them results in the creation of a new copy of the modified value alongside the old value. Mutable data types can be changed and are not copied when changed. Therefore, when you concatenate a string, a new value with the complete string is created in memory, alongside the two original strings.

The following code example creates five string values as it runs.

### Concatenation by Using the + Operator

```
string address = "23";
address = address + ", Main Street";
address = address + ", Buffalo";
```

An alternative approach is to use the **StringBuilder** class, which enables you to build a string dynamically and much more efficiently.

The following code example shows how to use the **StringBuilder** class.

### Concatenation by Using the **StringBuilder** Class

```
StringBuilder address = new StringBuilder();

address.Append("23");
address.Append(", Main Street");
address.Append(", Buffalo");

string concatenatedAddress = address.ToString();
```

### Validating Strings

When acquiring input from the user interface of an application, data is often provided as strings that you need to validate and then convert into a format that your application logic expects. For example, a text box control in a WPF application will return its contents as a **string**, even if a user specified an **integer** value. It is important that you validate such input so that you minimize the risk of errors, such as **InvalidCastException**.

Regular expressions provide a mechanism that enables you to validate input. The .NET Framework provides the **System.Text.RegularExpressions** namespace that includes the **Regex** class. You can use the **Regex** class in your applications to test a **string** to ensure that it conforms to the constraints of a regular expression.

The following code example shows how to use the **Regex.IsMatch** method to see if a **string** value contains any numerical digits.

#### Regex.IsMatch Method

```
var textToTest = "hell10 wOrld";
var regularExpression = "\d";

var result = Regex.IsMatch(textToTest, regularExpression, RegexOptions.None);

if (result)
{
 // Text matched expression.
}
```

Regular expressions provide a selection of expressions that you can use to match to a variety of data types. For example, the **\d** expression will match any numeric characters.



**Additional Reading:** For more information about using regular expressions, see the **Regex Class** page at <https://aka.ms/moc-20483c-m1-pg3>.

## Lesson 3

# Visual C# Programming Language Constructs

When developing an application, you will often need to execute logic based on a condition, or to repeatedly execute a section of logic until a condition is met. You may also want to store a collection of related data in a single variable. Visual C# provides a number of constructs than enable you model complex behavior in your applications.

In this lesson, you will learn how to implement decision and iteration statements and how to store collections of related data. You will also learn how to structure the API of your application by using namespaces, and how to use some of the debugging features that Visual Studio provides.

## Lesson Objectives

After completing this lesson, you will be able to:

- Use conditional statements.
- Use iteration statements.
- Create and use arrays.
- Describe the purpose of namespaces.
- Use breakpoints in Visual Studio.

## Implementing Conditional Logic

Application logic often needs to run different sections of code depending on the state of data in the application. For example, if a user requests to close a file, they may be asked whether they wish to save any changes. If they do, the application must execute code to save the file. If they don't, the application logic can simply close the file. Visual C# uses conditional statements to determine which code section to run.

The primary conditional statement in Visual C# is the **if** statement. There is also a **switch** statement that you can use for more complex decisions.

```
• if statements
if (response == "connection_failed") { . . . }
else if (response == "connection_error") { . . . }
else { }

• select statements
switch (response)
{
 case "connection_failed":
 . . .
 break;
 case "connection_success":
 . . .
 break;
 default:
 . . .
 break;
}
```

### Conditional Statements

You use **if** statements to test the truth of a statement. If the statement is **true**, the block of code associated with the **if** statement is executed, if the statement is **false**, control passes over the block.

The following code shows how to use an **if** statement to determine if a **string** contains the value **connection\_failed**.

MCT USE ONLY. STUDENT LICENSE PROHIBITED

### If Statement

```
string response = "...";
if (response == "connection_failed")
{
 // Block of code to execute if the value of the response variable is
 "connection_failed".
}
```

**if** statements can have associated **else** clauses. The **else** block executes when the **if** statement is **false**.

The following code example shows how to use an **if else** statement to execute code when a condition is **false**.

### If else Statements

```
string response = "...";
if (response == "connection_failed")
{
 // Block of code executes if the value of the response variable is
 "connection_failed".
}
else
{
 // Block of code executes if the value of the response variable is not
 "connection_failed".
}
```

**if** statements can also have associated **else if** clauses. The clauses are tested in the order that they appear in the code after the **if** statement. If any of the clauses returns **true**, the block of code associated with that statement is executed and control leaves the block of code associated with the entire **if** construct.

The following code example shows how to use an **if** statement with an **else if** clause.

### else if Statements

```
string response = "...";
if (response == "connection_failed")
{
 // Block of code executes if the value of the response variable is
 "connection_failed".
}
else if (response == "connection_error")
{
 // Block of code executes if the value of the response variable is
 "connection_error".
}
else
{
 // Block of code executes if the value of the response variable is not
 "connection_failed" or "connection_error".
}
```

## Selection Statements

If there are too many **if/else** statements, code can become messy and difficult to follow. In this scenario, a better solution is to use a **switch** statement. The **switch** statement simply replaces multiple **if/else** statements.

The following sample shows how you can use a **switch** statement to replace a collection of **else if** clauses.

### switch Statement

```
string response = "...";
switch (response)
{
 case "connection_failed":
 // Block of code executes if the value of response is "connection_failed".
 break;
 case "connection_success":
 // Block of code executes if the value of response is "connection_success".
 break;
 case "connection_error":
 // Block of code executes if the value of response is "connection_error".
 break;
 default:
 // Block executes if none of the above conditions are met.
 break;
}
```

In each **case** statement, notice the **break** keyword. This causes control to jump to the end of the switch after processing the block of code. A **switch case** must end with a jump statement, such as **break**, **return** or **goto**. If you omit the jump statement, your code will not compile.

Notice that there is a block labeled **default**: This block of code will run when none of the other blocks match.

 **Additional Reading:** For more information about selection statements, see the [Selection Statements \(C# Reference\)](#) page at <https://aka.ms/moc-20483c-m1-pg4>.

## Implementing Iteration Logic

Iteration provides a convenient way to execute a block of code multiple times. For example, iterating over a collection of items in an array or just executing a function multiple times. Visual C# provides a number of standard constructs known as loops that you can use to implement iteration logic.

### For Loops

The **for** loop executes a block of code repeatedly until the specified expression evaluates to false. You can define a **for** loop as follows.

```
for ([initializers]; [expression]; [iterators])
{
 [body]
}
```

When using a **for** loop, you first initialize a value as a counter. On each loop, you check that the value of the counter is within the range to execute the **for** loop, and if so, execute the body of the loop.

The following code example shows how to use a **for** loop to execute a code block 10 times.

- **for** loop  
`for (int i = 0 ; i < 10; i++) { ... }`
- **foreach** loop  
`string[] names = new string[10];
foreach (string name in names) { ... }`
- **while** loop  
`bool dataToEnter = CheckIfUserWantsToEnterData();
while (dataToEnter)
{
 ...
 dataToEnter = CheckIfUserHasMoreData();
}`
- **do** loop  
`do
{
 ...
 moreDataToEnter = CheckIfUserHasMoreData();
} while (moreDataToEnter);`

### for Loop

```
for (int i = 0 ; i < 10; i++)
{
 // Code to execute.
}
```

In this example, **i = 0**; is the initializer, **i < 10**; is the expression, and **i++**; is the iterator.

### For Each Loops

While a **for** loop is easy to use, it can be tricky to get right. For example, when iterating over a collection or an array, you have to know how many elements the collection or array contains. In many cases this is straightforward, but sometimes it can be easy to get wrong. Therefore, it is sometimes better to use a **foreach** loop.

The following code example shows how to use a **foreach** loop to iterate a **string** array.

### foreach Loop

```
string[] names = new string[10];

// Process each name in the array.
foreach (string name in names)
{
 // Code to execute.
}
```

### While Loops

A **while** loop enables you to execute a block of code while a given condition is **true**. For example, you can use a **while** loop to process user input until the user indicates that they have no more data to enter.

The following code example shows how to use a **while** loop.

### while Loop

```
bool dataToEnter = CheckIfUserWantsToEnterData();
while (dataToEnter)
{
 // Process the data.
 dataToEnter = CheckIfUserHasMoreData();
}
```

### Do Loops

A **do** loop is very similar to a **while** loop, with the exception that a **do** loop will always execute at least once. Whereas if the condition is not initially met, a **while** loop will never execute. For example, you can use a **do** loop if you know that this code will only execute in response to a user request to enter data. In this scenario, you know that the application will need to process at least one piece of data, and can therefore use a **do** loop.

The following code example shows how to use a **do** loop.

### do Loop

```
do
{
 // Process the data.
 moreDataToEnter = CheckIfUserHasMoreData();
} while (moreDataToEnter);
```



**Additional Reading:** For more information about loops, see the [Iteration Statements \(C# Reference\)](#) page at <http://go.microsoft.com/fwlink/?LinkId=267778>.

## Creating and Using Arrays

An array is a set of objects that are grouped together and managed as a unit. You can think of an array as a sequence of elements, all of which are the same type. You can build simple arrays that have one dimension (a list), two dimensions (a table), three dimensions (a cube), and so on. Arrays in Visual C# have the following features:

- Every element in the array contains a value.
- Arrays are zero-indexed, that is, the first item in the array is element 0.
- The size of an array is the total number of elements that it can contain.
- Arrays can be single-dimensional, multidimensional, or jagged.
- The rank of an array is the number of dimensions in the array.

Arrays of a particular type can only hold elements of that type. If you need to manipulate a set of unlike objects or value types, consider using one of the collection types that are defined in the **System.Collections** namespace.

### Creating Arrays

When you declare an array, you specify the type of data that it contains and a name for the array.

Declaring an array brings the array into scope, but does not actually allocate any memory for it. The CLR physically creates the array when you use the **new** keyword. At this point, you should specify the size of the array.

The following list describes how to create single-dimensional, multidimensional, and jagged arrays:

- Single-dimensional arrays. To declare a single-dimensional array, you specify the type of elements in the array and use brackets, [] to indicate that a variable is an array. Later, you specify the size of the array when you allocate memory for the array by using the **new** keyword. The size of an array can be any integer expression. The following code example shows how to create a single-dimensional array of integers with elements zero through nine.

```
int[] arrayName = new int[10];
```

- Multidimensional arrays. An array can have more than one dimension. The number of dimensions corresponds to the number of indexes that are used to identify an individual element in the array. You can specify up to 32 dimensions, but you will rarely need more than three. You declare a multidimensional array variable just as you declare a single-dimensional array, but you separate the dimensions by using commas. The following code example shows how to create an array of integers with three dimensions.

```
int[, ,] arrayName = new int[10,10,10];
```

- C# supports:
  - Single-dimensional arrays
  - Multidimensional arrays
  - Jagged arrays

- Creating an array:

```
int[] arrayName = new int[10];
```

- Accessing data in an array:

- By index

```
int result = arrayName[2];
```

- In a loop

```
for (int i = 0; i < arrayName.Length; i++)
{
 int result = arrayName[i];
}
```

MCT USE ONLY STUDENT USE PROHIBITED

- Jagged arrays. A jagged array is simply an array of arrays, and the size of each array can vary. Jagged arrays are useful for modeling sparse data structures where you might not always want to allocate memory for every item if it is not going to be used. The following code example shows how to declare and initialize a jagged array. Note that you must specify the size of the first array, but you must not specify the size of the arrays that are contained within this array. You allocate memory to each array within a jagged array separately, by using the **new** keyword.

```
int[][] jaggedArray = new int[10][];
jaggedArray[0] = new Type[5]; // Can specify different sizes.
jaggedArray[1] = new Type[7];
...
jaggedArray[9] = new Type[21];
```

## Accessing Data in an Array

You can access data in an array in several ways, such as by specifying the index of a specific element that you require or by iterating through the entire collection and returning each element in sequence.

The following code example uses an index to access the element at index two.

### Accessing Data by Index

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
int number = oldNumbers[2];
```

 **Note:** Arrays are zero-indexed, so the first element in any dimension in an array is at index zero. The last element in a dimension is at index N-1, where N is the size of the dimension. If you attempt to access an element outside this range, the CLR throws an **IndexOutOfRangeException** exception.

You can iterate through an array by using a **for** loop. You can use the **Length** property of the array to determine when to stop the loop.

The following code example shows how to use a **for** loop to iterate through an array.

### Iterating Over an Array

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
for (int i = 0; i < oldNumbers.Length; i++)
{
 int number = oldNumbers[i];
 ...
}
```

 **Additional Reading:** For more information about arrays, see the **Arrays (C# Programming Guide)** page at <http://go.microsoft.com/fwlink/?LinkId=267779>.

## Referencing Namespaces

The Microsoft .NET Framework consists of many namespaces that organize its classes into logically related hierarchies. You can use namespaces in your own applications to similarly organize your classes into hierarchies.

Namespaces function as both an internal system for organizing your application and as an external way to avoid name clashes between your code and other applications. Each namespace contains types that you can use in your program, such as classes, structures, enumerations, delegates, and interfaces. Because different classes can have the same name, you use namespaces to differentiate the same named class into two different hierarchies to avoid interoperability issues.

- Use namespaces to organize classes into a logically related hierarchy

- .NET class library includes:
  - **System.Windows**
  - **System.Data**
  - **System.Web**

- Define your own namespaces:

```
namespace FourthCoffee.Console
{
 class Program { . . . }
```

- Use namespaces:
  - Add reference to containing library
  - Add **using** directive to code file

### .NET Framework Class Library Namespaces

The most important namespace in the .NET Framework is the **System** namespace, which contains the classes that most applications use to interact with the operating system. A few of the namespaces provided by the .NET Framework through the **System** namespace are listed in the following table:

Namespace	Definition
System.Windows	Provides the classes that are useful for building WPF applications.
System.IO	Provides classes for reading and writing data to files.
System.Data	Provides classes for data access.
System.Web	Provides classes that are useful for building web applications.

### User-Defined Namespaces

User-defined namespaces are namespaces defined in your code. It is good practice to define all your classes in namespaces. The Visual Studio environment follows this recommendation by using the name of your project as the top-level namespace in a project.

The following code example shows how to define a namespace with the name **FourthCoffee.Console**, which contains the **Program** class.

#### Defining a Namespace

```
namespace FourthCoffee.Console
{
 class Program
 {
 static void Main(string[] args)
 {
 }
 }
}
```

### Using Namespaces

When you create a Visual C# project in Visual Studio, the most common base class assemblies are already referenced. However, if you need to use a type that is in an assembly that is not already referenced in your

project, you will need to add a reference to the assembly by using the **Add Reference** dialog box. Then at the top of your code file, you list the namespaces that you use in that file, prefixed with the **using** directive. The **using** directive instructs your application to import the types in the namespace to allow them to be called directly, the same as if they existed in the current namespace.

The following code example shows how to import the **System** namespace and use the **Console** class.

### Importing a Namespace

```
using System;
...
Console.WriteLine("Hello, World");
```

You can call any type in a referenced assembly provided you specify that type's entire name, including its namespace. This is referred to as the type's fully qualified name.

The following code example shows how to use the **Console** class without importing the **System** namespace.

### Calling Console by its fully qualified name

```
System.Console.WriteLine("Hello, World");
```



**Additional Reading:** For more information about namespaces, see the [namespace \(C# Reference\)](#) page at <https://aka.ms/moc-20483c-m1-pg2>.

## Using Breakpoints in Visual Studio 2017

Debugging is an essential part of application development. You may notice errors as you write code, but some errors, especially logic errors, may only occur in circumstances that you do not predict. Users may report these errors to you and you will have to correct them.

Visual Studio 2017 provides several tools to help you debug code. You might use these while you develop code, during a test phase, or after the application has been released. You will use the tools in the same way regardless of the circumstances. You can run an application with or without debugging enabled. When debugging is enabled, your application is said to be in **Debug** mode.

- Breakpoints enable you to view and modify the contents of variables:
  - Immediate Window
  - Autos, Locals, and Watch panes
- Debug menu and toolbar functions enable you to:
  - Start and stop debugging
  - Enter break mode
  - Restart the application
  - Step through code

### Using Breakpoints

If you know the approximate location of the issue in your code, you can use a breakpoint to make the Visual Studio debugger enter break mode before executing a specific line of code. This enables you to use the debugging tools to review or modify the status of your application to help you rectify the bug. To add a breakpoint to a line of code, on the **Debug** menu, click **Toggle Breakpoint**.

When you are in break mode, you can hover over variable names to view their current value. You can also use the **Immediate Window** and the **Autos**, **Locals**, and **Watch** panes to view and modify the contents of variables.

MCT USE ONLY STUDENT USE PROHIBITED

## Using Debug Controls

After viewing or modifying variables in break mode, you will likely want to move through the subsequent lines of code in your application. You might want to simply run the remainder of the application or you might want to run one line of code at a time. Visual Studio provides a variety of commands on the **Debug** menu that enable you to do this and more. The following table lists the key items on the **Debug** menu and the **Debug** toolbar, and the corresponding keyboard shortcuts for navigating through your code.

Menu item	Toolbar button	Keyboard shortcut	Description
Start Debugging	Start/continue	F5	This button is available when your application is not running and when you are in break mode. It will start your application in Debug mode or resume the application if you are in break mode.
Break All	Break all	Ctrl+Alt+Break	This button causes application processing to pause and break mode to be entered. The button is available when an application is running.
Stop Debugging	Stop	Shift+F5	This button stops debugging. It is available when an application is running or is in break mode.
Restart	Restart	Ctrl+Shift+F5	This button is equivalent to stop followed by start. It will cause your application to be restarted from the beginning. It is available when an application is running or is in break mode.
Step Into	Step into	F11	This button is used for stepping into method calls.
Step Over	Step over	F10	This button is used for stepping over method calls.
Step Out	Step out	Shift+F11	This button is used for executing the remaining code in the method and returning to the next statement in the calling method.



**Additional Reading:** For more information about debugging, see the [Debugging in Visual Studio](#) page at <https://aka.ms/moc-20483c-m1-pg6>.

## Demonstration: Developing the Class Enrollment Application Lab

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: Developing the Class Enrollment Application Lab** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD01\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_DEMO.md).

# Lab: Developing the Class Enrollment Application

## Scenario

You are a Visual C# developer working for a software development company that is writing applications for The School of Fine Arts, an elementary school for gifted children.

The school administrators require an application that they can use to enroll students in a class. The application must enable an administrator to add and remove students from classes, as well as to update the details of students.

You have been asked to write the code that implements the business logic for the application.

 **Note:** During the labs for the first two modules in this course, you will write code for this class enrollment application.

When The School of Fine Arts ask you to extend the application functionality, you realize that you will need to test proof of concept and obtain client feedback before writing the final application, so in the lab for Module 3, you will begin developing a prototype application and continue with this until the end of Module 8.

In the lab for Module 9, after gaining signoff for the final application, you will develop the user interface for the production version of the application, which you will work on for the remainder of the course.

## Objectives

After completing this lab, you will be able to:

- Write Visual C# code that implements the logic necessary to edit the details of a student.
- Write Visual C# code that implements the logic necessary to add new students.
- Write Visual C# code that implements the logic necessary to remove students from a class.
- Perform simple data transformations for displaying information.

## Lab Setup

Estimated Time: **105 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD01\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD01\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD01_LAK.md).

## Exercise 1: Implementing Edit Functionality for the Students List

### Scenario

In this exercise, you will write the code that enables an administrator using the application to edit a student's details.

A list of students is displayed in the user interface of the application. When the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Enter. If they did press Enter, you will write code to display the student's details in a separate form, which the user can use to modify the details. When the user closes the form, you will copy the updated details back to the list box displaying the list of students. Finally, you will run the application to verify that your code functions as expected, and then use the debugging tools to examine code as it runs.

## Exercise 2: Implementing Insert Functionality for the Students List

### Scenario

In this exercise, you will write code that enables an administrator using the application to add a new student to the students list.

A list of students is displayed in the user interface of the application. When the user presses a key on the keyboard, you will check whether the key they pressed was Insert. If they did press Insert, you will write code to display a form in which the user can enter the details of a new student, including their first name, last name, and date of birth. When the user closes the form, you will add the new student to the list of students and display the details in the list box. Finally, you will run the application to verify that your code functions as expected.

## Exercise 3: Implementing Delete Functionality for the Students List

### Scenario

In this exercise, you will write code that enables an administrator to remove a student from the students list.

A list of students is displayed in the user interface of the application. If the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Delete. If they did press Delete, you will write code to prompt the user to confirm that they want to remove the selected student from the class. If they do, the student will be deleted from the students list for the appropriate class, otherwise nothing changes. Finally, you will run the application to verify that your code functions as expected.

## Exercise 4: Displaying a Student's Age

### Scenario

In this exercise, you will update the application to display a student's age instead of their date of birth.

You will write code in the **AgeConverter** class that is linked to the grid column displaying student ages. In this class, you will write code to work out the difference between the current date and the date of birth of the student, and then convert this value into years. Then you will run the application to verify that the Age column now displays age in years instead of the date of birth.

## Module Review and Takeaways

In this module, you learned about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You also learned about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

### Review Questions

#### Check Your Knowledge

Question
What Visual Studio template would you use to create a .dll?
Select the correct answer.
<input type="checkbox"/> Console application
<input type="checkbox"/> Windows Forms application
<input type="checkbox"/> WPF application
<input type="checkbox"/> Class library
<input type="checkbox"/> WCF Service application

#### Check Your Knowledge

Question
Given the following for loop statement, what is the value of the count variable once the loop has finished executing?
<pre>var count = 0; for (int i = 5; i &lt; 12; i++) {     count++; }</pre>
Select the correct answer.
<input type="checkbox"/> 3
<input type="checkbox"/> 5
<input type="checkbox"/> 7
<input type="checkbox"/> 9
<input type="checkbox"/> 11

# Module 2

## Creating Methods, Handling Exceptions, and Monitoring Applications

### Contents:

Module Overview	2-1
<b>Lesson 1:</b> Creating and Invoking Methods	2-2
<b>Lesson 2:</b> Creating Overloaded Methods and Using Optional and Output Parameters	2-8
<b>Lesson 3:</b> Handling Exceptions	2-13
<b>Lesson 4:</b> Monitoring Applications	2-18
<b>Lab:</b> Extending the Class Enrollment Application Functionality	2-24
Module Review and Takeaways	2-26

## Module Overview

Applications often consist of logical units of functionality that perform specific functions, such as providing access to data or triggering some logical processing. Visual C# is an object-orientated language and uses the concept of methods to encapsulate logical units of functionality. A method can be as simple or as complex as you like, and therefore it is important to consider what happens to the state of your application when an exception occurs in a method.

In this module, you will learn how to create and use methods and how to handle exceptions. You will also learn how to use logging and tracing to record the details of any exceptions that occur.

### Objectives

After completing this module, you will be able to:

- Create and invoke methods.
- Create overloaded methods and use optional parameters.
- Handle exceptions.
- Monitor applications by using logging, tracing, and profiling.

## Lesson 1

# Creating and Invoking Methods

Every application exists to execute some algorithm. Wikipedia describes an algorithm as “*an unambiguous specification of how to solve a class of problems*”. In simple terms, an algorithm is the description of every action necessary to perform some process. For example, the algorithm for clicking an icon could be described as:

1. Locate the icon.
2. Move the mouse cursor to that icon.
3. Double-click the left mouse button.

Notice that each of these actions can be split further. Every time small actions are combined into one cohesive action a new level of abstraction is added.

In object-oriented languages such as Visual C#, a method is a unit of code that performs a discrete piece of work. This allows you to create new levels of abstractions as you see fit and divide your solution into manageable logical components.

In this lesson, you will learn how to create and invoke methods.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of methods.
- Create methods.
- Invoke methods.
- Debug methods.

### What Is a Method?

The ability to define and call methods is a fundamental component of object-oriented programming, because methods enable you to encapsulate operations that protect data that is stored inside a type.

Typically, any application that you develop by using the Microsoft .NET Framework and Visual C# will have many methods, each with a specific purpose. Some methods are fundamental to the operation of an application. For example, all Visual C# desktop applications must have a method called **Main** that defines the entry point for the application. When the user runs a Visual C# application, the common language runtime (CLR) executes the **Main** method for that application.

- Methods encapsulate operations that protect data
- .NET Framework applications contain a **Main** entry point method
- The .NET Framework provides many methods in the base class library

Methods can be designed for internal use by a type, and as such are hidden from other types. Public methods may be designed to enable other types to request that an object performs an action, and are exposed outside of the type.

The .NET Framework itself is built from classes that expose methods that you can call from your applications to interact with the user and the computer.

## Creating Methods

A method comprises of two elements:

1. The method's specification.
2. The method's body.

The method specification defines the name of the method, the parameters that the method can take, the return type of the method, and the accessibility of the method. The combination of the name of the method and its parameter list are referred to as the method signature. The definition of the return value of a method is not regarded as part of the signature. Each method in a class must have a unique signature.

- Methods comprise two elements:
  - Method specification (return type, name, parameters)
  - Method body
- Use the **ref** keyword to pass parameter references

```
void StartService(int upTime, bool shutdownAutomatically)
{
 // Perform some processing here.
}
```
- Use the **return** keyword to return a value from the method

```
string GetServiceName()
{
 return "FourthCoffee.SalesService";
}
```

## Naming Methods

A method name has the same syntactic restrictions as a variable name. A method must start with a letter or an underscore and can only contain letters, underscores, and numeric characters. Visual C# is case sensitive, so a class can contain two methods that have the same name and differ only in the casing of one or more letters—although this is not a good coding practice.

The following guidelines are recommended best practices when you choose the name of a method:

- Use verbs or verb phrases to name methods. This helps other developers to understand the structure of your code.
- Use UpperCamelCase.

## Implementing a Method Body

The body of a method is a block of code that is implemented by using any of the available Visual C# programming constructs. The body is enclosed in braces.

You can define variables inside a method body, in which case they exist only while the method is running. When the method finishes, it is no longer in scope.

The following code example shows the body for the **StopService** method, which contains a variable named **isServiceRunning**. The **isServiceRunning** variable is only available inside the **StopService** code block. If you try to refer to the **isServiceRunning** variable outside the scope of the method, the compiler will raise a compile error with the message **The name 'isServiceRunning' does not exist in the current context.**

## Variable Method Scope

```
void StopService()
{
 var isServiceRunning = FourthCoffeeServices.Status;
 ...
}
```

## Specifying Parameters

Parameters are local variables that are created when the method runs and are populated with values that are specified when the method is called. All methods must have a list of parameters. You specify the parameters in parentheses following the method name. Each parameter is separated by a comma. If a method takes no parameters, you specify an empty parameter list.

For each parameter, you specify the type and the name. By convention, parameters are named by using lowerCamelCase.

The following code example shows a method that accepts an **int** parameter and a **Boolean** parameter.

### Passing Parameters to a Method

```
void StartService(int upTime, bool shutdownAutomatically)
{
 // Perform some processing here.
}
```

When defining the parameters that a method accepts, you can also prefix the parameter definition with the **ref** keyword. By using the **ref** keyword, you instruct the CLR to pass a reference to the parameter and not just the value of the parameter. You must initialize the **ref** parameter, and any changes to the parameter inside the method body will then be reflected in the underlying variable in the calling method.

The following code example shows how to define a parameter by using the **ref** keyword.

### Defining a Parameter by Using the **ref** Keyword

```
void StopAllServices(ref int serviceCount)
{
 serviceCount = FourthCoffeeServices.ActiveServiceCount;
}
```



**Additional Reading:** For more information about the **ref** keyword, see the [ref \(C# Reference\)](#) page at <http://go.microsoft.com/fwlink/?LinkId=267782>.

## Specifying a Return Type

All methods must have a return type. A method that does not return a value has the **void** return type. You specify the return type before the method name when you define a method. When you declare a method that returns data, you must include a **return** statement in the method block.

The following code example shows how to return a **string** from a method.

### Returning Data from a Method

```
string GetServiceName()
{
 return "FourthCoffee.SalesService";
}
```

The expression that the **return** statement specifies must have the same type as the method. When the return statement runs, this expression is evaluated and passed back to the statement that called the method. The method then finishes, so any other statements that occur after a **return** statement has been executed will not run.

However, each execution path in the method must call the **return** keyword eventually (or throw an exception). The compiler will produce an error if there's a possibility to run the method without reaching a **return** statement.

The following code example will generate a compiler error since it's possible to execute the method without returning a value.

#### This code will not compile

```
// Error CS0161 'GetServiceName()': not all code paths return a value
string GetServiceName(string language)
{
 If(language == "en")
 return "FourthCoffee.SalesService";
}
```

## Invoking Methods

You call a method to run the code in that method from part of your application. You do not need to understand how the code in a method works. You may not even have access to the code, if it is in a class in an assembly for which you do not have the source, such as the .NET Framework class library.

To call a method, you specify the method name and provide any arguments that correspond to the method parameters in brackets.

The following code example shows how to invoke the **StartService** method, passing **int** and **Boolean** variables to satisfy the parameter requirements of the method's signature.

To call a method specify:

- Method name
- Any arguments to satisfy parameters

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.
void StartService(int upTime, bool shutdownAutomatically)
{
 // Perform some processing here.
}
```

#### Invoking a Method Passing Parameters

```
var upTime = 2000;
var shutdownAutomatically = true;
StartService(upTime, shutdownAutomatically);

// StartService method.
void StartService(int upTime, bool shutdownAutomatically)
{
 // Perform some processing here.
}
```

If the method returns a value, you specify how to handle this value, typically by assigning it to a variable of the same type, in your calling code.

The following code example shows how to capture the return value of the **GetServiceName** method in a variable named **serviceName**.

#### Capturing a Method Return Value

```
var serviceName = GetServiceName();

string GetServiceName()
{
 return "FourthCoffee.SalesService";
}
```



**Additional Reading:** For more information about methods, see the Methods (C# Programming Guide) page at <http://go.microsoft.com/fwlink/?LinkId=267774>.

## Debugging Methods

When you are debugging your application, you can step through code one statement at a time. This is an extremely useful feature because it enables you to test the logic that your application uses one step at a time.

Visual Studio provides a number of debugging tools that enable you to step through code in exactly the way you want to. For example, you can step through each line in each method that is executed, or you can ignore the statements inside a method that you know are working correctly. You can also step over code completely, preventing some statements from executing.

- Visual Studio provides debug tools that enable you to step through code
- When debugging methods you can:
  - Step into the method
  - Step over the method
  - Step out of the method

When debugging methods, you can use the following three debug features to control whether you step over, step into, or step out of a method:

- The Step Into feature executes the statement at the current execution position. If the statement is a method call, the current execution position will move to the code inside the method. After you have stepped into a method you can continue executing statements inside the method, one line at a time. You can also use the Step Into button to start an application in debug mode. If you do this, the application will enter break mode as soon as it starts.
- The Step Over feature executes the statement at the current execution position. However, this feature does not step into code inside a method. Instead, the code inside the method executes and the executing position moves to the statement after the method call. The exception to this is if the code for the method or property contains a breakpoint. If this is the case, execution will continue up to the breakpoint. Using Step Over when the application is closed will also start it in the debug mode and break on the first line.
- The Step Out feature enables you to execute the remaining code in a method. Execution will continue to the statement that called the method, and then pause at that point.



**Additional Reading:** For more information about stepping through code, see the Tutorial: Learn to debug using Visual Studio page at <https://aka.ms/moc-20483c-m2-pg1>.

## Demonstration: Creating, Invoking, and Debugging Methods

In this demonstration, you will create a method, invoke the method, and then debug the method.

### Demonstration Steps

You will find the steps in the “**Demonstration: Creating, Invoking, and Debugging Methods**” section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD02\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_DEMO.md).

## Lesson 2

# Creating Overloaded Methods and Using Optional and Output Parameters

You have seen that you can define a method that accepts a fixed number of parameters. However, sometimes you might write one generic method that requires different sets of parameters depending on the context in which it is used. You can create overloaded methods with unique signatures to support this need. In other scenarios, you may want to define a method that has a fixed number of parameters, but enables an application to specify arguments for only the parameters that it needs. You can do this by defining a method that takes optional parameters and then using named arguments to satisfy the parameters by name.

In this lesson, you will learn how to create overloaded methods, define and use optional parameters, named arguments, and output parameters.

### Lesson Objectives

After completing this lesson, you will be able to:

- Create an overloaded method.
- Use optional parameters.
- Use named arguments.
- Define output parameters.

### Creating Overloaded Methods

When you define a method, you might realize that it requires different sets of information in different circumstances. You can define overloaded methods to create multiple methods with the same functionality that accept different parameters depending on the context in which they are called.

Overloaded methods have the same name as each other to emphasize their common intent. However, each overloaded method must have a unique signature, to differentiate it from the other overloaded versions of the method in the class.

The signature of a method includes its name and its parameter list. The return type is not part of the signature. Therefore, you cannot define overloaded methods that differ only in their return type.

The following code example shows three versions of the **StopService** method, all with a unique signature.

#### Overloaded Methods

```
void StopService()
{
 ...
}

void StopService(string serviceName)
{
```

• Overloaded methods share the same method name

• Overloaded methods have a unique signature

```
void StopService()
{
 ...
}

void StopService(string serviceName)
{
 ...
}

void StopService(int serviceId)
{
 ...
}
```

```

 ...
}

void StopService(int serviceId)
{
 ...
}

```

When you invoke the **StopService** method, you have choice of which overloaded version you use. You simply provide the relevant arguments to satisfy a particular overload, and then the compiler works out which version to invoke based on the arguments that you passed.

## Creating Methods that Use Optional Parameters

By defining overloaded methods, you can implement different versions of a method that take different parameters. When you build an application that uses overloaded methods, the compiler determines which specific instance of each method it should use to satisfy each method call.

There are other languages and technologies that developers can use for building applications and components that do not follow these rules. A key feature of Visual C# is the ability to interoperate with applications and components that are written by using other technologies. One of the principal technologies that Windows uses is the Component Object Model (COM). COM does not support overloaded methods, but instead uses methods that can take optional parameters. To make it easier to incorporate COM libraries and components into a Visual C# solution, Visual C# also supports optional parameters.

Optional parameters are also useful in other situations. They provide a compact and simple solution when it is not possible to use overloading because the types of the parameters do not vary sufficiently to enable the compiler to distinguish between implementations.

The following code example shows how to define a method that accepts one mandatory parameter and two optional parameters.

### Defining a Method with Optional Parameters

```

void StopService(bool forceStop, string serviceName = null, int serviceId = 1)
{
 ...
}

```

When defining a method that accepts optional parameters, you must specify all mandatory parameters before any optional parameters.

The following code example shows a method definition that uses optional parameters that throws a compile error.

### Incorrect Optional Parameter Definition

```

void StopService(string serviceName = null, bool forceStop, int serviceId = 1)
{
 ...
}

```

- Define all mandatory parameters first

```

void StopService(
 bool forceStop,
 string serviceName = null,
 int serviceId = 1)
{
 ...
}

```

- Satisfy parameters in sequence

```

var forceStop = true;
StopService(forceStop);

// OR

var forceStop = true;
var serviceName = "FourthCoffee.SalesService";
StopService(forceStop, serviceName);

```

{}

You can call a method that takes optional parameters in the same way that you call any other method. You specify the method name and provide any necessary arguments. The difference with methods that take optional parameters is that you can omit the corresponding arguments, and the method will use the default value when the method runs.

The following code example shows how to invoke the **StopService** method, passing only an argument for the **forceStop** mandatory parameter.

#### Invoking a Method Specifying Only Mandatory Arguments.

```
var forceStop = true;
StopService(forceStop);
```

The following code example shows how to invoke the **StopService** method, passing an argument for the **forceStop** mandatory parameter, and an argument for the **serviceName** parameter.

#### Invoking a Method Specifying Mandatory and Optional Arguments

```
var forceStop = true;
var serviceName = "FourthCoffee.SalesService";
StopService(forceStop, serviceName);
```

## Calling a Method by Using Named Arguments

Traditionally, when calling a method, the order and position of arguments in the method call corresponds to the order of parameters in the method signature. If the arguments are misaligned and the types mismatched, you receive a compile error.

In Visual C#, you can specify parameters by name, and therefore supply arguments in a sequence that differs from that defined in the method signature. To use named arguments, you supply the parameter name and corresponding value separated by a colon.

- Specify parameters by name
- Supply arguments in a sequence that differs from the method's signature
- Supply the parameter name and corresponding value separated by a colon

```
StopService(true, serviceID: 1);
```

The following code example shows how to invoke the **StopService** method by using named arguments to pass the **serviceID** parameter.

#### Using Named Arguments

```
StopService(true, serviceID: 1);
```

When using named arguments in conjunction with optional parameters, you can easily omit parameters. Any optional parameters will receive their default value. However, if you omit any mandatory parameters, your code will not compile.

You can mix positional and named arguments. However, you must specify all positional arguments before any named arguments.

 **Additional Reading:** For more information about using named arguments, see the Named and Optional Arguments (C# Programming Guide) page at <http://go.microsoft.com/fwlink/?LinkId=267784>.

## Creating Methods that Use Output Parameters

A method can pass a value back to the code that calls it by using a **return** statement. If you need to return more than a single value to the calling code, you can use output parameters to return additional data from the method. When you add an output parameter to a method, the method body is expected to assign a value to that parameter. When the method completes, the value of the output parameter is assigned to a variable that is specified as the corresponding argument in the method call.

To define an output parameter, you prefix the parameter in the method signature with the **out** keyword.

The following code example shows how to define a method that uses output parameters

### Defining Output Parameters

```
bool IsServiceOnline(string serviceName, out string statusMessage)
{
 var isOnline = FourthCoffeeServices.GetStatus(serviceName);

 if (isOnline)
 {
 statusMessage = "Services is currently running.";
 }
 else
 {
 statusMessage = "Services is currently stopped.";
 }

 return isOnline;
}
```

- Use the **out** keyword to define an output parameter

```
bool IsServiceOnline(string serviceName, out string statusMessage)
{
 ...
}
```

- Provide a variable for the corresponding argument when you call the method

```
var statusMessage = string.Empty;
var isServiceOnline = IsServiceOnline(
 "FourthCoffee.SalesService",
 out statusMessage);
```

A method can have as many output parameters as required. When you declare an output parameter, you must assign a value to the parameter before the method returns, otherwise the code will not compile.

To use an output parameter, you must provide a variable for the corresponding argument when you call the method, and prefix that argument with the **out** keyword. If you attempt to specify an argument that is not a variable or if you omit the **out** keyword, your code will not compile.

MCT USE ONLY STUDENT ICE PROHIBITED

The following code example shows how to invoke a method that accepts an output parameter.

#### Invoking a Method that Accepts an Output Parameter

```
var statusMessage = string.Empty;
var isServiceOnline = IsServiceOnline("FourthCoffee.SalesService", out statusMessage);
```

New versions of Visual C# let you define the variable in line with the **out** keyword. This allows the removal of the first line defining the empty variable. This also works with **ref** parameters.

The following code example shows how to invoke a method that accepts an output parameter, and define the variable in line

#### Invoking a Method that Accepts an Output Parameter and Define the Parameter in line

```
var isServiceOnline = IsServiceOnline("FourthCoffee.SalesService", out var
statusMessage);
```



**Additional Reading:** For more information about output parameters, see the **out** parameter modifier (C# Reference) page at <http://go.microsoft.com/fwlink/?LinkId=267785>.

## Lesson 3

# Handling Exceptions

Exception handling is an important concept to ensure a good user experience and to limit data loss. Applications should be designed with exception handling in mind.

In this lesson, you will learn how to implement effective exception handling in your applications and how you can use exceptions in your methods to elegantly indicate an error condition to the code that calls your methods.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of an exception.
- Handle exceptions by using a try/catch block.
- Use a finally block to run code after an exception.
- Throw an exception.

### What Is an Exception?

Many things can go wrong as an application runs. Some errors may occur due to flaws in the application logic, but others may be due to conditions outside the control of your application. For example, your application cannot guarantee that a file exists on the file system or that a required database is online. When you design an application, you should consider how to ensure that your application can recover gracefully when such problems arise. It is common practice to simply check the return values from methods to ensure that they have executed correctly,

however, this methodology is not always sufficient to handle all errors that may occur because:

- An exception is an indication of an error or exceptional condition
- The .NET Framework provides many exception classes:
  - **Exception**
  - **SystemException**
  - **ApplicationException**
  - **NullReferenceException**
  - **FileNotFoundException**
  - **SerializationException**

- Not all methods return a value.
- You need to know why the method call has failed, not just that it has failed.
- Unexpected errors such as running out of memory cannot be handled in this way.

Traditionally, applications used the concept of a global error object. When a piece of code caused an error, it would set the data in this object to indicate the cause of the error and then return to the caller. It was the responsibility of the calling code to examine the error object and determine how to handle it. However, this approach is not robust, because it is too easy for a programmer to forget to handle errors appropriately.

### How Exceptions Propagate

The .NET Framework uses exceptions to help overcome these issues. An exception is an indication of an error or exceptional condition. A method can throw an exception when it detects that something unexpected has happened, for example, the application tries to open a file that does not exist.

MCT USE ONLY. STUDENT USE PROHIBITED

When a method throws an exception, the calling code must be prepared to detect and handle this exception. If the calling code does not detect the exception, the code is aborted and the exception is automatically propagated to the code that invoked the calling code. This process continues until a section of code takes responsibility for handling the exception. Execution continues in this section of code after the exception-handling logic has completed. If no code handles the exception, then the process will terminate and display a message to the user.

### The Exception Type

When an exception occurs, it is useful to include information about the original cause so that the method that handles the exception can take the appropriate corrective action. In the .NET Framework, exceptions are based on the **Exception** class, which contains information about the exception. When a method throws an exception, it creates an **Exception** object and can populate it with information about the cause of the error. The **Exception** object is then passed to the code that handles the exception.

The following table describes some of the exception classes provided by the .NET Framework.

Exception Class	Namespace	Description
<b>Exception</b>	<b>System</b>	Represents any exception that is raised during the execution of an application.
<b>SystemException</b>	<b>System</b>	Represents all exceptions raised by the CLR. The <b>SystemException</b> class is the base class for all the exception classes in the <b>System</b> namespace.
<b>ApplicationException</b>	<b>System</b>	Represents all non-fatal exceptions raised by applications and not the CLR.
<b>NullReferenceException</b>	<b>System</b>	Represents an exception that is caused when trying to use an object that is null.
<b>FileNotFoundException</b>	<b>System.IO</b>	Represents an exception caused when a file does not exist.
<b>SerializationException</b>	<b>System.Runtime.Serialization</b>	Represents an exception that occurs during the serialization or deserialization process.



**Additional Reading:** For more information about the **Exception** class, see the **Exception Class** page at <https://aka.ms/moc-20483c-m2-pg2>.

## Handling Exception by Using a Try/Catch Block

The try/catch block is the key programming construct that enables you to implement Structured Exception Handling (SEH) in your applications. You wrap code that may fail and cause an exception in a try block, and add one or more catch blocks to handle any exceptions that may occur.

The following code example shows the syntax for defining a try/catch block.

- Use try/catch blocks to handle exceptions
- Use one or more catch blocks to catch different types of exceptions

```
try
{
}
catch (NullReferenceException ex)
{
 // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
 // Catch all other exceptions.
}
```

### Try/Catch Syntax

```
try
{
 // Try block.
}
catch ([catch specification 1])
{
 // Catch block 1.
}
catch ([catch specification n])
{
 // Catch block n.
}
```

The statements that are enclosed in the braces in the try block can be any Visual C# statements, and can invoke methods in other objects. If any of these statements cause an exception to be thrown, execution passes to the appropriate catch block. The catch specification for each block determines which exceptions it will catch and the variable, if any, in which to store the exception. You can specify catch blocks for different types of exceptions. It is good practice to include a catch block for the general **Exception** type at the end of the catch blocks to catch all exceptions that have not been handled otherwise.

In the following code example, if the code in the try block causes a **NullReferenceException** exception, the code in the corresponding catch block runs. If any other type of exception occurs, the code in the catch block for the **Exception** type runs.

### Handling NullReferenceException and Exception exceptions

```
try
{
}
catch (NullReferenceException ex)
{
 // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
 // Catch all other exceptions.
}
```

When defining more than one catch block, you must ensure that you place them in the correct order. When an exception is thrown, the CLR attempts to match the exception against each catch block in turn. You must put more specific catch blocks before less specific catch blocks, otherwise your code will not compile.



**Additional Reading:** For more information about try/catch blocks, see the try-catch (C# Reference) page at <https://aka.ms/moc-20483c-m2-pg3>.

## Using a Finally Block

Some methods may contain critical code that must always be run, even if an unhandled exception occurs. For example, a method may need to ensure that it closes a file that it was writing to or releases some other resources before it terminates. A finally block enables you to handle this situation.

You specify a finally block after any catch handlers in a try/catch block. It specifies code that must be performed when the block finishes, irrespective of whether any exceptions, handled or unhandled, occur. If an exception is caught and handled, the exception handler in the catch block will run before the finally block.

You can also add a finally block to code that has no catch blocks. In this case, all exceptions are unhandled, but the finally block will always run.

The following code example shows how to implement a try/catch/finally block.

### Try/Catch/Finally Blocks

```
try
{
}
catch (NullReferenceException ex)
{
 // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
 // Catch all other exceptions.
}
finally
{
 // Code that always runs.
}
```

- Use a finally block to run code whether or not an exception has occurred

```
try
{
}
catch (NullReferenceException ex)
{
 // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
 // Catch all other exceptions.
}
finally
{
 // Code that always runs.
}
```



**Additional Reading:** For more information about try/catch/finally blocks, see the try-catch-finally (C# Reference) page at <https://aka.ms/moc-20483c-m2-pg4>.

## Throwing Exceptions

You can create an instance of an exception class in your code and throw the exception to indicate that an exception has occurred. When you throw an exception, execution of the current block of code terminates and the CLR passes control to the first available exception handler that catches the exception.

To throw an exception, you use the **throw** keyword and specify the exception object to throw.

The following code example shows how to create an instance of the **NullReferenceException** class and then throw the **ex** object.

- Use the **throw** keyword to throw a new exception

```
var ex =
 new NullReferenceException("The 'Name' parameter is null.");
throw ex;
```

- Use the **throw** keyword to rethrow an existing exception

```
try
{
}
catch (NullReferenceException ex)
{
}
catch (Exception ex)
{
}
...
throw;
}
```

### Creating and Throwing an Exception

```
var ex = new NullReferenceException("The 'Name' parameter is null.");
throw ex;
```

A common strategy is for a method or block of code to catch any exceptions and attempt to handle them. If the catch block for an exception cannot resolve the error, it can rethrow the exception to propagate it to the caller.

The following code example shows how to rethrow an exception that has been caught in a catch block.

### Rethrowing an Exception

```
try
{
}
catch (NullReferenceException ex)
{
 // Catch all NullReferenceException exceptions.
}
catch (Exception ex)
{
 // Attempt to handle the exception
 ...
 // If this catch handler cannot resolve the exception,
 // throw it to the calling code
 throw;
}
```

## Lesson 4

# Monitoring Applications

When you develop real-world applications, writing code is just one part of the process. You are likely to spend a significant amount of time resolving bugs, troubleshooting problems, and optimizing the performance of your code. Visual Studio and the .NET Framework provide various tools that can help you to perform these tasks more effectively.

In this lesson, you will learn how to use a range of tools and techniques to monitor and troubleshoot your applications.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use logging and tracing in your code.
- Use application profiling in Visual Studio.
- Use performance counters to monitor the performance of your application.

### Using Logging and Tracing

Logging and tracing are similar, but distinct, concepts. When you implement logging in your application, you add code that writes information to a destination log, such as a text file or the Windows event log. Logging enables you to provide users and administrators with more information about what your code is doing. For example, if your application handles an exception, you might write the details to the Windows event log to enable the user or a system administrator to resolve any underlying problems.

- *Logging* provides information to users and administrators
  - Windows event log
  - Text files
  - Custom logging destinations
- *Tracing* provides information to developers
  - Visual Studio Output window
  - Custom tracing destinations

By contrast, developers use tracing to monitor the execution of an application. When you implement tracing, you add code that writes messages to a trace listener, which in turn directs your output to a specified target. By default, your trace messages are shown in the **Output** window in Visual Studio. You typically use tracing to provide information about variable values or condition results, to help you find out why your application behaves in a particular way. You can also use tracing techniques to interrupt the execution of an application in response to conditions that you define.

### Writing to the Windows Event Log

Writing to the Windows event log is one of the more common logging requirements you might encounter. The **System.Diagnostics.EventLog** class provides various static methods that you can use to write to the Windows event log. In particular, the **EventLog.WriteEntry** method includes several overloads that you can use to log various combinations of information. To write to the Windows event log, you need to provide a minimum of three pieces of information:

- The *event log*. This is the name of the Windows event log you want to write to. In most cases you will write to the **Application** log.
- The *event source*. This identifies where the event came from, and is typically the name of your application. When you create an event source, you associate it with an event log.

- The *message*. This is the text that you want to add to the log.

You can also use the **WriteEntry** method to specify a category, an event ID, and an event severity if required.

 **Additional Reading:** Writing to the Windows event log requires a high level of permissions. If your application does not run with sufficient permissions, it will throw a **SecurityException** when you attempt to create an event source or write to the event log.

The following example shows how to write a message to the event log:

### Writing to the Windows Event Log

```
string eventLog = "Application";
string eventSource = "Logging Demo";
string eventMessage = "Hello from the Logging Demo application";

// Create the event source if it does not already exist.
If (!EventLog.SourceExists(eventSource))
 EventLog.CreateEventSource(eventSource, eventLog);

// Log the message.
EventLog.WriteEntry(eventSource, eventMessage);
```

 **Additional Reading:** For more information on writing to the Windows event log, see the How to write to an event log by using Visual C# page at <https://aka.ms/moc-20483c-m2-pg6>.

### Debugging and Tracing

The **System.Diagnostics** namespace includes two classes, **Debug** and **Trace**, which you can use to monitor the execution of your application. These two classes work in a similar way and include many of the same methods. However, **Debug** statements are only active if you build your solution in **Debug** mode, whereas **Trace** statements are active in both **Debug** and **Release** mode builds.

The **Debug** and **Trace** classes include methods to write format strings to the **Output** window in Visual Studio, as well as to any other listeners that you configure. You can also write to the **Output** window only when certain conditions are met, and you can adjust the indentation of your trace messages. For example, if you are writing details of every object within an enumeration to the **Output** window, you might want to indent these details to distinguish them from other output.

The **Debug** and **Trace** classes also include a method named **Assert**. The **Assert** method enables you to specify a condition (an expression that must evaluate to **true** or **false**) together with a format string. If the condition evaluates to **false**, the **Assert** method interrupts the execution of the program and displays a dialog box with the message you specify. This method is useful if you need to identify the point in a long-running program at which an unexpected condition arises.

The following example shows how to use the **Debug** class to write messages to the **Output** window, and to interrupt execution if unexpected conditions arise.

### Using the Debug Class

```
int number;
Console.WriteLine("Please type a number between 1 and 10, and then press Enter");
string userInput = Console.ReadLine();

Debug.Assert(int.TryParse(userInput, out number),
 string.Format("Unable to parse {0} as integer", userInput);

Debug.WriteLine(The current value of userInput is: {0}, userInput);
Debug.WriteLine(The current value of number is: {0}", number);

Console.WriteLine("Press Enter to finish");
Console.ReadLine();
```

 **Additional Reading:** For more information on tracing, see the How to trace and debug in Visual C# page at <http://go.microsoft.com/fwlink/?LinkId=267790>.

## Using Application Profiling

When you develop applications, making your code work without bugs is only part of the challenge. You also have to ensure that your code runs efficiently. You need to review how long your code takes to accomplish tasks and whether it uses excessive processor, memory, disk, or network resources.

Visual Studio includes a range of tools, collectively known as the Visual Studio Profiling Tools, that can help you to analyze the performance of your applications. At a high level, running a performance analysis in Visual Studio consists of three high-level steps:

- Create and run a *performance session*
- Analyze the *profiling report*
- Revise your code and repeat

1. Create and run a *performance session*. All performance analysis takes place within a performance session. You can create and run a performance session by launching the Performance Wizard from the **Analyze** menu in Visual Studio. When the performance session is running, you run your application as you usually would. While your application is running, you typically aim to use functionality that you suspect may be causing performance issues.
2. Analyze the *profiling report*. When you finish running your application, Visual Studio displays the profiling report. This includes a range of information that can provide insights into the performance of your application. For example, you can:
  - See which functions consume the most CPU time.
  - View a timeline that shows what your application was doing when.
  - View warnings and suggestions on how to improve your code.
3. Revise your code and repeat the analysis. When your analysis is complete, you should make changes to your code to fix any issues that you identified. You can then run a new performance session and

generate a new profiling report. The Visual Studio Profiling Tools enable you to compare two reports to help you identify and quantify how the performance of your code has changed.

Performance sessions work by sampling. When you create a performance session, you can choose whether you want to sample CPU use, .NET memory allocation, concurrency information for multi-threaded applications, or whether you want to use instrumentation to collect detailed timing information about every function call. In most cases you will want to start by using CPU sampling, which is the default option. CPU sampling uses statistical polling to determine which functions are using the most CPU time. This provides an insight into the performance of your application, without consuming many resources and slowing down your application.

 **Additional Reading:** For more information on application profiling, see the Analyzing Application Performance by Using Profiling Tools page at <https://aka.ms/moc-20483c-m2-pg5>.

## Using Performance Counters

Performance counters are system tools that collect information on how resources are used. Viewing performance counters can provide additional insights into what your application is doing, and can help you to troubleshoot performance problems. Performance counters fall into three main groups:

- *Counters that are provided by the operating system and the underlying hardware platform.* This group includes counters that you can use to measure processor use, physical memory use, disk use, and network use. The details of the counters available will vary according to the hardware that the computer contains.
- *Counters that are provided by the .NET Framework.* The .NET Framework includes counters that you can use to measure a wide range of application characteristics. For example, you can look at the number of exceptions thrown, view details of locks and thread use, and examine the behavior of the garbage collector.
- *Counters that you create yourself.* You can create your own performance counters to examine specific aspects of the behavior of your application. For example, you can create a performance counter to count the number of calls to a particular method or to count the number of times a specific exception is thrown.

- Create performance counters and categories in code or in Server Explorer
- Specify:
  - A name
  - Some help text
  - The base performance counter type
- Update custom performance counters in code
- View performance counters in Performance Monitor (perfmon.exe)

## Browsing and Using Performance Counters

Performance counters are organized into *categories*. This helps you to find the counters you want when you are capturing and reviewing performance data. For example, the **PhysicalDisk** category typically includes counters for the percentage of time spent reading and writing to disk, amounts of data read from and written to disk, and the queue lengths to read data from and write data to disk.

 **Note:** You can browse the performance counters available on your computer from Visual Studio. In Server Explorer, expand **Servers**, expand the name of your computer, and then expand **Performance Counters**.

Typically, you capture and view data from performance counters in Performance Monitor (perfmon.exe). Performance Monitor is included in the Windows operating system and enables you to view or capture data from performance counters in real time. When you use Performance Monitor, you can browse performance counter categories and add multiple performance counters to a graphical display. You can also create *data collector sets* to capture data for reporting or analysis.

## Creating Custom Performance Counters

You can use the **PerformanceCounter** and **PerformanceCounterCategory** classes to interact with performance counters in a variety of ways. For example, you can:

- Iterate over the performance counter categories available on a specified computer.
- Iterate over the performance counters within a specified category.
- Check whether specific performance counter categories or performance counters exist on the local computer.
- Create custom performance counter categories or performance counters.

You typically create custom performance counter categories and performance counters during an installation routine, rather than during the execution of your application. After a custom performance counter is created on a specific computer, it remains there. You do not need to recreate it every time you run your application. To create a custom performance counter, you must specify a base counter type by using the **PerformanceCounterType** enumeration.

The following example shows how to programmatically create a custom performance counter category. This example creates a new performance counter category named **FourthCoffeeOrders**. The category contains two performance counters. The first performance counter tracks the total number of coffee orders placed, and the second tracks the number of orders placed per second.

### Programmatically Creating Performance Counter Categories and Performance Counters

```
if (!PerformanceCounterCategory.Exists("FourthCoffeeOrders"))
{
 CounterCreationDataCollection counters = new CounterCreationDataCollection();

 CounterCreationData totalOrders = new CounterCreationData();
 totalOrders.CounterName = "# Orders";
 totalOrders.CounterHelp = "Total number of orders placed";
 totalOrders.CounterType = PerformanceCounterType.NumberOfItems32;
 counters.Add(totalOrders);

 CounterCreationData ordersPerSecond = new CounterCreationData();
 ordersPerSecond.CounterName = "# Orders/Sec";
 ordersPerSecond.CounterHelp = "Number of orders placed per second";
 ordersPerSecond.CounterType = PerformanceCounterType.RateOfCountsPerSecond32;
 counters.Add(ordersPerSecond);

 PerformanceCounterCategory.Create("FourthCoffeeOrders", "A custom category for
demonstration",
 PerformanceCounterCategoryType.SingleInstance, counters);
}
```

 **Note:** You can also create performance counter categories and performance counters from Server Explorer in Visual Studio.

When you have created custom performance counters, your application must provide the performance counters with data. Performance counters provide various methods that enable you to update the counter

value, such as the **Increment** and **Decrement** methods. How the counter processes the value will depend on the base type you selected when you created the counter.

The following example shows how to programmatically update custom performance counters.

### Using Custom Performance Counters

```
// Get a reference to the custom performance counters.
PerformanceCounter counterOrders = new PerformanceCounter("FourthCoffeeOrders", "#
Orders", false);
PerformanceCounter counterOrdersPerSec = new PerformanceCounter("FourthCoffeeOrders", "#
Orders/Sec", false);

// Update the performance counter values at appropriate points in your code.
public void OrderCoffee()
{
 counterOrders.Increment();
 counterOrdersPerSec.Increment();

 // Coffee ordering logic goes here.
}
```

When you have created a custom performance counter category, you can browse to your category and select individual performance counters in Performance Monitor. When you run your application, you can then use Performance Monitor to view data from your custom performance counters in real time.

## Demonstration: Extending the Class Enrollment Application Functionality Lab

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: Extending the Class Enrollment Application Functionality Lab** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD02\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_DEMO.md).

# Lab: Extending the Class Enrollment Application Functionality

## Scenario

You have been asked to refactor the code that you wrote in the lab exercises for module 1 into separate methods to avoid the duplication of code in the Class Enrollment Application.

Also, you have been asked to write code that validates the student information that the user enters and to enable the updated student information to be written back to the database, handling any errors that may occur.

## Objectives

After completing this lab, you will be able to:

- Refactor code to facilitate reusability.
- Write Visual C# code that validates data entered by a user.
- Write Visual C# code that saves changes back to a database.

## Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD02\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD02\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_LAK.md).

## Exercise 1: Refactoring the Enrollment Code

### Scenario

In this exercise, you will refactor the existing code to avoid writing duplicate code.

The application currently enables a user to edit a student's details by pressing Enter, but you now want them to also be able to initiate the edit process by double-clicking on a student in the list. You will begin by creating a new method that contains the code for editing a student's details. This will avoid duplicating and maintaining the code in both event handlers. You will then call the new method from both the **studentsList\_MouseDoubleClick** and **StudentsList\_KeyDown** events. While doing this, you also decide to refactor the code for adding and deleting students into separate methods, so that it can be called from other parts of the application if the need arises. You will then run the application and verify that users can press Enter or double-click on a student to edit the student's details, can press Insert to add a new student, and can press Delete to remove a student.

## Exercise 2: Validating Student Information

### Scenario

In this exercise, you will write code that validates the information that a user enters for a student.

Up until this point, almost anything can be entered as student data, and fields can be left blank. This means, for example, that a student could be added to the student list with no last name or with an invalid date of birth.

You will write code to check that when adding or editing a student, the first name and last name fields for the student contain data. You will also write code to check that the date of birth entered is a valid date

and that the student is at least five years old. Finally, you will run the application and test your validation code.

## Exercise 3: Saving Changes to the Class List

### Scenario

In this exercise, you will write code that saves changes in the student list to the database.

Every time the user closes and opens the application, they are presented with the original student list as it existed when they first ran the application, regardless of any changes they may have made. You will write code to save changes back to the database when the user clicks the **Save Changes** button. You will then add exception handling code to catch concurrency, update, and general exceptions, and handle the exceptions gracefully. Finally, you will run your application and verify that changes you make to student data are persisted between application sessions.

## Module Review and Takeaways

In this module, you learned how to create and use methods, and how to handle exceptions. You also learned how to use logging and tracing to record the details of any exceptions that occur.

### Review Questions

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

Statement	Answer
The return type of a method forms part of a methods signature.	

### Check Your Knowledge

Question
When using output parameters in a method signature, which one of the following statements is true?
Select the correct answer.
<input type="checkbox"/> You cannot return data by using a return statement in a method that use output parameters.
<input type="checkbox"/> You can only use the type object when defining an output parameter.
<input type="checkbox"/> You must assign a value to an output parameter before the method returns.
<input type="checkbox"/> You define an output parameter by using the output keyword.

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

Statement	Answer
A finally block enables you to run code in the event of an error occurring?	

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

Statement	Answer
<b>Trace</b> statements are active in both <b>Debug</b> and <b>Release</b> mode builds.	

# Module 3

## Basic Types and Constructs of Visual C#

### Contents:

Module Overview	3-1
Lesson 1: Implementing Structs and Enums	3-2
Lesson 2: Organizing Data into Collections	3-9
Lesson 3: Handling Events	3-17
Lab: Writing the Code for the Grades Prototype Application	3-22
Module Review and Takeaways	3-24

## Module Overview

To create effective applications by using Windows Presentation Foundation (WPF) or other .NET Framework platforms, you must first learn some basic Visual C# constructs. You need to know how to create simple structures to represent the data items you are working with. You need to know how to organize these structures into collections, so that you can add items, retrieve items, and iterate over your items. Finally, you need to know how to subscribe to events so that you can respond to the actions of your users.

In this module, you will learn how to create and use structs and enums, organize data into collections, and create and subscribe to events.

### Objectives

After completing this module, you will be able to:

- Create and use structs and enums.
- Use collection classes to organize data.
- Create and subscribe to events.

## Lesson 1

# Implementing Structs and Enums

The .NET Framework includes various built-in data types, such as **Int32**, **Decimal**, **String**, and **Boolean**.

However, suppose you want to create an object that represented a coffee. Which type would you use?

You might use built-in types to represent the properties of a coffee, such as the country of origin (a string) or the strength of the coffee (an integer). However, you need a way to represent coffee as a discrete entity, so that you can perform actions such as add a coffee to a collection or compare one coffee to another.

In this lesson, you will learn how to use structs and enums to create your own simple types.

### Lesson Objectives

After completing this lesson, you will be able to:

- Create and use enums.
- Create and use structs.
- Define constructors to instantiate structs.
- Create properties to get and set field values in a struct.
- Create indexers to expose struct members by using an integer index.

### Creating and Using Enums

An enumeration type, or *enum*, is a structure that enables you to create a variable with a fixed set of possible values. The most common example is to use an enum to define the day of the week. There are only seven possible values for days of the week, and you can be reasonably certain that these values will never change.

The following example shows how to create an enum:

- Create variables with a fixed set of possible values

```
enum Day { Sunday, Monday, Tuesday, Wednesday, ... };
```

- Set instance to the member you want to use

```
Day favoriteDay = Day.Friday;
```

- Set enum variables by name or by value

```
Day day1 = Day.Friday;
// is equivalent to
Day day1 = (Day)4;
```

#### Declaring an Enum

```
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
```

To use the enum, you create an instance of your enum variable and specify which enum member you want to use.

The following example shows how to use an enum:

#### Using an Enum

```
Day favoriteDay = Day.Friday;
```

Using enums has several advantages over using text or numerical types:

- *Improved manageability.* By constraining a variable to a fixed set of valid values, you are less likely to experience invalid arguments and spelling mistakes.

- *Improved developer experience.* In Visual Studio, the IntelliSense feature will prompt you with the available values when you use an enum.
- *Improved code readability.* The enum syntax makes your code easier to read and understand.

Each member of an enum has a *name* and a *value*. The name is the string you define in the braces, such as Sunday or Monday. By default, the value is an integer. If you do not specify a value for each member, the members are assigned incremental values starting with 0. For example, **Day.Sunday** is equal to 0 and **Day.Monday** is equal to 1.

The following example shows how you can use names and values interchangeably:

### Using Enum Names and Values Interchangeably

```
// Set an enum variable by name.
Day favoriteDay = Day.Friday;

// Set an enum variable by value.
Day favoriteDay = (Day)4;
```



**Reference Links:** For more information about enums, see the Enumeration Types (C# Programming Guide) page at <https://aka.ms/moc-20483c-m3-pg1>.

## Creating and Using Structs

In Visual C#, a *struct* is a programming construct that you can use to define custom types. Structs are essentially lightweight data structures that represent related pieces of information as a single item. For example:

- A struct named **Point** might consist of fields to represent an x-coordinate and a y-coordinate.
- A struct named **Circle** might consist of fields to represent an x-coordinate, a y-coordinate, and a radius.
- A struct named **Color** might consist of fields to represent a red component, a green component, and a blue component.

• Use structs to create simple custom types:  
 • Represent related data items as a single logical entity  
 • Add fields, properties, methods, and events

• Use the **struct** keyword to create a struct  
`public struct Coffee { ... }`

• Use the **new** keyword to instantiate a struct  
`Coffee coffee1 = new Coffee();`

Most of the built-in types in Visual C#, such as **int**, **bool**, and **char**, are defined by structs. You can use structs to create your own types that behave like built-in types.

### Creating a Struct

You use the **struct** keyword to declare a struct, as shown by the following example:

### Declaring a Struct

```
public struct Coffee
{
 public int Strength;
 public string Bean;
 public string CountryOfOrigin;

 // Other methods, fields, properties, and events.
}
```

The **struct** keyword is preceded by an *access modifier*—**public** in the above example—that specifies where you can use the type. You can use the following access modifiers in your struct declarations:

Access Modifier	Details
<b>public</b>	The type is available to code running in any assembly.
<b>internal</b>	The type is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier.
<b>private</b>	The type is only available to code within the struct that contains it. You can only use the private access modifier with nested structs.

Structs can contain a variety of members, including fields, properties, methods, and events.

### Using a Struct

To create an instance of a **struct**, you use the **new** keyword, as shown by the following example:

#### Instantiating a Struct

```
Coffee coffee1 = new Coffee();
coffee1.Strength = 3;
coffee1.Bean = "Arabica";
coffee1.CountryOfOrigin = "Kenya";
```

## Initializing Structs

You might have noticed that the syntax for instantiating a struct, for example, **new Coffee()**, is similar to the syntax for calling a method. This is because when you instantiate a struct, you are actually calling a special type of method called a *constructor*. A constructor is a method in the struct that has the same name as the struct.

When you instantiate a struct with no arguments, such as **new Coffee()**, you are calling the *default constructor* which is created by the Visual C# compiler. If you want to be able to specify default field values when you instantiate a struct, you can add constructors that accept parameters to your struct.

The following example shows how to create a constructor in a struct:

#### Adding a Constructor

```
public struct Coffee
{
 // This is the custom constructor.
 public Coffee(int strength, string bean, string countryOfOrigin)
 {
 this.Strength = strength;
 this.Bean = bean;
 this.CountryOfOrigin = countryOfOrigin;
 }

 // These statements declare the struct fields and set the default values.
 public int Strength;
```

- Use constructors to initialize a struct

```
public struct Coffee
{
 public Coffee(int strength, string bean, string origin)
 { ... }
```

- Provide arguments when you instantiate the struct

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

- Add multiple constructors with different combinations of parameters

```

 public string Bean;
 public string CountryOfOrigin;

 // Other methods, fields, properties, and events.
}

```

The following example shows how to use this constructor to instantiate a **Coffee** item:

### Calling a Constructor

```
// Call the custom constructor by providing arguments for the three required parameters.
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
```

You can add multiple constructors to your struct, with each constructor accepting a different combination of parameters. However, you cannot add a default constructor to a struct because it is created by the compiler.

## Creating Properties

In Visual C#, a *property* is a programming construct that enables client code to get or set the value of private fields within a struct or a class. To consumers of your struct or class, the property behaves like a public field. Within your struct or class, the property is implemented by using *accessors*, which are a special type of method. A property can include one or both of the following:

- A **get** accessor to provide read access to a field.
- A **set** accessor to provide write access to a field.

- Properties use get and set accessors to control access to private fields

```
private int strength;
public int Strength
{
 get { return strength; }
 set { strength = value; }
}
```

- Properties enable you to:

- Control access to private fields
- Change accessor implementations without affecting clients
- Data-bind controls to property values

The following example shows how to implement a property in a struct:

### Implementing a Property

```

public struct Coffee
{
 private int strength;
 public int Strength
 {
 get { return strength; }
 set { strength = value; }
 }
}

```

Within the property, the **get** and **set** accessors use the following syntax:

- The **get** accessor uses the **return** keyword to return the value of the private field to the caller.
- The **set** accessor uses a special local variable named **value** to set the value of the private field. The **value** variable contains the value provided by the client code when it accessed the property.

The following example shows how to use a property:

### Using a Property

```
Coffee coffee1 = new Coffee();

// The following code invokes the set accessor.
coffee1.Strength = 3;

// The following code invokes the get accessor.
int coffeeStrength = coffee1.Strength;
```

The client code uses the property as if it was a public field. However, using public properties to expose private fields offers the following advantages over using public fields directly:

- You can use properties to control external access to your fields. A property that includes only a get accessor is read-only, while a property that includes only a set accessor is write-only.

```
// This is a read-only property.
public int Strength
{
 get { return strength; }
}
// This is a write-only property.
public string Bean
{
 set { bean = value; }
}
```

- You can change the implementation of properties without affecting client code. For example, you can add validation logic, or call a method instead of reading a field value.

```
public int Strength
{
 get { return strength; }
 set
 {
 if(value < 1)
 { strength = 1; }
 else if(value > 5)
 { strength = 5; }
 else
 { strength = value; }
 }
}
```

- You can also create a **const** property by simply returning a literal value.

```
public string Hello { get { return "world"; } }
```

- Properties are required for data binding in WPF. For example, you can bind controls to property values, but you cannot bind controls to field values.

When you want to create a property that simply gets and sets the value of a private field without performing any additional logic, you can use an abbreviated syntax.

- To create a property that reads and writes to a private field, you can use the following syntax:

```
public int Strength { get; set; }
```

- To create a property that can be publicly read, but set only by its containing class, you can use the following syntax:

```
public int Strength { get; private set; }
```

- To create a **readonly** property where only the constructor can set the value to this property, you can use the following syntax:

```
public int Strength { get; }
```

- To create a property that writes to a private field, you can use the following syntax:

```
public int Strength { private get; set; }
```

In each case, the compiler will implicitly create a private field and map it to your property. These are known as *auto-implemented properties*. You can change the implementation of your property at any time.

 **Additional Reading:** In addition to controlling access to a property by omitting **get** or **set** accessors, you can also restrict access by applying access modifiers (such as **private** or **protected**) to your accessors. For example, you might create a property with a public **get** accessor and a protected **set** accessor. For more information, see the Restricting Accessor Accessibility (C# Programming Guide) page at <https://aka.ms/moc-20483c-m3-pg2>.

## Creating Indexers

In some scenarios, you might want to use a struct or a class as a container for an array of values. For example, you might create a struct to represent the beverages available at a coffee shop. The struct might use an array of strings to store the list of beverages.

The following example shows a struct that includes an array:

- Use the **this** keyword to declare an indexer
- Use **get** and **set** accessors to provide access to the collection

```
public int this[int index]
{
 get { return this.beverages[index]; }
 set { this.beverages[index] = value; }
}
```

- Use the instance name to interact with the indexer

```
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
```

### Creating a Struct that Includes an Array

```
public struct Menu
{
 public string[] beverages;

 public Menu(string bev1, string bev2)
 {
 beverages = new string[] { "Americano", "Café au Lait", "Café Macchiato",
 "Cappuccino", "Espresso" };
 }
}
```

When you expose the array as a public field, you would use the following syntax to retrieve beverages from the list:

### Accessing Array Items Directly

```
Menu myMenu = new Menu();
string firstDrink = myMenu.beverages[0];
```

A more intuitive approach would be if you could access the first item from the menu by using the syntax **myMenu[0]**. You can do this by creating an *indexer*. An indexer is similar to a property, in that it uses get

and set accessors to control access to a field. More importantly, an indexer enables you to access collection members directly from the name of the containing struct or class by providing an integer index value. To declare an indexer, you use the **this** keyword, which indicates that the property will be accessed by using the name of the struct instance.

The following example shows how to define an indexer for a struct:

### Creating an Indexer

```
public struct Menu
{
 private string[] beverages;

 // This is the indexer.
 public string this[int index]
 {
 get { return this.beverages[index]; }
 set { this.beverages[index] = value; }
 }

 // Enable client code to determine the size of the collection.
 public int Length
 {
 get { return beverages.Length; }
 }
}
```

When you use an indexer to expose the array, you use the following syntax to retrieve the beverages from the list:

### Accessing Array Items by Using an Indexer

```
Menu myMenu = new Menu();
string firstDrink = myMenu[0];
int numberofChoices = myMenu.Length;
```

Just like a property, you can customize the **get** and **set** accessors in an indexer without affecting client code. You can create a read-only indexer by including only a **get** accessor, and you can create a write-only indexer by including only a **set** accessor.



**Reference Links:** For more information about indexers, see the Using Indexers (C# Programming Guide) page at <https://aka.ms/moc-20483c-m3-pg3>.

## Demonstration: Creating and Using a Struct

In this demonstration, you will create a struct named **Coffee** and add several properties to the struct. You will then create an instance of the **Coffee** struct, set some property values, and display these property values in a console window.

### Demonstration Steps

You will find the steps in the **Demonstration: Creating and Using a Struct** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD03\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md).

## Lesson 2

# Organizing Data into Collections

When you create multiple items of the same type, regardless of whether they are integers, strings, or a custom type such as **Coffee**, you need a way of managing the items as a set. You need to be able to count the number of items in the set, add items to or remove items from the set, and iterate through the set one item at a time. To do this, you can use a *collection*.

Collections are an essential tool for managing multiple items. They are also central to developing graphical applications. Controls such as drop-down list boxes and menus are typically data-bound to collections.

In this lesson, you will learn how to use a variety of collection classes in Visual C#.

### Lesson Objectives

After completing this lesson, you will be able to:

- Choose appropriate collections for different scenarios.
- Manage items in a collection.
- Use Language Integrated Query (LINQ) syntax to query a collection.

### Choosing Collections

All collection classes share various common characteristics. To manage a collection of items, you must be able to:

- Add items to the collection.
- Remove items from the collection.
- Retrieve specific items from the collection.
- Count the number of items in the collection.
- Iterate through the items in the collection, one item at a time.

- *List* classes store linear collections of items
- *Dictionary* classes store collections of key/value pairs
- *Queue* classes store items in a first in, first out collection
- *Stack* classes store items in a last in, first out collection

Every collection class in Visual C# provides methods and properties that support these core operations. Beyond these operations, however, you will want to manage collections in different ways depending on the specific requirements of your application. Collection classes in Visual C# fall into the following broad categories:

- *List* classes store linear collections of items. You can think of a list class as a one-dimensional array that dynamically expands as you add items. For example, you might use a list class to maintain a list of available beverages at your coffee shop.
- *Dictionary* classes store a collection of key/value pairs. Each item in the collection consists of two objects—the *key* and the *value*. The value is the object you want to store and retrieve, and the key is the object that you use to index and look up the value. In most dictionary classes, the key must be unique, whereas duplicate values are perfectly acceptable. For example, you might use a dictionary class to maintain a list of coffee recipes. The key would contain the unique name of the coffee, and the value would contain the ingredients and the instructions for making the coffee.

- *Queue* classes represent a first in, first out collection of objects. Items are retrieved from the collection in the same order they were added. For example, you might use a queue class to process orders in a coffee shop to ensure that customers receive their drinks in turn.
- *Stack* classes represent a last in, first out collection of objects. The item that you added to the collection last is the first item you retrieve. For example, you might use a stack class to determine the 10 most recent visitors to your coffee shop.

When you choose a built-in collection class for a specific scenario, ask yourself the following questions:

- Do you need a list, a dictionary, a stack, or a queue?
- Will you need to sort the collection?
- How large do you expect the collection to get?
- If you are using a dictionary class, will you need to retrieve items by index as well as by key?
- Does your collection consist solely of strings?

If you can answer all of these questions, you will be able to select the Visual C# collection class that best meets your needs.

## Standard Collection Classes

The **System.Collections** namespace provides a range of general-purpose collections that includes lists, dictionaries, queues, and stacks. The following table shows the most important collection classes in the **System.Collections** namespace:

Class	Description
ArrayList	<ul style="list-style-type: none"> <li>• General-purpose list collection</li> <li>• Linear collection of objects</li> </ul>
BitArray	<ul style="list-style-type: none"> <li>• Collection of Boolean values</li> <li>• Useful for bitwise operations and Boolean arithmetic (for example, AND, NOT, and XOR)</li> </ul>
Hashtable	<ul style="list-style-type: none"> <li>• General-purpose dictionary collection</li> <li>• Stores key/value object pairs</li> </ul>
Queue	<ul style="list-style-type: none"> <li>• First in, first out collection</li> </ul>
SortedList	<ul style="list-style-type: none"> <li>• Dictionary collection sorted by key</li> <li>• Retrieve items by index as well as by key</li> </ul>
Stack	<ul style="list-style-type: none"> <li>• Last in, first out collection</li> </ul>

Class	Description
<b>ArrayList</b>	The <b>ArrayList</b> is a general-purpose list that stores a linear collection of objects. The <b>ArrayList</b> includes methods and properties that enable you to add items, remove items, count the number of items in the collection, and sort the collection.
<b>BitArray</b>	The <b>BitArray</b> is a list class that represents a collection of bits as Boolean values. The <b>BitArray</b> is most commonly used for bitwise operations and Boolean arithmetic, and includes methods to perform common Boolean operations such as AND, NOT, and XOR.
<b>Hashtable</b>	The <b>Hashtable</b> class is a general-purpose dictionary class that stores a collection of key/value pairs. The <b>Hashtable</b> includes methods and properties that enable you to retrieve items by key, add items, remove items, and check for particular keys and values within the collection.
<b>Queue</b>	The <b>Queue</b> class is a first in, last out collection of objects. The <b>Queue</b> includes methods to add objects to the back of the queue ( <b>Enqueue</b> ) and retrieve objects from the front of the queue ( <b>Dequeue</b> ).

Class	Description
<b>SortedList</b>	The <b>SortedList</b> class stores a collection of key/value pairs that are sorted by key. In addition to the functionality provided by the <b>Hashtable</b> class, the <b>SortedList</b> enables you to retrieve items either by key or by index.
<b>Stack</b>	The <b>Stack</b> class is a first in, first out collection of objects. The <b>Stack</b> includes methods to view the top item in the collection without removing it ( <b>Peek</b> ), add an item to the top of the stack ( <b>Push</b> ), and remove and return the item at the top of the stack ( <b>Pop</b> ).

 **Reference Links:** For more information about the classes listed in the previous table, see the System.Collections Namespace page at <https://aka.ms/moc-20483c-m3-pg4>.

## Specialized Collection Classes

The **System.Collections.Specialized** namespace provides collection classes that are suitable for more specialized requirements, such as specialized dictionary collections and strongly typed string collections. The following table shows the most important collection classes in the **System.Collections.Specialized** namespace:

Class	Description
ListDictionary	<ul style="list-style-type: none"> <li>Dictionary collection</li> <li>Optimized for small collections (&lt;10)</li> </ul>
HybridDictionary	<ul style="list-style-type: none"> <li>Dictionary collection</li> <li>Implemented as ListDictionary when small, changes to Hashtable as collection grows larger</li> </ul>
OrderedDictionary	<ul style="list-style-type: none"> <li>Unsorted dictionary collection</li> <li>Retrieve items by index as well as by key</li> </ul>
NameValueCollection	<ul style="list-style-type: none"> <li>Dictionary collection in which both keys and values are strings</li> <li>Retrieve items by index as well as by key</li> </ul>
StringCollection	<ul style="list-style-type: none"> <li>List collection in which all items are strings</li> </ul>
StringDictionary	<ul style="list-style-type: none"> <li>Dictionary collection in which both keys and values are strings</li> </ul>
BitVector32	<ul style="list-style-type: none"> <li>Fixed size 32-bit structure</li> <li>Represent values as Booleans or integers</li> </ul>

Class	Description
<b>ListDictionary</b>	The <b>ListDictionary</b> is a dictionary class that is optimized for small collections. As a general rule, if your collection includes 10 items or fewer, use a <b>ListDictionary</b> . If your collection is larger, use a <b>Hashtable</b> .
<b>HybridDictionary</b>	The <b>HybridDictionary</b> is a dictionary class that you can use when you cannot estimate the size of the collection. The <b>HybridDictionary</b> uses a <b>ListDictionary</b> implementation when the collection size is small, and switches to a <b>Hashtable</b> implementation as the collection size grows larger.
<b>OrderedDictionary</b>	The <b>OrderedDictionary</b> is an indexed dictionary class that enables you to retrieve items by key or by index. Note that unlike the <b>SortedList</b> class, items in an <b>OrderedDictionary</b> are not sorted by key.
<b>NameValueCollection</b>	The <b>NameValueCollection</b> is an indexed dictionary class in which both the key and the value are strings. The <b>NameValueCollection</b> will throw an error if you attempt to set a key or a value to anything other than a string. You can retrieve items by key or by index.
<b>StringCollection</b>	The <b>StringCollection</b> is a list class in which every item in the collection is a string. Use this class when you want to store a simple, linear collection of strings.

Class	Description
<b>StringDictionary</b>	The <b>StringDictionary</b> is a dictionary class in which both the key and the value are strings. Unlike the <b>NameValueCollection</b> class, you cannot retrieve items from a <b>StringDictionary</b> by index.
<b>BitVector32</b>	The <b>BitVector32</b> is a struct that can represent a 32-bit value as both a bit array and an integer value. Unlike the <b>BitArray</b> class, which can expand indefinitely, the <b>BitVector32</b> struct is a fixed 32-bit size. As a result, the <b>BitVector32</b> is more efficient than the <b>BitArray</b> for small values. You can divide a <b>BitVector32</b> instance into sections to efficiently store multiple values.



**Reference Links:** For more information about the classes and structs listed in the previous table, see the System.Collections.Specialized Namespace page at <https://aka.ms/moc-20483cm3-pg5>.

## Using List Collections

The **ArrayList** stores items as a linear collection of objects. You can add objects of any type to an **ArrayList** collection, but the **ArrayList** represents each item in the collection as a **System.Object** instance. When you add an item to an **ArrayList** collection, the **ArrayList** implicitly casts, or converts, your item to the **Object** type. When you retrieve items from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from an **ArrayList** collection:

- Add objects of any type

```
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
ArrayList beverages = new ArrayList();
beverages.Add(coffee1);
```

- Retrieve items by index

```
Coffee firstCoffee = (Coffee)beverages[0];
```

- Use a foreach loop to iterate over the collection

```
foreach(Coffee c in beverages)
{
 // Console.WriteLine(c.CountryOfOrigin);
}
```

### Adding and Retrieving Items from an ArrayList

```
// Create a new ArrayList collection.
ArrayList beverages = new ArrayList();

// Create some items to add to the collection.
Coffee coffee1 = new Coffee(4, "Arabica", "Columbia");
Coffee coffee2 = new Coffee(3, "Arabica", "Vietnam");
Coffee coffee3 = new Coffee(4, "Robusta", "Indonesia");

// Add the items to the collection.
// Items are implicitly cast to the Object type when you add them.
beverages.Add(coffee1);
beverages.Add(coffee2);
beverages.Add(coffee3);

// Retrieve items from the collection.
// Items must be explicitly cast back to their original type.
Coffee firstCoffee = (Coffee)beverages[0];
Coffee secondCoffee = (Coffee)beverages[1];
```

When you work with collections, one of your most common programming tasks will be to iterate over the collection. Essentially, this means that you retrieve each item from the collection in turn, usually to render a list of items, to evaluate each item against some criteria, or to extract specific member values from each

item. To iterate over a collection, you use a **foreach** loop. The **foreach** loop exposes each item from the collection in turn, using the variable name you specify in the loop declaration.

The following example shows how to iterate over an **ArrayList** collection:

### Iterating Over a List Collection

```
foreach(Coffee coffee in beverages)
{
 Console.WriteLine("Bean type: {0}", coffee.Bean);
 Console.WriteLine("Country of origin: {0}", coffee.CountryOfOrigin);
 Console.WriteLine("Strength (1-5): {0}", coffee.Strength);
}
```

 **Reference Links:** For more information on the **ArrayList** class, see the **ArrayList Class** page at <https://aka.ms/moc-20483c-m3-pg6>.

## Using Dictionary Collections

Dictionary classes store collections of key/value pairs. One useful dictionary class is the **Hashtable**. When you add an item to a **Hashtable** collection, you must specify a *key* and a *value*. Both the key and the value can be instances of any type, but the **Hashtable** implicitly casts both the key and the value to the **Object** type. When you retrieve values from the collection, you must explicitly cast the object back to its original type.

The following example shows how to add and retrieve items from a **Hashtable** collection. In this case both the key and the value are strings:

- Specify both a key and a value when you add an item

```
Hashtable ingredients = new Hashtable();
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");
```

- Retrieve items by key

```
string recipeMocha = ingredients["Café Mocha"];
```

- Iterate over key collection or value collection

```
foreach(string key in ingredients.Keys)
{
 Console.WriteLine(ingredients[key]);
}
```

### Adding and Retrieving Items from a Hashtable

```
// Create a new Hashtable collection.
Hashtable ingredients = new Hashtable();

// Add some key/value pairs to the collection.
ingredients.Add("Café au Lait", "Coffee, Milk");
ingredients.Add("Café Mocha", "Coffee, Milk, Chocolate");
ingredients.Add("Cappuccino", "Coffee, Milk, Foam");
ingredients.Add("Irish Coffee", "Coffee, Whiskey, Cream, Sugar");
ingredients.Add("Macchiato", "Coffee, Milk, Foam");

// Check whether a key exists.
if(ingredients.ContainsKey("Café Mocha"))
{
 // Retrieve the value associated with a key.
 Console.WriteLine("The ingredients of a Café Mocha are: {0}", ingredients["Café Mocha"]);
}
```

Dictionary classes, such as the **Hashtable**, actually contain two enumerable collections—the keys and the values. You can iterate over either of these collections. In most scenarios, however, you are likely to iterate through the key collection, for example to retrieve the value associated with each key in turn.

The following example shows how to iterate over the keys in a **Hashtable** collection and retrieve the value associated with each key:

#### Iterating Over a Dictionary Collection

```
foreach(string key in ingredients.Keys)
{
 // For each key in turn, retrieve the value associated with the key.
 Console.WriteLine("The ingredients of a {0} are {1}", key, ingredients[key]);
}
```



**Reference Links:** For more information on the **Hashtable** class, see the Hashtable Class page at <https://aka.ms/moc-20483c-m3-pg7>.

## Querying a Collection

LINQ is a query technology that is built in to .NET languages such as Visual C#. LINQ enables you to use a standardized, declarative query syntax to query data from a wide range of data sources, such as .NET collections, SQL Server databases, ADO.NET datasets, and XML documents. A basic LINQ query uses the following syntax:

```
from <variable names> in <data source>
where <selection criteria>
orderby <result ordering criteria>
select <variable names>
```

For example, suppose you use a **Hashtable** to maintain a price list for beverages at Fourth Coffee. You might use a LINQ expression to retrieve all the drinks that meet certain price criteria, such as drinks that cost less than \$2.00.

The following example shows how to use a LINQ expression to query a **Hashtable**:

#### Using LINQ to Query a Collection

```
// Create a new Hashtable and add some drinks with prices.
Hashtable prices = new Hashtable();
prices.Add("Café au Lait", 1.99M);
prices.Add("Caffe Americano", 1.89M);
prices.Add("Café Mocha", 2.99M);
prices.Add("Cappuccino", 2.49M);
prices.Add("Espresso", 1.49M);
prices.Add("Espresso Romano", 1.59M);
prices.Add("English Tea", 1.69M);
prices.Add("Juice", 2.89M);

// Select all the drinks that cost less than $2.00, and order them by cost.
var bargains =
 from string drink in prices.Keys
 where (Decimal)prices[drink] < 2.00M
 orderby prices[drink] ascending
 select drink;

// Display the results.
foreach(string bargain in bargains)
```

- Use LINQ expressions to query collections

```
var drinks =
 from string drink in prices.Keys
 orderby prices[drink] ascending
 select drink;
```

- Use extensions methods to retrieve specific items from results

```
decimal lowestPrice = drinks.FirstOrDefault();
decimal highestPrice = drinks.Last();
```

```
{
 Console.WriteLine(bargain);
}
Console.ReadLine();
```

 **Note:** Appending the suffix **m** or **M** to a number indicates that the number should be treated as a **decimal** type.

In addition to this basic query syntax, you can call a variety of methods on your query results. For example:

- Call the **FirstOrDefault** method to get the first item from the results collection, or a default value if the collection contains no results. This method is useful if you have ordered the results of your query.
- Call the **Last** method to get the last item from the results collection. This method is useful if you have ordered the results of your query.
- Call the **Max** method to find the largest item in the results collection.
- Call the **Min** method to find the smallest item in the results collection.

 **Note:** Most built-in types provide methods that enable you to compare one instance to another to determine which is considered larger or smaller. The **Max** and **Min** methods rely on these methods to find the largest or smallest items. If your collection contains numerical types, these methods will return the highest and lowest values, respectively. If your collection contains strings, members are compared alphabetically—for example, "Z" is considered greater than "A". If your collection contains custom types, the **Max** and **Min** methods will use the comparison logic created by the type developer.

For example, if you have ordered your results by ascending cost, the first item in the results collection will be the cheapest and the last item in the results collection will be the most expensive. As such, you can use the **FirstOrDefault** and **Last** methods to find the cheapest and most expensive drinks, respectively.

The following example shows how to retrieve the smallest and largest items from a collection based on the sort criteria in the LINQ expression:

### Using the **FirstOrDefault** and **Last** Methods

```
// Query the Hashtable to order drinks by cost.
var drinks =
 from string drink in prices.Keys
 orderby prices[drink] ascending
 select drink;

Console.WriteLine("The cheapest drink is {0}: ", drinks.FirstOrDefault());
// Output: "The cheapest drink is Espresso"

Console.WriteLine("The most expensive drink is {0}: ", drinks.Last());
// Output: "The most expensive drink is Café Mocha"

Console.WriteLine("The maximum is {0}: ", drinks.Max());
// Output: "The maximum is Juice"
// "Juice" is the largest value in the collection when ordered alphabetically.

Console.WriteLine("The minimum is {0}: ", drinks.Min());
// Output: "The minimum is Café au Lait"
// "Café au Lait" is the smallest value in the collection when ordered alphabetically.
```



**Note:** The return type of a LINQ expression is **IEnumerable<T>**, where **T** is the type of the items in the collection. **IEnumerable<T>** is an example of a *generic type*. The methods you use on a results set, such as **FirstOrDefault**, **Last**, **Max**, and **Min**, are extension methods. Generic types and extension methods are covered later in this course.



**Reference Links:** For more information about using LINQ to query collections, see the LINQ Query Expressions (C# Programming Guide) page at <https://aka.ms/moc-20483c-m3-pg8>.

## Lesson 3

# Handling Events

Events are mechanisms that enable objects to notify other objects when something happens. For example, controls on a web page or in a WPF user interface generate events when a user interacts with the control, such as by clicking a button. You can create code that *subscribes* to these events and takes some action in response to an event.

Without events, your code would need to constantly read control values to look for any changes in state that require action. This would be a very inefficient way of developing an application. In this lesson, you will learn how to create, or *raise*, events, and how to subscribe to events.

In this lesson, you will learn how to handle events.

### Lesson Objectives

After completing this lesson, you will be able to:

- Create events and delegates.
- Raise events.
- Subscribe to events and unsubscribe from events.

### Delegates and Events - Introduction

There are many cases when building an application, where you'll want to perform an action that's unknown in your current class or context, and that action will be supplied from an external source. For example, say there is a method that writes text to an output. You may want to let the user of that method configure the output beforehand: set the font, text size, etc. Perhaps the user may want to run some other complex logic before writing the text. Your method will need some way to receive and run that supplied code.

#### • Delegates

- Delegates are reference to methods.
  - They can be saved as field and passed as parameters.
  - They allow one to execute code that is provided from external sources.
  - Delegates can be executed exactly like methods.
- #### • Events
- An event wraps a delegate like a property wraps a field.
  - Events allow the class to notify other consumers of actions performed within it.
  - An event may only be raised by its containing class.

This is the problem that the delegates solve. A delegate can be thought of as reference to a method. You can save it as a field or variable and pass it along like any other reference type. The only difference is that where type fields save data, a Delegate saves logic. As such, a delegate can be invoked exactly like a method. It can accept parameters, and it can return a result.

Just as regular fields have properties to restrict access and protect the usage of their underlying field, delegates have events. Events encapsulate their delegate and prevent anyone from raising them or overriding their value. A consuming class can never assign a value to an event, like it can a delegate. It can only subscribe and unsubscribe to and from the event. Only the containing class may raise (invoked) the event.

Although it's easy to technically equate events to properties as they relate to their respective delegates and fields, it's not exactly true. While properties simply provide a protected access to their field, events are conceptually different from delegates, even though they rely on them. While delegates are often used to run the logic provided to the process from outside, events are used to notify other classes or consumers of the class that a certain action has just occurred (in other words – that an event has happened). Therefore,

events are seldom used to influence the current process (though that's possible), but to start other processes in the subscribed targets.

## Creating Events and Delegates

When you create an event in a struct or a class, you need a way of enabling other code to subscribe to your event. In Visual C#, you accomplish this by creating a *delegate*. A delegate is a special type that defines a method signature; in other words, the return type and the parameters of a method. As the name suggests, a delegate behaves like a representative for methods with matching signatures.

When you define an event, you associate a delegate with your event. To subscribe to the event from client code, you need to:

- Create a method with a signature that matches the event delegate. This method is known as the *event handler*.
- Subscribe to the event by giving the name of your event handler method to the *event publisher*, in other words, the object that will raise the event.

When the event is raised, the delegate invokes all the event handler methods that have subscribed to the event.

Suppose you create a struct named **Coffee**. One of the responsibilities of this struct is to keep track of the stock level for each **Coffee** instance. When the stock level drops below a certain point, you might want to raise an event to warn an ordering system that you are running out of beans.

The first thing you need to do is to define a delegate. To define a delegate, you use the **delegate** keyword. A delegate includes two parameters:

- The first parameter is the object that raised the event—in this case, a **Coffee** instance.
- The second parameter is the event arguments—in other words, any other information that you want to provide to consumers. This must be an instance of the **EventArgs** class, or an instance of a class that derives from **EventArgs**.

Next, you need to define the event. To define an event, you use the **event** keyword. You precede the name of your event with the name of the delegate you want to associate with your event.

The following example shows how to define delegates and events:

### Defining a Delegate and an Event

```
public struct Coffee
{
 public EventArgs e;
 public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);
 public event OutOfBeansHandler OutOfBeans;
}
```

In this example, you define an event named **OutOfBeans**. You associate a delegate named **OutOfBeansHandler** with your event. The **OutOfBeansHandler** delegate takes two parameters, an

instance of **Coffee** that will represent the object that raised the event and an instance of **EventArgs** that could be used to provide more information about the event.

## Raising Events

After you have defined an event and a delegate, you can write code that raises the event when certain conditions are met. When you raise the event, the delegate associated with your event will invoke any event handler methods that have subscribed to your event.

To raise an event, you need to do two things:

1. Check whether the event is null. The event will be null if no code is currently subscribing to it.
2. Invoke the event and provide arguments to the delegate.

- Check whether the event is null
  - Raise the event by using method syntax
- ```
if (OutOfBeans != null)
{
    OutOfBeans(this, e);
}
```

For example, suppose a **Coffee** struct includes a method named **MakeCoffee**. Every time you call the **MakeCoffee** method, the method reduces the stock level of the **Coffee** instance. If the stock level drops below a certain point, the **MakeCoffee** method will raise an **OutOfBeans** event.

The following example shows how to raise an event:

Raising an Event

```
public struct Coffee
{
    // Declare the event and the delegate.
    public EventArgs e = null;
    public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);
    public event OutOfBeansHandler OutOfBeans;

    int currentStockLevel;
    int minimumStockLevel;

    public void MakeCoffee()
    {
        // Decrement the stock level.
        currentStockLevel--;

        // If the stock level drops below the minimum, raise the event.
        if (currentStockLevel < minimumStockLevel)
        {
            // Check whether the event is null.
            if (OutOfBeans != null)
            {
                // Raise the event.
                OutOfBeans(this, e);
            }
        }
    }
}
```

To raise the event, you use a similar syntax to calling a method. You provide arguments to match the parameters required by the delegate. The first argument is the object that raised the event. Note how the **this** keyword is used to indicate the current **Coffee** instance. The second parameter is the **EventArgs** instance, which can be null if you do not need to provide any other information to subscribers.

Subscribing to Events

If you want to handle an event in client code, you need to do two things:

- Create a method with a signature that matches the delegate for the event.
- Use the addition assignment operator (+=) to attach your event handler method to the event.

For example, suppose you have created an instance of the **Coffee** struct named **coffee1**. In your **Inventory** class, you want to subscribe to the **OutOfBeans** that may be raised by **coffee1**.

- Create a method that matches the delegate signature

```
public void HandleOutOfBeans(Coffee c, EventArgs e)
{
    // Do something useful here.
}
```

- Subscribe to the event

```
coffee1.OutOfBeans += HandleOutOfBeans;
```

- Unsubscribe from the event

```
coffee1.OutOfBeans -= HandleOutOfBeans;
```



Note: The previous topic shows how the **Coffee** struct, the **OutOfBeans** event, and the **OutOfBeansHandler** delegate are defined.

The following example shows how to subscribe to an event:

Subscribing to an Event

```
public class Inventory
{
    public void HandleOutOfBeans(Coffee sender, EventArgs args)
    {
        string coffeeBean = sender.Bean;
        // Reorder the coffee bean.
    }

    public void SubscribeToEvent()
    {
        coffee1.OutOfBeans += HandleOutOfBeans;
    }
}
```

In this example, the signature of the **HandleOutOfBeans** method matches the delegate for the **OutOfBeans** event. When you call the **SubscribeToEvent** method, the **HandleOutOfBeans** method is added to the list of subscribers for the **OutOfBeans** event on the **coffee1** object.

To unsubscribe from an event, you use the subtraction assignment operator (-=) to remove your event handler method from the event. It's safe to unsubscribe from an event to which you're not subscribed. In that case nothing will happen, no exception will be thrown.

The following example shows how to unsubscribe from an event:

Unsubscribing from an Event

```
public void UnsubscribeFromEvent()
{
    coffee1.OutOfBeans -= HandleOutOfBeans;
}
```

It's important to unsubscribe to events when you are done with them or the subscribed object. A subscription to an event saves a reference to the subscribed object instance. Failing to unsubscribe may result in a memory leak.



Reference Links: For more information see How to: Subscribe to and Unsubscribe from Events (C# Programming Guide) page at <https://aka.ms/moc-20483c-m3-pg9>.

Demonstration: Working with Events in XAML

Visual Studio provides tools that make it easy to work with events in WPF applications. In this demonstration, you will learn how to subscribe to events raised by WPF controls.

Demonstration Steps

You will find the steps in the **Demonstration: Working with Events in XAML** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md.

Demonstration: Writing Code for the Grades Prototype Application Lab

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: Writing Code for the Grades Prototype Application Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_DEMO.md.

Lab: Writing the Code for the Grades Prototype Application

Scenario

The School of Fine Arts has decided that they want to extend their basic class enrollment application to enable teachers to record the grades that students in their class have achieved for each subject, and to allow students to view their own grades. This functionality necessitates implementing application log on functionality to authenticate the user and to determine whether the user is a teacher or a student.

You decide to start by developing parts of a prototype application to test proof of concept and to obtain client feedback before embarking on the final application. The prototype application will use basic WPF views rather than separate forms for the user interface. These views have already been designed and you must add the code to navigate among them.

You also decide to begin by storing the user and grade information in basic structs, and to use a dummy data source in the application to test your log on functionality.

Objectives

After completing this lab, you will be able to:

- Navigate among views.
- Create and use collections of structs.
- Handle events.

Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD03_LAK.md.

Exercise 1: Adding Navigation Logic to the Grades Prototype Application

Scenario

In this exercise, you will add navigation logic to the **Grades Prototype** application.

First, you will examine the window and views in the application so that you are familiar with the existing structure of the application. You will define a public event handler named **LogonSuccess** that will be raised when a user successfully logs on to the application. You will add dummy code to the **Logon_Click** event handler to store the username and role of the logged on user and raise the **LogonSuccess** event.

Then you will add markup to the **LogonPage** XAML code to connect the **Logon** button to the **Logon_Click** event handler. Next, you will add code to the **GotoLogon** method to display the logon view and to hide the other views. You will implement the **Logon_Success** method to handle a successful log on by displaying the logged on views, and then you will add markup to the **MainWindow** XAML code to connect the **LogonSuccess** event to the **Logon_Success** method. You will add code to the **MainWindow** to determine whether the user is a teacher or a student, display their name in the application, and display either the **StudentsPage** view for teachers or the **StudentProfile** view for students. You will then add code to the **StudentsPage** view that catches a student name being clicked and raises the **StudentSelected** event for that student and displays their student profile. Finally, you will run the application and verify that the appropriate views are displayed for students and teachers upon a successful log on.

Exercise 2: Creating Data Types to Store User and Grade Information

Scenario

In this exercise, you will define basic structs that will hold the teacher, student, and grade information for the application. You will then examine the dummy data source that the application uses to populate the collections in this exercise.

Exercise 3: Displaying User and Grade Information

Scenario

In this exercise, you will first define a public event handler named **LogonFailed** that will be raised when a user fails to log on successfully. You will add code to the **Logon_Click** event handler to validate the username and password entered by the user against the Users collection in the **MainWindow** window. If the user is a teacher or a student, you will store their details in the global context and then raise the **LogonSuccess** event, but if the user is not validated, you will raise the **LogonFailed** event. You will handle log on failure in the **Logon_Failed** method to display a message to the user and then you will add markup to the **MainWindow** XAML code to connect the **LogonFailed** event to the **Logon_Failed** method. You will add code to the **StudentsPage** view to display students for the current teacher, and to display the details for a student when the user clicks their name. You will then use data binding to display the details and grades for the current student in the **StudentProfile** view, and to display only the **Back** button if the user is a teacher. Finally, you will run the application and verify that only valid users can log on and that valid users can see only data appropriate to their role.

Module Review and Takeaways

In this module, you have learned how to implement structs and enums, organize data into collections, and work with events and delegates.

Review Questions

Check Your Knowledge

| Question |
|--|
| You want to create a string property named CountryOfOrigin. You want to be able to read the property value from any code, but you should only be able to write to the property from within the containing struct. How should you declare the property? |
| Select the correct answer. |
| <code>public string CountryOfOrigin { get; set; }</code> |
| <code>public string CountryOfOrigin { get; }</code> |
| <code>public string CountryOfOrigin { set; }</code> |
| <code>public string CountryOfOrigin { get; private set; }</code> |

Check Your Knowledge

| Question |
|--|
| You want to create a collection to store coffee recipes. You must be able to retrieve each coffee recipe by providing the name of the coffee. Both the name of the coffee and the coffee recipe will be stored as strings. You also need to be able to retrieve coffee recipes by providing an integer index. Which collection class should you use? |
| Select the correct answer. |
| <code>ArrayList</code> |
| <code>Hashtable</code> |
| <code>SortedList</code> |
| <code>NameValueCollection</code> |
| <code>StringDictionary</code> |

Check Your Knowledge

| Question | |
|---|---|
| <p>You are creating a method to handle an event named OutOfBeans. The delegate for the event is as follows:</p> <pre>public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args);</pre> <p>Which of the following methods should you use to subscribe to the event?</p> | |
| <p>Select the correct answer.</p> | |
| | <pre>public void HandleOutOfBeans(delegate OutOfBeansHandler)</pre> <pre>{}</pre> <pre>}</pre> |
| | <pre>public void HandleOutOfBeans(Coffee c, EventArgs e)</pre> <pre>{}</pre> <pre>}</pre> |
| | <pre>public Coffee HandleOutOfBeans(EventArgs e)</pre> <pre>{}</pre> <pre>}</pre> |
| | <pre>public Coffee HandleOutOfBeans(Coffee coffee, EventArgs args)</pre> <pre>{}</pre> <pre>}</pre> |
| | <pre>public void HandleOutOfBeans(Coffee coffee)</pre> <pre>{}</pre> <pre>}</pre> |

MCT USE ONLY. STUDENT USE PROHIBITED

Module 4

Creating Classes and Implementing Type-Safe Collections

Contents:

| | |
|---|------|
| Module Overview | 4-1 |
| Lesson 1: Creating Classes | 4-2 |
| Lesson 2: Defining and Implementing Interfaces | 4-12 |
| Lesson 3: Implementing Type-Safe Collections | 4-21 |
| Lab: Adding Data Validation and Type-Safety to the Application | 4-34 |
| Module Review and Takeaways | 4-36 |

Module Overview

Classes enable you to create your own custom, self-contained, and reusable types. Interfaces enable you to define a set of inputs and outputs that classes must implement in order to ensure compatibility with consumers of the classes. In this module, you will learn how to use interfaces and classes to define and create your own custom, reusable types. You will also learn how to create and use enumerable, type-safe collections of any type.

Objectives

After completing this module, you will be able to:

- Create and instantiate classes.
- Create and instantiate interfaces.
- Use generics to create type-safe collections.

Lesson 1

Creating Classes

In Visual C#, you can define your own custom types by creating *classes*. As a programming construct, the class is central to object-oriented programming in Visual C#. It enables you to encapsulate the behaviors and characteristics of any logical entity in a reusable and extensible way.

In this lesson, you will learn how to create, use, and test classes in your own applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Create classes.
- Create objects by instantiating classes.
- Use constructors to set values or to run logic when classes are instantiated.
- Explain the difference between reference types and value types.
- Create static classes and members.
- Describe the high-level process for testing class functionality.

Creating Classes and Members

In Visual C#, a *class* is a programming construct that you can use to define your own custom types. When you create a class, you create a new type, which is effectively creating a blueprint for the instance. The class defines the behaviors and characteristics, or class members, which exists in all instances of the class. You represent these behaviors and characteristics by defining methods, fields, properties, and events within your class.

Declaring a Class

For example, suppose you create a class named **DrinksMachine**.

You use the **class** keyword to declare a class, as shown in the following example:

Declaring a Class

```
public class DrinksMachine
{
    // Methods, fields, properties, and events go here.
}
```

- Use the **class** keyword

```
public class DrinksMachine
{
    // Methods, fields, properties, and events.
}
```

- Specify an access modifier:

- public
- internal
- private

- Add methods, fields, properties, and events

The **class** keyword is preceded by an *access modifier*, such as **public** in the above example, which specifies where you can use the type. You can use the following access modifiers in your class declarations:

| Access modifier | Description |
|-----------------|---|
| public | The type is available to code running in any assembly that references the assembly in which the class is contained. |

| Access modifier | Description |
|-----------------|--|
| internal | The type is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier. |
| private | The type is only available to code within the class that contains it. You can only use the private access modifier with nested classes. |

Adding Members to a Class

You would use fields and properties to define the characteristics of a drinks machine, such as the make, model, age, and service interval of the machine. You would create methods to represent the things that a drinks machine can do, such as make an espresso or make a cappuccino. Finally, you would define events to represent actions that might require your attention, such as replacing coffee beans when the machine has run out of coffee beans.

Within your class, you can add methods, fields, properties, and events to define the behaviors and characteristics of your type, as shown in the following example:

Defining Class Members

```
public class DrinksMachine
{
    // The following statements define a property with a private field.
    private int _age;
    public int Age
    {
        get
        {
            return _age;
        }
        set
        {
            if (value>0)
                _age = value;
        }
    }

    // The following statements define properties.
    public string Make;
    public string Model;

    // The following statements define methods.
    public void MakeCappuccino()
    {
        // Method logic goes here.
    }

    public void MakeEspresso()
    {
        // Method logic goes here.
    }

    // The following statement defines an event. The delegate definition is not shown.
    public event OutOfBeansHandler OutOfBeans;
}
```

Instantiating Classes

A class is just the definition of a type. To use the behaviors and characteristics that you define within a class, you need to create *instances* of the class.

To create a new instance of a class, you use the **new** keyword, as shown in the following example:

Instantiating a Class

```
DrinksMachine dm = new DrinksMachine();
```

When you instantiate a class in this way, you are actually doing two things:

- You are creating a new *object* in memory based on the **DrinksMachine** type.
- You are creating an *object reference* named **dm** that refers to the new **DrinksMachine** object.

- To instantiate a class, use the **new** keyword
`DrinksMachine dm = new DrinksMachine();`
- To infer the type of the new object, use the **var** keyword
`var dm = new DrinksMachine();`
- To call members on the instance, use the dot notation
`dm.Model = "BeanCrusher 3000";
dm.Age = 2;
dm.MakeEspresso();`

Instantiating a Class by Using Type Inference

```
var dm = new DrinksMachine();
```

In this case, the compiler does not know in advance the type of the **dm** variable. When the **dm** variable is initialized as a reference to a **DrinksMachine** object, the compiler deduces that the type of **dm** is **DrinksMachine**. Using type inference in this way causes no change in how your application runs; it is simply a shortcut for you to avoid typing the class name twice. In some circumstances, type inference can make your code easier to read, while in other circumstances it may make your code more confusing. As a general rule, consider using type inference when the type of variable is absolutely clear.

After you have instantiated your object, you can use any of the members—methods, fields, properties, and events—that you defined within the class, as shown in the following example:

Using Object Members

```
var dm = new DrinksMachine();
dm.Make = "Fourth Coffee";
dm.Model = "Beancrusher 3000";
dm.Age = 2;
dm.MakeEspresso();
```

This approach to calling members on an instance variable is known as *dot notation*. You type the variable name, followed by a period, followed by the member name. The IntelliSense feature in Visual Studio will prompt you with member names when you type a period after a variable.

Using Constructors

In the previous topics, you might have noticed that the syntax for instantiating a class—for example, `new DrinksMachine()`—looks similar to the syntax for calling a method. This is because when you instantiate a class, you are actually calling a special method called a *constructor*. A constructor is a method in the class that has the same name as the class.

Constructors are often used to specify initial or default values for data members within the new object, as shown by the following example:

- Constructors are a type of method:
 - Share the name of the class
 - Called when you instantiate a class
- A default constructor accepts no arguments

```
public class DrinksMachine
{
    public void DrinksMachine()
    {
        // This is a default constructor.
    }
}
```

- Classes can include multiple constructors
- Use constructors to initialize member variables

Adding a Constructor

```
public class DrinksMachine
{
    public int Age { get; set; }
    public DrinksMachine()
    {
        Age = 0;
    }
}
```

A constructor that takes no parameters is known as the *default constructor*. This constructor is called whenever someone instantiates your class without providing any arguments. If you do not include a constructor in your class, the Visual C# compiler will automatically add an empty public default constructor to your compiled class.

In many cases, it is useful for consumers of your class to be able to specify initial values for data members when the class is instantiated. For example, when someone creates a new instance of **DrinksMachine**, it might be useful if they can specify the make and model of the machine at the same time. Your class can include multiple constructors with different signatures that enable consumers to provide different combinations of information when they instantiate your class.

The following example shows how to add multiple constructors to a class:

Adding Multiple Constructors

```
public class DrinksMachine
{
    public int Age { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public DrinksMachine(int age)
    {
        this.Age = age;
    }

    public DrinksMachine(string make, string model)
    {
        this.Make = make;
        this.Model = model;
    }

    public DrinksMachine(int age, string make, string model)
    {
        this.Age = age;
        this.Make = make;
    }
}
```

```
        this.Model = model;
    }
```

Consumers can use any of the constructors to create instances of your class, depending on the information that is available to them at the time. For example:

Calling Constructors

```
var dm1 = new DrinksMachine(2);
var dm2 = new DrinksMachine("Fourth Coffee", "BeanCrusher 3000");
var dm3 = new DrinksMachine(3, "Fourth Coffee", "BeanToaster Turbo");
```

Reference Types and Value Types

Now that you know how to create and instantiate classes, you will learn about the differences between classes and structs.

First, however, let's see how Visual C# works behind the scenes.

The Common Language Runtime (CLR)

The .Net framework consists of several components, built one upon the other. The first is the CLR. This is the engine that envelops and runs all .Net applications. The CLR is responsible for - among other things: the execution of .Net code, Memory management and garbage collection for the application, exception handling, etc.

Above the CLR is the Base Class Library (BCL), that includes all the built-in types in the .Net Framework.

Above that are the specific implementation of the different languages .Net supports.

- Value types
 - Contain data directly
 - int First = 100;
 - int Second = First;
 - In this case, **First** and **Second** are two distinct items in memory
- Reference types
 - Point to an object in memory
 - object First = new Object();
 - object Second = First;
 - In this case, **First** and **Second** point to the same item in memory



Additional Reading: For more information on the CLR, see: <https://aka.ms/moc-20483cm4-pg3>.

The .Net Memory Model

Now let's address how variables are actually saved in the computer memory. .Net uses two main methods of saving its working memory.

The Stack

The stack is used to save local value variables in the current scope. It functions just like the stack collection, with each new variable being assigned right next to the last one used. When the code exits its current scope, all the associated data in the stack is cleared, and the stack reverts to its previous scope. A scope in .Net is generally defined as the currently running code block surrounded by curly brackets. The most common use of that is the method. When a method is entered, the stack will save all local value variables created in that method, as well as any value parameters passed to the method. When the code exits the method, all of this data is cleared. However, if another method is called from the current method, the new data will be saved right after the current data. This allows the stack to easily load the appropriate data when a scope is exited.

The stack has a fixed size during the lifetime of the application. Exceeding it will result in a **StackOverflowException**.

The Heap

The heap is used to save large reference variables. Memory is allocated dynamically from the heap as needed. When a new large object is created, the CLR looks for the next continuous free memory space large enough to contain the object and allocates that memory to it.

Memory is not automatically cleared from the heap when the scope is exited like it was with the stack. Rather, the garbage collector is responsible to clear the heap. When it decides so, the garbage collector will initiate a scan of the objects in the heap. It continues to delete any object it determines is no longer in use. Lastly the garbage collector defragments the memory left in use to eliminate small gaps between existing objects. This is to avoid waste since objects on the heap are only allocated to continuous free memory spaces.

The heap doesn't have a fixed size and can grow during the lifetime of the application. However, it does have an upper limit to its size. If the application reaches that upper limit, it throws an **OutOfMemoryException** exception.

Define Reference and Value Types

Now that we know where and how the CLR stores our application's variables, let see what it saves where and how to interact with them

Reference Types

These are assumed to be rather large objects and are therefore referred to by their address only (hence the name). When you create a reference type object, you do allocate all the memory it requires, however, what you receive back is not the object itself, but only its address in the memory. When you "copy" a reference type, as shown below:

```
Var x = new MyClass();
Var y = x;
```

Only the reference is copied. Any change made to data within the instance will show when either x or y is queried, since they are, in fact, the same instance.

```
x.MyProp = 42;
Console.WriteLine(y.MyProp); // Print 42
y.MyProp += 100;
Console.WriteLine(x.MyProp); // Print 142
```

This also means any change made to a referenced type passed as a parameter to a method will be present when the method exits. Reference types are only saved on the heap.

In Visual C#, classes and delegates, are always reference types. To create a new reference type yourself, you only need to define a new class. This applies to built-in types as well, you can see if a certain type is a reference type by looking up its definition.

Value Types

These are assumed to be smaller objects and are not referenced indirectly. When a value type variable is created, that variable will keep the variable's value itself. When you copy a value type to another it really is copied, resulting in two different instances with the same value. A change to the first will not affect the other.

```
Var x = 1;
Var y = x;
x = 42
Console.WriteLine(x); // Print 42
```

```
Console.WriteLine(y); // Print 1
y += 100;
Console.WriteLine(x); // Print 42
Console.WriteLine(y); // Print 101
```

When a value type is assigned as a method parameter, its value is again copied, and the method will use the copied value. Meaning that any change to the parameter's value will not be seen when the method returns.

Value types are sometimes saved on the heap, and sometimes on the stack, depending on their usage. Generally, only local method members and parameters are saved on the stack. While other usages (like class fields and properties) are saved on the heap as a part of their parent object.

In Visual C#, only **structs** are value types. To create a new value type yourself, you only need to define a new **struct**. This applies to built-in types as well, you can see if a certain type is a value type by looking up its definition.



Note: Most of the built-in types in Visual C#, such as **int**, **bool**, **byte**, and **char**, are value types. For more information about built-in types, see the Built-In Types Table (C# Reference) page at <http://go.microsoft.com/fwlink/?LinkId=267800>.

Boxing and Unboxing

In some scenarios you may need to convert value types to reference types, and vice versa. For example, some collection classes will only accept reference types. This is less likely to be an issue with the advent of generic collections. However, you still need to be aware of the concept, because a fundamental concept of Visual C# is that you can treat any type as an object.

The process of converting a value type to a reference type is called *boxing*. To box a variable, you assign it to an object reference:

Boxing

```
int i = 100;
object o = i;
```

The boxing process is implicit. When you assign an object reference to a value type, the Visual C# compiler automatically creates an object to wrap the value and stores it in memory. If you copy the object reference, the copy will point to the same object wrapper in memory.

The process of converting a reference type to a value type is called *unboxing*. Unlike the boxing process, to unbox a value type you must explicitly cast the variable back to its original type:

Unboxing

```
int j;
j = (int)o;
```

Note: Note that boxing and unboxing is a slow and expensive process. It's usually better keep the type specified if possible. Using generics is a good solution to the problem. Learn more about generics in Lesson 3 - Implementing Type-Safe Collections.

Demonstration: Comparing Reference Types and Value Types

In this demonstration, you will create a value type to store an integer value and a reference type to store an integer value. You will create a copy of each type, and then observe what happens when you change the value of the copy in each case.

Demonstration Steps

You will find the steps in the “**Demonstration: Comparing Reference Types and Value Types**” section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md.

Creating Static Classes and Members

In some cases, you may want to create a class purely to encapsulate some useful functionality, rather than to represent an instance of anything. For example, suppose you wanted to create a set of methods that convert imperial weights and measures to metric weights and measures, and vice versa. It would not make sense if you had to instantiate a class in order to use these methods, because you do not need to store or retrieve any instance-specific data. In fact, the concept of an instance is meaningless in this case.

- Use the **static** keyword to create a static class

```
public static class Conversions
{
    // Static members go here.
}
```

- Call members directly on the class name

```
double weightInKilos = 80;
double weightInPounds =
    Conversions.KilosToPounds(weightInKilos);
```

- Add static members to non-static classes

In scenarios like this, you can create a *static class*.

A static class is a class that cannot be instantiated. To create a static class, you use the **static** keyword. Any members within the class must also use the **static** keyword, as shown in the following example:

Static Classes

```
public static class Conversions
{
    public static double PoundsToKilos(double pounds)
    {
        // Convert argument from pounds to kilograms
        double kilos = pounds * 0.4536;
        return kilos;
    }

    public static double KilosToPounds(double kilos)
    {
        // Convert argument from kilograms to pounds
        double pounds = kilos * 2.205;
        return pounds;
    }
}
```

To call a method on a static class, you call the method on the class name itself instead of on an instance name, as shown by the following example:

Calling Methods on a Static Class

```
double weightInKilos = 80;
double weightInPounds = Conversions.KilosToPounds(weightInKilos);
```

Static Members

Non-static classes can include static members. This is useful when some behaviors and characteristics relate to the instance (instance members), while some behaviors and characteristics relate to the type itself (static members). Methods, fields, properties, and events can all be declared static. Static properties are often used to return data that is common to all instances, or to keep track of how many instances of a class have been created. Static methods are often used to provide utilities that relate to the type in some way, such as comparison functions.

To declare a static member you use the **static** keyword before the return type of the member, as shown by the following example:

Static Members in Non-static Classes

```
public class DrinksMachine
{
    public int Age { get; set; }
    public string Make { get; set; }
    public string Model { get; set; }

    public static int CountDrinksMachines()
    {
        // Add method logic here.
    }
}
```

Regardless of how many instances of your class exist, there is only ever one instance of a static member. You do not need to instantiate the class in order to use static members. You access static members through the class name rather than the instance name, as shown by the following example:

Access Static Members

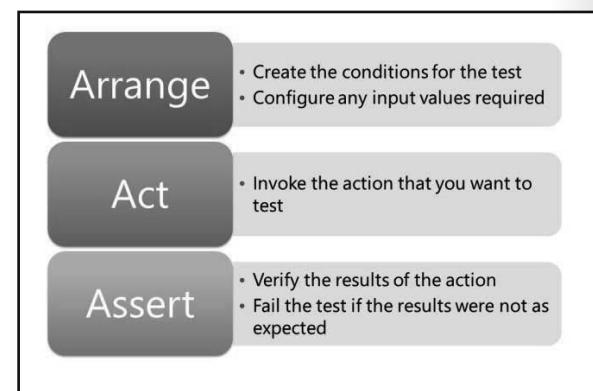
```
int drinksMachineCount = DrinksMachine.CountDrinksMachines();
```

Testing Classes

Classes often represent self-contained units of functionality. In many cases, you will want to test the functionality of your classes in isolation before you integrate them with other classes in your applications.

To test functionality in isolation, you create a unit test. A unit test presents the code under test with known inputs, performs an action on the code under test (for example by calling a method), and then verifies that the outputs of the operation are as expected. In this way, the unit test represents a contract that your code must fulfill. However, when you change the implementation of a class or method, the unit test ensures that your code always returns particular outputs in response to particular inputs.

For example, consider the case where you create a simple class to represent a customer. To help you target your marketing efforts, the **Customer** class includes a **GetAge** method that returns the current age of the customer in years:



Class Under Test

```
public class Customer
{
    public DateTime DateOfBirth { get; set; }

    public int GetAge()
    {
        TimeSpan difference = DateTime.Now.Subtract(DateOfBirth);
        int ageInYears = (int)(difference.Days / 365.25);
        // Note: converting a double to an int rounds down to the nearest whole number.
        return ageInYears;
    }
}
```

In this case, you might want to create a unit test that ensures the **GetAge** method behaves as expected. As such, your test method needs to instantiate the **Customer** class, specify a date of birth, and then verify that the **GetAge** method returns the correct age in years. Depending on the unit test framework you use, your test method might look something like the following:

Example Test Method

```
[TestMethod]
public void TestGetAge()
{
    // Arrange.
    DateTime dob = DateTime.Today;
    dob.AddDays(7);
    dob.AddYears(-24);
    Customer testCust = new Customer();
    testCust.DateOfBirth = dob;
    // The customer's 24th birthday is seven days away, so the age in years should be 23.
    int expectedAge = 23;

    // Act.
    int actualAge = testCust.GetAge();

    // Assert.
    // Fail the test if the actual age and the expected age are different.
    Assert.IsTrue(actualAge == expectedAge, "Age not calculated correctly");
}
```

Notice that the unit test method is divided into three conceptual phases:

- **Arrange.** In this phase, you create the conditions for the test. You instantiate the class you want to test, and you configure any input values that the test requires.
- **Act.** In this phase, you perform the action that you want to test.
- **Assert.** In this phase, you verify the results of the action. If the results were not as expected, the test fails.

The **Assert.IsTrue** method is part of the Microsoft Unit Test Framework that is included in Visual Studio 2012. This particular method throws an exception if the specified condition does not evaluate to **true**. However, the concepts described here are common to all unit testing frameworks.

Lesson 2

Defining and Implementing Interfaces

An *interface* is a little bit like a class without an implementation. It specifies a set of characteristics and behaviors by defining signatures for methods, properties, events, and indexers, without specifying how any of these members are implemented. When a class implements an interface, the class provides an implementation for each member of the interface. By implementing the interface, the class is thereby guaranteeing that it will provide the functionality specified by the interface.

In this lesson, you will learn how to define and implement interfaces.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the purpose of interfaces in Visual C#.
- Create interfaces.
- Create classes that implement a single interface.
- Create classes that implement multiple interfaces.
- Implement the **IComparable** interface.
- Implement the **IComparer** interface.

Introducing Interfaces

In Visual C#, an interface specifies a set of characteristics and behaviors; it does this by defining methods, properties, events, and indexers. The interface itself does not specify how these members are implemented. Instead, classes can *implement* the interface and provide their own implementations of the interface members. You can think of an interface as a contract. By implementing a particular interface, a class guarantees to consumers that it will provide specific functionality through specific members, even though the actual implementation is not part of the contract.

- Interfaces define a set of characteristics and behaviors
 - Member signatures only
 - No implementation details
 - Cannot be instantiated
- Interfaces are implemented by classes or structs
 - Implementing class or struct must implement every member
 - Implementation details do not matter to consumers
 - Member signatures must match definitions in interface
- By implementing an interface, a class or struct guarantees that it will provide certain functionality

For example, suppose that you want to develop a loyalty card scheme for Fourth Coffee. You might start by creating an interface named **ILoyaltyCardHolder** that defines:

- A read-only integer property named **TotalPoints**.
- A method named **AddPoints** that accepts a decimal argument.
- A method named **ResetPoints**.

The following example shows an interface that defines one read-only property and two methods:

Defining an Interface

```
public interface ILoyaltyCardHolder
{
    int TotalPoints { get; }
    int AddPoints(decimal transactionValue);
    void ResetPoints();
}
```

 **Note:** Programming convention dictates that all interface names should begin with an "I".

Notice that the methods in the interface do not include method bodies. Similarly, the properties in the interface indicate which accessors to include but do not provide any implementation details. The interface simply states that any implementing class must include and provide an implementation for the three members. The creator of the implementing class can choose how the methods are implemented. For example, any implementation of the **AddPoints** method will accept a decimal argument (the cash value of the customer transaction) and return an integer (the number of points added). The class developer could implement this method in a variety of ways. For example, an implementation of the **AddPoints** method could:

- Calculate the number of points to add by multiplying the transaction value by a fixed amount.
- Get the number of points to add by calling a service.
- Calculate the number of points to add by using additional factors, such as the location of the loyalty cardholder.

The following example shows a class that implements the **ILoyaltyCardHolder** interface:

Implementing an Interface

```
public class Customer : ILoyaltyCardHolder
{
    private int totalPoints;
    public int TotalPoints
    {
        get { return totalPoints; }
    }

    public int AddPoints(decimal transactionValue)
    {
        int points = Decimal.ToInt32(transactionValue);
        totalPoints += points;
    }

    public void ResetPoints()
    {
        totalPoints = 0;
    }

    // Other members of the Customer class.
}
```

The details of the implementation do not matter to calling classes. By implementing the **ILoyaltyCardHolder** interface, the implementing class is indicating to consumers that it will take care of the **AddPoints** operation. One of the key advantages of interfaces is that they enable you to modularize your code. You can change the way in which your class implements the interface at any point, without having to update any consumer classes that rely on an interface implementation.

Defining Interfaces

The syntax for defining an interface is similar to the syntax for defining a class.

You use the **interface** keyword to declare an interface, as shown by the following example:

Declaring an Interface

```
public interface IBeverage
{
    // Methods, properties, events, and indexers go here.
}
```

- Use the **interface** keyword

```
public interface IBeverage
{
    // Methods, properties, events, and indexers.
}
```

- Specify an access modifier:

- public
- internal

- Add interface members:

- Methods, properties, events, and indexers
- Signatures only, no implementation details

Similar to a class declaration, an interface declaration can include an *access modifier*. You can use the following access modifiers in your interface declarations:

| Access modifier | Description |
|-----------------|---|
| public | The interface is available to code running in any assembly. |
| internal | The interface is available to any code within the same assembly, but not available to code in another assembly. This is the default value if you do not specify an access modifier. |

Adding Interface Members

An interface defines the signature of members but does not include any implementation details. Interfaces can include methods, properties, events, and indexers:

- To define a method, you specify the name of the method, the return type, and any parameters:

```
int GetServingTemperature(bool includesMilk);
```

- To define a property, you specify the name of the property, the type of the property, and the property accessors:

```
bool IsFairTrade { get; set; }
```

- To define an event, you use the **event** keyword, followed by the event handler delegate, followed by the name of the event:

```
event EventHandler OnSoldOut;
```

- To define an indexer, you specify the return type and the accessors:

```
string this[int index] { get; set; }
```

Interface members do not include access modifiers. The purpose of the interface is to define the members that an implementing class should expose to consumers, so that all interface members are public. Interfaces cannot include members that relate to the internal functionality of a class, such as fields, constants, operators, and constructors.

Implementing Interfaces

To create a class that implements an interface, you add a colon followed by the name of the interface to your class declaration.

The following example shows how to create a class that implements the **IBeverage** interface:

Declaring a Class that Implements an Interface

```
public class Coffee : IBeverage
{
}
```

Within your class, you must provide an implementation for every member of the interface. Your class can include additional members that are not defined by the interface. In fact, most classes will include additional members, because generally the class *extends* the interface. However, you cannot omit any interface members from the implementing class. The way you implement the interface members does not matter, as long as your implementations have the same *signatures* (that is, the same names, types, return types, and parameters) as the member definitions in the interface.

The following example shows a trivial interface, together with a class that implements the interface:

Implementing an Interface

```
public interface IBeverage
{
    int GetServingTemperature(bool includesMilk);
    bool IsFairTrade { get; set; }
}

public class Coffee : IBeverage
{
    private int servingTempWithoutMilk { get; set; }
    private int servingTempWithMilk { get; set; }
    public int GetServingTemperature(bool includesMilk)
    {
        if(includesMilk)
        {
            return servingTempWithMilk;
        }
        else
        {
            return servingTempWithoutMilk;
        }
    }
    public bool IsFairTrade { get; set; }

    // Other non-interface members go here.
}
```

- Add the name of the interface to the class declaration
public class Coffee : IBeverage
- Implement all interface members
- Use the interface type and the derived class type interchangeably
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();

The **coffee2** variable will only expose members defined by the **IBeverage** interface

Interface Polymorphism

As it relates to interfaces, polymorphism states that you can represent an instance of a class as an instance of any interface that the class implements. Interface polymorphism can help to increase the flexibility and modularity of your code. Suppose you have several classes that implement the **IBeverage** interface, such as **Coffee**, **Tea**, **Juice**, and so on. You can write code that works with any of these classes as instances of **IBeverage**, without knowing any details of the implementing class. For example, you can build a collection of **IBeverage** instances without needing to know the details of every class that implements **IBeverage**.

For example, If the **Coffee** class implements the **IBeverage** interface, you can represent a new **Coffee** object as an instance of **Coffee** or an instance of **IBeverage**:

Representing an Object as an Interface Type

```
Coffee coffee1 = new Coffee();
IBeverage coffee2 = new Coffee();
```

You can use an implicit cast to convert to an interface type, because you know that the class must include all the interface members.

Casting to an Interface Type

```
IBeverage beverage = coffee1;
```

You must use an explicit cast to convert from an interface type to a derived class type, as the class may include members that are not defined in the interface.

Casting an Interface Type to a Derived Class Type

```
Coffee coffee3 = beverage as Coffee;
// OR
Coffee coffee4 = (Coffee)beverage;
```

Implementing Multiple Interfaces

In many cases, you will want to create classes that implement more than one interface. For example, you might want to:

- Implement the **IDisposable** interface to enable the .NET runtime to dispose of your class correctly.
- Implement the **IComparable** interface to enable collection classes to sort instances of your class.
- Implement your own custom interface to define the functionality of your class.

- Add the names of each interface to the class declaration

```
public class Coffee : IBeverage, IInventoryItem
```
- Implement every member of every interface
- Use explicit implementation if two interfaces have a member with the same name

```
// This is an implicit implementation.
public bool IsFairTrade { get; set; }
```



```
//These are explicit implementations.
public bool IInventoryItem.IsFairTrade { get; }
public bool IBeverage.IsFairTrade { get; set; }
```

To implement multiple interfaces, you add a comma-separated list of the interfaces that you want to implement to your class declaration. Your class must implement every member of every interface you add to your class declaration.

The following example shows how to create a class that implements multiple interfaces:

Declaring a Class that Implements Multiple Interfaces

```
public class Coffee: IBeverage, IInventoryItem
{}
```

MCIT USE ONLY STUDENT USE PROhibited

Implicit and Explicit Implementation

When you create a class that implements an interface, you can choose whether to implement the interface implicitly or explicitly. To implement an interface implicitly, you implement each interface member with a signature that matches the member definition in the interface. To implement an interface explicitly, you fully qualify each member name so that it is clear that the member belongs to a particular interface.

The following example shows an explicit implementation of the **IBeverage** interface:

Implementing an Interface Explicitly

```
public class Coffee : IBeverage
{
    private int servingTempWithoutMilk { get; set; }
    private int servingTempWithMilk { get; set; }
    public int IBeverage.GetServingTemperature(bool includesMilk)
    {
        if(includesMilk)
        {
            return servingTempWithMilk;
        }
        else
        {
            return servingTempWithoutMilk;
        }
    }
    public bool IBeverage.IsFairTrade { get; set; }

    // Other non-interface members.
}
```

Some developers prefer explicit interface implementation because doing so can make the code easier to understand. The only scenario in which you must use explicit interface implementation is if you are implementing two interfaces that share a member name. For example, if you implement interfaces named **IBeverage** and **IInventoryItem**, and both interfaces declare a Boolean property named **IsAvailable**, you would need to implement at least one of the **IsAvailable** members explicitly. In this scenario, the compiler would be unable to resolve the **IsAvailable** reference without an explicit implementation.

There is a difference in the usage of the class, however. When implementing an interface implicitly the members can be used as normal public members of the class and can be referenced without any special designation. When an interface is implemented explicitly, though, the members are made only available by casting the instance to the corresponding interface. Otherwise, they'll remain hidden, and the code calling them will not compile.

The following example shows the usage of an explicit implementation of the **IBeverage** interface. Assume the **Coffee** class is the same as the previous example.

Consuming an Explicit Interface implementation

```
Coffee myCoffee = new Coffee();

// This will not compile
myCoffee.GetServingTemperature(true)

IBeverage myBeverage = (IBeverage)myCoffee;

// This will work as expected
myBeverage.GetServingTemperature(true)
```

Implementing the **IComparable** Interface

The .NET Framework includes various collection classes that enable you to sort the contents of the collection. These classes, such as the **ArrayList** class, include a method named **Sort**. When you call this method on an **ArrayList** instance, the collection orders its contents.

How does the **ArrayList** instance know how items in the collection should be ordered? In the case of simple types, such as integers, this appears fairly straightforward. Intuitively, three follows two and two follows one. However, suppose you create a collection of **Coffee** objects. How would the

ArrayList instance determine whether one coffee is greater or lesser than another coffee? The answer is that the **Coffee** class needs to provide the **ArrayList** instance with logic that enables it to compare one coffee with another. To do this, the **Coffee** class must implement the **IComparable** interface.

The following example shows the **IComparable** interface:

The **IComparable** Interface

```
public interface IComparable
{
    int CompareTo(Object obj);
}
```

As you can see, the **IComparable** interface declares a single method named **CompareTo**. Implementations of this method must:

- Compare the current object instance with another object of the same type (the argument).
- Return an integer value that indicates whether the current object instance should be placed before, in the same position, or after the passed-in object instance.

The integer values returned by the **CompareTo** method are interpreted as follows:

- Less than zero indicates that the current object instance precedes the supplied instance in the sort order.
- Zero indicates that the current object instance occurs at the same position as the supplied instance in the sort order.
- More than zero indicates that the current object instance follows the supplied instance in the sort order.

The following example illustrates what happens if you use the **CompareTo** method to compare two integers:

CompareTo Example

```
int number1 = 5;
int number2 = 100;
int result = number1.CompareTo(number2);
// The value of result is -1, indicating that number1 should precede number2 in the sort order.
```

- If you want instances of your class to be sortable in collections, implement the **IComparable** interface

```
public interface IComparable
{
    int CompareTo(Object obj);
}
```

- The **ArrayList.Sort** method calls the **IComparable.CompareTo** method on collection members to sort items in a collection

 **Note:** All the built-in value types in the .NET Framework implement the **IComparable** interface. For more information about the **IComparable** interface, see the **IComparable** Interface page at <https://aka.ms/moc-20483c-m4-pg1>.

When you call the **Sort** method on an **ArrayList** instance, the **Sort** method calls the **CompareTo** method of the collection members to determine the correct order for the collection.

When you implement the **IComparable** interface in your own classes, you determine the criteria by which objects should be compared. For example, you might decide that coffees should be sorted alphabetically by variety.

The following example shows how to implement the **IComparable** interface:

Implementing the **IComparable** Interface

```
public class Coffee : IComparable
{
    public double AverageRating { get; set; }
    public string Variety { get; set; }

    int IComparable.CompareTo(object obj)
    {
        Coffee coffee2 = obj as Coffee;
        return String.Compare(this.Variety, coffee2.Variety);
    }
}
```

In this example, because the values we want to compare are strings, we can use the **String.Compare** method. This method returns -1 if the current string precedes the supplied string in an alphabetical sort order, 0 if the strings are identical, and 1 if the current string follows the supplied string in an alphabetical sort order.

Implementing the **IComparer** Interface

When you call the **Sort** method on an **ArrayList** instance, the **ArrayList** sorts the collection based on the **IComparable** interface implementation in the underlying type. For example, if you sort an **ArrayList** collection of integers, the sort criteria is defined by the **IComparable** interface implementation in the **Int32** type. The creator of the **ArrayList** instance has no control over the criteria that are used to sort the collection.

In some cases, developers may want to sort instances of your class using alternative sort criteria. For example, suppose you want to sort a collection of **Coffee** instances by the value of the **AverageRating** property rather than the **Variety** property. To sort an **ArrayList** instance by using custom sort criteria, you need to do two things:

1. Create a class that implements the **IComparer** interface to provide your custom sort functionality.
2. Call the **Sort** method on the **ArrayList** instance, and pass in an instance of your **IComparer** implementation as a parameter.

The following example shows the **IComparer** interface:

- To sort collections by custom criteria, implement the **IComparer** interface

```
public interface IComparer
{
    int Compare(Object x, Object y);
}
```

- To use an **IComparer** implementation to sort an **ArrayList**, pass an **IComparer** instance to the **ArrayList.Sort** method

```
ArrayList coffeeList = new ArrayList();
// Add some items to the collection.
coffeeList.Sort(new CoffeeRatingComparer());
```

The **IComparer** Interface

```
public interface IComparer
{
    int Compare(Object x, Object y)
}
```

As you can see, the **IComparer** interface declares a single method named **Compare**. Implementations of this method must:

- Compare two objects of the same type.
- Return an integer value that indicates whether the current object instance should be placed before, in the same position, or after the passed-in object instance.

The following example shows how to implement the **IComparer** interface:

Implementing the **IComparer** Interface

```
public class CoffeeRatingComparer : IComparer
{
    public int Compare(Object x, Object y)
    {
        Coffee coffee1 = x as Coffee;
        Coffee coffee2 = y as Coffee;
        double rating1 = coffee1.AverageRating;
        double rating2 = coffee2.AverageRating;
        return rating1.CompareTo(rating2);
    }
}
```

In the above example, because the values we want to compare are doubles, we can make use of the **Double.CompareTo** method. This returns -1 if the current double is less than the supplied double, 0 if the current double is equal to the supplied double, and 1 if the current double is greater than the supplied double. It is always better to make use of a built-in comparison function, if one exists, rather than creating your own.

The following example shows how to use a custom **IComparer** implementation:

Using an **IComparer** Implementation

```
// Create some instances of the Coffee class.
Coffee coffee1 = new Coffee();
coffee1.Rating = 4.5;
Coffee coffee2 = new Coffee();
coffee2.Rating = 8.1;
Coffee coffee3 = new Coffee();
coffee3.Rating = 7.1;

// Add the Coffee instances to an ArrayList.
ArrayList coffeeList = new ArrayList();
coffeeList.Add(coffee1);
coffeeList.Add(coffee2);
coffeeList.Add(coffee3);

// Sort the ArrayList by average rating.
coffeeList.Sort(new CoffeeRatingComparer());
```

To sort the **ArrayList** using a custom comparer, you call the **Sort** method and pass in a new instance of your **IComparer** implementation as an argument.

Lesson 3

Implementing Type-Safe Collections

Almost every application that you create will use collection classes in one form or another. In most cases, collections contain a set of objects of the same type. When you interact with a collection, you often rely on the collection to provide objects of a specific type. Historically, this created various challenges. You had to create exception handling logic in case a collection contained items of the wrong type. You also had to box value types in order to add them to collection classes, and unbox them on retrieval. Visual C# removes many of these challenges by using *generics*.

In this lesson, you will learn how to create and use generic classes to create strongly typed collections of any type.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe generics.
- Identify the advantages of generic classes over non-generic classes.
- Apply constraints to type parameters.
- Use generic list collections.
- Use generic dictionary collections.
- Create custom generic collections.
- Create enumerable generic collections.

Introducing Generics

Generics enable you to create and use strongly typed collections that are type safe, do not require you to cast items, and do not require you to box and unbox value types. Generic classes work by including a type parameter, T, in the class or interface declaration. You do not need to specify the type of T until you instantiate the class. To create a generic class, you need to:

- Add the type parameter T in angle brackets after the class name.
- Use the type parameter T in place of type names in your class members.

- Create classes and interfaces that include a type parameter

```
public class CustomList<T>
{
    public T this[int index] { get; set; }
    public void Add(T item) { ... }
    public void Remove(T item) { ... }
}
```

- Specify the type argument when you instantiate the class

```
CustomList<Coffee> coffees =
    new CustomList<Coffee>();
```

The following example shows how to create a generic class:

Creating a Generic Class

```
public class CustomList<T>
{
    public T this[int index] { get; set; }
    public void Add(T item)
    {
        // Method logic goes here.
    }
    public void Remove(T item)
    {
        // Method logic goes here.
    }
}
```

When you create an instance of your generic class, you specify the type you want to supply as a type parameter. For example, if you want to use your custom list to store objects of type **Coffee**, you would supply **Coffee** as the type parameter.

The following example shows how to instantiate a generic class:

Instantiating a Generic Class

```
CustomList<Coffee> clc = new CustomList<Coffee>;
 
Coffee coffee1 = new Coffee();
Coffee coffee2 = new Coffee();

clc.Add(coffee1);
clc.Add(coffee2);

Coffee firstCoffee = clc[0];
```

When you instantiate a class, every instance of **T** within the class is effectively replaced with the type parameter you supply. For example, if you instantiate the **CustomList** class with a type parameter of **Coffee**:

- The **Add** method will only accept an argument of type **Coffee**.
- The **Remove** method will only accept an argument of type **Coffee**.
- The indexer will always provide a return value of type **Coffee**.

Advantages of Generics

The use of generic classes, particularly for collections, offers three main advantages over non-generic approaches: type safety, no casting, and no boxing and unboxing.

Type Safety

Consider an example where you use an **ArrayList** to store a collection of **Coffee** objects. You can add objects of any type to an **ArrayList**. Suppose a developer adds an object of type **Tea** to the collection. The code will build without complaint. However, a runtime exception will occur if the **Sort** method is called, because the collection is unable to compare objects of different types. Furthermore, when you retrieve an object from the

Generic types offer three advantages over non-generic types:

- Type safety
- No casting
- No boxing and unboxing

collection, you must cast the object to the correct type. If you attempt to cast the object to the wrong type, an invalid cast runtime exception will occur.

The following example shows the type safety limitations of the **ArrayList** approach:

Type Safety Limitations for Non-Generic Collections

```
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var arrayList1 = new ArrayList();
arrayList1.Add(coffee1);
arrayList1.Add(coffee2);
arrayList1.Add(tea1);

// The Sort method throws a runtime exception because the collection is not homogenous.
arrayList1.Sort();

// The cast throws a runtime exception because you cannot cast a Tea instance to a Coffee
instance.
Coffee coffee3 = (Coffee)arrayList1[2];
```

As an alternative to the **ArrayList**, suppose you use a generic **List<T>** to store a collection of **Coffee** objects. When you instantiate the list, you provide a type argument of **Coffee**. In this case, your list is guaranteed to be homogenous, because your code will not build if you attempt to add an object of any other type. The **Sort** method will work because your collection is homogenous. Finally, the indexer returns objects of type **Coffee**, rather than **System.Object**, so there is no risk of invalid cast exceptions.

The following example shows an alternative to the **ArrayList** approach using the generic **List<T>** class:

Type Safety in Generic Collections

```
var coffee1 = new Coffee();
var coffee2 = new Coffee();
var tea1 = new Tea();
var genericList1 = new List<Coffee>();
genericList1.Add(coffee1);
genericList1.Add(coffee2);

// This line causes a build error, as the argument is not of type Coffee.
genericList1.Add(tea1);

// The Sort method will work because the collection is guaranteed to be homogenous.
arrayList1.Sort();

// The indexer returns objects of type Coffee, so there is no need to cast the return
value.
Coffee coffee3 = genericList[1];
```

No Casting

Casting is a computationally expensive process. When you add items to an **ArrayList**, your items are implicitly cast to the **System.Object** type. When you retrieve items from an **ArrayList**, you must explicitly cast them back to their original type. Using generics to add and retrieve items without casting improves the performance of your application.

No Boxing and Unboxing

If you want to store value types in an **ArrayList**, the items must be boxed when they are added to the collection and unboxed when they are retrieved. Boxing and unboxing incurs a large computational cost and can significantly slow your applications, especially when you iterate over large collections. By contrast, you can add value types to generic lists without boxing and unboxing the value.

The following example shows the difference between generic and non-generic collections with regard to boxing and unboxing:

Boxing and Unboxing: Generic vs. Non-Generic Collections

```
int number1 = 1;
var arrayList1 = new ArrayList();

// This statement boxes the Int32 value as a System.Object.
arrayList1.Add(number1);

// This statement unboxes the Int32 value.
int number2 = (int)arrayList1[0];

var genericList1 = new List<Int32>();

//This statement adds an Int32 value without boxing.
genericList1.Add(number1);

//This statement retrieves the Int32 value without unboxing.
int number3 = genericList1[0];
```

Constraining Generics

In some cases, you may need to restrict the types that developers can supply as arguments when they instantiate your generic class. The nature of these constraints will depend on the logic you implement in your generic class. For example, if a collection class uses a property named

AverageRating to sort the items in a collection, you would need to constrain the type parameter to classes that include the **AverageRating** property. Suppose the **AverageRating** property is defined by the **IBeverage** interface. To implement this restriction, you would constrain the type parameter to classes that implement the **IBeverage** interface by using the **where** keyword.

The following example shows how to constrain a type parameter to classes that implement a particular interface:

Constraining Type Parameters by Interface

```
public class CustomList<T> where T : IBeverage
{
}
```

You can constrain type parameters in six ways:

- where T : <name of interface>
- where T : <name of base class>
- where T : U
- where T : new()
- where T : struct
- where T : class

You can apply the following six types of constraint to type parameters:

| Constraint | Description |
|---|--|
| where T : <name of interface> | The type argument must be, or implement, the specified interface. |
| where T : <name of base class> | The type argument must be, or derive from, the specified class. |
| where T : U | The type argument must be, or derive from, the supplied type argument U. |
| where T : new() | The type argument must have a public default constructor. |
| where T : struct | The type argument must be a value type. |
| where T : class | The type argument must be a reference type. |

You can also apply multiple constraints to the same class, as shown by the following example:

Apply Multiple Type Constraints

```
public class CustomList<T> where T : IBeverage, IComparable<T>, new()
{
}
```

Using Generic List Collections

One of the most common and important uses of generics is in collection classes. Generic collections fall into two broad categories: generic list collections and generic dictionary collections. A generic list stores a collection of objects of type T.

The List<T> Class

The **List<T>** class provides a strongly-typed alternative to the **ArrayList** class. Like the **ArrayList** class, the **List<T>** class includes methods to:

- Add an item.
- Remove an item.
- Insert an item at a specified index.
- Sort the items in the collection by using the default comparer or a specified comparer.
- Reorder all or part of the collection.

Generic list classes store collections of objects of type T:

- **List<T>** is a general purpose generic list
- **LinkedList<T>** is a generic list in which each item is linked to the previous item and the next item in the collection
- **Stack<T>** is a last in, first out collection
- **Queue<T>** is a first in, first out collection

The following example shows how to use the **List<T>** class.

Using the List<T> Class

```
string s1 = "Latte";
string s2 = "Espresso";
string s3 = "Americano";
string s4 = "Cappuccino";
string s5 = "Mocha";

// Add the items to a strongly-typed collection.
var coffeeBeverages = new List<String>();
coffeeBeverages.Add(s1);
coffeeBeverages.Add(s2);
coffeeBeverages.Add(s3);
coffeeBeverages.Add(s4);
coffeeBeverages.Add(s5);

// Sort the items using the default comparer.
// For objects of type String, the default comparer sorts the items alphabetically.
coffeeBeverages.Sort();

// Write the collection to a console window.
foreach(String coffeeBeverage in coffeeBeverages)
{
    Console.WriteLine(coffeeBeverage);
}
Console.ReadLine ("Press Enter to continue");
```

Other Generic List Classes

The **System.Collections.Generic** namespace also includes various generic collections that provide more specialized functionality:

- The **LinkedList<T>** class provides a generic collection in which each item is linked to the previous item in the collection and the next item in the collection. Each item in the collection is represented by a **LinkedListNode<T>** object, which contains a value of type T, a reference to the parent **LinkedList<T>** instance, a reference to the previous item in the collection, and a reference to the next item in the collection.
- The **Queue<T>** class represents a strongly typed first in, last out collection of objects.
- The **Stack<T>** class represents a strongly typed last in, last out collection of objects.

Using Generic Dictionary Collections

Dictionary classes store collections of key value pairs. The value is the object you want to store, and the key is the object you use to index and retrieve the value. For example, you might use a dictionary class to store coffee recipes, where the key is the name of the coffee and the value is the recipe for that coffee. In the case of generic dictionaries, both the key and the value are strongly typed.

- Generic dictionary classes store key-value pairs
- Both the key and the value are strongly typed
- **Dictionary< TKey, TValue >** is a general purpose, generic dictionary class
- **SortedList< TKey, TValue >** and **SortedDictionary< TKey, TValue >** collections are sorted by key

MCT USE ONLY - STUDIO - IS PROHIBITED

The Dictionary<TKey, TValue> Class

The **Dictionary<TKey, TValue>** class provides a general purpose, strongly typed dictionary class. You can add duplicate values to the collection, but the keys must be unique. The class will throw an **ArgumentException** if you attempt to add a key that already exists in the dictionary.

The following example shows how to use the **Dictionary<TKey, TValue>** class:

Using the Dictionary<TKey, TValue> Class

```
// Create a new dictionary of strings with string keys.
var coffeeCodes = new Dictionary<String, String>();

// Add some entries to the dictionary.
coffeeCodes.Add("CAL", "Café Au Lait");
coffeeCodes.Add("CSM", "Cinammon Spice Mocha");
coffeeCodes.Add("ER", "Espresso Romano");
coffeeCodes.Add("RM", "Raspberry Mocha");
coffeeCodes.Add("IC", "Iced Coffee");

// This statement would result in an ArgumentException because the key already exists.
// coffeeCodes.Add("IC", "Instant Coffee");

// To retrieve the value associated with a key, you can use the indexer.
// This will throw a KeyNotFoundException if the key does not exist.
Console.WriteLine("The value associated with the key \"CAL\" is {0}",
coffeeCodes["CAL"]);

// Alternatively, you can use the TryGetValue method.
// This returns true if the key exists and false if the key does not exist.
string csmValue = "";
if(coffeeCodes.TryGetValue("CSM", out csmValue))
{
    Console.WriteLine("The value associated with the key \"CSM\" is {0}", csmValue);
}
else
{
    Console.WriteLine("The key \"CSM\" was not found");
}

// You can also use the indexer to change the value associated with a key.
coffeeCodes["IC"] = "Instant Coffee";
```

Other Generic Dictionary Classes

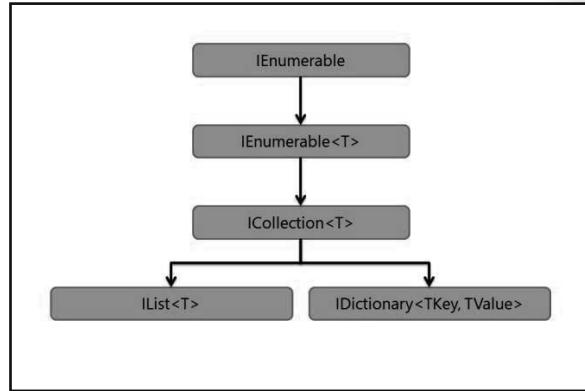
The **SortedList<TKey, TValue>** and **SortedDictionary<TKey, TValue>** classes both provide generic dictionaries in which the entries are sorted by key. The difference between these classes is in the underlying implementation:

- The **SortedList** generic class uses less memory than the **SortedDictionary** generic class.
- The **SortedDictionary** class is faster and more efficient at inserting and removing unsorted data.

Using Collection Interfaces

The **System.Collections.Generic** namespace provides a range of generic collections to suit various scenarios. However, there will be circumstances when you will want to create your own generic collection classes in order to provide more specialized functionality. For example, you might want to store data in a tree structure or create a circular linked list.

Where should you start when you want to create a custom collection class? All collections have certain things in common. For example, you will typically want to be able to enumerate the items in the collection by using a **foreach** loop, and you will need methods to add items, remove items, and clear the list. The picture on the slide shows that the .NET Framework provides a hierarchical set of interfaces that define the characteristics and behaviors of collections. These interfaces build on one another to define progressively more specific functionality.



The **IEnumerable** and **IEnumerable<T>** Interfaces

If you want to be able to use a **foreach** loop to enumerate over the items in your custom generic collection, you must implement the **IEnumerable<T>** interface. The **IEnumerable<T>** method defines a single method named **GetEnumerator()**. This method must return an object of type **IEnumerator<T>**. The **foreach** statement relies on this enumerator object to iterate through the collection.

The **IEnumerable<T>** interface *inherits* from the **IEnumerable** interface, which also defines a single method named **GetEnumerator()**. When an interface inherits from another interface, it exposes all the members of the parent interface. In other words, if you implement **IEnumerable<T>**, you also need to implement **IEnumerable**.

The **ICollection<T>** Interface

The **ICollection<T>** interface defines the basic functionality that is common to all generic collections. The interface inherits from **IEnumerable<T>**, which means that if you want to implement **ICollection<T>**, you must also implement the members defined by **IEnumerable<T>** and **IEnumerable**.

The **ICollection<T>** interface defines the following methods:

| Name | Description |
|-----------------|---|
| Add | Adds an item of type T to the collection. |
| Clear | Removes all items from the collection. |
| Contains | Indicates whether the collection contains a specific value. |
| CopyTo | Copies the items in the collection to an array. |
| Remove | Removes a specific object from the collection. |

The **ICollection<T>** interface defines the following properties:

| Name | Description |
|-------------------|--|
| Count | Gets the number of items in the collection. |
| IsReadOnly | Indicates whether the collection is read-only. |

The **IList<T>** Interface

The **IList<T>** interface defines the core functionality for generic list classes. You should implement this interface if you are defining a linear collection of single values. In addition to the members it inherits from **ICollection<T>**, the **IList<T>** interface defines methods and properties that enable you to use indexers to work with the items in the collection. For example, if you create a list named **myList**, you can use **myList[0]** to access the first item in the collection.

The **IList<T>** interface defines the following methods:

| Name | Description |
|-----------------|--|
| Insert | Inserts an item into the collection at the specified index. |
| RemoveAt | Removes the item at the specified index from the collection. |

The **IList<T>** interface defines the following properties:

| Name | Description |
|----------------|--|
| IndexOf | Determines the position of a specified item in the collection. |

The **IDictionary< TKey, TValue >** Interface

The **IDictionary< TKey, TValue >** interface defines the core functionality for generic dictionary classes. You should implement this interface if you are defining a collection of key-value pairs. In addition to the members it inherits from **ICollection<T>**, the **IDictionary<T>** interface defines methods and properties that are specific to working with key-value pairs.

The **IDictionary< TKey, TValue >** interface defines the following methods:

| Name | Description |
|----------------------|---|
| Add | Adds an item with the specified key and value to the collection. |
| ContainsKey | Indicates whether the collection includes a key-value pair with the specified key. |
| GetEnumerator | Returns an enumerator of KeyValuePair< TKey, TValue > objects. |
| Remove | Removes the item with the specified key from the collection. |
| TryGetValue | Attempts to set the value of an output parameter to the value associated with a specified key. If the key exists, the method returns true. If the key does not exist, the method returns false and the output parameter is unchanged. |

The **IDictionary< TKey, TValue >** interface defines the following properties:

| Name | Description |
|---------------|--|
| Item | Gets or sets the value of an item in the collection, based on a specified key. This property enables you to use indexer notation, for example myDictionary[myKey] = myValue . |
| Keys | Returns the keys in the collection as an ICollection< T > instance. |
| Values | Returns the values in the collection as an ICollection< T > instance. |



Reference Links: For comprehensive information and examples of all of the generic interfaces covered in this topic, see the System.Collections.Generic Namespace page at <https://aka.ms/moc-20483c-m4-pg2>.

Creating Enumerable Collections

To enumerate over a collection, you typically use a **foreach** loop. The **foreach** loop exposes each item in the collection in turn, in an order that is appropriate to the collection. The **foreach** statement masks some of the complexities of enumeration. For the **foreach** statement to work, a generic collection class must implement the **IEnumerable< T >** interface. This interface exposes a method, **GetEnumerator**, which must return an **IEnumerator< T >** type.

- Implement **IEnumerable< T >** to support enumeration (**foreach**)
- Implement the **GetEnumerator** method by either:
 - Creating an **IEnumerator< T >** implementation
 - Using an iterator
- Use the **yield return** statement to implement an iterator

The **IEnumerator< T >** Interface

The **IEnumerator< T >** interface defines the functionality that all enumerators must implement.

The **IEnumerator< T >** interface defines the following methods:

| Name | Description |
|-----------------|---|
| MoveNext | Advances the enumerator to the next item in the collection. |
| Reset | Sets the enumerator to its starting position, which is before the first item in the collection. |

MCT USE ONLY. STUDENT ICE PROHIBITED

The **IEnumerator<T>** interface defines the following properties:

| Name | Description |
|----------------|---|
| Current | Gets the item that the enumerator is pointing to. |

An enumerator is essentially a pointer to the items in the collection. The starting point for the pointer is before the first item. When you call the **MoveNext** method, the pointer advances to the next element in the collection. The **MoveNext** method returns **true** if the enumerator was able to advance one position, or **false** if it has reached the end of the collection. At any point during the enumeration, the **Current** property returns the item to which the enumerator is currently pointing.

When you create an enumerator, you must define:

- Which item the enumerator should treat as the first item in the collection.
- In what order the enumerator should move through the items in the collection.

The **IEnumerable<T>** Interface

The **IEnumerable<T>** interface defines a single method named **GetEnumerator**. This returns an **IEnumerator<T>** instance.

The **GetEnumerator** method returns the *default* enumerator for your collection class. This is the enumerator that a **foreach** loop will use, unless you specify an alternative. However, you can create additional methods to expose alternative enumerators.

The following example shows a custom collection class that implements a default enumerator, together with an alternative enumerator that enumerates the collection in reverse order:

Default and Alternative Enumerators

```
class CustomCollection<T> : IEnumerable<T>
{
    public Ienumerator<T> Backwards()
    {
        // This method returns an alternative enumerator.
        // The implementation details are not shown.
    }

    #region IEnumerable<T> Members
    public Ienumerator<T> GetEnumerator()
    {
        // This method returns the default enumerator.
        // The implementation details are not shown.
    }
    #endregion

    #region IEnumerable Members
    IEnumerator IEnumerable.GetEnumerator()
    {
        // This method is required because IEnumerable<T> inherits from IEnumerable
        throw new NotImplementedException();
    }
    #endregion
}
```

The following example shows how you can use a default enumerator or an alternative enumerator to iterate through a collection:

Enumerating a Collection

```
CustomCollection<Int32> numbers = new CustomCollection<Int32>();
// Add some items to the collection.
```

```
// Use the default enumerator to iterate through the collection:  
foreach (int number in numbers)  
{  
    // ...  
}  
  
// Use the alternative enumerator to iterate through the collection:  
foreach(int number in numbers.Backwards())  
{  
    // ...  
}
```

Implementing the Enumerator

You can provide an enumerator by creating a custom class that implements the **IEnumerator<T>** interface. However, if your custom collection class uses an underlying enumerable type to store data, you can use an *iterator* to implement the **IEnumerable<T>** interface without actually providing an **IEnumerator<T>** implementation. The best way to understand iterators is to start with a simple example.

The following example shows how you can use an iterator to implement an enumerator:

Implementing an Enumerator by Using an Iterator

```
using System;  
using System.Collections;  
using System.Collections.Generic;  
  
class BasicCollection<T> : IEnumerable<T>  
{  
    private List<T> data = new List<T>();  
  
    public void FillList(params T [] items)  
    {  
        foreach (var datum in items)  
            data.Add(datum);  
    }  
  
    IEnumerator<T> IEnumerable<T>.GetEnumerator()  
    {  
        foreach (var datum in data)  
        {  
            yield return datum;  
        }  
    }  
  
    IEnumerator IEnumerable.GetEnumerator()  
    {  
        throw new NotImplementedException();  
    }  
}
```

The example shows a custom generic collection class that uses a **List<T>** instance to store data. The **List<T>** instance is populated by the **FillList** method. When the **GetEnumerator** method is called, a **foreach** loop enumerates the underlying collection. Within the **foreach** loop, a **yield return** statement is used to return each item in the collection. It is this **yield return** statement that defines the iterator—essentially, the **yield return** statement pauses execution to return the current item to the caller before the next element in the sequence is retrieved. In this way, although the **GetEnumerator** method does not appear to return an **IEnumerator** type, the compiler is able to build an enumerator from the iteration logic that you provided.

Demonstration: Adding Data Validation and Type-Safety to the Application Lab

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 Data Validation and Type-Safety to the Application Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_DEMO.md.

Lab: Adding Data Validation and Type-Safety to the Application

Scenario

Now that the user interface navigation features are working, you decide to replace the simple structs with classes to make your application more efficient and straightforward.

You have also been asked to include validation logic in the application to ensure that when a user adds grades to a student, that the data is valid before it is written to the database. You decide to create a unit test project that will perform tests against the required validation for different grade scenarios.

Teachers who have seen the application have expressed concern that the students in their classes are displayed in a random order. You decide to use the **IComparable** interface to enable them to be displayed in alphabetical order.

Finally, you have been asked to add functionality to the application to enable teachers to add students to and remove students from a class, and to add student grades to the database.

Objectives

After completing this lab, you will be able to:

- Create classes.
- Write data validation code and unit tests.
- Implement the **IComparable** interface.
- Use generic collections.

Lab Setup

Estimated Time: **75 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD04_LAK.md.

Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes

Scenario

In this exercise, you will convert the existing Teacher, Student, and Grade structs into classes. This will enable you to implement the additional functionality required for each class, such as adding constructors, properties, and methods. In the Teacher and Student classes, you will make the password property write-only, add the **VerifyPassword** method, and then define their respective constructors. You will also modify the **Logon_Click** method to use the **VerifyPassword** method to verify passwords when a user logs on.

Finally, you will run the application and verify that it still functions correctly, allowing a student or a teacher to log on.

M
C
T
U
S
E
O
N
Y
S
T
U
D
E
N
T
U
S
E
P
R
O
H
I
B
I
T
E
D

Exercise 2: Adding Data Validation to the Grade Class

Scenario

In this exercise, you will define a public list of strings called **Subjects** to hold the names of each subject that can be assessed and then populate it with valid subject names. You will then add validation logic to the **Grade** class to ensure that the subject name appears in the list you created and that the assessment date and assessment grade contain allowed values. Finally, you will create a unit test project to verify that your validation code functions as expected.

Exercise 3: Displaying Students in Name Order

Scenario

In this exercise, you will write code to display the students in alphabetical order of last name and then first name.

The application currently displays students in no specific order when logged on as a teacher, but you now want them to be displayed in alphabetical order of last name and first name. To achieve this, you decide that the **Student** class should implement the **IComparable< >** interface to enable comparison of student data. You can then add code to the **CompareTo** method in the **Student** class, enabling students to be sorted based on their last name and first name. Currently, Students are stored in a non-type-safe **ArrayList** collection. You decide to modify this so they are stored in a type-safe **List** collection. Finally, you will sort the Students data and then run the application to verify that the students are retrieved and displayed in alphabetical order of their last name and first name.

Exercise 4: Enabling Teachers to Modify Class and Grade Data

Scenario

In this exercise, you will write code that enables a teacher to add a student and then enroll them in a class. This will be implemented as two separate steps, because a teacher may want to add a student before knowing which class they will be enrolled in. You will also enable a teacher to remove a student from a class. When adding or removing a student, you will display a prompt to confirm that the teacher wants to perform the action.

To enroll a student in a class or remove them from a class, you modify the **TeacherID** property of that student. The application now includes the **AssignStudentDialog** window, which displays a list of students who are not assigned to a class. You need to add code to this window to assign a student to the teacher's class and to update the list of students as appropriate. You also need to add code to remove a student from a class and to enable teachers to add grades to their students. After a student has been added to the database, that student will be able to log on to view their own grades.

Module Review and Takeaways

In this module, you have learned how to work with classes, interfaces, and generic collections in Visual C#.

Review Questions

Check Your Knowledge

| Question | |
|---|---------|
| Which of the following types is a reference type? | |
| Select the correct answer. | |
| <input type="checkbox"/> | Boolean |
| <input type="checkbox"/> | Byte |
| <input type="checkbox"/> | Decimal |
| <input type="checkbox"/> | Int32 |
| <input type="checkbox"/> | Object |

Check Your Knowledge

| Question | |
|--|------------|
| Which of the following types of member CANNOT be included in an interface? | |
| Select the correct answer. | |
| <input type="checkbox"/> | Events |
| <input type="checkbox"/> | Fields |
| <input type="checkbox"/> | Indexers |
| <input type="checkbox"/> | Methods |
| <input type="checkbox"/> | Properties |

MCT USE ONLY. STUDENT USE PROHIBITED

Check Your Knowledge

Question

You want to create a custom generic class. The class will consist of a linear collection of values, and will enable developers to queue items from either end of the collection. Which of the following should your class declaration resemble?

Select the correct answer.

- public class DoubleEndedQueue<T> : IEnumerable<T>
- public class DoubleEndedQueue<T> : ICollection<T>
- public class DoubleEndedQueue<T> : IList<T>
- public class DoubleEndedQueue<T> : IList<T>, IEnumerable<T>
- public class DoubleEndedQueue<T> : IDictionary<TKey, TValue>

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5

Creating a Class Hierarchy by Using Inheritance

Contents:

| | |
|--|------|
| Module Overview | 5-1 |
| Lesson 1: Creating Class Hierarchies | 5-2 |
| Lesson 2: Extending .NET Framework Classes | 5-11 |
| Lab: Refactoring Common Functionality into the User Class | 5-19 |
| Module Review and Takeaways | 5-21 |

Module Overview

The concept of inheritance is central to object-oriented programming in any language. It is also one of the most powerful tools in your developer toolbox. Essentially, inheritance enables you to create new classes by inheriting characteristics and behaviors from existing classes. When you inherit from an existing class and add some functionality of your own, your class becomes a more specialized instance of the existing class. Not only does this save you time by reducing the amount of code you need to write, it also enables you to create hierarchies of related classes that you can then use interchangeably, depending on your requirements.

In this module, you will learn how to use inheritance to create class hierarchies and to extend .NET Framework types.

Objectives

After completing this module, you will be able to:

- Create base classes and derived classes by using inheritance.
- Create classes that inherit from .NET Framework classes.

Lesson 1

Creating Class Hierarchies

Rather than creating new classes from nothing, in many cases you can use an existing class as a base for your new class. This is known as inheritance. Your class inherits all the members from the base class, and you simply include the functionality that you want to add to the base class's capabilities. This way, your class becomes a more specialized version of the base class. This concept of inheritance is one of the main pillars of object-oriented programming.

In this lesson, you will learn how to use inheritance to create class hierarchies.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe inheritance.
- Create base classes.
- Create base class members.
- Create classes that inherit from base classes.
- Call base class methods and constructors from within a derived class.

What Is Inheritance?

In Visual C#, a class can *inherit* from another class. When you create a class that inherits from another class, your class is known as the *derived class* and the class that you inherit from is known as the *base class*. The derived class inherits all the members of the base class, including constructors, methods, properties, fields, and events.

Inheritance enables you to build hierarchies of progressively more specialized classes. Rather than creating a class from nothing, you can inherit from a more general base class to provide a starting point for your functionality. Inheritance can help to simplify maintenance of your code.

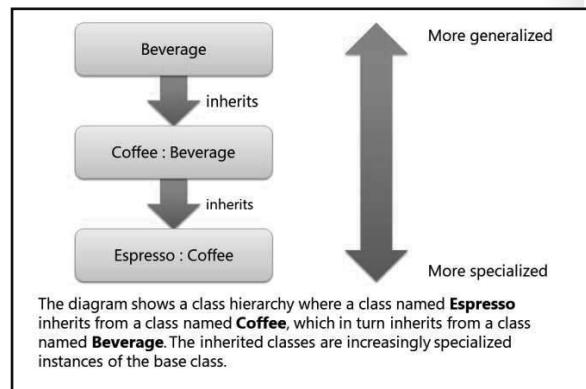
For example, you define a class named **Beverage**, as shown below:

The Beverage Class

```
public class Beverage
{
    protected int servingTemperature;

    public string Name { get; set; }
    public bool IsFairTrade { get; set; }

    public int GetServingTemperature()
    {
        return servingTemperature;
    }
}
```



Now you want to create a class to represent coffees. Coffee is a type of beverage. It shares all the characteristics and behaviors of a beverage. Rather than creating a class from scratch to represent coffees, you can create a class that inherits from the **Beverage** class. The derived class inherits all the members of the Beverage class, such as the **servingTemperature** field, the **Name** property, the **IsFairTrade** property, and the **GetServingTemperature** method. Within the derived class, you just need to add members that are specific to coffees.

The following example shows how to create a class for coffees that inherits from the **Beverage** class:

The Coffee Class

```
public class Coffee : Beverage
{
    public string Bean { get; set; }
    public string Roast { get; set; }
    public string CountryOfOrigin { get; set; }
}
```

 **Note:** In object-oriented programming, the terms *derives* and *inherits* are used interchangeably. Saying that the **Coffee** class *derives* from the **Beverage** class means the same as saying the **Coffee** class *inherits* from the **Beverage** class.

As you can see in the previous examples, the syntax for inheriting from a class is similar to the syntax for implementing an interface. This is because inheriting from a class and implementing an interface are both examples of inheritance. However, you do not need to duplicate the base class members in the derived class. By adding the base class to your class declaration, you make all the members of the base class available to consumers of your derived class.

The following example shows how to use base class members in a derived class:

Calling Base Class Members

```
Coffee coffee1 = new Coffee();

// Use base class members.
coffee1.Name = "Fourth Espresso";
coffee1.IsFairTrade = true;
int servingTemp = coffee1.GetServingTemperature();

// Use derived class members.
coffee1.Bean = "Arabica";
coffee1.Roast = "Dark";
coffee1.CountryOfOrigin = "Columbia";
```

As shown in the above example, you can call members of a base class in the same way that you call members of the class itself.

Creating Base Classes

When you create a class, you should consider whether you or other developers will need to use it as a base for derived classes. You have full control over whether, and how, your class can be inherited.

Abstract Classes and Members

As part of an object-oriented design, you may want to create classes that serve solely as base classes for other types. The base class may contain missing or incomplete functionality, and you may not want to enable code to instantiate your class directly. In these scenarios, you can add the **abstract** keyword to your class declaration.

The following example shows how to declare an abstract class:

Declaring an Abstract Class

```
abstract class Beverage
{
}
```

The **abstract** keyword indicates that the class cannot be instantiated directly; it exists solely to define or implement members for derived classes. If you attempt to instantiate an abstract class you will get an error when you build your code.

An abstract class can contain both abstract and non-abstract members. Abstract members are also declared with the **abstract** keyword and are conceptually similar to interface members, in that abstract members define the signature of the member but do not provide any implementation details. Any class that inherits from the abstract class must provide an implementation for the abstract members. Non-abstract members, however, can be used directly by derived classes.

The following example illustrates the difference between abstract and non-abstract members:

Abstract and Non-Abstract Members

```
abstract class Beverage
{
    // Non-abstract members.
    // Derived classes can use these members without modifying them.
    public string Name { get; set; }
    public bool IsFairTrade { get; set; }

    // Abstract members.
    // Derived classes must override and implement these members.
    public abstract int GetServingTemperature();
}
```



Note: You can only include abstract members within abstract classes. A non-abstract class cannot include abstract members.

Sealed Classes

You may want to create classes that cannot be inherited from. You can prevent developers from inheriting from your classes by marking the class with the **sealed** keyword.

The following example shows how to use the **sealed** modifier:

Creating a Sealed Class

```
public sealed class Tea : Beverage
{
    // ...
}
```

You can apply the **sealed** modifier to classes that inherit from other classes and classes that implement interfaces. You cannot use the **sealed** modifier and the **abstract** modifier on the same class, as a sealed class is the opposite of an abstract class—a sealed class cannot be inherited, whereas an abstract class must be inherited.

Any static class is also a sealed class. You can never inherit from a static class. Similarly, any static members within non-static classes are not inherited by derived classes.

Creating Base Class Members

You may want to implement a method in your base class, but allow derived classes to replace your method implementation with more specific functionality. To create a member that developers can override in a derived class, you use the **virtual** keyword.

- Use the **virtual** keyword to create members that you can override in derived classes

```
public virtual int GetServingTemperature()
```

- Use the **protected** access modifier to make members available to derived types

```
protected int servingTemperature;
```

The following example shows how to create a virtual method in a class:

Adding Virtual Members

```
public class Beverage
{
    protected int servingTemperature;

    public string Name { get; set; }
    public bool IsFairTrade { get; set; }

    public virtual int GetServingTemperature()
    {
        // This is the default implementation of the GetServingTemperature method.
        // Because the method is declared virtual, you can override the implementation in
        derived classes.
        return servingTemperature;
    }
}
```

When you create a class, you can use access modifiers to control whether the members of your class are accessible to derived types. You can use the following access modifiers for class members:

| Access Modifier | Details |
|------------------|--|
| public | The member is available to code running in any assembly. |
| protected | The member is available only within the containing class or in classes |

| Access Modifier | Details |
|---------------------------|---|
| | derived from the containing class. |
| internal | The member is available only to code within the current assembly. |
| protected internal | The member is available to any code within the current assembly, and to types derived from the containing class in any assembly. |
| private | The member is available only within the containing class. |
| private protected | The member is available to types derived from the containing class, but only within its containing assembly. (Only available since Visual C# 7.2) |

Members of a class are private by default. Private members are not inherited by derived classes. If you want members that would otherwise be private to be accessible to derived classes, you must prefix the member with the **protected** keyword.

Inheriting from a Base Class

To inherit from another class, you add a colon followed by the name of the base class to your class declaration.

The following example shows how to inherit from a base class:

Declaring a Class that Inherits From a Base Class

```
public class Coffee : Beverage
{
}
```

- To inherit from a base class, add the name of the base class to the class declaration
`public class Coffee : Beverage`
- To override virtual base class members, use the **override** keyword
`public override int GetServingTemperature()`
- To prevent classes further down the class hierarchy from overriding your override methods, use the **sealed** keyword
`sealed public override int GetServingTemperature()`

The derived class inherits every member from the base class. Within your derived class, you can add new members to extend the functionality of the base type.

A class can only inherit from one base class. However, your class can implement one or more interfaces in addition to deriving from a base type.

Overriding Base Class Members

In some cases you may want to change the way a base class member works in your derived class.

For example, the **Beverage** base class includes a method named **GetServingTemperature**:

Adding Virtual Members

```
public class Beverage
{
    protected int servingTemperature;

    public virtual int GetServingTemperature()
    {
        return servingTemperature;
    }

    // Other class members not shown.
}
```

Because **GetServingTemperature** is a virtual method, you can override it in a derived class. To override a virtual method in a derived class, you create a method with the same signature and prefix it with the **override** keyword.

The following example shows how to override a virtual method in a derived class:

Overriding a Virtual Method by Using the **override** Keyword

```
public class Coffee : Beverage
{
    protected bool includesMilk;
    private int servingTempWithMilk;
    private int servingTempWithoutMilk;

    public override int GetServingTemperature()
    {
        if(includesMilk) return servingTempWithMilk
        else return servingTempWithoutMilk;
    }
}
```

You can use the same approach to override properties, indexers, and events. In each case, you can only override a base class member if the member is marked as virtual in the base class. You cannot override constructors.

You can also override a base class member by using the **new** keyword:

Overriding a Virtual Method by Using the **new** Keyword

```
public class Coffee : Beverage
{
    public new int GetServingTemperature()
    {
        // ...
    }
}
```

When you use the **override** keyword, your method *extends* the base class method. By contrast, when you use the **new** keyword, your method *hides* the base class method. This causes subtle but important differences in how the compiler treats your base class and derived class types.

 **Note:** **Override** can be used for both **virtual** and **abstract** members. The difference is that overriding virtual members is optional, while overriding abstract members is mandatory. Only an **abstract** class must implement every **abstract** member defined by its ancestors.

 **Reference Links:** For a detailed explanation of the differences in behavior between the **override** keyword and the **new** keyword, see Knowing When to Use Override and New Keywords (C# Programming Guide) at <https://aka.ms/moc-20483c-m5-pg1>.

Sealing Overridden Members

When you override a base class member, you can prevent classes that derive from your class from overriding your implementation of the base class member by using the **sealed** keyword.

The following example shows how to seal an overridden base class member:

Sealing an Overridden Member

```
public class Coffee : Beverage
{
    sealed public override int GetServingTemperature()
    {
        // Derived classes cannot override this method.
    }
}
```

By sealing an overridden member, you force any classes that derive from your class to use your implementation of the base class member, rather than creating their own. This can be useful when you need to control the behavior of your classes and ensure that derived classes do not attempt to modify how specific members work.

You can only apply the **sealed** modifier to a member if the member is an **override** member. Remember that members are inherently sealed unless they are marked as **virtual**. In this case, because the base class method is marked as **virtual**, any descendants are able to override the method unless you seal it at some point in the class hierarchy.

Calling Base Class Constructors and Members

You can use the **base** keyword to access base class methods and constructors from within a derived class. This is useful in the following scenarios:

- You override a base class method, but you still want to execute the functionality in the base class method in addition to your own additional functionality.
- You create a new method, but you want to call a base class method as part of your method logic.
- You create a constructor and you want to call a base class constructor as part of your initialization logic.
- You want to call a base class method from a property accessor.

- To call a base class constructor from a derived class, add the **base** constructor to your constructor declaration

```
public Coffee(string name, bool isFairTrade, int temp)
    : base(name, isFairTrade, servingTemp)
```

- Pass parameter names to the base constructor as arguments
- Do not use the **base** keyword within the constructor body

- To call base class methods from a derived class, use the **base** keyword like an instance variable

```
base.GetServingTemperature();
```

Calling Base Class Constructors from a Derived Class

You cannot override constructors in derived classes. Instead, when you create constructors in a derived class, your constructors will automatically call the default base class constructor before they execute any of the logic that you have added. However, in some circumstances, you might want your constructors to call an alternative base class constructor. In these cases, you can use the **base** keyword in your constructor declarations to specify which base class constructor you want to call.

The following example shows how to call base class constructors:

Calling Base Class Constructors

```
public class Beverage
{
    public Beverage()
    {
        // Implementation details not shown.
    }
    public Beverage(string name, bool isFairTrade, int servingTemp)
    {
        // Implementation details are not shown.
    }

    // Other class members are not shown.
}

public class Coffee : Beverage
{
    public Coffee()
    {
        // This constructor implicitly calls the default Beverage constructor.
        // The declaration is implicitly equivalent to public Coffee() : base()
        // You can include additional code here.
    }

    public Coffee(string name, bool isFairTrade, int servingTemp, string bean, string
roast)
        : base(name, isFairTrade, servingTemp)
    {
        // This calls the Beverage(string, bool, int) constructor.
        // You can include additional code here:
        Bean = bean;
        Roast = roast;
    }

    public string Bean { get; set; }
    public string Roast { get; set; }
    public string CountryOfOrigin { get; set; }
}
```

As the previous example shows, you must call the base class constructor from your constructor declaration. You cannot call the base class constructor from within your constructor method body. You can use the named parameters from your constructor declaration as arguments to the base class constructor.

Calling Base Class Methods from a Derived Class

You can call base class methods from within method bodies or property accessors in a derived class. To do this, you use the **base** keyword as you would use an instance variable.

The following example shows how to call base class methods:

Calling Base Class Methods

```
public class Beverage
{
    protected int servingTemperature;
    public virtual int GetServingTemperature()
    {
        return servingTemperature;
    }

    // Constructors and additional class members are not shown.
}
```

```
public class Coffee : Beverage
{
    bool iced = false;
    protected int servingTempIced = 40;
    public override int GetServingTemperature()
    {
        if(iced)
        {
            return servingTempIced;
        }
        else
        {
            return base.GetServingTemperature();
        }
    }
}
```

Remember that the rules of inheritance do not apply to static classes and members. As such, you cannot use the **base** keyword within a static method.

Demonstration: Calling Base Class Constructors

In this demonstration, you will step through the execution of an application. The solution includes a class named **Coffee** that inherits from a class named **Beverage**. The **Coffee** class includes two constructors—one that implicitly calls the default base class constructor, and one that explicitly calls an alternative base class constructor. The application creates instances of the **Coffee** class by using both of these constructors. In both cases, you can observe how derived class constructors call base class constructors. You will also see how derived class constructors pass argument values to base class constructors.

Demonstration Steps

You will find the steps in the **Demonstration: Calling Base Class Constructors** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md.

Lesson 2

Extending .NET Framework Classes

The .NET Framework contains several thousand classes that provide a wide range of functionality. When you create your own classes, you should look to build on these classes by inheriting from .NET Framework types wherever possible. Not only does this reduce the amount of code you need to write, it also helps to ensure that your classes work in a standardized way.

The .NET Framework also enables you to create *extension methods* to add functionality to sealed .NET Framework types. This enables you to extend the functionality of built-in types, such as the **String** class, when the inheritance approach is not permitted.

In this lesson, you will learn how to extend .NET Framework types by using inheritance and extension methods.

Lesson Objectives

After completing this lesson, you will be able to:

- Create classes that inherit from .NET Framework types.
- Create custom exception classes.
- Throw and catch custom exceptions.
- Create classes that inherit from generic types.
- Create extension methods for .NET Framework types.

Inheriting from .NET Framework Classes

There are almost 15,000 public types in the .NET Framework. Although not all of these are extendable classes, many of them are. When you want to develop a class, in many cases there is a built-in .NET Framework class that can provide a foundation for your code.

There are two key advantages to creating a class that inherits from a .NET Framework class, rather than developing a class from scratch:

- **Reduced development time.** By inheriting from an existing class, you reduce the amount of logic that you have to create yourself.
- **Standardized functionality.** Just like implementing an interface, inheriting from a standard base class means that your class will work in a standardized way. You can also represent instances of your class as instances of the base class, which makes it easier for developers to use your class alongside other types that derive from the same base class.

- Inherit from .NET Framework classes to:
 - Reduce development time
 - Standardize functionality
- Inherit from any .NET Framework type that is not **sealed** or **static**
- Override any base class members that are marked as **virtual**
- Implement any base class members that are marked as **abstract**

The rules of inheritance apply to built-in .NET Framework classes in the same way they apply to custom classes:

- You can create a class that derives from a .NET Framework class, providing that the class is not *sealed* or *static*.
- You can override any base class members that are marked as *virtual*.

MCT USE ONLY. STUDENT LICENCE PROHIBITED

- If you inherit from an *abstract* class, you must provide implementations for all abstract members.

When you create a class, select a base class that minimizes the amount of coding and customization required. If you find yourself replicating functionality that is available in built-in classes, you should probably choose a more specific base class. On the other hand, if you find that you need to override several members, you should probably choose a more general base class.

For example, consider that you want to create a class that stores a linear list of values. The class must enable you to remove duplicate items from the list. Rather than creating a new list class from nothing, you can accomplish this by creating a class that inherits from the generic **List<T>** class and adding a single method to remove duplicate items. In addition, you can take advantage of the **Sort** method in the **List<T>** class. If you call the **Sort** method, any duplicate items will be adjacent in the collection, which can make it easier to identify and remove them.

The following example shows how to extend the **List<T>** class:

Extending a .NET Framework Class

```
public class UniqueList<T> : List<T>
{
    public void RemoveDuplicates()
    {
        base.Sort();
        for (int i = this.Count - 1; i > 0; i--)
        {
            if(this[i].Equals(this[i-1]))
            {
                this.RemoveAt(i);
            }
        }
    }
}
```

When you use this approach, consumers of your class have access to all the functionality provided by the base **List<T>** class. They also have access to the additional **RemoveDuplicates** method that you provided in your derived class.

Creating Custom Exceptions

The .NET Framework contains built-in exception classes to represent most common error conditions. For example:

- If you invoke a method with a null argument value, and the method cannot handle null argument values, the method will throw an **ArgumentNullException**.
- If you attempt to divide a numerical value by zero, the runtime will throw a **DivideByZeroException**.
- If you attempt to retrieve an indexed item from a collection, and the index is outside the bounds of the collection, the indexer will throw an **IndexOutOfRangeException**.

To create a custom exception type:

1. Inherit from the **System.Exception** class
2. Implement three standard constructors:
 - `base()`
 - `base(string message)`
 - `base(string message, Exception inner)`
3. Add additional members if required



Note: Most built-in exception classes are defined in the **System** namespace. For more information about the **System** namespace, see the System Namespace page at <https://aka.ms/moc-20483c-m5-pg2>.

When you need to throw exceptions in your code, you should reuse existing .NET Framework exception types wherever possible. However, there may be circumstances when you want to create your own custom exception types.

When Should You Create a Custom Exception Type?

You should consider creating a custom exception type when:

- Existing exception types do not adequately represent the error condition you are identifying.
- The exception requires very specific remedial action that differs from how you would handle built-in exception types.

Remember that the primary purpose of exception types is to enable you to handle specific error conditions in specific ways by catching a specific exception type. The exception type is not designed to communicate the precise details of the problem. All exception classes include a message property for this purpose. Therefore, you should not create a custom exception class just to communicate the nature of an error condition. Create a custom exception class only if you need to handle that error condition in a distinct way.

Creating Custom Exception Types

All exception classes ultimately derive from the **System.Exception** class. This class provides a range of properties that you can use to provide more detail about the error condition. For example:

- The **Message** property enables you to provide more information about what happened as a text string.
- The **InnerException** property enables you to identify another **Exception** instance that caused the current instance.
- The **Source** property enables you to specify the item or application that caused the error condition.
- The **Data** property enables you to provide more information about the error condition as key-value pairs.

When you create a custom exception type, you should make use of these existing properties wherever possible, rather than creating your own alternative properties. At a high level, the process for creating a custom exception class is as follows:

1. Create a class that inherits from the **System.Exception** class.
2. Map your class constructors to base class constructors.
3. Add any members if required.

The following example shows how to create a custom exception class:

Creating a Custom Exception Type

```
using System;

public class LoyaltyCardNotFoundException : Exception
{
    public LoyaltyCardNotFoundException()
    {
        // This implicitly calls the base class constructor.
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
public LoyaltyCardNotFoundException( string message) : base(message)
{
}

public LoyaltyCardNotFoundException(string message, Exception inner) : base(message,
inner)
{
}
```



Note: When you create a custom exception class, it is a best practice to include the word **Exception** at the end of your class name.

Throwing and Catching Custom Exceptions

After you have created your custom exception type, you can throw and catch custom exceptions in the same way that you would throw and catch any other exceptions. To throw a custom exception, you use the **throw** keyword and create a new instance of your exception type.

The following code shows how you can throw a custom exception:

- Use the **throw** keyword to throw a custom exception

```
throw new LoyaltyCardNotFoundException();
```

- Use a try/catch block to catch the exception

```
try
{
    // Perform the operation that could cause the exception.
}
catch(LoyaltyCardNotFoundException ex)
{
    // Use the exception variable, ex, to get more information.
}
```

Throwing a Custom Exception

```
public LoyaltyCard
{
    public static int GetBalance(string loyaltyCardNumber)
    {
        var customer = LoyaltyCard.GetCustomer(loyaltyCardNumber);
        if(customer == null)
        {
            throw new LoyaltyCardNotFoundException("The card number provided was not
found");
        }
        else
        {
            return customer.TotalPoints;
        }
    }
    // Other class members are not shown.
}
```

To catch the exception, you use a try/catch block. Remember that you should always attempt to catch the most specific exceptions first, and the most general exception (typically **System.Exception**) last.

The following example shows how you can catch a custom exception:

Catching a Custom Exception

```
public bool PayWithPoints(int costInPoints, string cardNumber)
try
{
    int totalPoints = LoyaltyCard.GetBalance(cardNumber);
    // Throws a LoyaltyCardNotFoundException if the card number is invalid.

    if(totalPoints >= costInPoints)
    {
        LoyaltyCard.DeductPoints(costInPoints);
        return true;
    }
    else return false;
}
catch(LoyaltyCardNotFoundException)
{
    // Take appropriate action to remedy the invalid card number.
    return false;
}
catch(Exception)
{
    // Catches other unanticipated exceptions.
    return false;
}
```

Inheriting from Generic Types

When you inherit from a generic class, you must decide how you want to manage the type parameters of the base class. You can handle type parameters in two ways:

- Leave the type parameter of the base type unspecified.
- Specify a type argument for the base type.

Consider an example where you want to create a custom list class that inherits from `List<T>`. If you leave the type parameter of the base type unspecified, you must include the same type parameter in your class declaration.

The following example shows how to inherit from a generic base type without specifying a type argument:

Inheriting from a Generic Base Type Without Specifying a Type Argument

```
public class CustomList<T> : List<T>
{ }
```

In the above example, when you instantiate the `CustomList` class and provide a type argument for `T`, the same type argument is applied to the base class.

Alternatively, you can specify a type argument for the base type in your class declaration. When you use this approach, any references to the type parameter in the base type are replaced with the type you specify in your class declaration.

For each base type parameter, you must either:

- Provide a type argument in your class declaration

```
public class CustomList : List<int>
```

- Include a matching type parameter in your class declaration

```
public class CustomList<T> : List<T>
```

The following example shows how to specify a type argument for a base type:

Inheriting from a Generic Base Type by Specifying a Type Argument

```
public class CustomList : List<int>
{ }
```

In the above example, when you instantiate the **CustomList** class, you do not need to specify a type parameter. Any base class methods or properties that referenced the type parameter are now strongly typed to the **Int32** type. For example, the **List.Add** method will only accept arguments of type **Int32**.

If the base class that you are inheriting from contains multiple type parameters, you can specify type arguments for any number of them. The important thing to remember is that you must *either* provide a type argument *or* add a matching type parameter to your class declaration for each type parameter in the base type.

The following example shows the different ways in which you can inherit from a base type with multiple type parameters:

Inheriting from a Base Type with Multiple Type Parameters

```
// Pass all the base type parameters on to the derived class.
public class CustomDictionary1<TKey, TValue> : Dictionary<TKey, TValue> { }

// Provide an argument for one of the base type parameters and pass the other one to the
// derived class.
public class CustomDictionary2<TValue> : Dictionary<int, TValue> { }

// Provide arguments for both of the base type parameters.
public class CustomDictionary3 : Dictionary <int, string> { }
```

Regardless of how many—if any—type parameters the base type includes, you can add additional type parameters to your derived class declarations.

The following example shows how to add additional type parameters to derived class declarations:

Adding Type Parameters to Derived Class Declarations

```
// Pass the base type parameter on to the derived class, and add an additional type
// parameter.
public class CustomCollection1<T, U> : List <T>

// Provide an argument for the base type parameter, but add a new type parameter.
public class CustomCollection2<T> : List<int>

//Inherit from a non-generic class, but add a type parameter.
public class CustomCollection3<T> : CustomBaseClass
```

Creating Extension Methods

In most cases, if you want to extend the functionality of a class, you use inheritance to create a derived class. However, this is not always possible. Many built-in types are sealed to prevent inheritance. For example, you cannot create a class that extends the **System.String** type.

As an alternative to using inheritance to extend a type, you can create *extension methods*. When you create extension methods, you are creating methods that you can call on a particular type without actually modifying the underlying type. An extension method is a type of static method.

To create an extension method, you create a static method within a static class. The first parameter of the method specifies the type you want to extend. By preceding the parameter with the **this** keyword, you indicate to the compiler that your method is an extension method to that type.

The following example shows how to create an extension method for the **System.String** type:

Creating an Extension Method

```
namespace FourthExtensionMethods;
{
    public static class FourthCoffeeExtensions
    {
        public static bool ContainsNumbers(this String s)
        {
            // Use regular expressions to determine whether the string contains any
            // numerical digits.
            return Regex.IsMatch(s, @"\d");
        }
    }
}
```

To use an extension method, you must explicitly import the namespace that contains your extension method by using a **using** directive:

Bringing an Extension Method Into Scope

```
using FourthExtensionMethods;
```

- Create a static method in a static class
- Use the first parameter to indicate the type you want to extend
- Precede the first parameter with the **this** keyword

```
public static bool ContainsNumbers(this string s) {...}
```

- Call the method like a regular instance method

```
string text = "Text with numb3r5";
if(text.ContainsNumbers)
{
    // Do something.
}
```

You can then call the extension method as if it was an instance method on the type that it extends:

Calling an Extension Method

```
Console.WriteLine("Please type some text that contains numbers and then press Enter");
string text = Console.ReadLine();
if(text.ContainsNumbers)
{
    Console.WriteLine("Your text contains numbers. Well done!");
}
else
{
    Console.WriteLine("Your text does not contain numbers. Please try again.");
}
```

Demonstration: Refactoring Common Functionality into the User Class Lab

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: Refactoring Common Functionality into the User Class Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_DEMO.md.

Lab: Refactoring Common Functionality into the User Class

Scenario

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

Objectives

After completing this lab, you will be able to:

- Use inheritance to factor common functionality into a base class.
- Implement polymorphism by using an abstract method.
- Create a custom exception class.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD05_LAK.md.

Exercise 1: Creating and Inheriting from the User Base Class

Scenario

In this exercise, you will create an abstract base class called **User** that contains the **UserName** and **Password** properties, and the **VerifyPassword** method that is common to the **Student** and **Teacher** classes.

You will modify the definitions of the **Student** and **Teacher** classes to inherit from the **User** class, and remove the **UserName** and **Password** properties and the **VerifyPassword** method from these classes. Finally, you will build and run the application without making any other changes to the application, and then verify that it still works correctly.

Exercise 2: Implementing Password Complexity by Using an Abstract Method

Scenario

In this exercise, you will add an abstract method called **SetPassword** to the **User** class. In the **Teacher** and **Student** classes you will implement the **SetPassword** method. This method will set the password for the user (either a teacher or a student). The **SetPassword** method for a teacher will check that the password is at least eight characters long and contains at least two numeric characters. The **SetPassword** method for a student will check that the password is at least six characters long. If the password meets these requirements, it is set and the method will return true, otherwise it will return false. You will then modify the set accessor of the **Password** property in the **User** class to call the **SetPassword** method to change the user's password. Next, you will integrate this feature into the user interface of the application to enable a user to change their password. Finally, you will build and run the application to test the password functionality.

Exercise 3: Creating the **ClassFullException** Custom Exception

Scenario

In this exercise, you will create a new custom exception class called **ClassFullException**. You will modify the **EnrollInClass** method of the **Teacher** class to raise this exception if too many students are added to a teacher's class. You will update the application to catch this exception, and then you will build and run the application to test this feature.

Module Review and Takeaways

In this module, you have learned how to use inheritance and extension methods to extend the functionality of existing types.

Review Questions

Check Your Knowledge

| Question | |
|---|--------------------|
| Which of the following types of method must you implement in derived classes? | |
| Select the correct answer. | |
| | Abstract methods. |
| | Protected methods. |
| | Public methods. |
| | Static methods. |
| | Virtual methods. |

Check Your Knowledge

| Question | |
|--|--|
| You want to create an extension method for the String class. You create a static method within a static class. How do you indicate that your method extends the String type? | |
| Select the correct answer. | |
| | The return type of the method must be a String. |
| | The first parameter of the method must be a String. |
| | The class must inherit from the String class. |
| | The method declaration must include String as a type argument. |
| | The method declaration must be preceded by String. |

MCT USE ONLY. STUDENT USE PROHIBITED

Module 6

Reading and Writing Local Data

Contents:

| | |
|--|------|
| Module Overview | 6-1 |
| Lesson 1: Reading and Writing Files | 6-2 |
| Lesson 2: Serializing and Deserializing Data | 6-12 |
| Lesson 3: Performing I/O by Using Streams | 6-26 |
| Lab: Generating the Grades Report | 6-34 |
| Module Review and Takeaways | 6-36 |

Module Overview

Reading and writing data are core requirements for many applications, such as a text file editor saving a file to the file system, or a Windows service writing error messages to a custom log file. The local file system provides the perfect environment for an application to read and write such data, because access is fast and the location is readily available. The Microsoft .NET Framework provides a variety of I/O classes that simplify the process of implementing I/O functionality in your applications.

In this module, you will learn how to read and write data by using transactional file system I/O operations, how to serialize and deserialize data to the file system, and how to read and write data to the file system by using streams.

Objectives

After completing this module, you will be able to:

- Read and write data to and from the file system by using file I/O.
- Convert data into a format that can be written to or read from a file or other data source.
- Use streams to send and receive data to or from a file or data source

Lesson 1

Reading and Writing Files

The .NET Framework provides the **System.IO** namespace, which contains a number of classes that help simplify applications that require I/O functionality.

In this lesson, you will learn how to use the classes in this namespace to read and write data to and from files, and to manipulate files and directories on the file system.

Lesson Objectives

After completing this lesson, you will be able to:

- Read and write data by using the **File** class.
- Manipulate files by using the **FileInfo** and the **File** classes.
- Manipulate directories by using the **DirectoryInfo** and **Directory** classes.
- Manipulate file and directory paths by using the **Path** class.

Reading and Writing Data by Using the File Class

The **File** class in the **System.IO** namespace exposes several static methods that you can use to perform transactional operations for direct reading and writing of files. These methods are transactional because they wrap several underlying functions into a single method call. Typically, to read data from a file, you:

1. Acquire the file handle.
2. Open a stream to the file.
3. Buffer the data from the file into memory.
4. Release the file handle so that it can be reused.

- The **System.IO namespace** contains classes for manipulating files and directories
- The **File** class contains atomic read methods, including:
 - **ReadAllText(...)**
 - **ReadAllLines(...)**
- The **File** class contains atomic write methods, including:
 - **WriteAllText(...)**
 - **AppendAllText(...)**

The static methods that the **File** class exposes are convenient because they encapsulate intricate, low-level functions. However their convenience and the fact that they shield the developer from the underlying functionality means in some cases they don't offer the control or flexibility that applications require. For example, the **ReadAllText** method will read the entire contents of a file into memory. For small files this will be fine, but for large files it can present scalability issues, and may result in an unresponsive UI in your application.

Reading Data from Files

The **File** class provides several methods that you can use to read data from a file. The format of your data and how your application intends to process it will influence the method that you should use. The following list describes some of these methods:

- The **ReadAllText** method enables you to read the contents of a file into a single string variable. The following code example shows how to read the contents of the settings.txt file into a string named **settings**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
```

```
string settings = File.ReadAllText(filePath);
```

- The **ReadAllLines** method enables you to read the contents of a file and store each line at a new index in a string array. The following code example shows how to read the contents of the settings.txt file and store each line in the string array named **settingsLineByLine**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string[] settingsLineByLine = File.ReadAllLines(filePath);
```

- The **ReadAllBytes** method enables you to read the contents of a file as binary data and store the data in a byte array. The following code example shows how to read the contents of the settings.txt file into a byte array named **rawSettings**.

```
string filePath = @"C:\fourthCoffee\settings.txt";
byte[] rawSettings = File.ReadAllBytes(filePath);
```

 **Note:** During these examples and for the rest of this module, you'll see the notation @"". Normally, when the character '\' appears in a string, it's treated as an escape character, transforming the next character to a special Unicode or ASCII character. For example, the string "\n" will be transformed to the new line character. To write '\', you need to add another to escape it, like this: "\\\".

String starting with @, called verbatim strings do not treat '\' as an **escape character**, and anything written in it will be treated literally. It's especially useful to shorten paths, negating the need for multiple "\\\".

You can learn more about strings here: [string \(C# Reference\)](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/string), <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/string>.

Each of these methods enables you to read the contents of a file into memory. You could use the **ReadAllText** method if you wanted to cache the entire file in memory in a single operation. Alternatively, if you wanted to process a file line-by-line, you could use the **ReadAllLines** method to read each line into an array.

Writing Data to Files

The **File** class also provides methods that you can use to write different types of data to a file. For each of the different types of data you can write, the **File** class provides two methods:

- If the specified file does not exist, the **Writexxx** methods create a new file with the new data. If the file does exist, the **Writexxx** methods overwrite the existing file with the new data.
- If the specified file does not exist, the **Appendxxx** methods also create a new file with the new data. However, if the file does exist, the new data is written to the end of the existing file.

The following list describes some of these methods:

- The **WriteAllText** method enables you to write the contents of a string variable to a file. If the file exists, its contents will be overwritten. The following code example shows how to write the contents of a string named **settings** to a new file named settings.txt.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string settings = "companyName=fourth coffee;";
File.WriteAllText(filePath, settings);
```

- The **WriteAllLines** method enables you to write the contents of a string array to a file. Each entry in the string array represents a new line in the file. The following code example shows how to write the contents of a string array named **hosts** to a new file named hosts.txt.

```
string filePath = @"C:\fourthCoffee\hosts.txt ";
string[] hosts = { "86.120.1.203", "113.45.80.31", "168.195.23.29" };
File.WriteAllLines(filePath, hosts);
```

- The **WriteAllBytes** method enables you to write the contents of a byte array to a binary file. The following code example shows how to write the contents of a byte array named **rawSettings** to a new file named settings.txt.

```
string filePath = @"C:\fourthCoffee\setting.txt ";
byte[] rawSettings = {99,111,109,112,97,110,121,78,97,109,101,61,102,111,
117,114,116,104,32,99,111,102,102,101,101};
File.WriteAllBytes(filePath, rawSettings);
```

- The **AppendAllText** method enables you to write the contents of a string variable to the end of an existing file. The following code example shows how to write the contents of a string variable named **settings** to the end of the existing settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
string settings = "companyContact= Dean Halstead";
File.AppendAllText(filePath, settings);
```

- The **AppendAllLines** method enables you to write the contents of a string array to the end of an existing file. The following code example shows how to write the contents of a string array named **newHosts** to the existing hosts.txt file.

```
string filePath = @"C:\fourthCoffee\hosts.txt ";
string[] newHosts = { "97.11.1.195", "203.194.40.177" };
File.WriteAllLines(filePath, newHosts);
```

Each of these methods enables you to write data to a file. If you want to add data to an existing file that may already exist, then you should use an **Appendxxx** method. If you want to overwrite an existing file, then you should use a **Writexxx** method. Then, depending on how you want the information is stored (whether as binary data, a textual blob in a string, or an array of strings representing each individual line) use the **xxxAllBytes**, **xxxAllText**, or **xxxAllLines** method.

Manipulating Files

As well as reading from and writing to files, applications typically require the ability to interact with files stored on the file system. For example, your application may need to copy a file from the system directory to a temporary location before performing some further processing, or your application may need to read some metadata associated with the file, such as the file creation time. You can implement this type of functionality by using the **File** and **FileInfo** classes.

- The **File** class provides static members

```
File.Delete(...);
bool exists = File.Exists(...);
DateTime createdOn = File.GetCreationTime(...);
```

- The **FileInfo** class provides instance members

```
FileInfo file = new FileInfo(...);
...
string name = file.DirectoryName;
bool exists = file.Exists;
file.Delete();
```

File Manipulation by using the File Class

The **File** class provides static methods that you can use to perform basic file manipulation. The following list describes some of these methods:

- The **Copy** method enables you to copy an existing file to a different directory on the file system. The following code example shows how to copy the settings.txt file from the C:\fourthCoffee\ directory to the C:\temp\ directory.

```
string sourceSettingsPath = @"C:\fourthCoffee\settings.txt";
string destinationSettingsPath = @"C:\temp\settings.txt";
bool overWrite = true;
File.Copy(sourceSettingsPath, destinationSettingsPath, overWrite);
```

 **Note:** The overwrite parameter passed to the **Copy** method call indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** to the **Copy** method call, and the file already exists, the Common Language Runtime (CLR) will throw a **System.IO.IOException**.

- The **Delete** method enables you to delete an existing file from the file system. The following code example shows how to delete the existing settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
File.Delete(filePath);
```

- The **Exists** method enables you to check whether a file exists on the file system. The following code example shows how to check whether the settings.txt file exists.

```
string filePath = @"C:\fourthCoffee\settings.txt";
bool persistedSettingsExist = File.Exists(filePath);
```

- The **GetCreationTime** method enables you to read the date time stamp that describes when a file was created, from the metadata associated with the file. The following code example shows how you can determine when the settings.txt file was created.

```
string filePath = @"C:\fourthCoffee\settings.txt";
DateTime settingsCreatedOn = File.GetCreationTime(filePath);
```

There are many other operations and metadata associated with files that you can utilize in your applications. The **FileInfo** class provides access to these through a number of instance members.

File Manipulation by using the FileInfo class

The **FileInfo** class provides instance members that you can use to manipulate an existing file. In contrast to the **File** class that provides static methods for direct manipulation, the **FileInfo** class behaves like an in-memory representation of the physical file, exposing metadata associated with the file through properties, and exposing operations through methods.

The following code example shows how to create an instance of the **FileInfo** class that represents the settings.txt file.

Instantiating the FileInfo Class

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
```

After you have created an instance of the **FileInfo** class, you can use the properties and methods that it exposes to interact with the file. The following list describes some of these properties and methods:

- The **CopyTo** method enables you to copy an existing file to a different directory on the file system. The following code example shows how to copy the settings.txt file from the C:\fourthCoffee\ directory to the C:\temp\ directory.

```
string sourceSettingsPath = @"C:\fourthCoffee\settings.txt";
string destinationSettingsPath = @"C:\temp\settings.txt";
bool overwrite = true;
FileInfo settings = new FileInfo(sourceSettingsPath);
settings.CopyTo(destinationSettingsPath, overwrite);
```

 **Note:** The overwrite parameter passed to the **CopyTo** method call indicates that the copy process should overwrite an existing file if it exists at the destination path. If you pass **false** to the **CopyTo** method call, and the file already exists, the CLR will throw a **System.IO.IOException**.

- The **Delete** method enables you to delete a file. The following code example shows how to delete the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
settings.Delete();
```

- The **DirectoryName** property enables you to get the directory path to the file. The following code example shows how to get the path to the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
string directoryPath = settings.DirectoryName; // returns C:\\fourthCoffee
```

- The **Exists** method enables you to determine if the file exists within the file system. The following code example shows how to check whether the settings.txt file exists.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
bool persistedSettingsExist = settings.Exists;
```

- The **Extension** property enables you to get the file extension of a file. The following code example shows how to get the extension of a path returned from a method call.

```
string filePath = FourthCoffeeDataService.GetFilePath();
FileInfo settings = new FileInfo(filePath);
string extension = settings.Extension;
```

- The **Length** property enables you to get the length of the file in bytes. The following code example shows how to get the length of the settings.txt file.

```
string filePath = @"C:\fourthCoffee\settings.txt";
FileInfo settings = new FileInfo(filePath);
long length = settings.Length;
```

Manipulating Directories

It is a common requirement for applications to interact and manipulate the file system directory structure, whether to check that a directory exists before writing a file or to remove directories when running a system cleanup process. The .NET Framework class library provides the **Directory** and **DirectoryInfo** classes for such operations.

Manipulating Directories by using the Directory Class

Similar to the **File** class, the **Directory** class provides static methods that enable you to interact with directories, without instantiating a directory-related object in your code. The following list describes some of these static methods:

- The **CreateDirectory** method enables you to create a new directory on the file system. The following example shows how to create the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
Directory.CreateDirectory(directoryPath);
```

- The **Delete** method enables you to delete a directory at a specific path. The following code example shows how to delete the C:\fourthCoffee\tempData directory, and all its contents.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool recursivelyDeleteSubContent = true;
Directory.Delete(directoryPath, recursivelyDeleteSubContent);
```

 **Note:** The **recursivelyDeleteSubContent** parameter passed into the **Delete** method call indicates whether the delete process should delete any content that may exist in the directory. If you pass **false** into the **Delete** method call, and the directory is not empty, the CLR will throw a **System.IO.IOException**.

- The **Exists** method enables you to determine if a directory exists on the file system. The following code example shows how to determine if the C:\fourthCoffee\tempData directory exists.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool tempDataDirectoryExists = Directory.Exists(directoryPath);
```

- The **GetDirectories** method enables you to get a list of all subdirectories within a specific directory on the file system. The following code example shows how to get a list of all the sub directories in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
string[] subDirectories = Directory.GetDirectories(directoryPath);
```

- The **Directory** class provides static members

```
Directory.Delete(...);
bool exists = Directory.Exists(...);
string[] files = Directory.GetFiles(...);
```

- The **DirectoryInfo** class provides instance members

```
 DirectoryInfo directory = new DirectoryInfo(...);
...
string path = directory.FullName;
bool exists = directory.Exists;
FileInfo[] files = directory.GetFiles();
```

- The **GetFiles** method enables you to get a list of all the files within a specific directory on the file system. The following example shows how to get a list of all the files in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
string[] files = Directory.GetFiles(directoryPath);
```

The **DirectoryInfo** class provides instance members that enable you to access directory metadata and manipulate the directory structure.

Manipulating Directories by using the DirectoryInfo Class

The **DirectoryInfo** class acts as an in-memory representation of a directory. Before you can access the properties and execute the methods that the **DirectoryInfo** class exposes, you must create an instance of the class.

The following code example shows how to create an instance of the **DirectoryInfo** class that represents the **C:\fourthCoffee\tempData** directory.

Instantiating the DirectoryInfo Class

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
```

When you have created an instance of the **DirectoryInfo** class, you can then use its properties and methods to interact with the directory. The following list describes some of these properties and methods:

- The **Create** method enables you to create a new directory on the file system. The following example shows how to create the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
directory.Create();
```

- The **Delete** method enables you to delete a directory at a specific path. The following code example shows how to delete the C:\fourthCoffee\tempData directory, and all its contents.

```
string directoryPath = @"C:\fourthCoffee\tempData";
bool recursivelyDeleteSubContent = true;
DirectoryInfo directory = new DirectoryInfo(directoryPath);
directory.Delete(recursivelyDeleteSubContent);
```



Note: The **recursivelyDeleteSubContent** parameter passed to the **Delete** method call indicates whether the delete process should delete any content that may exist in the directory. If you pass **false** to the **Delete** method call, and the directory is not empty, the CLR will throw a **System.IO.IOException**.

- The **Exists** property enables you to determine if a directory exists on the file system. The following code example shows how to determine if the C:\fourthCoffee\tempData directory exists.

```
string directoryPath = @"C:\fourthCoffee\tempData";
DirectoryInfo directory = new DirectoryInfo(directoryPath);
bool tempDataDirectoryExists = directory.Exists;
```

- The **FullName** property enables you to get the full path to the directory. The following example shows how to get the full path to the tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
 DirectoryInfo directory = new DirectoryInfo(directoryPath);
 string fullPath = directory.FullName;
```

- The **GetDirectories** method enables you to get a list of all subdirectories within a specific directory on the file system. In contrast to the static **File.GetDirectories** method, this instance method returns an array of type **DirectoryInfo**, which enables you to use each of the instance properties for each subdirectory. The following code example shows how to get all of the sub directories in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
 DirectoryInfo directory = new DirectoryInfo(directoryPath);
 DirectoryInfo[] subDirectories = directory.GetDirectories();
```

- The **GetFiles** method enables you to get a list of all the files within a specific directory on the file system. In contrast to the static **File.GetFiles** method, this instance method returns an array of type **FileInfo**, which enables you to use each of the instance properties for each file. The following example shows how to get all of the files in the C:\fourthCoffee\tempData directory.

```
string directoryPath = @"C:\fourthCoffee\tempData";
 DirectoryInfo directory = new DirectoryInfo(directoryPath);
 FileInfo[] subFiles = directory.GetFiles();
```

Depending on whether you require a simple one-line-of-code approach to manipulate a directory, or something that offers slightly more flexibility, either the static **Directory** or instance **DirectoryInfo** class should fulfill your requirements.

Manipulating File and Directory Paths

All files and all directories have a name, which when combined to point to a file in a directory, constitute a path. Different file systems can have different conventions and rules for what constitutes a path. The .NET Framework provides the **Path** class, which encapsulates a variety of file system utility functions that you can use to parse and construct valid file names, directory names, and paths within the Windows file system. These functions can be useful if your application needs to write a file to a temporary location, extract an element from a file system path, or even generate a random file name.

The **Path** class encapsulates file system utility functions

```
string settingsPath = "..could be anything here..";
// Check to see if path has an extension.
bool hasExtension = Path.HasExtension(settingsPath);
...
// Get the extension from the path.
string pathExt = Path.GetExtension(settingsPath);
...
// Get path to temp file.
string tempPath = Path.GetTempFileName();
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```

The following code shows how to create a new directory on the root of the C: drive.

Creating a Temporary Directory the Hard Way

```
string tempDirectoryPath = @"C:\fourthCoffee\tempData";  
  
if (!Directory.Exists(tempDirectoryPath))  
    Directory.CreateDirectory(tempDirectoryPath);
```

However, with the above approach, you are making many assumptions, including whether your application has the necessary privileges to perform I/O at the root of the C drive, and whether the C drive actually exists.

A better way is to use the static **GetTempPath** method provided by the **Path** class to get the path to the current user's Windows temporary directory.

Getting the Path to the Windows Temporary Directory

```
string tempDirectoryPath = Path.GetTempPath();
```

The **Path** class includes many other static methods that provide a good starting point for any custom I/O type functionality that your application may require. These methods include the following:

- The **HasExtension** method enables you to determine if the path your application is processing has an extension. This provides a convenient way for you to determine if you are processing a file or a directory. The following example shows how to check whether the path has an extension.

```
string settingsPath = "..could be anything here..";  
bool hasExtension = Path.HasExtension(settingsPath);
```

- The **GetExtension** method enables you to get the extension from a file name. This method is particularly useful when you want to ascertain what type of file your application is processing. The following code example shows how to check whether the **settingsPath** variable contains a path that ends with the .txt extension.

```
string settingsPath = "..could be anything here..";  
string pathExt = Path.GetExtension(settingsPath);  
if (pathExt == ".txt")  
{  
    // More processing here.  
}
```

- The **GetTempFileName** enables you to create a new temp file in your local Windows temporary directory in a single transactional operation folder. This method then returns the absolute path to that file, ready for further processing. The following code shows how to invoke the **GetTempFileName** method.

```
string tempPath = Path.GetTempFileName();  
// Returns C:\Users\LeonidsP\AppData\Local\Temp\ABC.tmp
```



Additional Reading: For more information about **the Path class**, refer to the Path Class page at <https://aka.ms/moc-20483c-m6-pg1>.

Demonstration: Manipulating Files, Directories, and Paths

In this demonstration, you will use the **File**, **Directory**, and **Path** classes to build a utility that combines multiple files into a single file.

Demonstration Steps

You will find the steps in the **Demonstration: Manipulating Files, Directories, and Paths** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md.

Lesson 2

Serializing and Deserializing Data

Serialization is the process of converting data to a format that can be persisted or transported.

Deserialization is the process of converting serialized data back to objects.

In this lesson, you will learn how to serialize objects to binary, XML, and JavaScript Object Notation (JSON), and how to create a custom serializer so that you can serialize objects into any format you choose.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of serialization, and the formats that the .NET Framework supports.
- Create a custom type that is serializable.
- Serialize an object as binary.
- Serialize an object as XML.
- Serialize an object as JSON.
- Create a custom serializer by implementing the **IFormatter** interface.

What Is Serialization?

Applications typically process data. Data is read into memory, perhaps from a file or web service call, processed, and then passed to another component in the system for further processing. The components of a system may run on the same machine, but commonly components run on different platforms, on different hardware, and even in different geographical locations. The format of the data also needs to be lightweight so that it can be transported over a variety of protocols, such as HTTP or SOAP. Serialization is the process of converting the state of an object into a form that can be persisted or transported.

- Binary
10101010101010111101011010101101011111101010110110001
- XML
<SOAP-ENV:Envelope ...>
 <SOAP-ENV:Body>
 ...
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
- JSON
{
 "ConfigName": "FourthCoffee_Default",
 "DatabaseHostName": "database209.fourthcoffee.com"
}

Serializable Formats

The requirements of your system, and how you intend to transport the data, will influence the serialization format you choose. The following table describes some of the common formats available.

| Format | Description |
|--------|--|
| Binary | Serializing data into the machine-readable binary format enables you to preserve the fidelity and state of an object between different instances of your application. Binary serialization is also fast and lightweight, because the binary format does not require the processing and storage of unnecessary formatting constructs.
Binary serialization is commonly used when persisting and transporting objects between applications running on the same platform.
Binary Example
10000001011101000110111001010111000111111010111000110011001011011100 |

| Format | Description |
|--------|--|
| | <pre>11110010110101001000100001101100010000000011100000111000110000100 000011000100101010001110110010110100001110010100010110101010011111101 110101100001101011001101110110100100001111010100111100000101101111111 010111000110011001011011100101101010001000011011000100001101100010000000 0011110000011100011000010000011000100101010001110110010110100001110 01010001011010100111110111010100011000010000001100010010101000111 011001011010000</pre> |
| XML | <p>Serializing data into the XML format enables you to utilize an open standard that can be processed by any application, regardless of platform. In contrast to binary, XML does not preserve type fidelity; it only lets you serialize public members that your type exposes.</p> <p>The XML format is more verbose than binary, as the serialized data is formatted with XML constructs. This makes the XML format less efficient and more processor intensive during the serializing, deserializing, and transporting processes.</p> <p>However, the nature of XML as a plain text and free-form language allows messages to be transmitted across different applications and platforms. So long as the transmitter and receiver have agreed on a known contract, both can send and receive messages and convert them to the appropriate model within their respective environments.</p> <p>XML serialization was commonly used to serialize data that can be transported via the SOAP protocol to and from web services. However, due to SOAP's verbose and strict nature, this protocol has fallen out of favor, and is generally found today only in legacy environments.</p> <p>SOAP XML Example</p> <pre><SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP- ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <SOAP-ENV:Body> <a1:ServiceConfiguration id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/ FourthCoffeeSerializer%2C%20Version%3D1.0.0.0%2C%20 Culture%3Dneutral%2C%20PublicKeyToken%3Dnull"> <ConfigName id="ref-3"> FourthCoffee_Default </ConfigName> <DatabaseHostName id="ref-4"> database209.fourthcoffee.com </DatabaseHostName> <ApplicationDataPath id="ref-5"> C:\fourthcoffee\applicationdata\ </ApplicationDataPath></pre> |

| Format | Description |
|--------|--|
| | <pre></a1:ServiceConfiguration> </SOAP-ENV:Body> </SOAP-ENV:Envelope></pre> |
| JSON | <p>Serializing data into the JSON format enables you to utilize a lightweight, data-interchange format that is based on a subset of the JavaScript programming language. JSON is a simple text format that is human readable and also easy to parse by machine, irrespective of platform.</p> <p>JSON shares XML strengths as plain text and free form, making it cross platform. However, unlike XML, it has a much more concise syntax, which makes it cheaper to transmit and human readable. These advantages made JSON the prevalent language to transmit data today, and the de-facto standard in today's industry.</p> <p>JSON is commonly used to transport data between everything from Asynchronous JavaScript and XML (AJAX) calls to messages between webservices, because unlike SOAP, you are not limited to just communicating within the same domain.</p> <p>JSON Example</p> <pre>{ "ConfigName": "FourthCoffee_Default", "DatabaseHostName": "database209.fourthcoffee.com", "ApplicationDataPath": "C:\\fourthcoffee\\applicationdata\\" }</pre> |

Alternatively, if you want to serialize your data to a format that the .NET Framework does not natively support, you can implement your own custom serializer class.

Creating a Serializable Type

The .NET Framework provides many classes that are serializable. If you want to create your own types that are serializable, you need to ensure that the type definition includes the necessary configuration and functionality for the serializer to consume. The .NET Framework provides the **System** and **System.Runtime.Serialization** namespaces, which provide classes to enable serialization support.

To create a serializable type, perform the following steps:

1. Define a default constructor.

```
public class ServiceConfiguration
{
    public ServiceConfiguration()
    {
        ...
    }
}
```

Implement the **ISerializable** interface

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    ...
    public ServiceConfiguration(
        SerializationInfo info, StreamingContext ctxt)
    {
        ...
    }

    public void GetObjectData(
        SerializationInfo info, StreamingContext context)
    {
        ...
    }
}
```

2. Decorate the class with the **Serializable** attribute provided in the **System** namespace.

```
[Serializable]
public class ServiceConfiguration
{
    ...
}
```

3. Implement the **ISerializable** interface provided in the **System.Runtime.Serialization** namespace.

The **GetObjectData** method enables you to extract the data from your object during the serialization process.

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    ...
    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        ...
    }
}
```

4. Define a deserialization constructor, which accepts **SerializationInfo** and **StreamingContext** objects as parameters. This constructor enables you to rehydrate your object during the deserialization process.

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    ...
    public ServiceConfiguration(SerializationInfo info, StreamingContext ctxt)
    {
        ...
    }
}
```

5. Define the public members that you want to serialize. You can instruct the serializer to ignore private fields by decorating them with the **NonSerialized** attribute.

```
...
[NonSerialized]
private Guid _internalId;
public string ConfigName { get; set; }
public string DatabaseHostName { get; set; }
public string ApplicationDataPath { get; set; }
...
```

The following code example shows the complete **ServiceConfiguration** class, which is serializable by any of the .NET Framework **IFormatter** implementations.

Serializable Type

```
[Serializable]
public class ServiceConfiguration : ISerializable
{
    [NonSerialized]
    private Guid _internalId;

    public string ConfigName { get; set; }
    public string DatabaseHostName { get; set; }
    public string ApplicationDataPath { get; set; }

    public ServiceConfiguration()
```

```
{  
}  
  
public ServiceConfiguration(SerializationInfo info, StreamingContext ctxt)  
{  
    this.ConfigName  
        = info.GetValue("ConfigName", typeof(string)).ToString();  
    this.DatabaseHostName  
        = info.GetValue("DatabaseHostName", typeof(string)).ToString();  
    this.ApplicationDataPath  
        = info.GetValue("ApplicationDataPath", typeof(string)).ToString();  
}  
  
public void GetObjectData(SerializationInfo info, StreamingContext context)  
{  
    info.AddValue("ConfigName", this.ConfigName);  
    info.AddValue("DatabaseHostName", this.DatabaseHostName);  
    info.AddValue("ApplicationDataPath", this.ApplicationDataPath);  
}  
}
```

Serializing Objects as Binary

The .NET Framework provides the **BinaryFormatter** class in the **System.Runtime.Serialization.Formatters.Binary** namespace, which you can use to serialize and deserialize objects as binary.

 **Note:** The **BinaryFormatter** and **SoapFormatter** classes implement the **IFormatter** interface. You can also implement the **IFormatter** interface to create your own custom serializer.

• Serialize as binary

```
ServiceConfiguration config = ServiceConfiguration.Default;  
IFormatter formatter = new BinaryFormatter();  
FileStream buffer = File.Create("C:\\fourthcoffee\\config.txt");  
formatter.Serialize(buffer, config);  
buffer.Close();
```

• Deserialize from binary

```
IFormatter formatter = new BinaryFormatter();  
FileStream buffer = File.OpenRead("C:\\fourthcoffee\\config.txt");  
ServiceConfiguration config  
    = formatter.Deserialize(buffer) as ServiceConfiguration;  
buffer.Close();
```

Serialize an Object by Using the **BinaryFormatter** Class

To serialize an object by using the **BinaryFormatter** class, perform the following steps:

1. Obtain a reference to the object you want to serialize.
2. Create an instance of the **BinaryFormatter** that you want to use to serialize your type.
3. Create a stream that you will use as a buffer to store the serialized data.
4. Invoke the **BinaryFormatter.Serialize** method, passing in stream that the serialized data will be buffered to, and the object you want to serialize.

The following code example shows how to use the **BinaryFormatter** class to serialize an object as binary.

BinaryFormatter Serialize Example

```
// Create the object you want to serialize.  
ServiceConfiguration config = ServiceConfiguration.Default;  
  
// Create the formatter you want to use to serialize the object.  
IFormatter formatter = new BinaryFormatter();  
  
// Create the stream that the serialized data will be buffered to.  
FileStream buffer = File.Create(@"C:\\fourthcoffee\\config.txt");
```

```
// Invoke the Serialize method.  
formatter.Serialize(buffer, config);  
  
// Close the stream.  
buffer.Close();
```

This example, serializes the **ServiceConfiguration** object, and writes the serialized data to a file. It is important to note that serialization doesn't just imply writing data to a file. Serialization is the process of transforming a type into another format, which you can then write to a file or database, or send over HTTP to a web service.

Deserialize an Object by Using the **BinaryFormatter** Class

Deserializing is the process of transforming your serialized object back into a format that your application can process. To deserialize an object by using the **BinaryFormatter** class, perform the following steps:

1. Create an instance of the **BinaryFormatter** that you want to use to deserialize your type.
2. Create a stream to read the serialized data.
3. Invoke the **BinaryFormatter.Deserialize** method, passing in stream that contains the serialized data.
4. Cast the result of the **BinaryFormatter.Deserialize** method call into the type of object that you are expecting.

The following code example shows how to use the **BinaryFormatter** class to deserialize binary data to an object.

BinaryFormatter Deserialize Example

```
// Create the formatter you want to use to serialize the object.  
IFormatter formatter = new BinaryFormatter();  
  
// Create the stream that the serialized data will be buffered too.  
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");  
  
// Invoke the Deserialize method.  
ServiceConfiguration config = formatter.Deserialize(buffer) as ServiceConfiguration;  
  
// Close the stream.  
buffer.Close();
```

The above example reads the serialized binary data from a file, and then deserializes the binary into a **ServiceConfiguration** object. The process is the same for serializing and deserializing objects by using any formatters that implement the **IFormatter** interface. This includes the **SoapFormatter** class, and any custom formatters that you may implement.

Serializing Objects as XML

The .NET Framework provides the **SoapFormatter** class in the **System.Runtime.Serialization.Formatters.Soap** namespace, which you can use to serialize and deserialize objects as XML.

Serialize an Object by Using the SoapFormatter Class

The process for serializing data as XML is similar to the process of serializing to binary, with the exception that you use the **SoapFormatter** class.

The following code example shows how to use the **SoapFormatter** class to serialize an object as XML.

SoapFormatter Serialize Example

```
// Create the object you want to serialize.  
ServiceConfiguration config = ServiceConfiguration.Default;  
  
// Create the formatter you want to use to serialize the object.  
IFormatter formatter = new SoapFormatter();  
  
// Create the stream that the serialized data will be buffered too.  
FileStream buffer = File.Create(@"C:\fourthcoffee\config.xml");  
  
// Invoke the Serialize method.  
formatter.Serialize(buffer, config);  
  
// Close the stream.  
buffer.Close();
```

• Serialize as XML

```
ServiceConfiguration config = ServiceConfiguration.Default;  
IFormatter formatter = new SoapFormatter();  
FileStream buffer = File.Create(@"C:\fourthcoffee\config.xml");  
formatter.Serialize(buffer, config);  
buffer.Close();
```

• Deserialize from XML

```
IFormatter formatter = new SoapFormatter();  
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.xml");  
ServiceConfiguration config  
= formatter.Deserialize(buffer) as ServiceConfiguration;  
buffer.Close();
```

Deserialize an Object by Using the SoapFormatter Class

The process for deserializing data from XML to an object is identical to the process of deserializing binary data, with the exception that you use the **SoapFormatter** class.

The following code example shows how to use the **SoapFormatter** class to deserialize XML data to an object.

SoapFormatter Deserialize Example

```
// Create the formatter you want to use to serialize the object.  
IFormatter formatter = new SoapFormatter();  
  
// Create the stream that the serialized data will be buffered too.  
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.xml");  
  
// Invoke the Deserialize method.  
ServiceConfiguration config = formatter.Deserialize(buffer) as ServiceConfiguration;  
  
// Close the stream.  
buffer.Close();
```

Serializing Objects as JSON

The .NET Framework also supports serializing objects as JSON by using the **DataContractJsonSerializer** class in the **System.Runtime.Serialization.Json** namespace. The JSON serialization steps are different because the **DataContractJsonSerializer** class is derived from the abstract **XmlobjectSerializer** class, and it is not an implementation of the **IFormatter** interface.

• Serialize as JSON

```
ServiceConfiguration config = ServiceConfiguration.Default;
DataContractJsonSerializer jsonSerializer
    = new DataContractJsonSerializer(config.GetType());
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");
jsonSerializer.WriteObject(buffer, config);
buffer.Close();
```

• Deserialize from JSON

```
DataContractJsonSerializer jsonSerializer = new
    DataContractJsonSerializer(
        typeof(ServiceConfiguration));
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");
ServiceConfiguration config = jsonSerializer.ReadObject(buffer)
    as ServiceConfiguration;
buffer.Close();
```

Serialize an Object by Using the **DataContractJsonSerializer** Class

To serialize an object by using the **DataContractJsonSerializer** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Create an instance of the **DataContractJsonSerializer** class that you want to use to serialize your type. The constructor also requires you to pass in a **Type** object, representing the type of object you want to serialize.
3. Create a stream that you will use as a buffer to store the serialized data.
4. Invoke the **DataContractJsonSerializer.WriteObject** method, passing in stream that the serialized data will be buffered too, and the object you want to serialize.

The following code example shows how to use the **DataContractJsonSerializer** class to serialize an object as JSON.

DataContractJsonSerializer Serialize Example

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;

// Create a DataContractJsonSerializer object that you will use to serialize the
// object to JSON.
DataContractJsonSerializer jsonSerializer
    = new DataContractJsonSerializer(config.GetType());

// Create the stream that the serialized data will be buffered too.
FileStream buffer = File.Create(@"C:\fourthcoffee\config.txt");

// Invoke the WriteObject method.
jsonSerializer.WriteObject(buffer, config);

// Close the stream.
buffer.Close();
```

Deserialize an Object by using the **DataContractJsonSerializer** Class

To deserialize JSON to an object by using the **DataContractJsonSerializer** class, perform the following steps:

1. Create an instance of the **DataContractJsonSerializer** class that you want to use to serialize your type. The constructor also requires you to pass in a **Type** object, representing the type of object you want to deserialize.
2. Create a stream that will read the serialized JSON into memory.

3. Invoke the **DataContractJsonSerializer.ReadObject** method, passing in the stream that contains the serialized data.
4. Cast the result of the **DataContractJsonSerializer.ReadObject** method call into the type of object you are expecting.

The following code example shows how to use the **DataContractJsonSerializer** class to deserialize JSON data to an object.

DataContractJsonSerializer Deserialize Example

```
// Create a DataContractJsonSerializer object that you will use to
// deserialize the JSON.
DataContractJsonSerializer jsonSerializer
    = new DataContractJsonSerializer(typeof(ServiceConfiguration));

// Create a stream that will read the serialized data.
FileStream buffer = File.OpenRead(@"C:\fourthcoffee\config.txt");

// Invoke the ReadObject method.
ServiceConfiguration config = jsonSerializer.ReadObject(buffer) as ServiceConfiguration;

// Close the stream.
buffer.Close();
```

Serializing Objects as JSON by Using JSON.Net

While the .NET framework provides the built in **DataContractJsonSerializer** class to serialize objects to JSON, today's standard of using JSON in the .NET environment is Newtonsoft's **Json.NET** library. **Json.NET** provides an expansive library to create and query JSON. Most important for our purposes is their **JsonConvert** class that can serialize and deserialize classes to and from JSON.

To use the **Json.Net** library, you'll need to add it to your project by using **Nuget**, which is a package manager for .Net, allowing you to easily download and manage third party libraries in your project.

1. In **Solution Explorer**, right-click the project.
2. Select **Manage Nuget Packages**.
3. Select the **Browse** tab and search for **JSON**.
4. Select the **Newtonsoft.Json** package, and then click **Install**.

Unlike the .NET implementations of the **IFormatter** interface, **JsonConvert** doesn't require the data class to be decorated with the **Serializable** attribute. Any class can be converted to JSON without special treatment. However, **Json.NET** does provide a set of attributes to specify exactly how the model will be serialized, if necessary. The most useful of them is perhaps the **JsonProperty** attribute, which allows to determine the name of the property in the serialized JSON string. Unlike Visual C#, JSON's naming standard for properties can vary, and usually follow the lowerCamelCase, snake_case, or even the kebab-case convention. This feature can be very useful, especially when communicating with other applications.

A normal data object ready to be used with **JsonConvert** can look something like this:

• Serialize as JSON

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;

// Serialize the object to a string
var jsonString = JsonConvert.SerializeObject(config);
```

• Deserialize from JSON

```
// Deserialize to the desired type
var deserializedConfig =
JsonConvert.DeserializeObject<ServiceConfiguration>(jsonString);
```

The following code example shows the complete **ServiceConfiguration** class, ready to be serialized with **Json.Net**.

JSON Serializable Type

```
public class ServiceConfiguration
{
    // JsonConvert ignores private members by default
    private Guid _internalId;

    // Map the properties with json naming conventions
    [JsonProperty("configName")]
    public string ConfigName { get; set; }
    [JsonProperty("databaseHostName")]
    public string DatabaseHostName { get; set; }
    [JsonProperty("applicationDataPath")]
    public string ApplicationDataPath { get; set; }
}
```

Serialize an Object by Using the JsonConvert Class

To serialize an object by using the **JsonConvert** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Invoke the **JsonConvert.Serialize** method. The method will automatically infer the type processed, and serialize it accordingly.

The following code example shows how to use the **JsonConvert** class to serialize an object as JSON.

JsonConvert Serialize Example

```
// Create the object you want to serialize.
ServiceConfiguration config = ServiceConfiguration.Default;

var jsonString = JsonConvert.SerializeObject(config);
```

Deserialize an Object by Using the JsonConvert Class

To deserialize JSON to an object by using the **JsonConvert** class, perform the following steps:

1. Obtain the JSON string you need to deserialize.
2. Invoke the **JsonConvert.Deserialize<T>** method, passing in the string to be deserialized, as well as the target type as a generic type parameter.

The following code example shows how to use the **JsonConvert** class to deserialize JSON data to an object.

JsonConvert Deserialize Example

```
// Get the JSON string - Here we're assuming it's the same from the previous example.

// Deserialize to the desired type
var deserializedConfig = JsonConvert.DeserializeObject<ServiceConfiguration>(jsonString);
```

You might have noticed that unlike the previous topics, here we're serializing directly to a string, and not to a stream, and then to a file. You will find that most messages passed along are small enough not to warrant the use of a stream and saving them to a string is perfectly acceptable. Modern computers are powerful enough to handle quite large strings.

However, **Json.Net** allows serializing to and from streams very similarly to the internal .Net types, while still allowing to discard the **ISerializable** interface, and keeping **Json.Net's** lax usage of its attributes.

MCT USE ONLY. STUDENT USE PROHIBITED

Serialize an Object by Using the **JsonSerializer** Class

To serialize an object by using the **JsonSerializer** class, perform the following steps:

1. Obtain a reference to the object that you want to serialize.
2. Create an instance of the **JsonSerializer** class that you want to use to serialize your type.
3. Create a stream writer to write the object.
4. Invoke the **JsonSerializer.Serialize** method, passing in the stream writer that the serialized data will be buffered too, and the object you want to serialize. The method will automatically infer the type processed and will serialize it accordingly.

The following code example shows how to use the **JsonSerializer** class to serialize an object as JSON and save it to a file.

JsonSerializer Serialize Example

```
// Create the serializer
var serializer = new JsonSerializer();

// Open the stream to the file
var fileWriter = File.CreateText(@"C:\fourthcoffee\config.json");

// Serialize and write the object to the file
serializer.Serialize(fileWriter, config);

// Close the stream
fileWriter.Close();
fileWriter.Dispose();
```

Deserialize an Object by Using the **JsonSerializer** Class

To deserialize JSON to an object by using the **JsonSerializer** class, perform the following steps:

1. Create an instance of the **JsonSerializer** class that you want to use to serialize your type.
2. Create a stream, stream reader, and **JsonTextReader** that will read the serialized JSON into the memory.
3. Invoke the **JsonSerializer.Deserialize<T>** method, passing in the **JsonTextReader** that contains the serialized data, as well as the target type as a generic type parameter.
4. Close all the readers and stream.

The following code example shows how to use the **JsonSerializer** class to deserialize JSON data to an object.

JsonSerializer Deserialize Example

```
// Create the serializer
var serializer = new JsonSerializer();

// Open a stream to the file
var fileReader = File.OpenRead(@"C:\fourthcoffee\config.json");

// Create a stream and json text readers
var textReader = new StreamReader(fileReader);
var jsonReader = new JsonTextReader(textReader);

// Deserialize to the desired type
var deserializedConfig = serializer.Deserialize<ServiceConfiguration>(jsonReader);

// Close all the readers and the stream
jsonReader.Close();
textReader.Close();
```

```
textReader.Dispose();
fileReader.Close();
fileReader.Dispose();
```

 **Additional Reading:** For more information about the Json.Net library, you can turn to the following resources:

- Json.Net home page: <https://aka.ms/moc-20483c-m6-pg2>
- Json.Net Documentation: <https://aka.ms/moc-20483c-m6-pg3>
- JsonConvert class: <https://aka.ms/moc-20483c-m6-pg4>
- JsonPropertyAttribute class: <https://aka.ms/moc-20483c-m6-pg5>
- JsonSerializer class: <https://aka.ms/moc-20483c-m6-pg6>

Demonstration: Serializing Objects as JSON using JSON.NET

In this demonstration, you will see how to serialize and deserialize objects using JSON.NET.

Demonstration Steps

You will find the steps in the **Demonstration: Serializing Objects as JSON using JSON.NET** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md.

Creating a Custom Serializer

You may want to serialize data into a format other than binary, XML, or JSON. The .NET Framework provides the **IFormatter** interface in the **System.Runtime.Serialization** namespace, so you can create your own formatter. Your custom formatter will then follow the same pattern as the **BinaryFormatter** and **SoapFormatter** classes.

To create your own formatter, perform the following steps:

1. Create a class that implements the **IFormatter** interface.
2. Create implementations for the **SurrogateSelector**, **Binder**, and **Context** properties.
3. Create implementations for the **Deserialize** and **Serialize** methods.

The following code example shows a custom formatter that can serialize and deserialize objects to the .ini format.

Custom IniFormatter

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using System.Runtime.Serialization;

namespace FourthCoffee.Serializer
{
```

Implement the **IFormatter** interface

```
class IniFormatter : IFormatter
{
    public ISurrogateSelector SurrogateSelector { get; set; }
    public SerializationBinder Binder { get; set; }
    public StreamingContext Context { get; set; }

    public object Deserialize(Stream serializationStream)
    {
        ...
    }

    public void Serialize(Stream serializationStream, object graph)
    {
        ...
    }
}
```

```
class IniFormatter : IFormatter
{
    static readonly char[] _delim = new char[] { '=' };

    public ISurrogateSelector SurrogateSelector { get; set; }
    public SerializationBinder Binder { get; set; }
    public StreamingContext Context { get; set; }

    public IniFormatter()
    {
        this.Context
            = new StreamingContext(StreamingContextStates.All);
    }

    public object Deserialize(Stream serializationStream)
    {
        StreamReader buffer
            = new StreamReader(serializationStream);

        // Get the type from the serialized data.
        Type typeToDeserialize = this.GetType(buffer);

        // Create default instance of object using type name.
        Object obj
            = FormatterServices.GetUninitializedObject(typeToDeserialize);

        // Get all the members for the type.
        MemberInfo[] members
            = FormatterServices.GetSerializableMembers(obj.GetType(), this.Context);

        // Create dictionary to store the variable names and any serialized data.
        Dictionary<string, object> serializedMemberData
            = new Dictionary<string, object>();

        // Read the serialized data, and extract the variable names
        // and values as strings.
        while (buffer.Peek() >= 0)
        {
            string line = buffer.ReadLine();
            string[] sarr = line.Split(_delim);

            // key = variable name, value = variable value.
            serializedMemberData.Add(
                sarr[0].Trim(), // Variable name.
                sarr[1].Trim()); // Variable value.
        }

        // Close the underlying stream.
        buffer.Close();

        // Create a list to store member values as their correct type.
        List<object> dataAsCorrectTypes = new List<object>();

        // For each of the members, get the serialized values as their correct type.
        for (int i = 0; i < members.Length; i++)
        {
            FieldInfo field = members[i] as FieldInfo;

            if(!serializedMemberData.ContainsKey(field.Name))
                throw new SerializationException(field.Name);

            // Change the type of the value to the correct type
            // of the member.
            object valueAsCorrectType = Convert.ChangeType(
                serializedMemberData[field.Name],
                field.FieldType);
        }
    }
}
```

```
        dataAsCorrectTypes.Add(valueAsCorrectType);
    }

    // Populate the object with the deserialized values.
    return FormatterServices.PopulateObjectMembers(
        obj,
        members,
        dataAsCorrectTypes.ToArray());
}

public void Serialize(Stream serializationStream, object graph)
{
    // Get all the fields that you want to serialize.
    MemberInfo[] allMembers
        = FormatterServices.GetSerializableMembers(graph.GetType(), this.Context);

    // Get the field data.
    object[] fieldData = FormatterServices.GetObjectData(graph, allMembers);

    // Create a buffer to write the serialized data too.
    StreamWriter sw = new StreamWriter(serializationStream);

    // Write the name of the class to the firstline.
    sw.WriteLine("@ClassName={0}", graph.GetType().FullName);

    // Iterate the field data.
    for (int i = 0; i < fieldData.Length; ++i)
    {
        sw.WriteLine("{0}={1}",
            allMembers[i].Name,           // Member name.
            fieldData[i].ToString());   // Member value.
    }
    sw.Close();
}

private Type GetType(StreamReader buffer)
{
    string firstLine = buffer.ReadLine();

    string[] sarr = firstLine.Split(_delim);
    string nameOfClass = sarr[1];

    return Type.GetType(nameOfClass);
}
}
```

METHODS
IN CLASS
PROHIBITED

Lesson 3

Performing I/O by Using Streams

When you work with data, whether the data is stored in a file on the file system or on a web server accessible over HTTP, the data sometimes becomes too large to load into memory and transmit in a single transactional operation. For example, imagine trying to load a 200-gigabyte video file from the file system into memory in a single operation. Not only would the operation take a long time, but it would also consume a large amount of memory.

In this lesson, you will learn how to use streams to read from and write to files without having to cache the entire file in memory.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of a stream.
- Describe the different types of streams provided in the .NET Framework.
- Describe how to use a stream.

What are Streams?

The .NET Framework enables you to use streams. A stream is a sequence of bytes, which could come from a file on the file system, a network connection, or memory. Streams enable you to read from or write to a data source in small, manageable data packets. Typically, streams provide the following operations:

- Reading chunks of data into a type, such as a byte array.
- Writing chunks of data from a type to a stream.
- Querying the current position in the stream and modifying a specific selection of bytes at the current position.

• The **System.IO namespace** contains a number of stream classes, including:

- The abstract **Stream** base class
- The **FileStream** class
- The **MemoryStream** class

• Typical stream operations include:

- Reading chunks of data from a stream
- Writing chunks of data to a stream
- Querying the position of the stream

Streaming in the .NET Framework

The .NET Framework provides several stream classes that enable you to work with a variety of data and data sources. When choosing which stream classes to use, you need to consider the following:

- What type of data you are reading or writing, for example, binary or alphanumeric.
- Where the data is stored, for example, on the local file system, in memory, or on a web server over a network.

The .NET Framework class library provides several classes in the **System.IO** namespace that you can use to read and write files by using streams. At the highest level of abstraction, the **Stream** class defines the common functionality that all streams provide. The class provides a generic view of a sequence of bytes, together with the operations and properties that all streams provide. Internally, a **Stream** object maintains a pointer that refers to the current location in the data source. When you first construct a **Stream** object

over a data source, this pointer is positioned immediately before the first byte. As you read and write data, the **Stream** class advances this pointer to the end of the data that is read or written.

You cannot use the **Stream** class directly. Instead, you instantiate specializations of this class that are optimized to perform stream-based I/O for specific types of data sources. For example, the **FileStream** class implements a stream that uses a disk file as the data source, and the **MemoryStream** class implements a stream that uses a block of memory as the data source.

Types of Streams in the .NET Framework

The .NET Framework provides many stream classes that you can use to read and write different types of data from different types of data sources. The following table describes some of these stream classes, and when you might want to use them.

- Classes that enable access to data sources include:

| Class | Description |
|---------------|--|
| FileStream | Exposes a stream to a file on the file system. |
| MemoryStream | Exposes a stream to a memory location. |
| NetworkStream | Exposes a stream to a network location. |

- Classes that enable reading and writing to and from data source streams include:

| Class | Description |
|--------------|---|
| StreamReader | Read textual data from a source stream. |
| StreamWriter | Write textual data to a source stream. |
| BinaryReader | Read binary data from a source stream. |
| BinaryWriter | Write binary data to a source stream. |

| Stream class | Description |
|----------------------|---|
| FileStream | Enables you to establish a stream to a file on the file system. The FileStream class handles operations such as opening and closing the file, and provides access to the file through a raw sequence of bytes. |
| MemoryStream | Enables you to establish a stream to a location in memory. The MemoryStream class handles operations such as acquiring the in-memory storage, and provides access to the memory location through a raw sequence of bytes. |
| NetworkStream | Enables you to establish a stream to a network location in memory. The NetworkStream class handles operations such as opening and closing a connection to the network location, and provides access to the network location through a raw sequence of bytes. |

A stream that is established by using a **FileStream**, **MemoryStream**, or **NetworkStream** object is just a raw sequence of bytes. If the source data is structured, you must convert the byte sequence into the appropriate types. This can be a time-consuming and error-prone task. However, the .NET Framework contains classes that you can use to read and write textual data and primitive types in a stream that you have opened by using the **FileStream**, **MemoryStream**, or **NetworkStream** classes. The following table describes some of the stream reader and writer classes.

| Stream class | Description |
|---------------------|---|
| StreamReader | Enables you to read textual data from an underlying data source stream, such as a FileStream , MemoryStream , or NetworkStream object. |
| StreamWriter | Enables you to write textual data to an underlying data source stream, such as a FileStream , MemoryStream , or NetworkStream object. |
| BinaryReader | Enables you to read binary data from an underlying data source stream, |

| Stream class | Description |
|---------------------|---|
| | such as a FileStream , MemoryStream , or NetworkStream object. |
| BinaryWriter | Enables you to write binary data to an underlying data source stream, such as a FileStream , MemoryStream , or NetworkStream object. |

Reading and Writing Binary Data by Using Streams

Many applications store data in raw binary form for a number of reasons, such as the following:

- Writing binary data is fast.
- Binary data takes up less space on disk.
- Binary data is not human readable.

You can read and write data in a binary format in your .NET Framework applications by using the **BinaryReader** and **BinaryWriter** classes.

To read or write binary data, you construct a

BinaryReader or **BinaryWriter** object by

providing a stream that is connected to the source of the data that you want to read or write, such as a **FileStream** or **MemoryStream** object.

You can use the **BinaryReader** and **BinaryWriter** classes to stream binary data

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";
// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath,
// BinaryReader object exposes read operations on the underlying
// FileStream object.
BinaryReader reader = new BinaryReader(file);

// BinaryWriter object exposes write operations on the underlying
// FileStream object.
BinaryWriter writer = new BinaryWriter(file);
```

The following code example shows how to initialize the **BinaryReader** and **BinaryWriter** classes, passing a **FileStream** object.

Initializing a **BinaryReader** and **BinaryWriter** Object

```
string filePath = @"C:\\fourthcoffee\\applicationdata\\settings.txt";
FileStream file = new FileStream(filePath);
...
BinaryReader reader = new BinaryReader(file);
...
BinaryWriter writer = new BinaryWriter(file);
```

After you have created a **BinaryReader** object, you can use its members to read the binary data. The following list describes some of the key members:

- The **BaseStream** property enables you to access the underlying stream that the **BinaryReader** object uses.
- The **Close** method enables you to close the **BinaryReader** object and the underlying stream.
- The **Read** method enables you to read the number of remaining bytes in the stream from a specific index.
- The **ReadByte** method enables you to read the next byte from the stream, and advance the stream to the next byte.
- The **ReadBytes** method enables you to read a specified number of bytes into a byte array.

Similarly, the **BinaryWriter** object exposes various members to enable you to write data to an underlying stream. The following list describes some of the key members.

- The **BaseStream** property enables you to access the underlying stream that the **BinaryWriter** object uses.

- The **Close** method enables you to close the **BinaryWriter** object and the underlying stream.
- The **Flush** method enables you to explicitly flush any data in the current buffer to the underlying stream.
- The **Seek** method enables you to set your position in the current stream, thus writing to a specific byte.
- The **Write** method enables you to write your data to the stream, and advance the stream. The **Write** method provides several overloads that enable you to write all primitive data types to a stream.

Reading Binary Data

The following code example shows how to use the **BinaryReader** and **FileStream** classes to read a file that contains a collection of bytes. This example uses the **Read** method to advance through the stream of bytes in the file.

BinaryReader Example

```
// Source file path.  
string sourceFilePath =  
    @"C:\fourthcoffee\applicationdata\settings.txt";  
  
// Create a FileStream object so that you can interact with the file  
// system.  
FileStream sourceFile = new FileStream(  
    sourceFilePath, // Pass in the source file path.  
    FileMode.Open, // Open an existing file.  
    FileAccess.Read); // Read an existing file.  
  
// Create a BinaryReader object passing in the FileStream object.  
BinaryReader reader = new BinaryReader(sourceFile);  
  
// Store the current position of the stream.  
int position = 0;  
// Store the length of the stream.  
int length = (int)reader.BaseStream.Length;  
  
// Create an array to store each byte from the file.  
byte[] dataCollection = new byte[length];  
int returnedByte;  
while ((returnedByte = reader.Read()) != -1)  
{  
    // Set the value at the next index.  
    dataCollection[position] = (byte)returnedByte;  
  
    // Advance our position variable.  
    position += sizeof(byte);  
}  
  
// Close the streams to release any file handles.  
reader.Close();  
sourceFile.Close();
```

Writing Binary Data

The following code example shows how to use the **BinaryWriter** and **FileStream** classes to write a collection of four byte integers to a file.

BinaryWriter Example

```
string destinationFilePath = @"C:\fourthcoffee\applicationdata\settings.txt";

// Collection of bytes.
byte[] dataCollection = { 1, 4, 6, 7, 12, 33, 26, 98, 82, 101 };

// Create a FileStream object so that you can interact with the file
// system.
FileStream destFile = new FileStream(
    destinationFilePath, // Pass in the destination path.
    FileMode.Create,    // Always create new file.
    FileAccess.Write);  // Only perform writing.

// Create a BinaryWriter object passing in the FileStream object.
BinaryWriter writer = new BinaryWriter(destFile);

// Write each byte to stream.
foreach (byte data in dataCollection)
{
    writer.Write(data);
}

// Close both streams to flush the data to the file.
writer.Close();
destFile.Close();
```



Best Practice: It's best to use the **IDisposable** interface with a **using statement**, as opposed to directly using the **Open** and **Close** methods. You can learn more about them in Module 11, Lesson 2, Topic 2 – "Implementing the Dispose Pattern".

Reading and Writing Text Data by Using Streams

In addition to storing data as raw binary data, you can also store data as plain text. You may want to do this in your application if the persisted data needs to be human readable.

The process for reading from and writing plain text to a file is very similar to reading and writing binary data, except that you use the **StreamReader** and **StreamWriter** classes.

When you initialize the **StreamReader** or **StreamWriter** classes, you must provide a stream object to handle the interaction with the data source.

The following code example shows how to initialize the **StreamReader** and **StreamWriter** classes, passing a **FileStream** object.

You can use the **StreamReader** and **StreamWriter** classes to stream plain text

```
string filePath = "C:\\fourthcoffee\\applicationdata\\settings.txt";
// Underlying stream to file on the file system.
FileStream file = new FileStream(filePath);

// StreamReader object exposes read operations on the underlying
// FileStream object.
StreamReader reader = new StreamReader(file);

// StreamWriter object exposes write operations on the underlying
// FileStream object.
StreamWriter writer = new StreamWriter(file);
```

Initializing a **StreamReader** and **StreamWriter** Object

```
string destinationFilePath = @"C:\fourthcoffee\applicationdata\settings.txt";
FileStream file = new FileStream(destinationFilePath);
...
StreamReader reader = new StreamReader(file);
...
StreamWriter writer = new StreamWriter(file);
```

After you have created a **StreamReader** object, you can use its members to read the plain text. The following list describes some of the key members:

- The **Close** method enables you to close the **StreamReader** object and the underlying stream.
- The **EndOfStream** property enables you to determine whether you have reached the end of the stream.
- The **Peek** method enables you to get the next available character in the stream, but does not consume it.
- The **Read** method enables you to get and consume the next available character in the stream. This method returns an **int** variable that represents the binary of the character, which you may need to explicitly convert.
- The **ReadBlock** method enables you to read an entire block of characters from a specific index from the stream.
- The **ReadLine** method enables you to read an entire line of characters from the stream.
- The **ReadToEnd** method enables you to read all characters from the current position in the stream.

Similarly, the **StreamWriter** object exposes various members to enable you to write data to an underlying stream. The following list describes some of the key members:

- The **AutoFlush** property enables you to instruct the **StreamWriter** object to flush data to the underlying stream after every write call.
- The **Close** method enables you to close the **StreamWriter** object and the underlying stream.
- The **Flush** method enables you to explicitly flush any data in the current buffer to the underlying stream.
- The **NewLine** property enables you to get or set the characters that are used for new line breaks.
- The **Write** method enables you to write your data to the stream, and to advance the stream.
- The **WriteLine** method enables you to write your data to the stream followed by a new line break, and then advance the stream.

These members provide many options to suit many different requirements. If you do not want to store the entire file in memory in a single chunk, you can use a combination of the **Peek** and **Read** methods to read each character, one at a time. Similarly, if you want to write lines of text to a file one at time, you can use the **WriteLine** method.

Reading Plain Text

The following code example shows how to use the **StreamReader** and **FileStream** classes to read a file that contains plain text. This example uses the **Peek** method to advance through the stream of characters in the file.

MCT ICE ONLY STUDENT USE PROHIBITED

StreamReader Example

```
string sourceFilePath =
    @"C:\fourthcoffee\applicationdata\settings.txt ";

// Create a FileStream object so that you can interact with the file
// system.
FileStream sourceFile = new FileStream(
    sourceFilePath, // Pass in the source file path.
    FileMode.Open, // Open an existing file.
    FileAccess.Read); // Read an existing file.

StreamReader reader = new StreamReader(sourceFile);
StringBuilder fileContents = new StringBuilder();

// Check to see if the end of the file
// has been reached.
while (reader.Peek() != -1)
{
    // Read the next character.
    fileContents.Append((char)reader.Read());
}

// Store the file contents in a new string variable.
string data = fileContents.ToString();

// Always close the underlying streams release any file handles.
reader.Close();
sourceFile.Close();
```

Writing Plain Text

The following code example shows how to use the **StreamWriter** and **FileStream** classes to write a string to a new file on the file system.

StreamWriter Example

```
string destinationFilePath =
    @"C:\fourthcoffee\applicationdata\settings.txt ";

string data = "Hello, this will be written in plain text";

// Create a FileStream object so that you can interact with the file
// system.
FileStream destFile = new FileStream(
    destinationFilePath, // Pass in the destination path.
    FileMode.Create, // Always create new file.
    FileAccess.Write); // Only perform writing.

// Create a new StreamWriter object.
StreamWriter writer = new StreamWriter(destFile);

// Write the string to the file.
writer.WriteLine(data);

// Always close the underlying streams to flush the data to the file
// and release any file handles.
writer.Close();
destFile.Close();
```

Demonstration: Generating the Grades Report Lab

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: Generating the Grades Report Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_DEMO.md.

Lab: Generating the Grades Report

Scenario

You have been asked to upgrade the Grades Prototype application to enable users to save a student's grades as an XML file on the local disk. The user should be able to click a new button on the StudentProfile view that asks the user where they would like to save the file, displays a preview of the data to the user, and asks the user to confirm that they wish to save the file to disk. If they do, the application should save the grade data in XML format in the location that the user specified.

Objectives

After completing this lab, you will be able to:

- Serialize data to a memory stream.
- Deserialize data from a memory stream.
- Save serialized data to a file.

Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD06_LAK.md.

Exercise 1: Serializing Data for the Grades Report as XML

Scenario

In this exercise, you will write code that runs when the user clicks the **Save Report** button on the **Student Profile** view. You will enable a user to specify where to save the Grade Report, and to serialize the grades data so it is ready to save to a file.

You will use the **SaveFileDialog** object to ask the user for the file name and location where they want to save the file. You will extract the grade data from the application data source and store it in a list of Grade objects.

You will then write the **FormatAsXMLStream** method. This method will use an **XmlWriter** object to create an XML document and populate it with grade information from the list of Grade objects. Finally, you will debug the application and view the data held in the memory stream.

Exercise 2: Previewing the Grades Report

Scenario

In this exercise, you will write code to display a preview of the report to the user before saving it.

First, you will add code to the **SaveReport_Click** method to display the XML document to the user in a message box. To display the document, you need to build a string representation of the XML document that is stored in the **MemoryStream** object. Finally, you will verify that your code functions as expected by running the application and previewing the contents of a report.

Exercise 3: Persisting the Serialized Grade Data to a File

Scenario

In this exercise, you will write the grade data to a file on the local disk.

You will begin by modifying the existing preview dialog box to ask the user if they wish to save the file. If they wish to save the file, you will use a **FileStream** object to copy the data from the **MemoryStream** to a physical file. Then you will run the application, generate and save a report, and then verify that the report has been saved in the correct location in the correct format.

Module Review and Takeaways

In this module, you have learned how to work with the file system by using a number of classes in the System.IO namespace, and how to serialize application data to different formats.

Review Questions

Question: You are a developer working on the Fourth Coffee Windows Presentation Foundation (WPF) client application. You have been asked to store some settings in a plain text file in the user's temporary folder on the file system. Briefly explain which classes and methods you could use to achieve this.

Question: You are a developer working for Fourth Coffee. A bug has been raised and you have been asked to investigate. To help reproduce the error, you have decided to add some logic to persist the state of the application to disk, when the application encounters the error. All the types in the application are serializable, and it would be advantageous if the persisted state was human readable. What approach will you take?

Check Your Knowledge

| Question |
|---|
| You are a developer working for Fourth Coffee. You have been asked to write some code to process a 100 GB video file. Your code needs to transfer the file from one location on disk, to another location on disk, without reading the entire file into memory. Which classes would you use to read and write the file? |
| Select the correct answer. |
| The MemoryStream, BinaryReader and BinaryWriter classes. |
| The FileStream, BinaryReader and BinaryWriter classes. |
| The BinaryReader and BinaryWriter classes. |
| The FileStream, StreamReader and StreamWriter classes. |
| The MemoryStream, StreamReader and StreamWriter classes. |

Module 7

Accessing a Database

Contents:

| | |
|---|------|
| Module Overview | 7-1 |
| Lesson 1: Creating and Using Entity Data Models | 7-2 |
| Lesson 2: Querying Data by Using LINQ | 7-8 |
| Lab: Retrieving and Modifying Grade Data | 7-13 |
| Module Review and Takeaways | 7-15 |

Module Overview

Many applications require access to data that is stored in a database. Microsoft® Visual Studio® 2017 and the Microsoft .NET Framework provide tools and functionality that you can use to easily access, query, and update data.

In this module, you will learn how to create and use entity data models (EDMs) and how to query many types of data by using Language-Integrated Query (LINQ).

Objectives

After completing this module, you will be able to:

- Create, use, and customize an EDM.
- Query data by using LINQ.

Lesson 1

Creating and Using Entity Data Models

Data access applications have traditionally been tedious to develop. They often contain queries that are written as text strings that cannot be type-checked or syntax-checked at compile time, and results are returned as untyped data records. The ADO.NET Entity Framework solves these problems and simplifies the process of developing data access applications by using EDMs.

In this lesson, you will learn how to use the ADO.NET Entity Data Tools to create EDMs, how to customize the classes that the tools generate, and how to access the entities in the generated model.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the ADO.NET Entity Framework.
- Use the ADO.NET Entity Model Tools.
- Customize generated classes.
- Read and modify data by using the Entity Framework.

Introduction to the ADO.NET Entity Framework

Historically when you write code to access data that is stored in a database, you have to understand the structure of the data in the database and how it all interrelates. Often it is stored in a normalized fashion, where tables do not logically map to the real-life objects that they represent. The ADO.NET Entity Framework enables you to develop applications that target a conceptual model instead of the normalized database structure in the storage layer.

The ADO.NET Entity Framework provides the following:

- *EDMs*. These are models that you can use to map database tables and queries to .NET Framework objects.
- *Entity Structured Query Language (SQL)*. This is a storage independent query language that enables you to query and manipulate EDM constructs.
- *Object Services*. These are services that enable you to work with the Common Language Runtime (CLR) objects in a conceptual model.

These components enable you to:

- Write code against a conceptual model that includes types that support inheritance and relationships.
- Update applications to target a different storage model without rewriting or redistributing all of your data access code.
- Write standard code that is not dependent on the data storage system.
- Write data access code that supports compile-time type-checking and syntax-checking.

- The ADO.NET Entity Framework provides:
 - EDMs
 - Entity SQL
 - Object Services

- The ADO.NET Entity Framework supports:
 - Writing code against a conceptual model
 - Easy updating of applications to a different data source
 - Writing code that is independent from the storage system
 - Writing data access code that supports compile-time type-checking and syntax-checking

The conceptual model that you work with in the Entity Framework describes the semantics of the business view of the data. It defines entities and relationships in a business sense and is mapped to the logical model of the underlying data in the data source. For example, in a human resources application, entities may include employees, jobs, and branch locations. An entity is a description of the items and their properties, and they are linked by relationships, such as an employee being related to a particular branch location.

 **Additional Reading:** For more information about the ADO.NET Entity Framework, refer to the ADO.NET Entity Framework page at <http://go.microsoft.com/fwlink/?LinkId=267806>.

Using the ADO.NET Entity Data Model Tools

Visual Studio 2017 provides the Entity Data Model Tools that you can use to create and update EDMs in your applications. It supports both database-first design and code-first design:

- *Database-first design.* In database-first design, you design and create your database before you generate your model. This is commonly used when you are developing applications against an existing data source; however, this can limit the flexibility of the application in the long term.
- *Code-first design.* In code-first design, you design the entities for your application and then create the database structure around these entities. Developers prefer this method because it enables you to design your application around the business functionality that you require. However, in reality, you often have to work with an existing data source.

• Tools support:

- Database-first design by using the Entity Data Model Wizard
- Code-first design by using the Generate Database Wizard

• They also provide:

- Designer pane for viewing, updating, and deleting entities and their relationships
- Update Model Wizard for updating a model with changes that are made to the data source
- Mapping Details pane for viewing, updating, and deleting mappings

Using the Entity Data Model Tools

Visual Studio 2017 provides the ADO.NET Entity Data Model Tools, which include the Entity Data Model Designer for graphically creating and relating entities in a model and three wizards for working with models and data sources. The following table describes the wizards.

| Wizard | Description |
|--------------------------|--|
| Entity Data Model Wizard | Enables you to generate a new conceptual model from an existing data source by using the database-first design method. |
| Update Model Wizard | Enables you to update an existing conceptual model with changes that are made to the data source on which it is based. |
| Generate Database Wizard | Enables you to generate a database from a conceptual model that you have designed in the Entity Data Model Designer by using the code-first design method. |

When you create a model by using the Entity Data Model Wizard, the model opens in the Designer pane, displaying the entities that the wizard has generated and the relationships between them. You can use this pane to add, modify, and delete entities and relationships.

By default, when you create a model from a database, the Entity Designer automatically generates the mappings from the data source to the conceptual model. You can view, modify, and delete these mappings in the Mapping Details pane.

 **Additional Reading:** For more information about the Entity Data Model Tools, refer to the ADO.NET Entity Data Model Tools page at <http://go.microsoft.com/fwlink/?LinkId=267807>.

Demonstration: Creating an Entity Data Model

In this demonstration, you will use the Entity Data Wizard to generate an EDM for an existing database.

Demonstration Steps

You will find the steps in the “**Demonstration: Creating an Entity Data Model**” section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Customizing Generated Classes

When you use the Entity Data Model Wizard to create a model, it automatically generates classes that expose the entities in the model to your application code. These classes contain properties that provide access to the properties in the entities.

The following code example shows an **Employee** class that the Entity Data Model Wizard generated.

- Do not modify the automatically generated classes in a model
- Use partial classes and partial methods to add business functionality to the generated classes

```
public partial class Employee
{
    partial void OnDateOfBirthChanging(DateTime? value)
    {
        if (GetAge() < 16)
        {
            throw new Exception("Employees must be 16 or over");
        }
    }
}
```

Wizard-Generated Classes

```
namespace FourthCoffee.Employees
{
    using System;
    using System.Collections.Generic;

    public partial class Employee
    {
        public int EmployeeID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public Nullable<System.DateTime> DateOfBirth { get; set; }
        public Nullable<int> Branch { get; set; }
        public Nullable<int> JobTitle { get; set; }

        public virtual Branch Branch1 {get; set; }
        public virtual JobTitle JobTitle1 {get; set; }
    }
}
```

```
}
```

You may find that you want to add custom business logic to the entity classes; however, if at any time in the future you run the Update Model Wizard, the classes will be regenerated, and your code will be overwritten. However, the generated classes are defined as partial classes.

The **partial** keyword allows certain construct's definitions to be split into multiple parts, across multiple source files. Classes, structs, and interfaces can all be partial. When using the **partial** modifier, all definitions of that type must contain the **partial** keyword, and all must exist in the same namespace within the same assembly. Partial types cannot be split across multiple .dll and .exe files. When the compiler encounters a partial definition, it searches the assembly for others of the same type and combines all their content to a single type. A type may be split into as many or as few partial definitions as you'd like. A single partial definition is completely valid.

Entity Data Model Wizard will always produce its classes as partial; therefore, you can extend them to add custom functionality to the classes.

For example, if you have a date of birth property in your model, you could write a **GetAge** method in a partial class to enable a run-time calculation of the employee's age.

The following code example shows how you can add business logic to a generated class by using a partial class.

Adding Business Logic in a Partial Class

```
public partial class Employee
{
    public int GetAge()
    {
        DateTime DOB = (DateTime)_DateOfBirth;
        TimeSpan difference = DateTime.Now.Subtract(DOB);
        int ageInYears = (int)(difference.Days / 365.25);
        return ageInYears;
    }
}
```

 **Additional Reading:** For more information about partial classes, refer Partial Classes and Methods (C# Programming Guide) at <http://go.microsoft.com/fwlink/?LinkId=267808>.

Reading and Modifying Data by Using the Entity Framework

The automatically generated code files for a model also contains a partial class that inherits from the **System.Data.Entity.DbContext** class.

The **DbContext** class provides facilities for querying and working with entity data as objects. It contains a default constructor which initializes the class by using the connection string that the wizard generates in the application configuration file. This defines the data connection and model definition to use. The **DbContext** class also contains a **DbSet** property that exposes a **DbSet(TEntity)** class for each entity in your model. The **DbSet(TEntity)** class represents a typed entity set that you can use to read, create, update, and delete data.

• Reading data

```
FourthCoffeeEntities DBContext = new FourthCoffeeEntities();

// Print a list of employees.
foreach (FourthCoffee.Employees.Employee emp in
DBContext.Employees)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

• Modifying data

```
var emp = DBContext.Employees.First(e => e.LastName ==
"Prescott");
if (emp != null)
{
    emp.LastName = "Forsyth";
    DBContext.SaveChanges();
}
```

The following code example shows the class for the FourthCoffeeEntities model.

FourthCoffeeEntities Class

```
public partial class FourthCoffeeEntities : DbContext
{
    public FourthCoffeeEntities() : base("name=FourthCoffeeEntities")
    {

        public DbSet<Branch> Branches { get; set; }
        public DbSet<Employee> Employees { get; set; }
        public DbSet<JobTitle> JobTitles { get; set; }
    }
}
```

To use the typed entity set, you create an instance of the **DbContext** class and then access the properties by using the standard dot notation.

The following code example shows how to read and update data by using the **DbSet(TEntity)** class.

Reading Data

```
FourthCoffeeEntities DBContext = new FourthCoffeeEntities();

// Print a list of employees.
foreach (FourthCoffee.Employees.Employee emp in DBContext.Employees)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

The **DbSet(TEntity)** class implements the **IEnumerable** interface which provides a number of extension methods that enable you to easily locate specific data in the source. For example, the **First** extension method locates the first match for the specified condition, such as a last name of Prescott.

The following code example shows how to use the **First** extension method to locate an employee and then how to update the data by using standard dot notation.

Locating and Modifying Data

```
// Update the employee with a surname of "Prescott."
var emp = DBContext.Employees.First(e => e.LastName == "Prescott");
if (emp != null)
{
    emp.LastName = "Forsyth";
}
```



Additional Reading: For more information about the **DbSet(Entity)** class, refer to the **DbSet(Entity)** Class page at <https://aka.ms/moc-20483c-m7-pg1>.

For more information about the **Enumerable** methods, refer to the **Enumerable Methods** page at <https://aka.ms/moc-20483c-m7-pg2>.

After you change the data in the model, you must explicitly apply those changes to the data in the data source. You can do this by calling the **SaveChanges** method of the **ObjectContext** object.

The following code example shows how to use the **SaveChanges** method.

Persisting Changes To The Database

```
DbContext.SaveChanges();
```

Demonstration: Reading and Modifying Data in an EDM

In this demonstration, you will use the **ObjectSet(TEntity)** class to read and modify data in an EDM.

Demonstration Steps

You will find the steps in the **Demonstration: Reading and Modifying Data in an EDM** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Lesson 2

Querying Data by Using LINQ

As an alternative to using the Entity Framework for querying data, you can use LINQ. This also supports compile-time syntax-checking and type-checking and also uses Microsoft IntelliSense® in Visual Studio. LINQ defines a range of standard query operators that enable you to retrieve exactly the data that you require in a declarative way.

In this lesson, you will learn how to query data and use anonymous methods and how to force query execution to override the default deferred query execution behavior.

Lesson Objectives

After completing this lesson, you will be able to:

- Query data.
- Query data by using anonymous types.
- Force query execution.

Querying Data

You can use LINQ to query data from a wide range of data sources, including .NET Framework collections, Microsoft SQL Server® databases, ADO.NET data sets, and XML documents. In fact, you can use it to query any data source that implements the **IEnumerable** interface.

The syntax of all LINQ queries has the same basis, as follows:

```
from <variable names> in <data source>
select <variable names>
```

However, you can customize this syntax in many ways to retrieve exactly the data that you require in the format that you want. The following code examples all use LINQ to Entities to query data in an EDM; however, the syntax of the query itself does not change if you use a different type of data source.

Selecting Data

The following code example shows how to use a simple **select** clause to return all of the data in a single entity.

Using a select Clause

```
IQueryable<Employee> emps = from e in DBContext.Employees
                                select e;
```

• Use LINQ to query a range of data sources, including:

- .NET Framework collections
- SQL Server databases
- ADO.NET data sets
- XML documents

• Use LINQ to:

- Select data
- Filter data by row
- Filter data by column

The return data type from the query is an **IQueryable<Employee>**, enabling you to iterate through the data that is returned.

Filtering Data by Row

The following code example shows how to use the **where** keyword to filter the returned data by row to contain only employees with a last name of Prescott.

Using a where Clause

```
string _LastName = "Prescott";
IQueryable<Employee> emps = from e in DBContext.Employees
                                where e.LastName == _LastName
                                select e;
```

Filtering Data by Column

The following code example shows how to declare a new type in which to store a subset of columns that the query returns; in this case, just the **FirstName** and **LastName** properties of the **Employee** entity.

Using a New Class to Return a Subset of Columns

```
private class FullName
{
    public string Forename { get; set; }
    public string Surname { get; set; }
}

private void FilteringDataByColumn()
{
    IQueryable<FullName> names = from e in DBContext.Employees
                                    select new FullName { Forename
= e.FirstName, Surname = e.LastName };
}
```

Working with the Results

To then work with the data that is returned from any of these queries, you use dot notation to access the properties of the members of the **IQueryable<>** type, as the following code example shows.

Accessing the Returned Data

```
foreach (var emp in emps)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

Demonstration: Querying Data

In this demonstration, you will use LINQ to Entities to query data.

Demonstration Steps

You will find the steps in the **Demonstration: Querying Data** section on the following page:
https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Querying Data by Using Anonymous Types

In the examples in the previous topic and demonstration, the return data was always stored in a strongly typed **IQueryable<Type>** variable; however, in the filtering by column scenario, it is necessary to define the type containing a subset of columns before defining the query. Although this is a perfectly valid way of working, it can become tedious to explicitly define multiple classes.

You can use anonymous types to store the returned data by declaring the return type as an implicitly typed local variable, a **var**, and by using the **new** keyword in the **select** clause to create the instance of the type.

Use LINQ and anonymous types to:

- Filter data by column
- Group data
- Aggregate data
- Navigate data

Filtering Data by Column

The following code example shows how to use the **var** data type and the **new** keyword in the **select** clause to filter the returned data by column.

Using an Anonymous Type to Return a Subset of Columns

```
var names = from e in DBContext.Employees  
            select new { e.FirstName, e.LastName };
```

Anonymous types enable you to perform more complex queries in LINQ.

Grouping Data

The following code example shows how to use a **group** clause to group the returned employees by their job title ID.

Using a group Clause

```
var emps = from e in DBContext.Employees  
           group e by e.JobTitle into eGroup  
           select new { Job = eGroup.Key, Names = eGroup };
```

Aggregating Data

The following code example shows how to use a **group** clause with an aggregate function to count the number of employees with each job title.

Using a group Clause with an Aggregate Function

```
var emps = from e in DBContext.Employees  
           group e by e.JobTitle into eGroup  
           select new { Job = eGroup.Key, CountOfEmployees = eGroup.Count() };
```

Navigating Data

The following code example shows how to use navigation properties to retrieve data from the **Employees** entity and the related **JobTitles** entity.

Using Dot Notation to Navigate Related Entities

```
var emps = from e in DBContext.Employees
           select new
           {
               FirstName = e.FirstName, LastName = e.LastName, Job =
               e.JobTitle1.Job
           };
```

Demonstration: Querying Data by Using Anonymous Types

In this demonstration, you will use LINQ to Entities to query data by using anonymous types.

Demonstration Steps

You will find the steps in the **Demonstration: Querying Data by Using Anonymous Types** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Forcing Query Execution

By default, when you define a LINQ query that returns a sequence of values, it is not run until you actually try to use some of the returned data. This feature is known as *deferred query execution* and ensures that you can create a query to retrieve data in a multiple-user scenario and know that whenever it is executed you will receive the latest information.

In the following code example, the query is not actually executed until the start of the **foreach** block.

- Deferred query execution—default behavior for most queries
- Immediate query execution—default behavior for queries that return a singleton value
- Forced query execution—overrides deferred query execution:
 - **ToArray**
 - **ToDictionary**

```
IList<Employee> emp = (from e in FCEntities.Employees
                           orderby e.LastName
                           select e).ToList();
```

Deferred Query Execution

```
IQueryable<Employee> emps = from e in DBContext.Employees
                                select e;

foreach (var emp in emps)
{
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);
}
```

Conversely, when you define a LINQ query that returns a singleton value, for example, an **Average**, **Count**, or **Max** function, the query is run immediately. This is known as *immediate query execution* and is necessary in the singleton result scenario because the query must produce a sequence to calculate the singleton result.

You can override the default deferred query execution behavior for queries that do not produce a singleton result by calling one of the following methods on the query:

- **ToArray**
- **ToDictionary**
- **ToList**

In the following code example, the query is executed immediately after it is defined.

Forcing Query Execution

```
IList<Employee> emps = (from e in DBContext.Employees  
                           select e).ToList()  
  
foreach (var emp in emps)  
{  
    Console.WriteLine("{0} {1}", emp.FirstName, emp.LastName);  
}
```

Demonstration: Retrieving and Modifying Grade Data Lab

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: Retrieving and Modifying Grade Data Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_DEMO.md.

Lab: Retrieving and Modifying Grade Data

Scenario

You have been asked to upgrade the prototype application to use an existing SQL Server database. You begin by working with a database that is stored on your local machine and decide to use the Entity Data Model Wizard to generate an EDM to access the data. You will need to update the data access code for the Grades section of the application, to display grades that are assigned to a student and to enable users to assign new grades. You also decide to incorporate validation logic into the EDM to ensure that students cannot be assigned to a full class and that the data that users enter when they assign new grades conforms to the required values.

After completing this lab, you will be able to:

- Create an EDM from an existing database.
- Update data by using the .NET Entity Framework.
- Extend an EDM to validate data.

Lab Setup

Estimated Time: **75 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD07_LAK.md.

Exercise 1: Creating an Entity Data Model from The School of Fine Arts Database

Scenario

In this exercise, you will use the Entity Data Model Wizard to generate an EDM from the **SchoolGradesDB** SQL Server database and then review the model and the code that the wizard generates.

Exercise 2: Updating Student and Grade Data by Using the Entity Framework

Scenario

In this exercise, you will add functionality to the prototype application to display the grades for a user. The grade information in the database stores the subject ID for a grade, so you will add code to the application to convert this to the subject name for display purposes. You will also add code to display the **Add Grade** view to the user and then use the information that the user enters to add a grade for the current student. Finally, you will run the application and verify that the grade display and grade-adding functionality works as expected.

Exercise 3: Extending the Entity Data Model to Validate Data

Scenario

In this exercise, you will update the application to validate data that the user enters.

First, you will add code to check whether a class is full before enrolling a student and throw an exception if it is. Then you will add validation code to check that a user enters a valid date and assessment grade when adding a grade to a student. Finally, you will run the application and verify that the data validation works as expected.

Module Review and Takeaways

In this module, you learned how to create and use EDMs and how to query many types of data by using LINQ.

Review Questions

Question: What advantages does LINQ provide over traditional ways of querying data?

Check Your Knowledge

| Question |
|---|
| Fourth Coffee wants you to add custom functionality to an existing EDM in its Coffee Sales application. You need to write a method for adding a new product to the application. In which of the following locations should you write your code? |
| Select the correct answer. |
| <input type="radio"/> In the relevant generated class in the EDM project. |
| <input type="radio"/> In a partial class in the EDM project. |

MCT USE ONLY. STUDENT USE PROHIBITED

Module 8

Accessing Remote Data

Contents:

| | |
|--|------|
| Module Overview | 8-1 |
| Lesson 1: Accessing Data Across the Web | 8-2 |
| Lesson 2: Accessing Data by Using OData Connected Services | 8-11 |
| Lab: Retrieving and Modifying Grade Data Remotely | 8-21 |
| Module Review and Takeaways | 8-23 |

Module Overview

Systems often consist of many components and services; some might be hosted within your organization's infrastructure, whereas others could be hosted in data centers anywhere in the world. The ability for applications to be able to interact with such services is a common requirement in modern applications.

In this module, you will learn how to use the request and response classes in the **System.Net** namespace to directly manipulate remote data sources. You will also learn how to use Windows® Communication Foundation (WCF) Data Services to expose and consume an entity data model (EDM) over the web.

Objectives

After completing this module, you will be able to:

- Send data to and receive data from web services and other remote data sources.
- Access data by using WCF Data Services.

Lesson 1

Accessing Data Across the Web

Data is often exposed over the web through web services or other application programming interfaces (APIs). To be able to consume such data sources in your application, you need a way to send and receive messages so that you can establish a connection and ultimately send and receive data.

In this lesson, you will learn how to consume remote data sources such as web services and File Transfer Protocol (FTP) sites, which will include how to create a request, supply credentials for authentication, package data into the request, and process data that is returned in the response.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how the Microsoft® .NET Framework uses requests and responses.
- Expose types from web services by using data contracts.
- Create a request and process a response.
- Provide credentials or a security token to enable the remote data source to perform authentication.
- Send and receive data.

Overview of Web Connectivity in the .NET Framework

The .NET Framework provides the infrastructure to enable you to integrate your applications with remote data sources. The remote data source could be anything from an FTP site to an ASP.NET or WCF Web Service.

When consuming a remote data source in your application, the .NET Framework uses requests and responses to pass messages between the two (or more) systems. This involves the following steps:

1. Initiate a connection to the remote data source. This might include passing a security token or user credentials so that the remote data source can authenticate you.
2. Send a request message to the remote data source. This message may also include any data that the remote data source requires to satisfy the request, such as the identifier for the sales record you want to retrieve.
3. Wait for the remote data source to process your request and issue a response. As a developer, you have no control over how long it might take to receive a response from a web service.
4. Process the response, and extract any data that is included in the response.

- Use the request and response pattern
- Use the classes in the **System.Net** namespace:
 - **WebRequest** (abstract base class)
 - **WebResponse** (abstract base class)
 - **HttpWebRequest**
 - **HttpWebResponse**
 - **FtpWebRequest**
 - **FtpWebResponse**
 - **FileWebRequest**
 - **FileWebResponse**



Note: Not all communications have to include both a request and response message. Depending on the nature of the application, it might only be applicable to send one message. For example, if your application wants to let a web service know that it has finished processing a task, you only need to send a request. This is known as a one-way operation.

Web Connectivity in the .NET Framework

The .NET Framework provides the **System.Net** namespace, which contains several request and response classes that enable you to target different data sources. The following table describes some of these request and response classes.

| Class | Description |
|------------------------|---|
| WebRequest | An abstract class that provides the base infrastructure for any request to a Uniform Resource Identifier (URI). |
| WebResponse | An abstract class that provides the base infrastructure to process any response from a URI. |
| HttpWebRequest | A derivative of the WebRequest class that provides functionality for any HTTP web request. |
| HttpWebResponse | A derivative of the WebResponse class that provides functionality to process any HTTP web response. |
| FtpWebRequest | A derivative of the WebRequest class that provides functionality for any FTP request. |
| FtpWebResponse | A derivative of the WebResponse class that provides functionality to process any FTP response. |
| FileWebRequest | A derivative of the WebRequest class that provides functionality for requesting files. |
| FileWebResponse | A derivative of the WebResponse class that provides functionality to process a file response. |

Depending on whether you want to send a request to a web service by using HTTP or you want to download a file from an FTP site, the .NET Framework provides the necessary classes for you to consume these remote data sources in your applications.



Additional Reading: For more information about the **System.Net** namespace, refer to the **System.Net Namespace page** at <https://aka.ms/moc-20483c-m8-pg1>.

Defining a Data Contract

A remote data source can expose any type of data. For example, a web service can expose binary streams, scalar values, or custom objects. The choice of the type of data that you expose is determined by the requirements of your application, but how you expose it is controlled by the data contracts that you define in your web services.

If you want to expose a custom object from a web service, you must provide metadata that describes the structure of the object. The serialization process uses this metadata to convert your object

Use the **DataContract** and **DataMember** attributes to expose types from a web service

```
[DataContract]
public class SalesPerson
{
    [DataMember]
    public string FirstName { get; set; }

    [DataMember]
    public string LastName { get; set; }

    [DataMember]
    public string Area { get; set; }

    [DataMember]
    public string EmailAddress { get; set; }
}
```

into a transportable format, such as XML or JavaScript Object Notation (JSON). This metadata provides instructions to the serializer that enable you to control which types and members are serialized.

Data Contracts in the .NET Framework

The .NET Framework provides the **System.Runtime.Serialization** namespace, which includes the **DataContract** and **DataMember** attributes. You can use these attributes to define serializable types and members.

The following code example shows how to define a serializable type by using the **DataContract** and **DataMember** attributes.

Defining a Data Contract

```
[DataContract()]
public class SalesPerson
{
    [DataMember()]
    public string FirstName { get; set; }

    [DataMember()]
    public string LastName { get; set; }

    [DataMember()]
    public string Area { get; set; }

    [DataMember()]
    public string EmailAddress { get; set; }
}
```

 **Additional Reading:** For more information about the **DataContract** and **DataMember** attributes, refer to the **DataContractAttribute** Class page at <https://aka.ms/moc-20483c-m8-pg2>.

Creating a Request and Processing a Response

The protocol that your remote data source uses determines the request and response classes you must use. Irrespective of the classes you use, you can apply the same pattern to send a request and receive a response.

The **HttpWebRequest** and **HttpWebResponse** Classes

The following steps describe how to send an HTTP request to a web service and process the response by using the **HttpWebRequest** and **HttpWebResponse** classes:

- Get a URI

```
var uri =
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";
```
- Create a request object

```
var request = WebRequest.Create(uri) as HttpWebRequest;
```
- Get a response object from the request object

```
var response = request.GetResponse() as HttpWebResponse;
```
- Read the properties in the response object

```
var status = response.StatusCode;
// Returns OK if a response is received.
```

1. Get a URI to the web service to which you want to send a request. The following code example shows an HTTP URI that addresses the **GetSalesPerson** method in the Fourth Coffee Sales Service.

```
var uri = "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";
```

2. Create a request object to configure the request and access the response. The following code example shows how to create an **HttpWebRequest** object.

```
var request = WebRequest.Create(uri) as HttpWebRequest;
```

 **Note:** Regardless of the type of request object you require, you always use the static **Create** method that the **WebRequest** base class exposes and then cast to the type of request you require.

3. Get the response from the request object. The following code example shows how to get the response from an **HttpWebRequest** object.

```
var response = request.GetResponse() as HttpWebResponse;
```

 **Note:** Similar to creating a request object, you create a response object by invoking the **GetResponse** method on the **WebRequest** object, and you then cast to the type of response you require.

4. Access and process the response by using the various members that the **WebResponse** object provides. The following code example shows how to use and view the status of the response by using the **StatusCode** property.

```
var status = response.StatusCode; // Returns OK if a response is received.
```

If the remote data source uses a different protocol, such as FTP, you can apply the same pattern but use the **FtpWebRequest** and **FtpWebResponse** classes instead.

Handling Network Exceptions

When consuming any remote resources, whether an FTP site or an HTTP web service, you cannot guarantee that the resource is online and listening for your request. If you try to send a request to a web service by using the **HttpWebRequest** class and the web service is not online, the **GetResponse** method call will throw a **WebException** exception with the following message:

WebException – The remote server returned an error: (404) Not Found.

Similarly, if you try to access a secure web service with the wrong credentials or security token, the **GetResponse** method call will throw a **WebException** exception with the following message:

WebException – The remote server returned an error: 401 unauthorized

If you do not handle these exceptions in your code, they will cause your application to fail, offering a poor user experience. Therefore, as a minimum, you should enclose any logic that communicates with a remote data source in a **try/catch** statement, with the **catch** block handling exceptions of type **WebException**.

Authenticating a Web Request

Remote data sources are often protected to prevent unauthorized users from using the service and gaining access to data. Exposing an unprotected data source over the web can lead to unauthorized users sending requests and increasing the load on the data source. There are many ways in which you can secure remote data sources. One approach is to authenticate each user who attempts to connect to the remote data source.

Type of Authentication

The following table describes some of the common authentication techniques that you can use to secure remote data sources.

| Authentication mechanism | Description |
|--------------------------|---|
| Basic | Enables users to authenticate by using a user name and password. Basic authentication does not encrypt the credentials while in transit, which means that unauthorized users may access the credentials. |
| Digest | Enables users to authenticate by using a user name and password, but unlike basic authentication, the credentials are encrypted. |
| Windows | Enables clients to authenticate by using their Windows domain credentials. Windows authentication uses either hashing or a Kerberos token to securely authenticate users. Windows authentication is typically used to provide a single sign-on (SSO) experience in organizations. |
| Certificate | Enables only clients that have the correct certificate installed to authenticate with the service. |

The nature of the service and where it is hosted are likely to influence an organization's choice of authentication mechanism. For example, a service that is exposed over an organization's intranet might use Windows authentication so that users can authenticate by using their Active Directory® Domain Services (AD DS) credentials. Using Windows authentication in this scenario will provide the users with an SSO experience and not require them to remember credentials for each service they consume.

The .NET Framework provides a number of classes that you can use to authenticate with a secure remote data source.

Authenticating Users by Using Credentials

When communicating with a remote data source that requires a user name and password, you use the **Credentials** property that is exposed by any class that is derived from the **WebRequest** base class to pass the credentials to the data source. You can set the **Credentials** property to any object that implements the **ICredentials** interface:

- Create the request object

```
var uri =
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";
var request = WebRequest.Create(uri) as HttpWebRequest;
```

- Use the **NetworkCredential** class

```
var username = "jespera";
var password = "Pa$$w0rd";
request.Credentials = new NetworkCredential(username, password);
```

- Use the **CredentialCache** class

```
request.Credentials = CredentialCache.DefaultCredentials;
```

- Use the **X509Certificate2** class

```
var certificate = FourthCoffeeCertificateServices.GetCertificate();
request.ClientCertificates.Add(certificate);
```

- The **NetworkCredential** class implements the **ICredentials** interface and enables you to encapsulate a user name and password. The following code example shows how to instantiate a **NetworkCredential** object and pass values for the user name and password to the class constructor.

```
var uri = "http://Sales.FourthCoffee.com/SalesService.svc/GetSalesPerson";
var request = WebRequest.Create(uri) as HttpWebRequest;
var username = "jespera";
var password = "Pa$$w0rd";
request.Credentials = new NetworkCredential(username, password);
```

- The **CredentialCache** class provides a number of members that enable you to get credentials in the form of an **ICredentials** object. These members include the **DefaultCredentials** property, which returns an **ICredentials** object containing the credentials that the user is currently logged on with. The following code example shows how to use the **DefaultCredentials** property to get the current user's credentials.

```
var uri = "http://Sales.FourthCoffee.com/SalesService.svc/GetSalesPerson";
var request = WebRequest.Create(uri) as HttpWebRequest;
request.Credentials = CredentialCache.DefaultCredentials;
```

Authenticating users by using basic or Windows authentication is common, but it does require users to remember and provide credentials. Alternatively, you may choose to authenticate clients by using an X509 certificate.

 **Additional Reading:** For more information about the **NetworkCredential** class, refer to the **NetworkCredential Class page** at <https://aka.ms/moc-20483c-m8-pg3>.

Authenticating Users by Using an X509 Certificate

You can use an X509 certificate as a security token to authenticate users. Instead of users having to specify credentials when they access the remote data source, they will automatically gain access as long as the request that is sent includes the correct X509 certificate. To the users, this means they need the correct X509 certificate in their certificate store on the computer where the client application is running.

The following code example shows how you can create an **HttpWebRequest** object and add an X509 certificate to the request object by using the **ClientCertificates** property.

HttpWebRequest and x509 Certificates

```
var uri = "http://Sales.FourthCoffee.com/SalesService.svc/GetSalesPerson";
var request = WebRequest.Create(uri) as HttpWebRequest;
var certificate = FourthCoffeeCertificateServices.GetCertificate();
request.ClientCertificates.Add(certificate);
```

When the remote data source receives the request, it must extract and process the X509 certificate. If the X509 certificate is invalid, the remote data source can return a suitable response, such as a "The remote server returned an error: 401 unauthorized" message.

 **Note:** The **GetCertificate** method of the web service that is shown in the previous example returns an **X509Certificate2** object. This type provides programmatic access to the properties of the X509 certificate.

Sending and Receiving Data

You use requests and responses to send data to or retrieve data from a remote data source. For example, you may want to add a new record to a web service or retrieve a file from an FTP site.

Each derivative of the **WebRequest** and **WebResponse** classes enables you to send and receive data by using the specific protocol that the class implements. For example, the **HttpWebRequest** class enables you to send requests to a web service by using the HTTP protocol.

The **WebRequest** base class includes the following members that you can use to send data to a remote data source:

- **ContentType**. This property enables you to set the type of data that the request will send. For example, if you are sending JSON data by using the **HttpWebRequest** class, you use the **application/json** content type.
- **Method**. This property enables you to set the type of method that the **WebRequest** object will use to send the request. For example, if you are uploading a file by using the **FtpWebRequest** class, you use the **STOR** method.
- **ContentLength**. This property enables you to set the number of bytes that the request will send.
- **GetRequestStream**. This method enables you to access and write data to the underlying data stream in the request object.

The **WebResponse** class provides the **GetResponseStream** method, which enables you to access and stream data from the response.

Sending Data to a Remote Data Source

Whether you are sending data to a web service or uploading a file to an FTP site, the process for creating the request remains the same:

1. Get the URI to the remote data source and the data you want to send.
2. Create the request object.
3. Configure the request object, which includes setting the request method and the length of the data that the request will send.
4. Stream the data to the request object.

The following code example shows how to use a POST operation to send a JSON string to a web service by using the **HttpWebRequest** class.

Sending Data to a Web Service

```
// Get the URI and the data.  
var uri = "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";  
var rawData =  
Encoding.Default.GetBytes("{\"emailAddress\":\"jesper@fourthcoffee.com\"}");  
// Create the request object.  
var request = WebRequest.Create(uri) as HttpWebRequest;  
// Configure the type of data the request will send.  
request.Method = "POST";  
request.ContentType = "application/json";  
request.ContentLength = rawData.Length;
```

Send data

```
var uri =  
    "http://sales.fourthcoffee.com/SalesService.svc/GetSalesPerson";  
var rawData = Encoding.Default.GetBytes(  
    "{\"emailAddress\":\"jesper@fourthcoffee.com\"}");  
var request = WebRequest.Create(uri) as HttpWebRequest;  
request.Method = "POST";  
request.ContentType = "application/json";  
request.ContentLength = rawData.Length;  
var dataStream = request.GetRequestStream();  
dataStream.Write(rawData, 0, rawData.Length);  
dataStream.Close();
```

Process the response

```
var response = request.GetResponse() as HttpWebResponse;  
var stream = new StreamReader(response.GetResponseStream());  
// Code to process the stream.  
stream.Close();
```

```
// Stream the data to the request.  
var dataStream = request.GetRequestStream();  
dataStream.Write(rawData, 0, rawData.Length);  
dataStream.Close();
```

The following code example shows how to upload a file to an FTP site by using the **FtpWebRequest** class.

Uploading a File to an FTP Site

```
// Get the URI and the data.  
var uri = "ftp://sales.fourthcoffee.com/FileRepository/SalesForcast.xls";  
var rawData = File.ReadAllBytes("C:\\FourthCoffee\\Documents\\SalesForecast.xls");  
// Create the request object.  
var request = WebRequest.Create(uri) as FtpWebRequest;  
// Configure the type of data the request will send.  
request.Method = WebRequestMethods.Ftp.UploadFile;  
request.ContentLength = rawData.Length;  
// Stream the data to the request.  
var dataStream = request.GetRequestStream();  
dataStream.Write(rawData, 0, rawData.Length);  
dataStream.Close();
```

Receiving Data from a Remote Data Source

To get data that a response might contain, you use the **GetResponseStream** method of the response object. The **GetResponseStream** method returns a stream, which you can read to get the data.

The following code example shows how to use the **GetResponseStream** method to access the response data.

Reading Data from the Response

```
var request = WebRequest.Create(uri) as HttpWebRequest;  
...  
var response = request.GetResponse() as HttpWebResponse;  
var stream = new StreamReader(response.GetResponseStream());  
// Code to process the stream.  
stream.Close();
```

When you have acquired the response stream, you must then verify that the data is in the correct format.

For example, if the response that is returned is in the JSON format, you can use the

DataContractJsonSerializer class to serialize the raw JSON data into an object that you can consume in your code. Alternatively, if the data is in the XML format, you can use the **SoapFormatter** class to deserialize the data or Language-Integrated Query (LINQ) to XML to manually parse the XML.

 **Additional Reading:** For more information about LINQ to XML, refer to the LINQ to XML page at <https://aka.ms/moc-20483c-m8-pg4>.

Demonstration: Consuming a Web Service

In this demonstration, you will use the **HttpWebRequest** and **HttpWebResponse** classes to consume a web service over HTTP.

The application will use the **System.Net** classes to get sales people records from the Fourth Coffee Sale Service. You will send a request that will contain the email address of the sale person record you want to get in the JSON format, process the response, and then display the record details in the UI.

Demonstration Steps

You will find the steps in the **Demonstration: Consuming a Web Service**" section on the following page:
https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD08_DEMO.md.

Lesson 2

Accessing Data by Using OData Connected Services

WCF Data Services follows the Representational State Transfer (REST) architectural model and uses open web standards such as the Open Data Protocol (OData) to expose and consume data over the web. By following these standards, you can build solutions based on WCF Data Services that a wide variety of client applications can easily access, regardless of the technology that is used to implement the client application.

In this lesson, you will learn how to create a WCF Data Service and how to define which entities and operations you want to expose. You will also learn how to create a client library and how to consume the entities and operations that you have exposed.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of a WCF Data Service.
- Define a WCF Data Service.
- Expose a data model by using a WCF Data Service.
- Expose web methods by using a WCF Data Service.
- Create a client library and reference a WCF Data Service.
- Retrieve and manipulate entities that a WCF Data Service exposes.

What Is WCF Data Services?

WCF Data Services enables you to create and access data services over the web. You expose your data as resources that applications can access by using a URI. These resources are exposed as sets of entities that are related by associations, the same concepts as in an EDM. However, you can expose data from many types of storage, including databases and Common Language Runtime (CLR) classes.

WCF Data Services uses URIs to address data and simple, well-known formats to represent that data, such as XML and Atom. This results in data being served as a REST-style resource collection.

REST has become a popular model for implementing web services that need to access data (other models, such as those based on the **WebRequest/WebResponse** model described in the previous lesson, are more suited to invoking remote methods). REST describes a stateless, hierarchical scheme for representing resources and business objects over a network. Resources are accessed through URLs that identify the data to retrieve. For example, Fourth Coffee might choose to make the data for all of its sales people available through the following URI:

<http://FourthCoffee.com/SalesService.svc/SalesPersons>

The data for a specific sales person could be fetched by specifying the identifier (such as a sales person number) for that sales person, like this:

WCF Data Services:

- Enables you to create highly flexible data services
- Uses the REST model for data access

<http://FourthCoffee.com/SalesService.svc/SalesPersons>
<http://FourthCoffee.com/SalesService.svc/SalesPersons/99>
<http://FourthCoffee.com/SalesService.svc/SalesPersons?top=10>

- Enables you to query and modify data by using URLs with standard HTTP verbs: GET, PUT, POST, and DELETE

MCT USE ONLY. STUDENT LICENSED ONLY.

<http://FourthCoffee.com/SalesService.svc/SalesPersons/99>

Similarly, the details of products that it sells might be available through the following URI:

<http://FourthCoffee.com/SalesService.svc/Products>

The details for a specific product could be retrieved by including the product ID in the URI:

<http://FourthCoffee.com/SalesService.svc/Products/25>

 **Note:** The exact URI scheme that a web service uses to expose data is a decision of the organization that implements the web service, but the examples that are shown here illustrate a common pattern.

The REST model performs HTTP GET queries to retrieve data, but the REST model also supports insert, update, and delete operations by using HTTP PUT, POST, and DELETE requests.

The REST model enables a web service to extend URIs to support filtering, sorting, and paging of data. For example, the following URI sends a request that fetches the first 10 sales people only:

<http://FourthCoffee.com/SalesService.svc/SalesPersons?top=10>

The list of filters and other functionality depends on how the web service is implemented, but features such as these are available with WCF Data Services. Additionally, WCF Data Services enables you to extend your web services by writing service operations as methods that perform business logic at the server.

These methods are then accessible as URIs in a similar manner to resources. You can also define interceptors, which are called when you query, insert, update, or delete data and can validate or modify the data, enforce security, or reject the change.

 **Additional Reading:** For more information about WCF Data Services, refer to the [WCF Data Services page](http://go.microsoft.com/fwlink/?LinkId=267815) at <http://go.microsoft.com/fwlink/?LinkId=267815>.

Defining a WCF Data Service

By using WCF Data Services, you can expose data from relational data sources such as Microsoft SQL Server® through an EDM conceptual schema that is created by using the ADO.NET Entity Framework, and you can enable a client application to query and maintain data by using this schema.

 **Note:** WCF Data Services can also expose non-relational data, but this requires building customized classes. WCF Data Services operates most naturally with the model that the ADO.NET Entity Framework presents.

- WCF Data Services is based on the **System.Data.Services DataService** generic class

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    ...
}
```

- URLs are mapped to entity sets by a data service:

<http://FourthCoffee.com/SalesService.svc/SalesPersons>



A WCF Data Service is based on the **System.Data.Services DataService** generic class. This class expects a type parameter that is a collection containing at least one property that implements the **IQueryable** interface, such as the **DbContext** class for an entity set that is defined by using the Entity Framework. The

DataService type implements the basic functionality to expose the entities in this collection as a series of REST resources.

The following code example shows the definition of a WCF Data Service based on a **DbContext** class called **FourthCoffee** that is generated by using the ADO.NET Entity Framework.

Defining a Data Service

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    ...
}
```

You can implement methods in a WCF Data Service that specify the entities to expose from the underlying EDM and that configure the size of datasets that the data service presents. You can also override methods that are inherited from the **DataService** class to customize the way in which the service operates.

By default, WCF Data Services uses a simple addressing scheme that exposes the entity sets that are defined within the specified EDM. When you consume a WCF Data Service, you address these entity resources as an entity set that contains instances of an entity type. For example, suppose that the following URI (shown in the previous topic) returns all of the **SalesPerson** entities that were defined in the EDM that was used to construct a WCF Data Service:

<http://FourthCoffee.com/SalesService.svc/SalesPersons>

The "/SalesPersons" element of the URI points to the SalesPersons entity set, which is the container for SalesPerson instances.

 **Additional Reading:** For more information about defining a WCF Data Service, refer to the [Exposing Your Data as a Service \(WCF Data Services\) page](#) at <http://go.microsoft.com/fwlink/?LinkId=267816>.

Exposing a Data Model by Using WCF Data Services

For security reasons, WCF Data Services does not automatically expose any resources, such as entity collections, that the EDM implements. You specify a policy that enables or disables access to resources in the **InitializeService** method of your data service. This method takes a **DataServiceConfiguration** object, which has a **SetEntitySetAccessRule** property that you use to define the access policy.

The following code example shows how to allow access to all resources that the WCF Data Service exposes.

Configure the access rules on the WCF Data Service by using the **SetEntitySetAccessRule** method

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(
        DataServiceConfiguration config)
    {
        config.SetEntitySetAccessRule("*", EntitySetRights.All);
    }
}
```

Data Service Access to all Entities

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.SetEntitySetAccessRule("*", EntitySetRights.All);
    }
}
```

The parameters to the **SetEntitySetAccessRule** method are the name of a resource and the access rights to grant over that resource. You can specify a resource explicitly, or you can use wildcard characters. The * value that is shown in the code example is a shorthand way of specifying all resources that the WCF Data Service publishes. The **EntitySetRights.All** value grants unrestricted access to these resources.

 **Note:** Enumerations and partial classes that are implemented in an EDM do not propagate through WCF Data Services. If you want to expose enumerations and extensions to partial classes, you should include these types in a separate assembly that you can reference in the service and client application directly.

 **Additional Reading:** For more information about defining access rules in a WCF Data Service, refer to the **Configuring the Data Service (WCF Data Services)** page at <http://go.microsoft.com/fwlink/?LinkId=267817>.

Exposing Web Methods by Using WCF Data Services

The primary purpose of a WCF Data Service is to provide access to data. However, you can also implement custom operations that manipulate data. A WCF Data Service operation is simply a method of the data service that is visible to the REST infrastructure and that can be accessed by sending an HTTP GET, PUT, POST, or DELETE request.

WCF Data Services operations that are accessed by using a GET request should be annotated with the **WebGet** attribute. These operations typically return data, although they may run some business logic that does not return a value. Operations that are accessed by using PUT, POST, or DELETE requests should be annotated with the **WebInvoke** attribute. These operations typically modify the data that the service uses in some way.

Similar to entity sets, you must explicitly enable access to operations that a WCF Data Service exposes. You can do this by calling the **SetServiceOperationAccessRule** method of the **DataServiceConfiguration** object when you initialize the service. You specify the name of the operation and the appropriate access rights.

A data service operation can take parameters and returns one of the following data types:

- **IEnumerable<T>** or **IQueryable<T>** (where T represents an entity type in the service). If an operation returns an enumerable collection based on an entity type that the service recognizes, a client application can perform queries by specifying HTTP URIs in the manner shown in the previous topics in this lesson. Implementing an operation that returns an enumerable collection in this way gives you detailed control over the contents of the collection. It is the responsibility of your code to

Expose operations by using the **WebGet** and **WebInvoke** attributes

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(DataServiceConfiguration config)
    {
        config.SetServiceOperationAccessRule("SalesPersonByEmail",
            ServiceOperationRights.ReadMultiple);
    }

    [WebGet]
    [SingleResult]
    public SalesPerson SalesPersonByEmail(string emailAddress)
    {
        ...
    }
}
```

generate this collection, possibly based on information that the client application provides. The following code example shows an operation that retrieves sales people per area.

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(
        DataServiceConfiguration config)
    {
        ...
        config.SetServiceOperationAccessRule("SalesPersonByArea",
            ServiceOperationRights.ReadMultiple);
    }
    ...
    [WebGet]
    public IQueryable<SalesPerson> SalesPersonByArea(string area)
    {
        if (!String.IsNullOrEmpty(area))
        {
            return from p in this.CurrentDataSource.SalesPerson
                   where String.Equals(p.Area, area)
                   select p;
        }
        else
        {
            throw new ArgumentException("Area must be specified", "area");
        }
    }
}
```

You can invoke this operation by using the following URI:

<http://<hostName>/FourthCoffee/FourthCoffeeDataService.svc/SalesPersonByArea?area='snacks'>

- T (where T represents an entity type in the service). An operation can return a single instance of an entity. The following code example shows an operation that retrieves a sales person record that has a specific email address. Notice that you should also annotate an operation that returns a scalar value with the **SingleResult** attribute.

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(
        DataServiceConfiguration config)
    {
        ...
        config.SetServiceOperationAccessRule("SalesPersonByEmail",
            ServiceOperationRights.ReadMultiple);
    }
    ...
    [WebGet]
    [SingleResult]
    public SalesPerson SalesPersonByEmail(string emailAddress)
    {
        return (from p in this.CurrentDataSource.SalesPerson
                where String.Equals(p.Area, area)
                select p).SingleOrDefault();
    }
}
```

- A primitive value. The following code example shows an operation that retrieves a count of all of the sales people and returns the count value as an **int**.

```
public class FourthCoffeeDataService : DataService<FourthCoffee>
{
    public static void InitializeService(
        DataServiceConfiguration config)
    {
        ...
        config.SetServiceOperationAccessRule("SalesPersonCount",
            ServiceOperationRights.ReadSingle);
    }
    ...
    [WebGet]
    [SingleResult]
    public int SalesPersonCount()
    {
        return (from p in this.CurrentDataSource.SalesPerson
                select p).Count();
    }
}
```

- **void**. Not all operations have to return a value.

 **Additional Reading:** For more information about service operations, refer to the **Service Operations (WCF Data Services) page** at <http://go.microsoft.com/fwlink/?LinkId=267818>.

Referencing a WCF Data Source

The client library for a WCF Data Service consists of a class that is derived from the **DataServiceContext** type that exposes one or more **DataServiceQuery** objects as properties. The name of this class is usually the same as the name of the **DbContext** object that is used by the EDM on which the WCF Data Service is based. For example, the **FourthCoffeeDataService** WCF Data Service uses a **DbContext** object called **FourthCoffeeEntities** to connect to the underlying EDM, so the name of the **DataServiceContext** type that is generated for the client library is also **FourthCoffeeEntities**.

The **DataServiceContext** class performs a similar role to the **DbContext** class in the Entity Framework. A client application connects to the data source through a **DataServiceContext** object and fetches the data for the entities that the data service exposes by using the **DataServiceQuery** properties. Each **DataServiceQuery** property is a generic collection object that presents data from one of the underlying entities that provides the data for the WCF Data Service.

The client library also provides definitions of the types that each **DataServiceQuery** collection contains. A client application can perform LINQ queries against the **DataServiceQuery** collection properties, and the client library constructs the appropriate HTTP request to fetch the corresponding data. The WCF Data Service fetches the matching data and populates the **DataServiceQuery** collection. The client application can then iterate through this collection and retrieve the data for each item.

Client libraries:

- Are derived from the **DataServiceContext** class
- Expose entities that the **DataServiceQuery** collection contains

Create a client library by using:

- The **Add Service Reference** function in Visual Studio
- The **DataSvcUtil** command line utility

You can generate the client library for a WCF Data Service by using the **Add Service Reference** dialog box in Visual Studio or by using the WCF Data Service client utility, DataSvcUtil, from the command line.

Adding a Service Reference

You can use the **Add Service Reference** dialog box in a client application. This dialog box enables you to specify the URL of the WCF Data Service to connect to. The dialog box sends a metadata query to the specified URL, and it uses the response to generate the appropriate **DataServiceContext** class that contains the **DataServiceQuery** properties and the classes for each of the entities that the WCF Data Service exposes. The returned metadata is stored in the client project as an .edmx file. This is not the same as an .edmx file that is generated by using the ADO.NET Entity Data Model Designer (it has a different format), but you can view this metadata file by using the XML editor or any text editor.

To add a data service reference, perform the following steps:

1. If the data service is not part of the solution and is not already running, start the data service and note its URI.
2. In Solution Explorer, right-click the client project, and then select **Add Service Reference**.
3. If the data service is part of the current solution, click **Discover**.
4. Alternatively, if the data service is not part of the current solution, in the **Address** box, type the base URL of the data service, and then click **Go**.
5. Click **OK**.

After you have referenced the WCF Data Service, you can then consume the entities and service operations that it exposes.

 **Additional Reading:** For more information about creating a service reference by using the command line, refer to the **WCF Data Service Client Utility (DataSvcUtil.exe)** page at <http://go.microsoft.com/fwlink/?LinkId=267819>.

Retrieving and Updating Data in a WCF Data Service

After you have referenced the WCF Data Service by generating a client library, you can then consume the EDM and any service operations that the service exposes.

Retrieving Data

To retrieve data from a WCF Data Service by using the client library, perform the following steps:

1. Create an instance of the type that is derived from the **DataServiceContext** class in the client library, and then connect to the WCF Data Service. The constructor for this class expects a **Uri** object that contains the address of the service.
2. Retrieve data by querying the appropriate **DataServiceQuery** collection in the **DataServiceContext** object. When you query a **DataServiceQuery** collection, the client library constructs an HTTP request that specifies the resource and any criteria that is required. The query is transmitted to the WCF Data Service, and the data is returned and used to populate the **DataServiceQuery** object.

• Retrieve entities:

- Use the properties that are exposed by the context
- Invoke custom service operations
- Use eager or explicit loading to get related entities

• Modify entities:

- Use the **AddToXXXX** method to add a new entity
- Use the **DeleteObject** method to remove an existing entity
- Use the **UpdateObject** method to modify an existing entity

3. Iterate through the items in the **DataServiceQuery** collection and process the objects that are returned.

The following code example connects to the FourthCoffeeDataService WCF Data Service by using the **FourthCoffeeEntities** type in the client library (this is the class that is derived from **DataServiceContext**).

The parameter to the constructor is the address of the service. The code then queries the **SalesPersons** **DataServiceQuery** property to fetch all sales people and reads the email address for each record.

Querying the FourthCoffeeDataService WCF Data Service

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri  
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));  
  
foreach (SalesPerson person in context.SalesPersons)  
{  
    var email = product.EmailAddress;  
}
```

You can also modify data by invoking any custom service operations that you may have exposed in the WCF Data Service. You can invoke service operations by using the **Execute** method of the **DataServiceContext** class. The value that the **Execute** method returns is an enumerable collection. If the service operation returns a single, scalar value, you should extract that value from the collection by using a method such as **First**.

The following code example shows how to invoke the **SalesPersonByArea** service operation and iterate the results.

Querying the FourthCoffeeDataService WCF Data Service by Using a Service Operation

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri  
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));  
  
foreach (SalesPerson person in context.Execute<SalesPerson>  
    (new Uri("/SalesPersonByArea?area='snacks'", UriKind.Relative)))  
{  
    var email = product.EmailAddress;  
}
```

Modifying Data

After you have retrieved data, you can modify the entities as you would when working with an EDM and then save your changes back to the WCF Data Service.

The **DataServiceContext** class provides the following methods that enable you to work with the entities in your EDM.

- **AddToXXXX**. These methods enable you to add a new entity to the entity collection. The following code example shows how to use the **AddToSalesPersons** method to add a new **SalesPerson** entity.

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri  
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));  
...  
var newSalesPerson = new SalesPerson  
{  
    Area = "tea",  
    EmailAddress = "roy@fourthcoffee.com",  
    FirstName = "Roy",  
    LastName = "Antebi"  
};  
context.AddToSalesPersons(newSalesPerson);  
context.SaveChanges();
```

- **DeleteObject.** This method enables you to remove an existing object from the entity collection. The following code example shows how to use the **DeleteObject** method to delete a sales person with the email address **roya@fourthcoffee.com**.

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));
...
var salesPerson = (from p in context.SalesPersons
                  where p.EmailAddress.Equals("roy@fourthcoffee.com")
                  select p).Single();
context.DeleteObject(salesPerson);
context.SaveChanges();
```

- **UpdateObject.** This method enables you to update an existing object in the entity collection. The following code example shows how to use the **UpdateObject** method to change the area to which a sales person belongs.

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));
...
var salesPerson = (from p in context.SalesPersons
                  where p.EmailAddress.Equals("roy@fourthcoffee.com")
                  select p).Single();
salesperson.Area = "soft drinks";
context.UpdateObject(salesPerson);
context.SaveChanges();
```

Implementing Eager Loading of Entities

When retrieving data by using WCF Data Services, by default only the entity you requested is returned in the response. For example, the **SalesPerson** entity in the **FourthCoffeeEntities** object is related to the **Sales** entity. When you request a **SalesPerson** entity, the response will not include the related **Sales** entity. However, you can use the **Expand** or **LoadProperty** methods to get related entities.

Using the eager loading strategy that the **Expand** method implements causes the data for the specified related entities to be retrieved as part of the same request that fetches the primary data for the query. This approach is useful if you know that you will always need this related data, but it can be wasteful of bandwidth for the cases where you do not actually use these entities.

The following code example shows how to use the **Expand** method to fetch the sales associated with each **SalesPerson** object.

Eager Loading of Entities

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri
    ("http://FourthCoffee.com/FourthCoffeeDataService.svc"));

var salesPersons = (from s in context.SalesPersons.Expand("Sales")
                    select s).ToList();
```

As an alternative, you can use explicit loading. This strategy sends an additional query to the WCF Data Service that is requesting the related data for a specific object, but it has the advantage that it does not waste bandwidth by automatically fetching data that is not used.

You can implement explicit loading by using the **LoadProperty** method of the **DataServiceContext** object. You call the **LoadProperty** method each time you require data that is related to a particular entity; you specify the entity and the name of the **DataServiceQuery** collection property that holds the related data.

The following code example shows how to use the **LoadProperty** method to fetch the sales that are associated with each **SalesPerson** object.

Explicit Loading of Entities

```
FourthCoffeeEntities context = new FourthCoffeeEntities (new Uri  
("http://FourthCoffee.com/FourthCoffeeDataService.svc"));  
  
foreach (var salesPerson in context.SalesPersons)  
{  
    ...  
    context.LoadProperty(salesPerson, "Sales");  
  
    foreach (var sale in salesPerson.Sales)  
    {  
        ...  
    }  
}
```



Additional Reading: For more information about modifying entities, refer to the **How to: Add, Modify, and Delete Entities (WCF Data Services) page** at <http://go.microsoft.com/fwlink/?LinkId=267820>.

Demonstration: Retrieving and Modifying Grade Data Remotely

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: Retrieving and Modifying Grade Data Remotely** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD08_DEMO.md.

Lab: Retrieving and Modifying Grade Data Remotely

Scenario

Currently, the application retrieves data from a local database. However, you have decided to store the data in the cloud and must configure the application so that it can retrieve data across the web.

You must create a WCF Data Service for the **SchoolGrades** database that will be integrated into the application to enable access to the data.

Finally, you have been asked to write code that displays student images by retrieving them from across the web.

Objectives

After completing this lab, you will be able to:

- Create a WCF Data Service.
- Use an OData Connected Service.
- Retrieve data over the web.

Lab Setup

Estimated Time: **60 minutes**

Ensure that you have followed all the steps listed in the Setup Guide for this course. The Setup Guide contains installations instructions specific to this module's lab. The Setup guide can be found in the **Instructions** folder (<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Instructions>)

The specific installation resources can be found in the **Allfiles** directory:

<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Allfiles/Assets>,

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD08_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD08_LAK.md.

Exercise 1: Creating a WCF Data Service for the SchoolGrades Database

Scenario

In this exercise, you will create a WCF Data Service for the **SchoolGrades** database so that the client application can connect to the database over the web.

First, you will add a new ASP.NET Web Application project to the solution and configure it for the client application. You will then expose the entities in the EDM from a data service in the new project. Next, you will specify the data context for the data service and configure access rights to the entities that it exposes. Finally, you will add an operation to the data service that returns all of the students in a specified class.

Exercise 2: Integrating the Data Service into the Application

Scenario

In this exercise, you will integrate the WCF Data Service into the Grades Prototype application.

First, you will add a service reference in the GradesPrototype project that references the running WCF Data Service. You will then modify the code that accesses the local EDM to use the WCF Data Service instead. Next, you will modify the code that saves changes back to the database to do so through the data service. Finally, you will test the application to verify that it runs the same as if the data was being called locally.

Exercise 3: Retrieving Student Photographs Over the Web (If Time Permits)

Scenario

In this exercise, you will write code that displays student images by retrieving the image from across the web. You will modify the **StudentsPage** window (that displays the list of students in a class), the **StudentProfile** window (that displays the details for an individual student), and the **AssignStudentDialog** window (that displays a list of unassigned students) to include the student photographs. The data for each student contains an **ImageName** property that specifies the filename of the photograph for the student on the web server. These files are located in the **Images\Portraits** folder on the same web server that hosts the data service (in the Web.Grades project.) You will build a value converter class that generates the URL of an image from the **ImageName** property and then use an **Image** control to use the URL to fetch and render the image in each of the specified windows. Finally, you will run the application to verify that the images appear.

Module Review and Takeaways

In this module, you have learned how to use the request and response classes in the **System.Net** namespace to manipulate remote data sources directly and how to use WCF Data Services to expose and consume an entity data model over the web.

Review Questions

Check Your Knowledge

| Question | |
|---|---|
| Which of the following correctly describes how to access data that is provided in an HTTP response? | |
| Select the correct answer. | |
| | Invoke the GetResponseStream static method on the <code>HttpWebResponse</code> class. |
| | Read the <code>ContentLength</code> instance property on the <code>HttpWebResponse</code> object. |
| | Invoke the <code>GetRequestStream</code> instance method on the <code>HttpWebResponse</code> object. |
| | Invoke the <code>GetResponseStream</code> instance method on the <code>HttpWebResponse</code> object. |
| | Invoke the <code>GetResponseStream</code> instance method on the <code>HttpWebRequest</code> object. |

Check Your Knowledge

| Question | |
|--|---|
| When you create a WCF Data Service to provide remote access to an EDM, how do you specify which entity sets the data service should make available to client applications? | |
| Select the correct answer. | |
| | Do nothing. All entity sets in the EDM are automatically available to client applications. |
| | In the <code>InitializeService</code> method of the data service, use the <code>SetEntityAccessRule</code> method of the <code>DataServiceConfiguration</code> object to specify which entity sets should be made available to client applications. |
| | Create a certificate for each client that can connect to the service. Configure the service to only allow authenticated clients to connect and retrieve data. |
| | Define a data contract for each entity set. |
| | Configure the service to enable HTTP GET requests for each entity set. |

MCT USE ONLY
STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 9

Designing the User Interface for a Graphical Application

Contents:

| | |
|--|------|
| Module Overview | 9-1 |
| Lesson 1: Using XAML to Design a User Interface | 9-2 |
| Lesson 2: Binding Controls to Data | 9-12 |
| Lesson 3: Styling a UI | 9-19 |
| Lab: Customizing Student Photographs and Styling the Application | 9-25 |
| Module Review and Takeaways | 9-27 |

Module Overview

An effective and easy-to-use user interface (UI) is essential for graphical applications. In this module, you will learn how to use Extensible Application Markup Language (XAML) and Windows Presentation Foundation (WPF) to create engaging UIs.

Objectives

After completing this module, you will be able to:

- Use XAML to design a UI.
- Bind a XAML control to data.
- Apply styles to a XAML UI.

Lesson 1

Using XAML to Design a User Interface

XAML is a declarative, XML-based markup language that you can use to create UIs for .NET Framework applications. Defining a UI in declarative markup, rather than imperative code, makes your UI more flexible and portable, ensuring that the same application can be used on a variety of devices. Using XAML also helps to separate your application UI from its runtime logic.

In this lesson, you will learn how to use XAML to create simple graphical UIs.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain how to use XAML to define the layout of a UI.
- Describe some of the common controls used by WPF applications.
- Create controls and set properties in XAML.
- Handle events for XAML controls.
- Use layout controls in XAML.
- Create user controls in XAML.

Introducing XAML

XAML enables you to define UI elements by using a declarative, XML-based syntax. When you use XAML in a WPF application, you use XML elements and attributes to represent controls and their properties. You can use a variety of tools to create XAML files, such as Visual Studio, the Microsoft Expression® suite, and text editors. When you build a WPF application, the build engine converts the declarative markup in your XAML file into classes and objects.

- Use XML elements to create controls
- Use attributes to set control properties
- Create hierarchies to represent parent controls and child controls



Note: It's important to note that the XAML language itself is just a mapping of C# classes to declarative code. There's no special engine to treat XAML differently. Everything done with XAML in a declarative way can be achieved with normal C# code.

The following example shows the basic structure of a XAML UI:

Defining a Button in XAML

```
<Window x:Class="MyNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Title="Order Your Coffee Here" Height="350" Width="525">
    <Grid>
        <Button Content="Get Me a Coffee!" />
    </Grid>
</Window>
```

XAML uses a hierarchical approach to define a UI. The most common top-level element in a WPF XAML file is the **Window** element. The **Window** element can include several attributes. In the previous example, the **Window** element identifies various XML namespaces that make built-in controls available for use. It defines a title, which is displayed in the title bar of the application, and defines the initial height and width of the window.

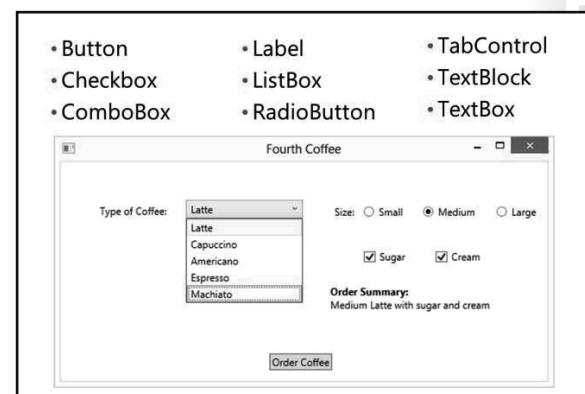
A **Window** element can contain a single child element that defines the content of the UI. In most applications, a UI requires more than a single control, so WPF defines *container* controls that you can use to combine other lower-level controls together and lay them out. The most commonly used container control is the **Grid** control, and when you add a new **Window** to a WPF project in Visual Studio, the **Window** template automatically adds a **Grid** control to the **Window**.

The **Grid** control can contain multiple child controls that it lays out in a grid style. In the example shown above, the **Grid** control contains a single **Button** control. The **Grid** control defines a default layout that consists of a single row and a single column, but you can customize this layout by defining additional rows and columns as attributes of the **Grid**. This mechanism enables you to define cells in the **Grid**, and you can then place controls in specific cells. You can also nest a grid control inside a cell if you need to provide finer control over the layout of certain parts of a window.

 **Additional Reading:** For more information about XAML, refer to the XAML Overview (WPF) page at <http://go.microsoft.com/fwlink/?LinkId=267821>.

Common Controls

The .NET Framework provides a comprehensive collection of controls that you can use to implement a UI. You can find the complete set in the Toolbox that is available when you design a window. You can also define your own custom user controls, as described in this lesson. The following table summarizes some of the most commonly used controls:



| Control | Description |
|--------------------|---|
| Button | Displays a button that a user can click to perform an action. |
| CheckBox | Enables a user to indicate whether an item should be selected (checked) or not (blank). |
| ComboBox | Displays a drop-down list of items from which the user can make a selection. |
| Label | Displays a piece of static text. |
| ListBox | Displays a scrollable list of items from which the user can make a selection. |
| RadioButton | Enables the user to select from a range of mutually exclusive options. |
| TabControl | Organizes the controls in a UI into a set of tabbed pages. |

| Control | Description |
|------------------|--|
| TextBlock | Displays a read-only block of text. |
| TextBox | Enables the user to enter and edit text. |

Setting Control Properties

When you add controls to a XAML window, you can define control properties in various ways. Most controls enable you to set simple property values by using attributes.

The following example shows how to set the properties of a button control by using attributes:

Using Attributes to Set Control Properties

```
<Button Content="Get Me a Coffee!"  
       Background="Yellow"  
       Foreground="Blue" />
```

- Use attribute syntax to define simple property values

```
<Button Content="Click Me" Background="Yellow" />
```

- Use property element syntax to define complex property values

```
<Button Content="Click Me">  
  <Button.Background>  
    <LinearGradientBrush StartPoint="0.5, 0.5"  
                        EndPoint="1.5, 1.5">  
      <GradientStop Color="AliceBlue" Offset="0" />  
      <GradientStop Color="Aqua" Offset="0.5" />  
    </LinearGradientBrush>  
  </Button.Background>  
</Button>
```

This attribute syntax is easy to use and intuitive for developers with XML experience. However, this syntax does not provide the flexibility that you need to define more complex property values. Suppose that instead of setting the button background to a simple text value such as **Yellow**, you want to apply a more complex gradient effect to the button. You cannot define all the nuances of a gradient effect within an attribute value. Instead, XAML supports an alternative way of setting control properties called *property element syntax*. Rather than specifying the **Background** property as an inline attribute of the **Button** element, you can add an element named **Button.Background** as a child element of the **Button** element, and then in the **Button.Background** element, you can add further child elements to define your gradient effect.

The following example shows how to set the properties of a button control by using property element syntax:

Using Property Element Syntax to Set Control Properties

```
<Button Content="Get Me a Coffee!">  
  <Button.Background>  
    <LinearGradientBrush StartPoint="0.5, 0.5" EndPoint="1.5, 1.5" >  
      <GradientStop Color="AliceBlue" Offset="0" />  
      <GradientStop Color="Aqua" Offset="0.5" />  
    </LinearGradientBrush>  
  </Button.Background>  
  <Button.Foreground>  
    <SolidColorBrush Color="Black" />  
  </Button.Foreground>  
</Button>
```

Many WPF controls include a **Content** property. The previous examples used an attribute to set the **Content** property of a button to a simple text value. However, the content of a control is rarely limited to text. Instead of specifying the **Content** property as an attribute, you can add the desired content between the opening and closing tags of the control. For example, you might want to replace the contents of the **Button** element with a picture of a cup of coffee.

The following example shows how to add content to a WPF control:

Adding Content to a WPF Control

```
<Button>
    <Image Source="Images/coffee.jpg" Stretch="Fill" />
</Button>
```

Handling Events

When you create a WPF application in Visual Studio, each XAML page has a corresponding code-behind file. For example, the *MainWindow.xaml* file that Visual Studio creates by default has a code-behind file named *MainWindow.xaml.cs*. You subscribe to event handlers in your XAML markup and then define your event handler logic in the code-behind file.

 **Note:** Visual Studio includes many features that make it easy to create handlers for events. For example, if you double-click a **Button** control at design time, Visual Studio will automatically create an event handler stub method in the code-behind file. It also automatically binds the **Click** event of the button to the event handler method.

- Specify the event handler method in XAML

```
<Button x:Name="btnMakeCoffee"
        Content="Make Me a Coffee!"
        Click="btnMakeCoffee_Click" />
```

- Handle the event in the code-behind class

```
private void btnMakeCoffee_Click(object sender,
                                RoutedEventArgs e)
{
    lblResult.Content = "Your coffee is on its way.";
}
```

- Events are bubbled to parent controls

Suppose you create a simple application that consists of a button and a label. When you click the button, you want an event handler to add some text to the label. To do this, you need to do three things:

1. Make sure the button and the label include a **Name** property, so that you can reference the controls in your code.
2. Set the **Click** attribute of the button to the name of an event handler method. This method runs when the **Click** event occurs.

 **Note:** The name of a control should be unique within the window that holds the control; no two controls in the same window should have the same name.

The following code example shows how to set the event handler method for the **Click** event of a **Button** control:

Handling Events in XAML

```
<Button x:Name="btnMakeCoffee"
        Content="Make Me a Coffee!"
        Click="btnMakeCoffee_Click" />
<Label x:Name="lblResult"
        Content="" />
```

In the code-behind file, you can add logic to the **btnMakeCoffee_Click** method to define what should happen when a user clicks the button.

The following example shows how to create an event handler method for a WPF control:

Creating Event Handler Methods

```
private void btnMakeCoffee_Click(object sender, RoutedEventArgs e)
{
    lblResult.Content = "Your coffee is on its way.";
}
```

 **Note:** WPF uses the concept of “routed events”. When an event is raised, WPF will attempt to run the corresponding event handler for the control that has the focus. If there is no event handler available, then WPF will examine the parent of the control that has the focus, and if it has a handler for the event it will run. If the parent has no suitable event handler, WPF examines the parent of the parent, and so on right up to the top-level window. This process is known as *bubbling*, and it enables a container control to implement default event-handling logic for any events that are not handled by its children.

When a control handles an event, the event is still bubbled to the parent in case the parent needs to perform any additional processing. An event handler is passed information about the event in the *RoutedEventArgs* parameter. An event handler can use the properties in this parameter to determine the source of the event (the control that had the focus when the event was raised). The *RoutedEventArgs* parameter also includes a Boolean property called *Handled*. An event handler can set this property to *true* to indicate that the event has been processed. When the event bubbles, the value of this property can be used to prevent the event from being handled by a parent control.

 **Additional Reading:** For more information about how routed events work in WPF, refer to the Routed Events Overview page at <http://go.microsoft.com/fwlink/?LinkId=267822>.

Using Layout Controls

Support for relative positioning is one of the core principles of WPF. The idea is that your application should render correctly regardless of how the user positions or resizes the application window. WPF includes several layout, or container, controls that enable you to position and size your child controls in different ways. The following table shows the most common layout controls:

- Canvas
- DockPanel
- Grid
- StackPanel
- VirtualizingStackPanel
- WrapPanel

Control	Description
Canvas	Child controls define their own layout by specifying canvas coordinates.
DockPanel	Child controls are attached to the edges of the DockPanel.
Grid	Child controls are added to rows and columns within the grid.
StackPanel	Child controls are stacked either vertically or horizontally.

Control	Description
VirtualizingStackPanel	Child controls are stacked either vertically or horizontally. At any one time, only child items that are visible on the screen are created.
WrapPanel	Child controls are added from left to right. If there are too many controls to fit on a single line, the controls wrap to the next line.

The following example shows how to define a grid with two rows and two columns:

Using a Grid Layout

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition MinHeight="100" MaxHeight="200" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="3*" />
        <ColumnDefinition Width="7*" />
    </Grid.ColumnDefinitions>
    <Label Content="This is Row 0, Column 0" Grid.Row="0" Grid.Column="0" />
    <Label Content="This is Row 0, Column 1" Grid.Row="0" Grid.Column="1" />
    <Label Content="This is Row 1, Column 0" Grid.Row="1" Grid.Column="0" />
    <Label Content="This is Row 1, Column 1" Grid.Row="1" Grid.Column="1" />
</Grid>
```

In the previous example, notice that you use **RowDefinition** and **ColumnDefinition** elements to define rows and columns respectively. For each row or column, you can specify a minimum and maximum height or width, or a fixed height or width. You can specify heights and widths in three ways:

- As numerical units. For example, **Width="200"** represents 200 units (where 1 unit equals 1/96th of an inch).
- As **Auto**. For example, **Width="Auto"** will set the column to the minimum width required to render all the child controls in the column.
- As a star value. For example, **Width="*"** will make the column use up any available space after fixed-width columns and auto-width columns are allocated. If you create two columns with widths of **3*** and **7***, the available space will be divided between the columns in the ratio 3:7.

To put a child control in a particular row or column, you add the **Grid.Row** and **Grid.Column** attributes to the child control element. Note that these properties are defined by the parent **Grid** element, rather than the child **Label** element. Properties such as **Grid.Row** and **Grid.Column** are known as attached properties—they are defined by a parent element to be specified by its child elements. Attached properties enable child elements to tell a parent element how they should be rendered.



Additional Reading: For more information about layout controls, refer:

- <https://aka.ms/moc-20483c-m9-pg1>.
- The DockPanel Class page at <https://aka.ms/moc-20483c-m9-pg2>.
- The Grid Class page at <https://aka.ms/moc-20483c-m9-pg3>.
- The StackPanel Class page at <https://aka.ms/moc-20483c-m9-pg4>.
- The VirtualizingStackPanel Class page at <https://aka.ms/moc-20483c-m9-pg5>.
- The WrapPanel Class page at <https://aka.ms/moc-20483c-m9-pg6>.

Demonstration: Using Design View to Create a XAML UI

In this demonstration, you will create a simple WPF application that contains a button and a label. When you click the button, an event handler will update the text on the label.

The demonstration illustrates various ways in which you can create and configure XAML files in Visual Studio. It illustrates how you can add controls to the design surface by double-clicking a control in the toolbox. It shows how you can configure controls through a combination of using designer tools and editing XAML markup directly. It also illustrates how Visual Studio can automatically connect event handlers to your controls.

Demonstration Steps

You will find the steps in the **Demonstration: Using Design View to Create a XAML UI** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_DEMO.md.

Creating User Controls

When you work with WPF, you can create your own self-contained, reusable user controls in XAML. User controls are sometimes called composite controls, because they are a composite of other controls. For example, if you use the same combination of a text box and a button multiple times in a UI, it might be more convenient to create and use a user control that consists of the text box and the button. Alternatively, you might create a user control to enable multiple developers to share the same custom control across multiple assemblies.

- To create a user control:
 - Define the control in XAML
 - Expose properties and events in the code-behind class
- To use a user control:
 - Add an XML namespace prefix for the assembly and namespace
 - Use the control like a standard XAML control

Like a regular XAML window, user controls consist of a XAML file and a corresponding code-behind file. In the XAML file, the principal difference is that the top-level element is a **UserControl** element rather than a **Window** element.

The following example shows a user control that enables people to select and order beverages:

Creating a User Control in XAML

```
<UserControl x:Class="DesignView.CoffeeSelector"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="200">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <StackPanel Grid.Row="0">
            <Label Content="Do you want coffee or tea?"/>
            <RadioButton x:Name="radCoffee" Content="Coffee" HorizontalAlignment="Left"
                VerticalAlignment="Top" Margin="5" GroupName="Beverage"
                IsChecked="True" Checked="radCoffee_Checked" />
```

```

<RadioButton x:Name="radTea" Content="Tea" HorizontalAlignment="Left"
    VerticalAlignment="Top" Margin="5" GroupName="Beverage"
    Checked="radTea_Checked"/>
</StackPanel>
<StackPanel Grid.Row="1">
    <Label Content="Do you want milk?"/>
    <RadioButton x:Name="radMilk" Content="Milk" HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="5" GroupName="Milk"
        IsChecked="True" Checked="radMilk_Checked" />
    <RadioButton x:Name="radNoMilk" Content="No Milk" HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="5" GroupName="Milk"
        Checked="radNoMilk_Checked"/>
</StackPanel>
<StackPanel Grid.Row="2">
    <Label Content="Do you want sugar?"/>
    <RadioButton x:Name="radSugar" Content="Sugar" HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="5" GroupName="Sugar"
        IsChecked="True" Checked="radSugar_Checked" />
    <RadioButton x:Name="radNoSugar" Content="No Sugar"
    HorizontalAlignment="Left"
        VerticalAlignment="Top" Margin="5" GroupName="Sugar"
        Checked="radNoSugar_Checked"/>
</StackPanel>
<Button x:Name="btnOrder" Content="Place Order" Margin="5" Grid.Row="3"
    Click="btnOrder_Click"/>
</Grid>
</UserControl>

```

As the previous example shows, creating the XAML for a user control is very similar to creating the XAML for a window. In the code-behind file for the user control, you can create event handler methods in the same way that you would for regular XAML windows. When you edit the code-behind file, you should also:

- Define any required public properties. Creating public properties enables the consumers of your user control to get or set property values, either in XAML or in code.
- Define any required public events. Raising events enables consumers of your user control to respond in the same way that they would respond to other control events, such as the **Click** event of a button.

The following example shows the code-behind class for a user control:

Creating the Code-Behind Class for a User Control

```

public partial class CoffeeSelector : UserControl
{
    public CoffeeSelector()
    {
        InitializeComponent();
    }

    private string beverage;
    private string milk;
    private string sugar;

    public string Order
    {
        get
        {
            return $"{beverage}, {milk}, {sugar}";
        }
    }

    public event EventHandler<EventArgs> OrderPlaced;

    private void btnOrder_Click(object sender, RoutedEventArgs e)
    {

```

```
        if(OrderPlaced!=null)
            OrderPlaced(this, EventArgs.Empty);
    }

    private void radCoffee_Checked(object sender, RoutedEventArgs e) { beverage =
"Coffee"; }
    private void radTea_Checked(object sender, RoutedEventArgs e) { beverage = "Tea"; }
    private void radMilk_Checked(object sender, RoutedEventArgs e) { milk = "Milk"; }
    private void radNoMilk_Checked(object sender, RoutedEventArgs e) { milk = "No Milk"; }
    private void radSugar_Checked(object sender, RoutedEventArgs e) { sugar = "Sugar"; }
    private void radNoSugar_Checked(object sender, RoutedEventArgs e) { sugar = "No
Sugar"; }
}
```



Note: In this example, you will notice a string usage—\$ strings, or string interpolation—that was not discussed before. Using \$ before a string allows you to use the string like a **String.Format** method, except that you don't specify the location of the values. Rather, you provide the values directly in the string itself. You can place anything in a string interpolation. Besides regular variables, you can use the results of calculations or even method return. For more, refer: \$ - string interpolation (C# Reference) <https://aka.ms/moc-20483c-m9-pg9>.

In the previous code example, the user control raises an **OrderPlaced** event when the user clicks the **Place Order** button. Applications that include the user control can subscribe to this event and take appropriate action.

To use your user control in a WPF application, you need to do two things:

1. Add a namespace prefix for your user control namespace and assembly to the **Window** element. This should take the following form:

xmlns:[your prefix] = "clr-namespace:[your namespace].[your assembly name]"



Note: If your application and your user control are in the same assembly, you can omit the assembly name from the namespace prefix declaration.

2. Add the control to your application in the same way that you would add a built-in control, with the namespace prefix you defined.

The following example shows how to add a user control in XAML:

Adding a User Control to a WPF Application

```
<Window x:Class="DesignView.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:coffee="clr-namespace:DesignView"
    Title="Order Your Coffee Here" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="8*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <coffee:CoffeeSelector x:Name="coffeeSelector1" Grid.Row="0"
            OrderPlaced="coffeeSelector1_OrderPlaced" />
        <Label x:Name="lblResult" Margin="5" Grid.Row="1" />
    </Grid>
</Window>
```

When you have added the user control, you can handle events and get or set property values in the same way that you would for a built-in control. In the previous example, the **OrderPlaced** event of the **CoffeeSelector** control is wired up to the **coffeeSelector1_OrderPlaced** method.

The following example shows how to interact with a user control in a code-behind class:

Programming with User Controls

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void coffeeSelector1_OrderPlaced(object sender, EventArgs e)
    {
        lblResult.Content = coffeeSelector1.Order;
    }
}
```

In the previous code example, the **coffeeSelector1_OrderPlaced** method handles the **OrderPlaced** event of the **CoffeeSelector** control. The method then retrieves the order details from the control and writes them to a label.

Lesson 2

Binding Controls to Data

Most applications work with data in one form or another. The data that drives your application can come from a wide variety of sources, such as files, databases, or web services. Almost every graphical application needs to connect UI controls to an underlying data source so that users can retrieve, enter, or edit information.

In this lesson, you will learn how to bind controls to data in WPF applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe how data binding works in WPF.
- Use XAML to bind controls to data.
- Use code to bind controls to data.
- Bind controls to collections of data.
- Create data templates to specify how data is rendered.

Introduction to Data Binding

Data binding is the act of connecting a data source to a UI element in such a way that if one changes, the other must also change.

Conceptually, data binding consists of three components:

- The *binding source*. This is the source of your data, and is typically a property of a custom .NET object. For example, you might bind a control to the **CountryOfOrigin** property of a **Coffee** object.
- The *binding target*. This is the XAML element you want to bind to your data source, and is typically a UI control. You must bind your data source to a property of your target object, and that property must be a *dependency property*. For example, you might bind data to the **Content** property of a **TextBox** element.
- The *binding object*. This is the object that connects the source to the target. The **Binding** object can also specify a converter, if the source property and the target property are of different data types.

• Data binding has three components:

- Binding source
- Binding target
- Binding object

• A data binding can be bidirectional or unidirectional:

- TwoWay
- OneWay
- OneTime
- OneWayToSource
- Default



Note: A dependency property is a special type of wrapper around a regular property. Dependency properties are registered with the .NET Framework runtime, which enables the runtime to notify any interested parties when the value of the underlying property changes. This ability to notify changes is what makes data binding work. Most built-in UI elements implement dependency properties and will support data binding. For more information about dependency properties, refer to the Dependency Properties Overview page at <http://go.microsoft.com/fwlink/?LinkId=267829>.

The following example shows a simple data binding expression:

Simple Data Binding

```
<TextBlock Text="{Binding Source={StaticResource coffee1}, Path=Bean}" />
```

In this example, the **Text** property of a **TextBlock** is set to a data binding expression. Note that:

- The **Binding** expression is enclosed in braces. This enables you to set properties on the **Binding** object before it is evaluated by the **TextBlock**.
- The **Source** property of the **Binding** object is set to **{StaticResource coffee1}**. This is an object instance defined elsewhere in the solution.
- The **Path** property of the **Binding** object is set to **Bean**. This indicates that you want to bind to the **Bean** property of the **coffee1** object.

As a result of this expression, the **TextBlock** will always display the value of the **Bean** property of the **coffee1** object. If the value of the **Bean** property changes, the **TextBlock** will update automatically. In terms of the concepts described at the start of this topic:

- The *binding source* is the **Bean** property of the **coffee1** object.
- The *binding target* is the **Text** property of the **TextBlock** element.
- The *binding object* is defined by the expression in braces.

You can also configure the direction of the data binding. For example, you might want to update the UI when the source data is changed or you might want to update the source data when the user edits a value in the UI. To specify the direction of the binding, you set the **Mode** property of the **Binding** object.

The **Mode** property can take the following values:

Mode Value	Details
TwoWay	Updates the target property when the source property changes, and updates the source property when the target property changes.
OneWay	Updates the target property when the source property changes.
OneTime	Updates the target property only when the application starts or when the DataContext property of the target is changed.
OneWayToSource	Updates the source property when the target property changes.
Default	Uses the default Mode value of the target property.

The following example shows how to configure a text box to use two-way data binding:

Specifying the Binding Direction

```
<TextBox Text="{Binding Source={StaticResource coffee1}, Path=Bean, Mode=TwoWay}" />
```

 **Additional Reading:** For more information about the concepts of data binding, refer to the Data Binding Overview page at <http://go.microsoft.com/fwlink/?LinkId=267830>.

Binding Controls to Data in XAML

You can bind controls to data in various ways. If your source data will not change during the execution of your application, you can create a *static resource* to represent your data in XAML. A static resource enables you to create instances of classes. For example, if your assembly contains a class named **Coffee**, you can define a static resource to create an instance of **Coffee** and set properties on it. To create a static resource, you must:

- Add an element to the **Resources** property of a container control, such as the top-level **Window**.
- Set the name of the element to the name of the class you want to instantiate. For example, if you want to create an instance of the **Coffee** class, create an element named **Coffee**. Use a namespace prefix declaration to identify the namespace and assembly that contains your class.
- Add an **x:Key** attribute to the element. This is the identifier by which you will specify the static resource in data binding expressions. You can create multiple instances of a resource in a window, but each instance should have a unique **x:Key** value.

The following example shows how to create a static resource for data binding:

Creating a Static Resource

```
<Window x:Class="DataBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:DataBinding"
        Title="Data Binding Example" Height="350" Width="525">
    <Window.Resources>
        <loc:Coffee x:Key="coffee1"
                    Name="Fourth Coffee Quencher"
                    Bean="Arabica"
                    CountryOfOrigin="Brazil"
                    Strength="3" />
        ...
    </Window.Resources>
    ...
</Window>
```

If you want to bind an individual UI element to this static resource, you can use a binding statement that specifies both a source and a path. You set the **Source** property to the static resource, and set the **Path** property to the specific property in the source object to which you want to bind.

The following example shows how to bind an individual item to a static resource:

Binding an Individual Item to a Static Resource

```
<TextBlock Text="{Binding Source={StaticResource coffee1}, Path=Name}" />
```

More commonly, you will want to bind multiple UI elements to different properties of a data source. In this case, you can set the **DataContext** property on a container object, such as a **Grid** or a **StackPanel**. Setting a **DataContext** property is similar to providing a partial data binding expression. It specifies the source object, but does not identify specific properties. Any child controls within the container object will

inherit this data context, unless you override it. This means that when you create data binding expressions for child controls, you can omit the source and simply specify the path to the property of interest.

The following example shows how to set a data context for a set of child controls:

Specifying a Data Context

```
<StackPanel>
    <StackPanel.DataContext>
        <Binding Source="{StaticResource coffee1}" />
    </StackPanel.DataContext>
    <TextBlock Text="{Binding Path=Name}" />
    <TextBlock Text="{Binding Path=Bean}" />
    <TextBlock Text="{Binding Path=CountryOfOrigin}" />
    <TextBlock Text="{Binding Path=Strength}" />
</StackPanel>
```

While you could specify a full data binding expression for each individual UI element, specifying a **DataContext** property typically makes your XAML easier to write, easier to read, and easier to maintain.

Binding Controls to Data in Code

In real-world applications, it is unlikely that your source data will be static. It is far more likely that you will retrieve data at runtime from a database or a web service. In these scenarios, you cannot use a static resource to represent your data. Instead, you must use code to specify the data source for any UI bindings at runtime.

In many cases you can use a mixture of XAML binding and code binding. For example, you might know that your UI elements will be bound to a Coffee instance at design time. In this case, you can specify the binding paths in XAML, and then set a **DataContext** property in code.

The following example shows how to specify binding paths in XAML:

Specifying Binding Paths in XAML

```
<StackPanel x:Name="stackCoffee">
    <TextBlock Text="{Binding Path=Name}" />
    <TextBlock Text="{Binding Path=Bean}" />
    <TextBlock Text="{Binding Path=CountryOfOrigin}" />
    <TextBlock Text="{Binding Path=Strength}" />
</StackPanel>
```

- Create data binding entirely in code
- Create **Path** bindings in XAML and set the **DataContext** in code

```
<StackPanel x:Name="stackCoffee">
    <TextBlock Text="{Binding Path=Name}" />
    <TextBlock Text="{Binding Path=Bean}" />
    <TextBlock Text="{Binding Path=CountryOfOrigin}" />
    <TextBlock Text="{Binding Path=Strength}" />
</StackPanel>
```

```
stackCoffee.DataContext = coffee1;
```

In this example, you have set the binding path for each individual text block in XAML. To complete the binding, all you need to do is to set the **DataContext** property of the parent **StackPanel** object to the **Coffee** instance that you want to display.

Binding Collections to Control

In many scenarios, you will want to data bind a control to a collection of objects. WPF includes controls that are designed to render collections, such as the **ListBox** control, the **ListView** control, the **ComboBox** control, and the **TreeView** control. These controls all inherit from the **ItemsControl** class and, as such, support a common approach to data binding.

To bind a collection to an **ItemsControl** instance, you need to:

- Specify the source data collection in the **ItemsSource** property of the **ItemsControl** instance.
- Specify the source property you want to display in the **DisplayMemberPath** property of the **ItemsControl** instance.

- Set the **ItemsSource** property to bind to an **IEnumerable** collection

```
IstCoffees.ItemsSource = coffees;
```

- Use the **DisplayMemberPath** property to specify the source field to display

```
<ListBox x:Name="IstCoffees"  
        DisplayMemberPath="Name" />
```

You can bind an **ItemsControl** instance to any collection that implements the **IEnumerable** interface. You can set the **ItemsSource** and **DisplayMemberPath** properties in XAML or in code. One common approach is to define the **DisplayMemberPath** property (or a data template) in XAML, and then to set the **ItemsSource** programmatically at runtime.

The following code example shows how to set the **DisplayMemberPath** property in XAML:

Setting the **DisplayMemberPath** Property

```
<ListBox x:Name="IstCoffees" DisplayMemberPath="Name" />
```

Having set the **DisplayMemberPath** property in XAML, you can now set the **ItemsSource** property in code to complete the data binding.

The following example shows how to set the **ItemsSource** property in code:

Setting the **ItemsSource** Property

```
// Create some Coffee instances.  
var coffee1 = new Coffee("Fourth Coffee Quencher");  
var coffee2 = new Coffee("Espresso Number Four");  
var coffee3 = new Coffee("Fourth Refresher");  
var coffee3 = new Coffee("Fourth Frenetic");  
  
// Add the items to an observable collection.  
var coffees = new ObservableCollection<Coffee>();  
coffees.Add(coffee1);  
coffees.Add(coffee2);  
coffees.Add(coffee3);  
coffees.Add(coffee4);  
  
// Set the ItemsSource property of the ListBox to the coffees collection.  
IstCoffees.ItemsSource = coffees;
```



Note: If you want a control displaying data in a collection to be updated automatically when items are added or removed, the collection must implement the **INotifyPropertyChanged** interface. This interface defines an event called **PropertyChanged** that the collection can raise

after making a change. The .NET Framework includes a class named **ObservableCollection<T>** that provides a generic implementation of **INotifyPropertyChanged**.

The control that binds to the collection receives the event, and can use it to refresh the data that it is displaying. Many of the WPF container controls catch and handle this event automatically.

 **Additional Reading:** For more information about the **ObservableCollection<T>** class, refer to the `ObservableCollection(T)` class at <https://aka.ms/moc-20483c-m9-pg7>. For more information about the **INotifyPropertyChanged** interface, refer to the `INotifyPropertyChanged` Interface page at <https://aka.ms/moc-20483c-m9-pg8>.

Creating Data Templates

When you use controls that derive from the **ItemsControl** or **ContentControl** control, you can create a *data template* to specify how your items are rendered. For example, suppose you want to use a **ListBox** control to display a collection of **Coffee** instances. Each **Coffee** instance includes several properties to represent the name of the coffee, the type of coffee bean, the country of origin, and the strength of the coffee. The data template specifies how each **Coffee** instance should be rendered, by mapping properties of the **Coffee** instance to child controls within the data template. Creating a data template gives you precise control over how your items are rendered and styled.

The following example shows how to define a data template for a **ListBox** control:

Creating a Data Template

```
<ListBox x:Name="lstCoffees" Width="200">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="2*" />
                    <RowDefinition Height="*" />
                    <RowDefinition Height="*" />
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
                <TextBlock Text="{Binding Path=Name}" Grid.Row="0"
                           FontSize="22" Background="Black" Foreground="White" />
                <TextBlock Text="{Binding Path=Bean}" Grid.Row="1" />
                <TextBlock Text="{Binding Path=CountryOfOrigin}" Grid.Row="2" />
                <TextBlock Text="{Binding Path=Strength}" Grid.Row="3" />
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

Specify how each item in a collection should be displayed

```
<DataTemplate>
    <Grid>
        ...
        <TextBlock Text="{Binding Path=Name}" Grid.Row="0"
                   FontSize="22" Background="Black"
                   Foreground="White" />
        <TextBlock Text="{Binding Path=Bean}" Grid.Row="1" />
        <TextBlock Text="{Binding Path=CountryOfOrigin}" Grid.Row="2" />
        <TextBlock Text="{Binding Path=Strength}" Grid.Row="3" />
    </Grid>
</DataTemplate>
```

When you set the **ItemsSource** property of this **ListBox** to a collection of **Coffee** objects, the data template specifies how the **ListBox** should render each **Coffee** object. In this example, the data template uses a simple grid layout and displays the name of the coffee with a larger font and a contrasting background. However, you can make your data templates as complex and sophisticated as you want.



Additional Reading: For more information about data templates in WPF, refer to the Data Templating Overview page at <http://go.microsoft.com/fwlink/?LinkId=267833>.

Lesson 3

Styling a UI

If you had to style a graphical application by setting properties on one control at a time, the development process would quickly become laborious. It would also be difficult to maintain your application, as you would often need to make the same change in multiple locations. XAML enables you to define styles as reusable resources that you can apply to multiple controls.

In this lesson, you will learn how to use styles and animations.

Lesson Objectives

After completing this lesson, you will be able to:

- Create reusable resources in XAML.
- Define styles that apply to multiple controls.
- Use property triggers to apply styles when conditions are met.
- Use animations to create dynamic effects and transformations.

Creating Reusable Resources in XAML

XAML enables you to create certain elements, such as data templates, styles, and brushes, as reusable resources. This has various advantages when you are developing a graphical application:

- You can define a resource once and use it in multiple places.
- You can edit a resource without editing every element that uses the resource.
- Your XAML files are shorter and easier to read.

- Define resources in a **Resources** collection
- Add an **x:Key** to uniquely identify the resource

```
<Window.Resources>
<SolidColorBrush x:Key="MyBrush" Color="Coral" />
...
</Window.Resources>
```
- Reference the resource in property values

```
<TextBlock Text="Foreground"
Foreground="{StaticResource MyBrush}" />
```
- Use a resource dictionary to manage large collections of resources

Every WPF control has a **Resources** property to which you can add resources. This is because the **Resources** property is defined by the **FrameworkElement** class from which all WPF elements ultimately derive. In most cases, you define resources on the root element in a file, such as the **Window** element or the **UserControl** element. The resources are then available to the root element and all of its descendants. You can also create resources for use across the entire application by defining them in the App.xaml file.

 **Note:** Every WPF application has an App.xaml file. It is a XAML file that can contain global resources used by all windows and controls in a WPF application. It is also the entry point for the application, and defines which window should appear when an application starts.

Resources are stored in a dictionary collection of type **ResourceDictionary**. When you create a reusable resource, you must give it a unique key by providing a value for the **x:Key** attribute.

MCT USE ONLY. STUDENT USE PROHIBITED

The following example shows how to create a brush as a window-level resource:

Creating and Using Resources

```
<Window x:Class="DataBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Reusable Resources" Height="350" Width="525">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Coral" />
        ...
    </Window.Resources>
    ...
</Window>
```

To reference a resource, you use the format **{StaticResource [resource key]}**. You can use the resource in any property that accepts values of the same type as the resource, provided that the resource is in scope. For example, if you create a brush as a resource, you can reference the brush in any property that accepts brush types, such as **Foreground**, **Background**, or **Fill**.

The following example shows how to reference a reusable resource in multiple places:

Referencing a Reusable Resource

```
<StackPanel>
    <Button Content="Click Me" Background="{StaticResource MyBrush}" />
    <TextBlock Text="Foreground" Foreground="{StaticResource MyBrush}" />
    <TextBlock Text="Background" Background="{StaticResource MyBrush}" />
    <Ellipse Height="50" Fill="{StaticResource MyBrush}" />
</StackPanel>
```

If you need to create several reusable resources, it can be useful to create your resources in a separate resource dictionary file. A resource dictionary is a XAML file with a top-level element of **ResourceDictionary**. You can add reusable resources within the **ResourceDictionary** element in the same way that you would add them to a **Window.Resources** element.

 **Note:** You can create a WPF Resource Dictionary from the **Add New Item** menu in Visual Studio.

In most cases, you make a resource dictionary available for use in your application by referencing it in the application-scoped App.xaml file. The following example shows how to reference a resource dictionary in the App.xaml file:

Referencing a Resource Dictionary

```
<Application x:Class="ReusableResources.App"
            xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="FourthCoffeeResources.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Defining Styles as Resources

In many cases, you will want to apply the same property values to multiple controls of the same type within an application. For example, if a page contains five textboxes, you will probably want each textbox to have the same foreground color, background color, font size, and so on. To make this consistency easier to manage, you can create **Style** elements as resources in XAML. **Style** elements enable you to apply a collection of property values to some or all controls of a particular type. To create a style, perform the following steps:

1. Add a **Style** element to a resource collection within your application (for example, the **Window.Resources** collection or a resource dictionary).
2. Use the **TargetType** attribute of the **Style** element to specify the type of control you want the style to target (for example, **TextBox** or **Button**).
3. Use the **x:Key** attribute of the **Style** element to enable controls to specify this style. Alternatively, you can omit the **x:Key** attribute and your style will apply to all controls of the specified type.
4. Within the **Style** element, use **Setter** elements to apply specific values to specific properties.

The following example shows how to create a style that targets **TextBlock** controls:

Creating Styles

```
<Window.Resources>
    <Style TargetType="TextBlock" x:Key="BlockStyle1">
        <Setter Property="FontSize" Value="20" />
        <Setter Property="Background" Value="Black" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="Orange" />
                        <GradientStop Offset="1.0" Color="Red" />
                    </LinearGradientBrush.GradientStops>
                </Setter.Value>
            </Setter>
        </Style>
    ...
</Window.Resources>
```

To apply this style to a **TextBlock** control, you need to set the **Style** attribute of the **TextBlock** to the **x:Key** value of the style resource.

The following example shows how to apply a style to a control:

Applying a Style

```
<TextBlock Text="Drink More Coffee" Style="{StaticResource BlockStyle1}" />
```

 **Additional Reading:** For more information about defining styles, refer to the Styling and Templating page at <http://go.microsoft.com/fwlink/?LinkId=267834>.

Using Property Triggers

When you create a style in XAML, you can specify property values that are only applied when certain conditions are true. For example, you might want to change the font style of a button when the user hovers over it, or you might want to apply a highlighting effect to selected items in a list box.

To apply style properties based on conditions, you add **Trigger** elements to your styles. The **Trigger** element identifies the property of interest and the value that should trigger the change. Within the **Trigger** element, you use **Setter** elements to apply changes to property values.

The following example shows how to make the text on a button bold while the user is hovering over the button:

Using a Property Trigger

```
<Window.Resources>
    <Style TargetType="Button">
        <Style.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter Property="FontWeight" Value="Bold" />
            </Trigger>
        </Style.Triggers>
    </Style>
    ...
</Window.Resources>
```

Use triggers to apply style properties based on conditions:

- Use the **Trigger** element to identify the condition
- Use **Setter** elements apply the conditional changes

```
<Style TargetType="Button">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="FontWeight" Value="Bold" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Creating Dynamic Transformations

Sophisticated graphical applications often use animations to make the user experience more engaging. Animations are sometimes used to make transitions between states less abrupt. For example, if you want to enlarge or rotate a picture, increasing the size or changing the orientation progressively over a short time period can look better than simply switching from one size or orientation to another.

To create and apply an animation effect in XAML, you need to do three things:

1. *Create an animation.*

WPF includes various classes that you can use to create animations in XAML. The most commonly used animation element is **DoubleAnimation**. The **DoubleAnimation** element specifies how a value should change over time, by specifying the initial value, the final value, and the duration over which the value should change.

• Use an **EventTrigger** to identify the event that starts the animation

- Use a **Storyboard** to identify the properties that should change

- Use a **DoubleAnimation** to define the changes

```
<EventTrigger RoutedEvent="Image.MouseDown">
    <BeginStoryboard>
        <Storyboard>
            <DoubleAnimation
                Storyboard.TargetProperty="Height"
                From="200" To="300" Duration="0:0:2" />
        </Storyboard>
    </BeginStoryboard>
</EventTrigger>
```

2. *Create a storyboard.*

To apply an animation to an object, you need to wrap your animation in a **Storyboard** element. The **Storyboard** element enables you to specify the object and the property you want to animate. It does this by providing the **TargetName** and **TargetProperty** attached properties, which you can set on child animation elements.

3. *Create a trigger.*

To trigger your animation in response to a property change, you need to wrap your storyboard in an **EventTrigger** element. The **EventTrigger** element specifies the control event that will trigger the animation. In the **EventTrigger** element, you can use a **BeginStoryboard** element to launch the animation.

You can add an **EventTrigger** element containing your storyboards and animations to the **Triggers** collection of a **Style** element, or you can add it directly to the **Triggers** collection of an individual control.

The following example shows how to use an animation to rotate and expand an image when the user clicks on it:

Creating an Animation Effect

```
<Window.Resources>
    <Style TargetType="Image" x:Key="CoffeeImageStyle">
        <Setter Property="Height" Value="200" />
        <Setter Property="RenderTransformOrigin" Value="0.5,0.5" />
        <Setter Property="RenderTransform">
            <Setter.Value>
                <RotateTransform Angle="0" />
            </Setter.Value>
        </Setter>
        <Style.Triggers>
            <EventTrigger RoutedEvent="Image.MouseDown">
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Storyboard.TargetProperty="Height"
                            From="200" To="300" Duration="0:0:2" />
                        <DoubleAnimation
                            Storyboard.TargetProperty="RenderTransform.Angle"
                            From="0" To="30" Duration="0:0:2" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
```



Additional Reading: For more information about animations, refer to the Animation Overview page at <http://go.microsoft.com/fwlink/?LinkId=267835>.

Demonstration: Customizing Student Photographs and Styling the Application Lab

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: Customizing Student Photographs and Styling the Application Lab** section on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_DEMO.md.

Lab: Customizing Student Photographs and Styling the Application

Scenario

Now that you and The School of Fine Arts are happy with the basic functionality of the application, you need to improve the appearance of the interface to give the user a nicer experience through the use of animations and a consistent look and feel.

You decide to create a **StudentPhoto** control that will enable you to display photographs of students in the student list and other views. You also decide to create a fluid method for a teacher to remove a student from their class. Finally, you want to update the look of the various views, keeping their look consistent across the application.

Objectives

After completing this lab, you will be able to:

- Create and use user controls.
- Use styles and animations.

Lab Setup

Estimated Time: **90 minutes**

You will find the high-level steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_LAB_MANUAL.md.

You will find the detailed steps on the following page: https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_LAK.md.

Exercise 1: Customizing the Appearance of Student Photographs

Scenario

In this exercise, you will customize the appearance of student photographs in the production application.

You will begin by creating a **StudentPhoto** user control that will host the photographs on the various pages in the UI. Then you will lay out the user controls and write code to raise the **Student_Click** method when a user clicks a student photograph.

Next, you will add a remove button with a red X to the user control that users can click to remove a student from a class. When a user hovers over the button, the opacity of the button and the photograph will change.

Finally, you will run the application to verify that the student's image is displayed correctly on the **StudentsPage** view.

Exercise 2: Styling the Logon View

Scenario

In this exercise, you will update the **LogonPage** control to have the same look and feel as the rest of the application.

First, you will define styles for the username and password text boxes on the **LogonPage** of the application. You will use the **Style** property of each control to apply the styles that you have defined. Then you will define some global styles for use across the entire application. You will define a style for labels and a style for text. Finally, you will run the application to verify that the styling of the text elements has changed throughout the application.

Exercise 3: Animating the StudentPhoto Control (If Time Permits)

Scenario

In this exercise, you will update the **StudentPhoto** control to animate when a user hovers over it.

First you will define an animation for the **StudentPhoto** control, which will cause a student's photograph to pulse when a user hovers over it. You will then add event handlers for this animation and apply the animation to the control. Finally, you will run the application to verify that the animation executes correctly.

Module Review and Takeaways

In this module, you learned how to create engaging UIs for graphical applications. You learned how to use XAML to create windows and user controls and how you bind controls to data items and collections. You also learned how to provide a consistent user experience by creating styles as reusable resources.

Review Questions

Check Your Knowledge

Question
You want to use rows and columns to lay out a UI. Which container control should you use?
Select the correct answer.
The Canvas control.
The DockPanel control.
The Grid control.
The StackPanel control.
The WrapPanel control.

Check Your Knowledge

Question
You are creating an application that enables users to place orders for coffees. The application should allow users to select the drink they want from a list. Each list item should display the name of the coffee, the description, the price, and an image of the coffee. How should you proceed?
Select the correct answer.
Create a ListBox control. Add child controls to the ListBox control to represent each field.
Create a ListBox control. Use a DataTemplate to specify how each field is displayed within a list item.
Create a ListBox control. Create a custom control that inherits from ListBoxItem, and use this custom control to specify how each field is displayed.
Create a ListBox control. Use the DisplayMemberPath property to specify the fields you want to display in each list item.
Create a ListBox control. Use a Style to specify how each field is displayed within a list item.

MCT USE ONLY STUDENT USE PROHIBITED

Check Your Knowledge

Question
You want to apply a highlighting effect to selected items in a ListBox. How should you proceed?
Select the correct answer.
Create a Style element and set the TargetType attribute to ListBox. Use a Setter element to apply the highlighting effect.
Create a Style element and set the TargetType attribute to ListBox. Use a Trigger element to apply the highlighting effect when a list box item is selected.
Create a Style element and set the TargetType attribute to ListBox. Use an EventTrigger element to apply the highlighting effect when a list box item is selected.
Create a Style element and set the TargetType attribute to ListBox. Use a Storyboard element to apply the highlighting effect when a list box item is selected.
Create a Style element and set the TargetType attribute to ListBox. Use a DoubleAnimation element to apply the highlighting effect when a list box item is selected.

Module 10

Improving Application Performance and Responsiveness

Contents:

Module Overview	10-1
Lesson 1: Implementing Multitasking	10-2
Lesson 2: Performing Operations Asynchronously	10-14
Lesson 3: Synchronizing Concurrent Access to Data	10-24
Lab: Improving the Responsiveness and Performance of the Application	10-30
Module Review and Takeaways	10-31

Module Overview

Modern processors use threads to concurrently run multiple operations. If your application performs all of its logic on a single thread, you do not make the best use of the available processing resources, which can result in a poor experience for your users. In this module, you will learn how to improve the performance of your applications by distributing your operations across multiple threads.

Objectives

After completing this module, you will be able to:

- Use the Task Parallel Library to implement multitasking.
- Perform long-running operations without blocking threads.
- Control how multiple threads can access resources concurrently.

Lesson 1

Implementing Multitasking

A typical graphical application consists of blocks of code that run when an event occurs; these events fire in response to actions such as the user clicking a button, moving the mouse, or opening a window. By default, this code runs by using the UI thread. However, you should avoid executing long-running operations on this thread because they can cause the UI to become unresponsive. Also, running all of your code on a single thread does not make good use of available processing power in the computer; most modern machines contain multiple processor cores, and running all operations on a single thread will only use a single processor core.

The Microsoft® .NET Framework now includes the Task Parallel Library. This is a set of classes that makes it easy to distribute your code execution across multiple threads. You can run these threads on different processor cores and take advantage of the parallelism that this model provides. You can assign long-running tasks to a separate thread, leaving the UI thread free to respond to user actions.

In this lesson, you will learn how to use the Task Parallel Library to create multithreaded and responsive applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Create tasks.
- Control how tasks are executed.
- Return values from tasks.
- Cancel long-running tasks.
- Run multiple tasks in parallel.
- Link tasks together.
- Handle exceptions that tasks throw.

Creating Tasks

The **Task** class lies at the heart of the Task Parallel Library in the .NET Framework. As the name suggests, you use the **Task** class to represent a task, or in other words, a unit of work. The **Task** class enables you to perform multiple tasks concurrently, each on a different thread. Behind the scenes, the Task Parallel Library manages the thread pool and assigns tasks to threads. You can implement sophisticated multitasking functionality by using the Task Parallel Library to chain tasks, pause tasks, wait for tasks to complete before continuing, and perform many other operations.

- Use an **Action** delegate
`Task task1 = new Task(new Action(MyMethod));`
- Use an anonymous delegate/anonymous method
`Task task2 = new Task(delegate
{
 Console.WriteLine("Task 2 reporting");
});`
- Use lambda expressions (recommended)
`Task task2 = new Task(() =>
{
 Console.WriteLine(" Task 2 reporting");
});`

Creating a Task

You create a new **Task** object by using the **Task** class. A **Task** object runs a block of code, and you specify this code as a parameter to the constructor. You can provide this code in a method and create an **Action** delegate that wraps this method.

 **Note:** The **.NET Framework class library** contains the built-in **Action** delegate that lets you quickly define a delegate that has no return value and up to 16 parameters that use generic parameter types. It also provides the **Func** delegate that returns a result. You can use these delegates instead of defining new delegates for specific cases.

The following code example shows how to create a task by using an **Action** delegate:

Creating a Task by Using an Action Delegate

```
Task task1 = new Task(new Action(GetTheTime));  
  
private static void GetTheTime()  
{  
    Console.WriteLine("The time now is {0}", DateTime.Now);  
}
```

Using an **Action** delegate requires that you have defined a method that contains the code that you want to run in a task. However, if the sole purpose of this method is to provide the logic for a task and it is not reused anywhere else, you can find yourself creating (and having to remember the names of) a substantial number of methods. This makes maintenance more difficult. A more common approach is to use an anonymous method. An anonymous method is a method without a name, and you provide the code for an anonymous method inline, at the point you need to use it. You can use the **delegate** keyword to convert an anonymous method into a delegate.

The following code example shows how to create a task by using an anonymous delegate.

Creating a Task by Using an Anonymous Delegate

```
Task task2 = new Task(delegate { Console.WriteLine("The time now is {0}", DateTime.Now);});
```

Using Lambda Expressions to Create Tasks

A lambda expression is a shorthand syntax that provides a simple and concise way to define anonymous delegates. When you create a **Task** instance, you can use a lambda expression to define the delegate that you want to associate with your task.

If you want your delegate to invoke a named method or a single line of code, you can use a lambda expression. A lambda expression provides a shorthand notation for defining a delegate that can take parameters and return a result. It has the following form:

(input parameters) => expression

In this case:

- The lambda operator, `=>`, is read as “goes to.”
- The left side of the lambda operator includes any variables that you want to pass to the expression. If you do not require any inputs—for example, if you are invoking a method that takes no parameters—you include empty parentheses () on the left side of the lambda operator.

The right side of the lambda operator includes the expression you want to evaluate. This could be a comparison of the input parameters—for example, the expression `(x, y) => x == y` will return **true** if **x** is

equal to **y**; otherwise, it will return **false**. Alternatively, you can call a method on the right side of the lambda operator.

The following code example shows how to use lambda expressions to represent a delegate that invokes a named method.

Using a Lambda Expression to Invoke a Named Method

```
Task task1 = new Task( () => MyMethod() );
// This is equivalent to: Task task1 = new Task( delegate(MyMethod) );
```

A lambda expression can be a simple expression or function call, as the previous example shows, or it can reference a more substantial block of code. To do this, specify the code in curly braces (like the body of a method) on the right side of the lambda operator:

(input parameters) => { Visual C# statements; }

The following code example shows how to use lambda expressions to represent a delegate that invokes an anonymous method.

Using a Lambda Expression to Invoke an Anonymous Method

```
Task task2 = new Task( () => { Console.WriteLine("Test") } );
// This is equivalent to: Task task2 = new Task( delegate { Console.WriteLine("Test") } );
```

As your delegates become more complex, lambda expressions offer a far more concise and easily understood way to express anonymous delegates and anonymous methods. As such, lambda expressions are the recommended approach when you work with tasks.



Reference Links: For more information about lambda expressions, refer Lambda Expressions at <https://aka.ms/moc-20483c-m10-pg1>.

Controlling Task Execution

The Task Parallel Library offers several different approaches that you can use to start tasks. There are also various different ways in which you can pause the execution of your code until one or more tasks have completed.

Starting Tasks

When your code starts a task, the Task Parallel Library assigns a thread to your task and starts running that task. The task runs on a separate thread, so your code does not need to wait for the task to complete. Instead, the task and the code that invoked the task continue to run in parallel.

If you want to queue the task immediately, you use the **Start** method.

- To start a task:
 - **Task.Start** instance method
 - **Task.Factory.StartNew** static method
 - **Task.Run** static method

- To wait for tasks to complete:
 - **Task.Wait** instance method
 - **Task.WaitAll** static method
 - **Task.WaitAny** static method

Using the Task.Start Method to Queue a Task

```
var task1 = new Task( () => Console.WriteLine("Task 1 has completed." ) );
task1.Start();
```

Alternatively, you can use the static **TaskFactory** class to create and queue a task with a single line of code. The **TaskFactory** class is exposed through the static **Factory** property of the **Task** class.

Using the TaskFactory.StartNew Method to Queue a Task

```
var task3 = Task.Factory.StartNew( () => Console.WriteLine("Task 3 has completed."));
```

The **Task.Factory.StartNew** method is highly configurable and accepts a wide range of parameters. If you simply want to queue some code with the default scheduling options, you can use the static **Task.Run** method as a shortcut for the **Task.Factory.StartNew** method.

Using the Task.Run Method to Queue a Task

```
var task4 = Task.Run( () => Console.WriteLine("Task 4 has completed. "));
```

Waiting for Tasks

In some cases, you may need to pause the execution of your code until a particular task has completed. Typically you do this if your code depends on the result from one or more tasks, or if you need to handle exceptions that a task may throw. The **Task** class offers various mechanisms to do this:

- If you want to wait for a single task to finish executing, use the **Task.Wait** method.
- If you want to wait for multiple tasks to finish executing, use the static **Task.WaitAll** method.
- If you want to wait for any one of a collection of tasks to finish executing, use the static **Task.WaitAny** method.

The following code example shows how to wait for a single task to complete.

Waiting for a Single Task to Complete

```
var task1 = Task.Run( () => LongRunningMethod() );
// Do some other work.
// Wait for task 1 to complete.
task1.Wait();
// Continue with execution.
```

If you want to wait for multiple tasks to finish executing, or for one of a collection of tasks to finish executing, you must add your tasks to an array. You can then pass the array of tasks to the static **Task.WaitAll** or **Task.WaitAny** methods.

The following code example shows how to wait for multiple tasks to complete.

Waiting for Multiple Tasks to Complete

```
Task[] tasks = new Task[3]
{
    Task.Run( () => LongRunningMethodA()),
    Task.Run( () => LongRunningMethodB()),
    Task.Run( () => LongRunningMethodC())
};

// Wait for any of the tasks to complete.
Task.WaitAny(tasks);

// Alternatively, wait for all of the tasks to complete.
Task.WaitAll(tasks);

// Continue with execution.
```

Returning a Value from a Task

For tasks to be effective in real-world scenarios, you need to be able to create tasks that can return values, or *results*, to the code that initiated the task. The regular **Task** class does not enable you to do this. However, the Task Parallel Library also includes the generic **Task<TResult>** class that you can use when you need to return a value.

When you create an instance of **Task<TResult>**, you use the type parameter to specify the type of the result that the task will return. The **Task<TResult>** class exposes a read-only property named **Result**. After the task has finished executing, you can use the **Result** property to retrieve the return value of the task. The **Result** property is the same type as the task's type parameter.

The following example shows how to use the **Task<TResult>** class.

Retrieving a Value from a Task

```
// Create and queue a task that returns the day of the week as a string.  
Task<string> task1 = Task.Run<string>(() => DateTime.Now.DayOfWeek.ToString());  
// Retrieve and display the task result.  
Console.WriteLine(task1.Result);
```

If you access the **Result** property before the task has finished running, your code will wait until a result is available before proceeding.

Cancelling Long-Running Tasks

Tasks are often used to perform long-running operations without blocking the UI thread, because of their asynchronous nature. In some cases, you will want to give your users the opportunity to cancel a task if they are tired of waiting. However, it would be dangerous to simply abort the task on demand, because this could leave your application data in an unknown state. Instead, the Task Parallel Library uses *cancellation tokens* to support a cooperative cancellation model. At a high level, the cancellation process works as follows:

1. When you create a task, you also create a cancellation token.
2. You pass the cancellation token as an argument to your delegate method.
3. On the thread that created the task, you *request* cancellation by calling the **Cancel** method on the cancellation token source.
4. In your task method, you can check the status of the cancellation token at any point. If the instigator has requested that the task be cancelled, you can terminate your task logic gracefully, possibly rolling back any changes resulting from the work that the task has performed.

- Use the **Task<TResult>** class
 - Specify the return type in the type argument
- ```
Task<string> task1 = Task.Run<string>(() =>
 DateTime.Now.DayOfWeek.ToString());
```
- Get the result from the **Result** property
- ```
Console.WriteLine("Today is {0}", task1.Result);
```

- Pass a cancellation token as an argument to the delegate method
- Request cancellation from the joining thread
- In the delegate method, check whether the cancellation token is cancelled
- Return or throw an **OperationCanceledException** exception

Typically, you would check whether the cancellation token has been set to canceled at one or more convenient points in your task logic. For example, if your task logic iterates over a collection, you might check for cancellation after each iteration.

The following code example shows how to cancel a task.

Cancelling a Task

```
// Create a cancellation token source and obtain a cancellation token.  
CancellationTokenSource cts = new CancellationTokenSource();  
CancellationToken ct = cts.Token;  
  
// Create and start a task.  
Task.Run( () => doWork(ct) );  
  
// Method run by the task.  
private void doWork(CancellationToken token);  
{  
    ...  
    // Check for cancellation.  
    if(token.IsCancellationRequested)  
    {  
        // Tidy up and finish.  
        ...  
        return;  
    }  
    // If the task has not been cancelled, continue running as normal.  
    ...  
}
```

This approach works well if you do not need to check whether the task ran to completion. Each task exposes a **Status** property that enables you to monitor the current status of the task in the task life cycle. If you cancel a task by returning the task method, as shown in the previous example, the task status is set to **RanToCompletion**. In other words, the task has no way of knowing why the method returned—it may have returned in response to a cancellation request, or it may simply have completed its logic.

If you want to cancel a task and be able to confirm that it was cancelled, you need to pass the cancellation token as an argument to the task constructor in addition to the delegate method. In your task method, you check the status of the cancellation token. If the instigator has requested the cancellation of the task, you throw an **OperationCanceledException** exception. When an **OperationCanceledException** exception occurs, the Task Parallel Library checks the cancellation token to verify whether a cancellation was requested. If it was, the Task Parallel Library handles the **OperationCanceledException** exception, sets the task status to **Canceled**, and throws a **TaskCanceledException** exception. In the code that created the cancellation request, you can catch this **TaskCanceledException** exception and deal with the cancellation accordingly.

To check whether a cancellation was requested and throw an **OperationCanceledException** exception if it was, you call the **ThrowIfCancellationRequested** method on the cancellation token.

The following code example shows how to cancel a task by throwing an **OperationCanceledException** exception.

Cancelling a Task by Throwing an Exception

```
// Create a cancellation token source and obtain a cancellation token.
CancellationTokenSource cts = new CancellationTokenSource();
CancellationToken ct = cts.Token;

// Create and start a task.
Task.Run( () => doWork(ct) );

// Method run by the task.
private void doWork(CancellationToken token);
{
    ...
    // Throw an OperationCanceledException if cancellation was requested.
    token.ThrowIfCancellationRequested();

    // If the task has not been cancelled, continue running as normal.
    ...
}
```



Reference Links: For more information about cancelling tasks, refer Task Cancellation at <http://go.microsoft.com/fwlink/?LinkId=267837> and How to: Cancel a Task and Its Children at <http://go.microsoft.com/fwlink/?LinkId=267838>.

Running Tasks in Parallel

The Task Parallel Library includes a static class named **Parallel**. The **Parallel** class provides a range of methods that you can use to execute tasks simultaneously.

Executing a Set of Tasks Simultaneously

If you want to run a fixed set of tasks in parallel, you can use the **Parallel.Invoke** method. When you call this method, you use lambda expressions to specify the tasks that you want to run simultaneously. You do not need to explicitly create each task—the tasks are created implicitly from the delegates that you supply to the **Parallel.Invoke** method.

The following code example shows how to use the **Parallel.Invoke** method to run several tasks in parallel.

Using the Parallel.Invoke Method

```
Parallel.Invoke( () => MethodForFirstTask(),
                () => MethodForSecondTask(),
                () => MethodForThirdTask() );
```

- Use **Parallel.Invoke** to run multiple tasks simultaneously

```
Parallel.Invoke( () => MethodForFirstTask(),
                () => MethodForSecondTask(),
                () => MethodForThirdTask() );
```

- Use **Parallel.For** to run **for** loop iterations in parallel
- Use **Parallel.ForEach** to run **foreach** loop iterations in parallel
- Use PLINQ to run LINQ expressions in parallel

Running Loop Iterations in Parallel

The **Parallel** class also provides methods that you can use to run **for** and **foreach** loop iterations in parallel. Clearly it will not always be appropriate to run loop iterations in parallel. For example, if you want to compare sequential values, you must run your loop iterations sequentially. However, if each loop iteration represents an independent operation, running loop iterations in parallel enables you to maximize your use of the available processing power.

To run **for** loop iterations in parallel, you can use the **Parallel.For** method. This method has several overloads to cater to many different scenarios. In its simplest form, the **Parallel.For** method takes three parameters:

- An **Int32** parameter that represents the start index for the operation, inclusive.
- An **Int32** parameter that represents the end index for the operation, exclusive.
- An **Action<Int32>** delegate that is executed once per iteration.

The following code example shows how to use a **Parallel.For** loop. In this example, each element of an array is set to the square root of the index value. This is a simple example of a loop in which the order of the iterations does not matter.

Using a Parallel.For Loop

```
int from = 0;
int to = 500000;
double[] array = new double[capacity];

// This is a sequential implementation:
for(int index = 0; index < 500000; index++)
{
    array[index] = Math.Sqrt(index);
}

// This is the equivalent parallel implementation:
Parallel.For(from, to, index =>
{
    array[index] = Math.Sqrt(index);
});
```

To run **foreach** loop iterations in parallel, you can use the **Parallel.ForEach** method. Like the **Parallel.For** method, the **Parallel.ForEach** method includes many different overloads. In its simplest form, the **Parallel.ForEach** method takes two parameters:

- An **IEnumerable<TSource>** collection that you want to iterate over.
- An **Action<TSource>** delegate that is executed once per iteration.

The following code example shows how to use a **Parallel.ForEach** loop. In this example, you iterate over a generic list of **Coffee** objects. For each item, you call a method named **CheckAvailability** that accepts a **Coffee** object as an argument.

Using a Parallel.ForEach Loop

```
var coffeeList = new List<Coffee>();
// Populate the coffee list...

// This is a sequential implementation:
foreach(Coffee coffee in coffeeList)
{
    CheckAvailability(coffee);
}

// This is the equivalent parallel implementation:
Parallel.ForEach(coffeeList, coffee => CheckAvailability(coffee));
```

 **Additional Reading:** For more information and examples about running data operations in parallel, refer Data Parallelism (Task Parallel Library) at <http://go.microsoft.com/fwlink/?LinkId=267839>.

Using Parallel LINQ

Parallel LINQ (PLINQ) is an implementation of Language-Integrated Query (LINQ) that supports parallel operations. In most cases, PLINQ syntax is identical to regular LINQ syntax. When you write a LINQ expression, you can “opt in” to PLINQ by calling the **AsParallel** extension method on your **IEnumerable** data source.

The following code example shows how to write a PLINQ query.

Using PLINQ

```
var coffeeList = new List<Coffee>();
// Populate the coffee list...

var strongCoffees =
    from coffee in coffeeList.AsParallel()
    where coffee.Strength > 3
    select coffee;
```



Additional Reading: For more information about PLINQ, refer Parallel LINQ (PLINQ) at <http://go.microsoft.com/fwlink/?LinkId=267840>.

Linking Tasks

Sometimes it is useful to sequence tasks. For example, you might require that if a task completes successfully, another task is run, or if the task fails, a different task is run that possibly needs to perform some kind of recovery process. A task that runs only when a previous task finishes is called a *continuation*. This approach enables you to construct a pipeline of background operations.

Additionally, a task may spawn other tasks if the work that it needs to perform is also multithreaded in nature. The *parent* task (the task that spawned the new or *nested* tasks) can wait for the nested tasks to complete before it finishes itself, or it can return and let the child tasks continue running asynchronously. Tasks that cause the parent task to wait are called *child* tasks.

- Use task continuations to chain tasks together:
 - **Task.ContinueWith** method links continuation task to antecedent task
 - Continuation task starts when antecedent task completes
 - Antecedent task can pass result to continuation task
- Use nested tasks if you want to start an *independent* task from a task delegate
- Use child tasks if you want to start a *dependent* task from a task delegate

Continuation Tasks

Continuation tasks enable you to chain multiple tasks together so that they execute one after another. The task that invokes another task on completion is known as the *antecedent*, and the task that it invokes is known as the *continuation*. You can pass data from the antecedent to the continuation, and you can control the execution of the task chain in various ways. To create a basic continuation, you use the **Task.ContinueWith** method.

The following code example shows how to create a task continuation.

Creating a Task Continuation

```
// Create a task that returns a string.
Task<string> firstTask = new Task<string>(() => return "Hello");
// Create the continuation task.
// The delegate takes the result of the antecedent task as an argument.
Task<string> secondTask = firstTask.ContinueWith((antecedent) =>
{
    // Task continuation logic here.
});
```

```

        return String.Format("{0}, World!", antecedent.Result));
    }
    // Start the antecedent task.
    firstTask.Start();
    // Use the continuation task's result.
    Console.WriteLine(secondTask.Result);
    // Displays "Hello, World!"
}

```

Notice that when you create the continuation task, you are providing the result of the first task as an argument to the delegate of the second task. For example, you might use the first task to collect some data and then invoke a continuation task to process the data. Continuation tasks do not have to return the same result type as their antecedent tasks.

 **Note:** Continuations enable you to implement the Promise pattern. This is a common idiom that many asynchronous environments use to ensure that operations perform in a guaranteed sequence.

 **Additional Reading:** For more information about continuation tasks, refer Continuation Tasks at <http://go.microsoft.com/fwlink/?LinkId=267841>.

Nested Tasks and Child Tasks

A nested task is simply a task that you create within the delegate of another task. When you create tasks in this way, the nested task and the outer task are essentially independent. The outer task does not need to wait for the nested task to complete before it finishes.

A child task is a type of nested task, except that you specify the **AttachedToParent** option when you create the child task. In this case, the child task and the parent task become far more closely connected. The status of the parent task depends on the status of any child tasks—in other words, a parent task cannot complete until all of its child tasks have completed. The parent task will also propagate any exceptions that its child tasks throw.

To illustrate the difference between nested tasks and child tasks, consider these two simple examples.

The following code example shows how to create a nested task.

Creating Nested Tasks

```

var outer = Task.Run( () =>
{
    Console.WriteLine("Outer task starting...");
    var inner = Task.Run( () =>
    {
        Console.WriteLine("Nested task starting...");
        Thread.Sleep(500000);
        Console.WriteLine("Nested task completing...");
    });
    outer.Wait();
    Console.WriteLine("Outer task completed.");
}

/* Output:
   Outer task starting...
   Nested task starting...
   Outer task completed.
   Nested task completing...
*/

```

In this case, due to the delay in the nested task, the outer task completes before the nested task.

The following code example shows how to create a child task.

MCITICE ONLY STUDENT USE PROHIBITED

Creating Child Tasks

```
var parent = Task.Run( () =>
{
    Console.WriteLine("Parent task starting...");
    var child = Task.Run( () =>
    {
        Console.WriteLine("Child task starting...");
        Thread.Sleep(500000);
        Console.WriteLine("Child task completing...");
    }, TaskCreationOptions.AttachedToParent);
});
parent.Wait();
Console.WriteLine("Parent task completed.");

/* Output:
   Parent task starting...
   Child task starting...
   Child task completing...
   Parent task completed.
*/
```

Note that the preceding example is essentially identical to the nested task example, except that in this case, the child task is created by using the **AttachedToParent** option. As a result, in this case, the parent task waits for the child task to complete before completing itself.

Nested tasks are useful because they enable you to break down asynchronous operations into smaller units that can themselves be distributed across available threads. By contrast, it is more useful to use parent and child tasks when you need to control synchronization by ensuring that certain child tasks are complete before the parent task returns.



Additional Reading: For more information about nested tasks and child tasks, refer Nested Tasks and Child Tasks at <http://go.microsoft.com/fwlink/?LinkId=267842>.

Handling Task Exceptions

When a task throws an exception, the exception is propagated back to the thread that initiated the task (the *joining thread*). The task might be linked to other tasks through continuations or child tasks, so multiple exceptions may be thrown. To ensure that all exceptions are propagated back to the joining thread, the Task Parallel Library bundles any exceptions together in an **AggregateException** object. This object exposes all of the exceptions that occurred through an **InnerExceptions** collection.

- Call **Task.Wait** to catch propagated exceptions
- Catch **AggregateException** in the **catch** block
- Iterate the **InnerExceptions** property and handle individual exceptions

```
try
{
    task1.Wait();
}
catch(AggregateException ae)
{
    foreach(var inner in ae.InnerExceptions)
    {
        // Deal with each exception in turn.
    }
}
```

To catch exceptions on the joining thread, you must wait for the task to complete. You do this by calling the **Task.Wait** method in a **try** block and then catching an **AggregateException** exception in the corresponding **catch** block. A common exception-handling scenario is to catch the **TaskCanceledException** exception that is thrown when you cancel a task.

The following code example shows how to handle task exceptions.

Catching Task Exceptions

```
static void Main(string[] args)
{
    // Create a cancellation token source and obtain a cancellation token.
    CancellationTokenSource cts = new CancellationTokenSource();
    CancellationToken ct = cts.Token;

    // Create and start a long-running task.
    var task1 = Task.Run(() => doWork(ct),ct);

    // Cancel the task.
    cts.Cancel();

    // Handle the TaskCanceledException.
    try
    {
        task1.Wait();
    }
    catch(AggregateException ae)
    {
        foreach(var inner in ae.InnerExceptions)
        {
            if(inner is TaskCanceledException)
            {
                Console.WriteLine("The task was cancelled.");
                Console.ReadLine();
            }
            else
            {
                // If it's any other kind of exception, re-throw it.
                throw;
            }
        }
    }
}

// Method run by the task.
private void doWork(CancellationToken token);
{
    for (int i = 0; i < 100; i++)
    {
        Thread.Sleep(500000);
        // Throw an OperationCanceledException if cancellation was requested.
        token.ThrowIfCancellationRequested();
    }
}
```



Additional Reading: For more information about handling task exceptions, refer Exception Handling (Task Parallel Library) at <http://go.microsoft.com/fwlink/?LinkId=267843>.

Lesson 2

Performing Operations Asynchronously

An asynchronous operation is an operation that runs on a separate thread; the thread that initiates an asynchronous operation does not need to wait for that operation to complete before it can continue.

Asynchronous operations are closely related to tasks. The .NET Framework 4.5 includes some new features that make it easier to perform asynchronous operations. These operations transparently create new tasks and coordinate their actions, enabling you to concentrate on the business logic of your application. In particular, the **async** and **await** keywords enable you to invoke an asynchronous operation and wait for the result within a single method, without blocking the thread.

In this lesson, you will learn various techniques for invoking and managing asynchronous operations.

Lesson Objectives

After completing this lesson, you will be able to:

- Use the **Dispatcher** object to run code on a specific thread.
- Use the **async** and **await** keywords to run asynchronous operations.
- Create methods that are compatible with the **await** operator.
- Create and invoke callback methods.
- Use the Task Parallel Library with traditional Asynchronous Programming Model (APM) implementations.
- Handle exceptions that asynchronous operations throw.

Using the Dispatcher

In the .NET Framework, each thread is associated with a **Dispatcher** object. The dispatcher is responsible for maintaining a queue of work items for the thread. When you work across multiple threads, for example, by running asynchronous tasks, you can use the **Dispatcher** object to invoke logic on a specific thread. You typically need to do this when you use asynchronous operations in graphical applications. For example, if a user clicks a button in a Windows® Presentation Foundation (WPF) application, the click event handler runs on the UI thread. If the event handler starts an asynchronous task, that task runs on the background thread. As a result, the task logic no longer has access to controls on the UI, because these are all owned by the UI thread. To update the UI, the task logic must use the **Dispatcher.BeginInvoke** method to queue the update logic on the UI thread.

- To update a UI element from a background thread:
 - Get the **Dispatcher** object for the thread that owns the UI element
 - Call the **BeginInvoke** method
 - Provide an **Action** delegate as an argument
- ```
lblTime.Dispatcher.BeginInvoke(new Action(() => SetTime(currentTime)));
```

Consider a simple WPF application that consists of a button named **btnGetTime** and a label named **lblTime**. When the user clicks the button, you use a task to get the current time. In this example, the task simply queries the **DateTime.Now** property, but in many scenarios, your applications might retrieve data from web services or databases in response to activity on the UI.

The following code example shows how you might attempt to update the UI from your task logic.

### The Wrong Way to Update a UI Object

```
public void btnGetTime_Click(object sender, RoutedEventArgs e)
{
 Task.Run(() =>
 {
 string currentTime = DateTime.Now.ToString("T");
 SetTime(currentTime);
 });
}

private void SetTime(string time)
{
 lblTime.Content = time;
}
```

If you were to run the preceding code, you would get an **InvalidOperationException** exception with the message "The calling thread cannot access this object because a different thread owns it." This is because the **SetTime** method is running on a background thread, but the **lblTime** label was created by the UI thread. To update the contents of the **lblTime** label, you must run the **SetTime** method on the UI thread. To do this, you can retrieve the **Dispatcher** object that is associated with the **lblTime** object and then call the **Dispatcher.BeginInvoke** method to invoke the **SetTime** method on the UI thread.

The following code example shows how to use the **Dispatcher.BeginInvoke** method to update a control on the UI thread.

### The Correct Way to Update a UI Object

```
public void buttonGetTime_Click(object sender, RoutedEventArgs e)
{
 Task.Run(() =>
 {
 string currentTime = DateTime.Now.ToString("T");
 lblTime.Dispatcher.BeginInvoke(new Action(() => SetTime(currentTime)));
 });
}

private void SetTime(string time)
{
 lblTime.Content = time;
}
```

Note that the **BeginInvoke** method will not accept an anonymous delegate. The previous example uses the **Action** delegate to invoke the **SetTime** method. However, you can use any delegate that matches the signature of the method you want to call.

## Using **async** and **await**

The **async** and **await** keywords were introduced in the .NET Framework 4.5 to make it easier to perform asynchronous operations. You use the **async** modifier to indicate that a method is asynchronous. Within the **async** method, you use the **await** operator to indicate points at which the execution of the method can be suspended while you wait for a long-running operation to return. While the method is suspended at an **await** point, the thread that invoked the method can do other work.

Unlike other asynchronous programming techniques, the **async** and **await** keywords enable you to run logic asynchronously on a single thread. This is particularly useful when you want to run logic on the UI thread, because it enables you to run logic asynchronously on the same thread without blocking the UI.

Consider a simple WPF application that consists of a button named **btnLongOperation** and a label named **lblResult**. When the user clicks the button, the event handler invokes a long-running operation. In this example, it simply sleeps for 10 seconds and then returns the result "Operation complete." In practice, however, you might make a call to a web service or retrieve information from a database. When the task is complete, the event handler updates the **lblResult** label with the result of the operation.

The following code example shows an application that performs a synchronous long-running operation on the UI thread.

### Running a Synchronous Operation on the UI Thread

```
private void btnLongOperation_Click(object sender, RoutedEventArgs e)
{
 lblResult.Content = "Commencing long-running operation...";
 Task<string> task1 = Task.Run<string>(() =>
 {
 // This represents a long-running operation.
 Thread.Sleep(10000);
 return "Operation Complete";
 })
 // This statement blocks the UI thread until the task result is available.
 lblResult.Content = task1.Result;
}
```

In this example, the final statement in the event handler blocks the UI thread until the result of the task is available. This means that the UI will effectively freeze, and the user will be unable to resize the window, minimize the window, and so on. To enable the UI to remain responsive, you can convert the event handler into an asynchronous method.

The following code example shows an application that performs an asynchronous long-running operation on the UI thread.

### Running an Asynchronous Operation on the UI Thread

```
private async void btnLongOperation_Click(object sender, RoutedEventArgs e)
{
 lblResult.Content = "Commencing long-running operation...";
 Task<string> task1 = Task.Run<string>(() =>
 {
 // This represents a long-running operation.
 Thread.Sleep(10000);
 })
 // This statement runs on the UI thread.
 await task1;
 lblResult.Content = task1.Result;
}
```

```

 return "Operation Complete";
 }
 // This statement is invoked when the result of task1 is available.
 // In the meantime, the method completes and the thread is free to do other work.
 lblResult.Content = await task1;
}

```

This example includes two key changes from the previous example:

- The method declaration now includes the **async** keyword.
- The blocking statement has been replaced by an **await** operator.

Notice that when you use the **await** operator, you do not await the result of the task—you await the task itself. When the .NET runtime executes an **async** method, it effectively bypasses the **await** statement until the result of the task is available. The method returns and the thread is free to do other work. When the result of the task becomes available, the runtime returns to the method and executes the **await** statement.



**Additional Reading:** For more information about using **async** and **await**, refer Asynchronous Programming with Async and Await at <https://aka.ms/moc-20483c-m10-pg2>.

## Creating Awaitable Methods

The **await** operator is always used to await the completion of a **Task** instance in a non-blocking manner. If you want to create an asynchronous method that you can wait for with the **await** operator, your method must return a **Task** object. When you convert a synchronous method to an asynchronous method, use the following guidelines:

- If your synchronous method returns **void** (in other words, it does not return a value), the asynchronous method should return a **Task** object.
- If your synchronous method has a return type of **TResult**, your asynchronous method should return a **Task<TResult>** object.

- The **await** operator is always used to wait for a task to complete
- If your synchronous method returns **void**, the asynchronous equivalent should return **Task**
- If your synchronous method has a return type of **T**, the asynchronous equivalent should return **Task<T>**

There's a major difference between returning **void** or **Task** from an **async** method. As long as the method returns **Task**, it's a part of the **Async** operation, and you'll have to use **await** to call the method. The operation will pause until the **async** operation is complete. However, returning **void** from an **async** method will not block the method, and the next line will be called immediately. An **async void** method is the launching point of an **async** operation, even though the method itself cannot be **awaited**.

As a result, **void async** methods are usually event handlers, or a commands-execute handler, as they're the launching point of the operation.

The following code example shows a synchronous method that waits ten seconds and then returns a string.

### A Long-Running Synchronous Method

```
private string GetData()
{
 var task1 = Task.Run<string>(() =>
 {
 // Simulate a long-running task.
 Thread.Sleep(10000);
 return "Operation Complete.";
 });
 return task1.Result;
}
```

To convert this into an awaitable asynchronous method, you must:

- Add the **async** modifier to the method declaration.
- Change the return type from **string** to **Task<string>**.
- Modify the method logic to use the **await** operator with any long-running operations.

The following code example shows how to convert the previous synchronous method into an asynchronous method.

### Creating an Awaitable Asynchronous Method

```
private async Task<string> GetData()
{
 var result = await Task.Run<string>(() =>
 {
 // Simulate a long-running task.
 Thread.Sleep(10000);
 return "Operation Complete.";
 });
 return result;
}
```

The **GetData** method returns a task, so you can use the method with the **await** operator. For example, you might call the method in the event handler for the click event of a button and use the result to set the value of a label named **lblResult**.

The following code example shows how to call an awaitable asynchronous method.

### Calling an Awaitable Asynchronous Method

```
private async void btnGetData_Click(object sender, RoutedEventArgs e)
{
 lblResult.Content = await GetData();
}
```

Note that you can only use the **await** keyword in an asynchronous method.



**Additional Reading:** For more information about return types for asynchronous methods, refer Async Return Types at <https://aka.ms/moc-20483c-m10-pg3>.

## Creating and Invoking Callback Methods

If you must run complex logic when an asynchronous method completes, you can configure your asynchronous method to invoke a callback method. The asynchronous method passes data to the callback method, and the callback method processes the data. You might also use the callback method to update the UI with the processed results.

To configure an asynchronous method to invoke a callback method, you must include a delegate for the callback method as a parameter to the asynchronous method. A callback method typically accepts one or more arguments and does not return a value. This makes the **Action<T>** delegate a good choice to represent a callback method, where *T* is the type of your argument. If your callback method requires multiple arguments, there are versions of the **Action** delegate that accept up to 16 type parameters.

Consider a WPF application that consists of a button named **btnGetCoffees** and a list named **IstCoffees**. When the user clicks the button, the event handler invokes an asynchronous method that retrieves a list of coffees. When the asynchronous data retrieval is complete, the method invokes a callback method. The callback method removes any duplicates from the results and then displays the updated results in the **listCoffees** list.

The following code example shows an asynchronous method that invokes a callback method.

### Invoking a Callback Method

```
// This is the click event handler for the button.
private async void btnGetCoffees_Click(object sender, RoutedEventArgs e)
{
 await GetCoffees(RemoveDuplicates);
}

// This is the asynchronous method.
public async Task GetCoffees(Action<IEnumerable<string>> callback)
{
 // Simulate a call to a database or a web service.
 var coffees = await Task.Run(() =>
 {
 var coffeeList = new List<string>();
 coffeeList.Add("Caffe Americano");
 coffeeList.Add("Café au Lait");
 coffeeList.Add("Café au Lait");
 coffeeList.Add("Espresso Romano");
 coffeeList.Add("Latte");
 coffeeList.Add("Macchiato");
 return coffeeList;
 })

 // Invoke the callback method asynchronously.
 await Task.Run(() => callback(coffees));
}

// This is the callback method.
private void RemoveDuplicates(IEnumerable<string> coffees)
{
 IEnumerable<string> uniqueCoffees = coffees.Distinct();
 Dispatcher.BeginInvoke(new Action(() =>
 {
 // Process the unique coffees here.
 }));
}
```

- Use the **Action<T>** delegate to represent your callback method

- Add the delegate to your asynchronous method parameters

```
public async Task
GetCoffees(Action<IEnumerable<string>> callback)
```

- Invoke the delegate asynchronously within your method

```
await Task.Run(() => callback(coffees));
```

```
 1stCoffees.ItemsSource = uniqueCoffees;
}
```

In the previous example, the **RemoveDuplicates** callback method accepts a single argument of type **IEnumerable<string>** and does not return a value. To support this callback method, you add a parameter of type **Action<IEnumerable<string>>** to your asynchronous method. When you invoke the asynchronous method, you supply the name of the callback method as an argument.



**Reference Links:** For more information, refer Action<T> Delegate at <https://aka.ms/moc-20483c-m10-pg4>.

## Working with APM Operations

Many .NET Framework classes that support asynchronous operations do so by implementing a design pattern known as APM. The APM pattern is typically implemented as two methods: a **BeginOperationName** method that starts the asynchronous operation and an **EndOperationName** method that provides the results of the asynchronous operation. You typically call the **EndOperationName** method from within a callback method. For example, the **HttpWebRequest** class includes methods named **BeginGetResponse** and **EndGetResponse**. The **BeginGetResponse** method submits an asynchronous request to an Internet or intranet resource, and the **EndGetResponse** method returns the actual response that the Internet resource provides.

- Use the **TaskFactory.FromAsync** method to call methods that implement the APM pattern

```
HttpWebRequest request =
 (HttpWebRequest)WebRequest.Create(url);

HttpWebResponse response =
 await Task<WebResponse>.Factory.FromAsync(
 request.BeginGetResponse,
 request.EndGetResponse,
 request) as HttpWebResponse;
```

Classes that implement the APM pattern use an **IAsyncResult** instance to represent the status of the asynchronous operation. The **BeginOperationName** method returns an **IAsyncResult** object, and the **EndOperationName** method includes an **IAsyncResult** parameter.

The Task Parallel Library makes it easier to work with classes that implement the APM pattern. Rather than implementing a callback method to call the **EndOperationName** method, you can use the **TaskFactory.FromAsync** method to invoke the operation asynchronously and return the result in a single statement. The **TaskFactory.FromAsync** method includes several overloads to accommodate APM methods that take varying numbers of arguments.

Consider a WPF application that verifies URLs that a user provides. The application consists of a box named **txtUrl**, a button named **btnSubmitUrl**, and a label named **lblResults**. The user types a URL in the box and then clicks the button. The click event handler for the button submits an asynchronous web request to the URL and then displays the status code of the response in the label. Rather than implementing a callback method to handle the response, you can use the **TaskFactory.FromAsync** method to perform the entire operation.

The following code example shows how to use the **TaskFactory.FromAsync** method to submit an asynchronous web request and handle the response.

### Using the **TaskFactory.FromAsync** Method

```
private async void btnCheckUrl_Click(object sender, RoutedEventArgs e)
{
 // Get the URL provided by the user.
```

```

string url = txtUrl.Text;
// Create an HTTP request.
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
// Submit the request and await a response.
HttpWebResponse response =
 await Task<WebResponse>.Factory.FromAsync(request.BeginGetResponse,
request.EndGetResponse, request)
 as HttpWebResponse;
// Display the status code of the response.
lblResult.Content = String.Format("The URL returned the following status code: {0}",
response.StatusCode);
}

```

 **Additional Reading:** For more information about using the Task Parallel Library with APM patterns, refer TPL and Traditional .NET Framework Asynchronous Programming at <http://go.microsoft.com/fwlink/?LinkId=267847>.

## Demonstration: Using the Task Parallel Library to Invoke APM Operations

This demonstration explains how to convert an application that uses a traditional APM implementation to use the Task Parallel Library instead. The demonstration illustrates how the Task Parallel Library approach results in shorter and simpler code.

### Demonstration Steps

You will find the steps in the **Demonstration: Using the Task Parallel Library to Invoke APM Operations** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD10\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md).

## Handling Exceptions from Awaitable Methods

When you perform asynchronous operations with the **async** and **await** keywords, you can handle exceptions in the same way that you handle exceptions in synchronous code, which is by using **try/catch** blocks.

The following code example shows how to catch an exception that an awaitable method has thrown.

- Use a conventional **try/catch** block to catch exceptions in asynchronous methods
- Subscribe to the **TaskScheduler.UnobservedTaskException** event to create an event handler of last resort

```

TaskScheduler.UnobservedTaskException +=
 (object sender, UnobservedTaskExceptionEventArgs e) =>
{
 // Respond to the unobserved task exception.
}

```

### Catching an Awaitable Method Exception

```

private async void btnThrowError_Click(object sender, RoutedEventArgs e)
{
 using (WebClient client = new WebClient())
 {
 try
 {
 string data = await client.DownloadStringTaskAsync("http://fourthcoffee/bogus");
 }
 catch (WebException ex)
 {

```

```
 lblResult.Content = ex.Message;
 }
}
```

In the previous example, the click event handler for a button calls the **WebClient.DownloadStringTaskAsync** method asynchronously by using the **await** operator. The URL that is provided is invalid, so the method throws a **WebException** exception. Even though the operation is asynchronous, control returns to the **btnThrowErrorHandler\_Click** method when the asynchronous operation is complete and the exception is handled correctly. This works because behind the scenes, the Task Parallel Library is catching the asynchronous exception and re-throwing it on the UI thread.

### Unobserved Exceptions

When a task raises an exception, you can only handle the exception when the joining thread accesses the task, for example, by using the **await** operator or by calling the **Task.Wait** method. If the joining thread never accesses the task, the exception will remain *unobserved*. When the .NET Framework garbage collector (GC) detects that a task is no longer required, the task scheduler will throw an exception if any task exceptions remain unobserved. By default, this exception is ignored. However, you can implement an exception handler of last resort by subscribing to the **TaskScheduler.UnobservedTaskException** event. Within the exception handler, you can set the status of the exception to **Observed** to prevent any further propagation.

The following code example shows how to subscribe to the **TaskScheduler.UnobservedTaskException** event.

### Implementing a Last-Resort Exception Handler

```
static void Main(string[] args)
{
 // Subscribe to the TaskScheduler.UnobservedTaskException event and define an event
 // handler.
 TaskScheduler.UnobservedTaskException += (object sender,
 UnobservedTaskEventArgs e) =>
 {
 foreach (Exception ex in ((AggregateException)e.Exception).InnerExceptions)
 {
 Console.WriteLine(String.Format("An exception occurred: {0}", ex.Message));
 }
 // Set the exception status to Observed.
 e.SetObserved();
 }

 // Launch a task that will throw an unobserved exception
 // by attempting to download an item from an invalid URL.
 Task.Run(() =>
 {
 using(WebClient client = new WebClient())
 {
 client.DownloadStringTaskAsync("http://fourthcoffee/bogus");
 }
 });

 // Give the task time to complete and then trigger garbage collection (for example
 // purposes only).
 Thread.Sleep(5000);
 GC.WaitForPendingFinalizers();
 GC.Collect();

 Console.WriteLine("Execution complete.");
 Console.ReadLine();
}
```

If you use a debugger to step through this code, you will notice that the **UnobservedTaskException** event is fired when the GC runs.

In the .NET Framework 4.5, the .NET runtime ignores unobserved task exceptions by default and allows your application to continue executing. This contrasts with the default behavior in the .NET Framework 4.0, where the .NET runtime would terminate any processes that throw unobserved task exceptions. You can revert to the process termination approach by adding a **ThrowUnobservedTaskExceptions** element to your application configuration file.

The following code example shows how to add a **ThrowUnobservedTaskExceptions** element to an application configuration file.

#### Configuring the ThrowUnobservedTaskExceptions Element

```
<configuration>
 ...
 <runtime>
 <ThrowUnobservedTaskExceptions enabled="true" />
 </runtime>
</configuration>
```

If you set **ThrowUnobservedTaskExceptions** to **true**, the .NET runtime will terminate any processes that contain unobserved task exceptions. A recommended best practice is to set this flag to **true** during application development and to remove the flag before you release your code.

## Lesson 3

# Synchronizing Concurrent Access to Data

Introducing multithreading into your applications has many advantages in terms of performance and responsiveness. However, it also introduces new challenges. When you can simultaneously update a resource from multiple threads, the resource can become corrupted or can be left in an unpredictable state.

In this lesson, you will learn how to use various synchronization techniques to ensure that you access resources in a *thread-safe* manner—in other words, in a way that prevents concurrent access from having unpredictable effects.

### Lesson Objectives

After completing this lesson, you will be able to:

- Use **lock** statements to prevent concurrent access to code.
- Use synchronization primitives to restrict and control access to resources.
- Use concurrent collections to store data in a thread-safe way.

### Using Locks

When you introduce multithreading into your applications, you can often encounter situations in which a resource is simultaneously accessed from multiple threads. If each of these threads can alter the resource, the resource can end up in an unpredictable state. For example, suppose you create a class that manages stock levels of coffee. When you place an order for a number of coffees, a method in the class will:

1. Check the current stock level.
2. If sufficient stock exists, make the coffees.
3. Subtract the number of coffees from the current stock level.

- Create a private object to apply the lock to
- Use the **lock** statement and specify the locking object
- Enclose your critical section of code in the **lock** block

```
private object lockingObject = new object();
lock (lockingObject)
{
 // Only one thread can enter this block at any one time.
}
```

If only one thread can call this method at any one time, everything will work fine. However, suppose two threads call this method at the same time. The current stock level might change between steps 1 and 2, or between steps 2 and 3, making it impossible to keep track of the current stock level and establish whether you have sufficient stock to make a coffee.

To solve this problem, you can use the **lock** keyword to apply a mutual-exclusion lock to critical sections of code. A mutual-exclusion lock means that if one thread is accessing the critical section, all other threads are locked out. To apply a lock, you use the following syntax:

**lock (object) { statement block }**

The first thing to notice is that you apply the lock to an object. This is because the lock works by ensuring that only one thread can access that object at any one time. This object should be private and should serve no other role in your logic; its purpose is to provide something for the **lock** keyword to make mutually exclusive. Next, you put your critical section of code inside the **lock** block. While the **lock** statement is in scope, only one thread can enter the critical section at any one time.

The following code example shows how to use the **lock** keyword to prevent concurrent access to a block of code.

### Using the Lock Keyword

```
class Coffee
{
 private object coffeeLock = new object();
 int stock;

 public Coffee(int initialStock)
 {
 stock = initialStock;
 }

 public bool MakeCoffees(int coffeesRequired)
 {
 lock (coffeeLock)
 {
 if (stock >= coffeesRequired)
 {
 Console.WriteLine(String.Format("Stock level before making coffees: {0}", stock));
 stock = stock - coffeesRequired;
 Console.WriteLine(String.Format("{0} coffees made", coffeesRequired));
 Console.WriteLine(String.Format("Stock level after making coffees: {0}", stock));
 return true;
 }
 else
 {
 Console.WriteLine("Insufficient stock to make coffees");
 return false;
 }
 }
 }
}
```

In the previous example, the **lock** statement ensures that only one thread can enter the critical section of code within the **MakeCoffees** method at any one time.

 **Note:** Internally, the **lock** statement actually uses another Microsoft Visual C#® locking mechanism, the **Monitor.Enter** and **Monitor.Exit** methods, to apply a mutual exclusion lock to a critical section of code. For more information, refer lock Statement (C# Reference) at <http://go.microsoft.com/fwlink/?LinkId=267848> and Thread Synchronization (C# and Visual Basic) at <https://aka.ms/moc-20483c-m10-pg5>.

## Demonstration: Using Lock Statements

This demonstration illustrates how the **lock** statement prevents concurrent access to critical sections of code. The demonstration solution contains a class named **Coffee** that includes a method named **MakeCoffees**. The **MakeCoffees** method does the following:

1. Checks the current coffee stock level.
2. Fulfills the order, if the current stock level is sufficient.
3. Subtracts the number of coffees ordered from the current stock level.

In the **Program** class, a **Parallel.For** loop creates 100 parallel operations. Each parallel operation places an order for between one and 100 coffees at random.

MCITUCE CANIV STIDENIT USE PROHIBITED

When you use a **lock** statement to apply a mutual-exclusion lock to the critical section of code in the **MakeCoffees** method, the method will always function correctly. However, if you remove the **lock** statement, the logic can fail, the stock level becomes negative, and the application throws an exception.

### Demonstration Steps

You will find the steps in the **Demonstration: Using Lock Statements** section on the following page:  
[https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD10\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md).

## Using Synchronization Primitives with the Task Parallel Library

A *synchronization primitive* is a mechanism by which an operating system enables its users, in this case the .NET runtime, to control the synchronization of threads. The Task Parallel Library supports a wide range of synchronization primitives that enable you to control access to resources in a variety of ways. These are the most commonly used synchronization primitives:

- The **ManualResetEventSlim** class enables one or more threads to wait for an event. A **ManualResetEventSlim** object can be in one of two states: *signaled* and *unsignaled*. If a thread calls the **Wait** method and the **ManualResetEventSlim** object is unsignaled, the thread is blocked until the **ManualResetEventSlim** object state changes to signaled. In your task logic, you can call the **Set** or **Reset** methods on the **ManualResetEventSlim** object to change the state to signaled or unsignaled respectively. You typically use the **ManualResetEventSlim** class to ensure that only one thread can access a resource at any one time.

- Use the **ManualResetEventSlim** class to limit resource access to one thread at a time
- Use the **SemaphoreSlim** class to limit resource access to a fixed number of threads
- Use the **CountdownEvent** class to block a thread until a fixed number of tasks signal completion
- Use the **ReaderWriterLockSlim** class to allow multiple threads to read a resource or a single thread to write to a resource at any one time
- Use the **Barrier** class to block multiple threads until they all satisfy a condition



**Reference Links:** For more information about the **ManualResetEventSlim** class, refer to the **ManualResetEventSlim Class** page at <https://aka.ms/moc-20483c-m10-pg6>.

- The **SemaphoreSlim** class enables you to restrict access to a resource, or a pool of resources, to a limited number of concurrent threads. The **SemaphoreSlim** class uses an integer counter to track the number of threads that are currently accessing a resource or a pool of resources. When you create a **SemaphoreSlim** object, you specify an initial value and an optional maximum value. When a thread wants to access the protected resources, it calls the **Wait** method on the **SemaphoreSlim** object. If the value of the **SemaphoreSlim** object is above zero, the counter is decremented and the thread is granted access. When the thread has finished, it should call the **Release** method on the **SemaphoreSlim** object, and the counter is incremented. If a thread calls the **Wait** method and the counter is zero, the thread is blocked until another thread calls the **Release** method. For example, if your coffee shop has three coffee machines and each can only process one order at a time, you might use a **SemaphoreSlim** object to prevent more than three threads simultaneously ordering a coffee.



**Additional Reading:** For more information about the **SemaphoreSlim** class, refer to the **SemaphoreSlim Class** page at <https://aka.ms/moc-20483c-m10-pg7>.

MCT USE ONLY STUDENT USE PROHIBITED

- The **CountdownEvent** class enables you to block a thread until a fixed number of threads have signaled the **CountdownEvent** object. When you create a **CountdownEvent** object, you specify an initial integer value. When a thread completes an operation, it can call the **Signal** method on the **CountdownEvent** object to decrement the integer value. Any threads that call the **Wait** method on the **CountdownEvent** object are blocked until the counter reaches zero. For example, if you need to run ten tasks before you continue with your code, you can create a **CountdownEvent** object with an initial value of ten, signal the **CountdownEvent** object from each task, and wait for the **CountdownEvent** object to reach zero before you proceed. This is useful because your code can dynamically set the initial value of the counter depending on how much work there is to be done.

 **Additional Reading:** For more information about the **CountdownEvent** class, refer to the CountdownEvent Class page at <https://aka.ms/moc-20483c-m10-pg8>

- The **ReaderWriterLockSlim** class enables you to restrict write access to a resource to one thread at a time, while permitting multiple threads to read from the resource simultaneously. If a thread wants to read the resource, it calls the **EnterReadLock** method, reads the resource, and then calls the **ExitReadLock** method. If a thread wants to write to the resource, it calls the **EnterWriteLock** method. If one or more threads have a read lock on the resource, the **EnterWriteLock** method blocks until all read locks are released. When the thread has finished writing to the resource, it calls the **ExitWriteLock** method. Calls to the **EnterReadLock** method are blocked until the write lock is released. As a result, at any one time, a resource can be locked by either one writer or multiple readers. This type of read/write lock is useful in a wide range of scenarios. For example, a banking application might permit multiple threads to read an account balance simultaneously, but a thread that wants to modify the account balance requires an exclusive lock.

 **Additional Reading:** For more information about the **ReaderWriterLockSlim** class, refer to the ReaderWriterLockSlim Class page at <https://aka.ms/moc-20483c-m10-pg9>.

- The **Barrier** class enables you to temporarily halt the execution of several threads until they have all reached a particular point. When you create a **Barrier** object, you specify an initial number of participants. You can change this number at a later date by calling the **AddParticipant** and **RemoveParticipant** methods. When a thread reaches the synchronization point, it calls the **SignalAndWait** method on the **Barrier** object. This decrements the **Barrier** counter and also blocks the calling thread until the counter reaches zero. When the counter reaches zero, all threads are allowed to continue. The **Barrier** class is often used in forecasting scenarios, where various tasks perform interrelated calculations on one time window and then wait for all of the other tasks to reach the same point before performing interrelated calculations on the next time window.

 **Additional Reading:** For more information about the **Barrier** class, refer to the Barrier Class page at <https://aka.ms/moc-20483c-m10-pg10>.

Many of these classes enable you to set timeouts in terms of the number of *spins*. When a thread is waiting for an event, it spins. The length of time a spin takes depends on the computer that is running the thread. For example, if you use the **ManualResetEventSlim** class, you can specify the maximum number of spins as an argument to the constructor. If a thread is waiting for the **ManualResetEventSlim** object to signal and it reaches the maximum number of spins, the thread is suspended and stops using processor resources. This helps to ensure that waiting tasks do not consume excessive processor time.

## Using Concurrent Collections

The standard collection classes in the .NET Framework are, by default, not thread-safe. When you access collections from tasks or other multithreading techniques, you must ensure that you do not compromise the integrity of the collections. One way to do this is to use the synchronization primitives described earlier in this lesson to control concurrent access to your collections. However, the .NET Framework also includes a set of collections that are specifically designed for thread-safe access. The following table describes some of the classes and interfaces that the **System.Collections.Concurrent** namespace provides.

The **System.Collections.Concurrent** namespace includes generic classes and interfaces for thread-safe collections:

- **ConcurrentBag<T>**
- **ConcurrentDictionary< TKey, TValue >**
- **ConcurrentQueue<T>**
- **ConcurrentStack<T>**
- **IProducerConsumerCollection<T>**
- **BlockingCollection<T>**

Class or interface	Description
ConcurrentBag<T>	This class provides a thread-safe way to store an unordered collection of items.
ConcurrentDictionary< TKey, TValue >	This class provides a thread-safe alternative to the <b>Dictionary&lt; TKey, TValue &gt;</b> class.
ConcurrentQueue<T>	This class provides a thread-safe alternative to the <b>Queue&lt;T&gt;</b> class.
ConcurrentStack<T>	This class provides a thread-safe alternative to the <b>Stack&lt;T&gt;</b> class.
IProducerConsumerCollection<T>	This interface defines methods for implementing classes that exhibit producer/consumer behavior; in other words, classes that distinguish between producers who add items to a collection and consumers who read items from a collection. This distinction is important because these collections need to implement a read/write locking pattern, where the collection can be locked either by a single writer or by multiple readers. The <b>ConcurrentBag&lt;T&gt;</b> , <b>ConcurrentQueue&lt;T&gt;</b> , and <b>ConcurrentStack&lt;T&gt;</b> classes all implement this interface.
BlockingCollection<T>	This class acts as a wrapper for collections that implement the <b>IProducerConsumerCollection&lt;T&gt;</b> interface. For example, it can block read requests until a read lock is available, rather than simply refusing a request if a lock is unavailable. You can also use the <b>BlockingCollection&lt;T&gt;</b> class to limit the size of the underlying collection. In this case, requests to add items are blocked until space is available.

Consider an order management system for Fourth Coffee that uses a **ConcurrentQueue<T>** object to represent the queue of orders that customers have placed. Orders are added to the queue at a single order point, but they can be picked up by one of three baristas working in the store. The following example simulates this scenario by creating one task that places an order every 0.25 seconds and three tasks that pick up orders as they become available.

The following code example shows how to use the **ConcurrentQueue<T>** class to queue and de-queue objects in a thread-safe way.

### Using the ConcurrentQueue<T> Collection

```
class Program
{
 static ConcurrentQueue<string> queue = new ConcurrentQueue<string>();

 static void PlaceOrders()
 {
 for (int i = 1; i <= 100; i++)
 {
 Thread.Sleep(250);
 String order = String.Format("Order {0}", i);
 queue.Enqueue(order);
 Console.WriteLine("Added {0}", order);
 }
 }

 static void ProcessOrders()
 {
 while (true) //continue indefinitely
 if (queue.TryDequeue(out order))
 {
 Console.WriteLine("Processed {0}", order);
 }
 }

 static void Main(string[] args)
 {
 var taskPlaceOrders = Task.Run(() => PlaceOrders());
 Task.Run(() => ProcessOrders());
 Task.Run(() => ProcessOrders());
 Task.Run(() => ProcessOrders());

 taskPlaceOrders.Wait();
 Console.WriteLine("Press ENTER to finish");
 Console.ReadLine();
 }
}
```



**Reference Links:** For more information about using concurrent collections, refer System.Collections.Concurrent Namespace at <https://aka.ms/moc-20483c-m10-pg11>.

## Demonstration: Improving the Responsiveness and Performance of the Application Lab

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: Improving the Responsiveness and Performance of the Application Lab.**” section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD10\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_DEMO.md).

MCT USE ONLY - STUDENT USE PROHIBITED

# Lab: Improving the Responsiveness and Performance of the Application

## Scenario

You have been asked to update the Grades application to ensure that the UI remains responsive while the user is waiting for operations to complete. To achieve this improvement in responsiveness, you decide to convert the logic that retrieves the list of students for a teacher to use asynchronous methods. You also decide to provide visual feedback to the user to indicate when an operation is taking place.

## Objectives

After completing this lab, you will be able to:

- Use the **async** and **await** keywords to implement asynchronous methods.
- Use events and user controls to provide visual feedback during long-running operations.

## Lab Setup

Estimated Time: **75 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD10\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD10\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD10_LAK.md).

## Exercise 1: Ensuring That the UI Remains Responsive When Retrieving Teacher Data

### Scenario

In this exercise, you will modify the functionality that retrieves data for teachers to make use of asynchronous programming techniques. First, you will modify the code that gets the details of the current user (when the user is a teacher) to run asynchronously. You will use an asynchronous task to run the LINQ query and use the **await** operator to return the results of the query. Next, you will modify the code that retrieves the list of students for a teacher. In this case, you will configure the code that retrieves the list of students to run asynchronously. When the operation is complete, your code will invoke a callback method to update the UI with the list of students. Finally, you will build and test the application and verify that the UI remains responsive while the application is retrieving data.

## Exercise 2: Providing Visual Feedback During Long-Running Operations

### Scenario

In this exercise, you will create a user control that displays a progress indicator while the Grades application is retrieving data. You will add this user control to the main page but will initially hide it from view. Next, you will modify the code that retrieves data so that it raises one event when the data retrieval starts and another event when the data retrieval is complete. You will create handler methods for these events that toggle the visibility of the progress indicator control, so that the application displays the progress indicator when data retrieval starts and hides it when data retrieval is complete. Finally, you will build and test the application and verify that the UI displays the progress indicator while the application is retrieving data.

# Module Review and Takeaways

In this module, you have learned a variety of asynchronous programming techniques for Visual C#, including how to use the Task Parallel Library, how to use the **async** and **await** keywords, and how to use synchronization primitives.

## Review Questions

### Check Your Knowledge

Question
You create and start three tasks named task1, task2, and task3. You want to block the joining thread until all of these tasks are complete. Which code example should you use to accomplish this?
Select the correct answer.
task1.Wait(); task2.Wait(); task3.Wait();
Task.WaitAll(task1, task2, task3);
Task.WaitAny(task1, task2, task3);
Task.WhenAll(task1, task2, task3);
Task.WhenAny(task1, task2, task3);

### Check Your Knowledge

Question
You have a synchronous method with the following signature: <code>public IEnumerable&lt;string&gt; GetCoffees(string country, int strength)</code> You want to convert this method to an asynchronous method. What should the signature of the asynchronous method be?
Select the correct answer.
<code>public async IEnumerable&lt;string&gt; GetCoffees(string country, int strength)</code>
<code>public async Task&lt;string&gt; GetCoffees(string country, int strength)</code>
<code>public async Task&lt;IEnumerable&lt;string&gt;&gt; GetCoffees(string country, int strength)</code>
<code>public async Task GetCoffees(string country, int strength, out string result)</code>
<code>public async Task GetCoffees(string country, int strength, out IEnumerable&lt;string&gt; result)</code>

MCT USE ONLY. STUDENT USE PROHIBITED

**Check Your Knowledge**

Question	
You want to ensure that no more than five threads can access a resource at any one time. Which synchronization primitive should you use?	
Select the correct answer.	
	The ManualResetEventSlim class.
	The SemaphoreSlim class.
	The CountdownEvent class.
	The ReaderWriterLockSlim class.
	The Barrier class.

# Module 11

## Integrating with Unmanaged Code

### Contents:

Module Overview	11-1
<b>Lesson 1:</b> Creating and Using Dynamic Objects	11-2
<b>Lesson 2:</b> Managing the Lifetime of Objects and Controlling Unmanaged Resources	11-7
<b>Lab:</b> Upgrading the Grades Report	11-13
Module Review and Takeaways	11-14

## Module Overview

Software systems can be complex and may involve applications that are implemented in a variety of technologies. For example, some applications may be managed Microsoft® .NET Framework applications, whereas others may be unmanaged C++ applications. You may want to use functionality from one application in another or use functionality that is exposed through Component Object Model (COM) assemblies, such as the Microsoft Word 14.0 Object Library assembly, in your applications.

In this module, you will learn how to interoperate unmanaged code in your applications and how to ensure that your code releases any unmanaged resources.

### Objectives

After completing this module, you will be able to:

- Integrate unmanaged code into a Microsoft Visual C#® application by using the Dynamic Language Runtime (DLR).
- Control the lifetime of unmanaged resources and ensure that your application releases resources.

## Lesson 1

# Creating and Using Dynamic Objects

There are many different programming languages available, each with its own set of advantages and disadvantages and scenarios where it is better suited. Having the ability to consume components that are implemented in other languages enables you to reuse functionality that may already exist in your organization.

In this lesson, you will learn how to use the DLR to interoperate with unmanaged code.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of dynamic objects.
- Describe the purpose of the DLR.
- Create a dynamic object.
- Invoke methods on a dynamic object.

### What Are Dynamic Objects?

Visual C# is a strongly typed static language.

When you create a variable, you specify the type of that variable, and you can only invoke methods and access members that this type defines. If you try to call a method that the type does not implement, your code will not compile. This is good because the compile time checks catch a large number of possible errors even before you run your code.

The .NET Framework provides dynamic objects so that you can define objects that are not constrained by the static type system in Visual C#.

Instead, dynamic objects are not type checked until run time, which enables you to write code quickly without worrying about defining every member until you need to call them in your code.

You can use dynamic objects in your .NET Framework applications to take advantage of dynamic languages and unmanaged code.

- Objects that do not conform to the strongly typed object model
- Objects that enable you to take advantage of dynamic languages, such as IronPython
- Objects that simplify the process of interoperating with unmanaged code

### Dynamic Languages

Languages such as IronPython and IronRuby are dynamic and enable you to write code that is not compiled until run time. Dynamic languages provide the following benefits:

- They enable faster development cycles because they do not require a build or compile process.
- They have increased flexibility over static languages because there are no static types to worry about.
- They do not have a strongly typed object model to learn.

Dynamic languages do suffer from slower execution times in comparison with compiled languages, because any optimization steps that a compiler may take when compiling the code are left out of the build process.

## Unmanaged Code

Visual C# is a managed language, which means that the Visual C# code you write is executed by a runtime environment known as the Common Language Runtime (CLR). The CLR provides other benefits, such as memory management, Code Access Security (CAS), and exception handling. Unmanaged code such as C++ executes outside the CLR and in general benefits from faster execution times and being extremely flexible.

When you build applications by using the .NET Framework, it is a common use case to want to reuse unmanaged code. Maybe you have a legacy C++ system that was implemented a decade ago, or maybe you just want to use a function in the user32.dll assembly that Windows provides. The process of reusing functionality that is implemented in technologies other than the .NET Framework is known as interoperability. These technologies can include:

- COM
- C++
- Microsoft ActiveX®
- Microsoft Win32 application programming interface (API)

Dynamic objects simplify the code that is required to interoperate with unmanaged code.



**Additional Reading:** For more information about dynamic objects, refer to the DynamicObject Class page at <https://aka.ms/moc-20483c-m11-pg1>.

## What Is the Dynamic Language Runtime?

The .NET Framework provides the DLR, which contains the necessary services and components to support dynamic languages and provides an approach for interoperating with unmanaged components. The DLR is responsible for managing the interactions between the executing assembly and the dynamic object, for example, an object that is implemented in IronPython or in a COM assembly.

The DLR simplifies interoperating with dynamic objects in the followings ways:

The DLR provides:

- Support for dynamic languages, such as IronPython
- Run-time type checking for dynamic objects
- Language binders to handle the intricate details of interoperating with another language

- It defers type-safety checking for unmanaged resources until run time. Visual C# is a type-safe language and, by default, the Visual C# compiler performs type-safety checking at compile time.



**Note:** Type safety relies on the compiler being able to compile or interpret each of the components that the solution references. When interoperating with unmanaged components, the compile-time type-safety check is not always possible.

- It abstracts the intricate details of interoperating with unmanaged components, including marshaling data between the managed and unmanaged environments.

The DLR does not provide functionality that is pertinent to a specific language but provides a set of language binders. A language binder contains instructions on how to invoke methods and marshal data

between the unmanaged language, such as IronPython, and the .NET Framework. The language binders also perform run-time type checks, which include checking that a method with a given signature actually exists in the object.



**Additional Reading:** For more information about the DLR, see the Dynamic Language Runtime Overview page at <http://go.microsoft.com/fwlink/?LinkId=267858>.

## Creating a Dynamic Object

The DLR infrastructure provides the **dynamic** keyword to enable you to define dynamic objects in your applications. When you use the **dynamic** keyword to declare a dynamic object, it has the following implications:

- It defines a variable of type **object**. You can assign any value to this variable and attempt to call any methods. At run time, the DLR will use the language binders to type check your dynamic code and ensure that the member you are trying to invoke exists.
- It instructs the compiler not to perform type checking on any dynamic code.
- It suppresses the Microsoft IntelliSense® feature because IntelliSense is unable to provide syntax assistance for dynamic objects.

• Dynamic objects are declared by using the **dynamic** keyword  
using Microsoft.Office.Interop.Word;  
...

• Dynamic objects are variables of type **object**  
• Dynamic objects do not support:

- Type checking at compile time
- Visual Studio IntelliSense

To create a dynamic object to consume the Microsoft.Office.Interop.Word COM assembly, perform the following steps:

1. In your .NET Framework project, add a reference to the Microsoft.Office.Interop.Word COM assembly.
2. Bring the **Microsoft.Office.Interop.Word** namespace into scope.

The following code example shows how to create an instance of the **Application** COM class in the **Microsoft.Office.Interop.Word** namespace by using the **dynamic** keyword.

### Dynamic Variable Declaration

```
using Microsoft.Office.Interop.Word;
...
dynamic word = new Application();
```

After you have created an instance of the class, you can use the members that it provides in the same way you would with any .NET Framework class.



**Additional Reading:** For more information about the **dynamic** keyword, refer to the Using Type dynamic (C# Programming Guide) page at <https://aka.ms/moc-20483c-m11-pg2>

## Invoking Methods on a Dynamic Object

After you have instantiated a dynamic object by using the **dynamic** keyword, you can access the properties and methods by using the standard Visual C# dot notation.

The following code example shows how you can use the **Add** method of the **Document** class in the Word interop library to create a new Word document.

- You can access members by using the dot notation

```
string filePath = "C:\\FourthCoffee\\Documents\\Schedule.docx";
...
dynamic word = new Application();
dynamic doc = word.Documents.Open(filePath);
doc.SaveAs(filePath);
```

- You do not need to:

- Pass **Type.Missing** to satisfy optional parameters
- Use the **ref** keyword
- Pass all parameters as type **object**

### Invoking a Method on a Dynamic Object

```
// Start Microsoft Word.
dynamic word = new Application();
...
// Create a new document.
dynamic doc = word.Documents.Add();
doc.Activate();
```

You can also pass parameters to dynamic object method calls as you would with any .NET Framework object.

The following code example shows how you can pass a string variable to a method that a dynamic object exposes.

### Passing Parameters to a Method

```
string filePath = "C:\\FourthCoffee\\Documents\\Schedule.docx";
...
dynamic word = new Application();
dynamic doc = word.Documents.Open(filePath);
doc.SaveAs(filePath);
```

Traditionally when consuming a COM assembly, the .NET Framework required you to use the **ref** keyword and pass a **Type.Missing** object to satisfy any optional parameters that a method contains. Also, any parameters that you pass must be of type **object**. This can result in simple method calls requiring a long list of **Type.Missing** objects and code that is verbose and difficult to read. By using the **dynamic** keyword, you can invoke the same methods but with less code.

 **Best Practice:** When using dynamic objects in your code to encapsulate classes that are exposed in COM assemblies, use the Object Browser feature in Microsoft Visual Studio® to view the COM API.

 **Additional Reading:** For more information about how to create and consume dynamic methods, see the How to: Define and Execute Dynamic Methods page at <http://go.microsoft.com/fwlink/?LinkId=267860>.

## Demonstration: Interoperating with Microsoft Word

In this demonstration, you will use dynamic objects to consume the Microsoft.Office.Interop.Word COM assembly in an existing .NET Framework application.

The application will use the Word object model to combine several text files that contain exception information into an exception report in Word format. The code uses the Word object model to add data from the text files, line breaks, and formatting to generate the final document.

### Demonstration Steps

You will find the steps in the **Interoperating with Microsoft Word** section on the following page:

[https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD11\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_DEMO.md).

## Lesson 2

# Managing the Lifetime of Objects and Controlling Unmanaged Resources

When interoperating with unmanaged resources, it is important that you manage and release these resources when you are no longer using them.

In this lesson, you will learn about the life cycle of a typical .NET Framework object and how to ensure that objects release the resources that they have been using when they are destroyed.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the life cycle of a .NET Framework object.
- Implement the dispose pattern.
- Manage the lifetime of an object.

### The Object Life Cycle

The life cycle of an object has several stages, which start at creation of the object and end in its destruction. To create an object in your application, you use the **new** keyword. When the CLR executes code to create a new object, it performs the following steps:

1. It allocates a block of memory large enough to hold the object.
2. It initializes the block of memory to the new object.

- When an object is created:
  1. Memory is allocated
  2. Memory is initialized to the new object
- When an object is destroyed:
  1. Resources are released
  2. Memory is reclaimed

The CLR handles the allocation of memory for all managed objects. However, when you use unmanaged objects, you may need to write code to allocate memory for the unmanaged objects that you create.

When you have finished with an object, you can dispose of it to release any resources, such as database connections and file handles, that it consumed. When you dispose of an object, the CLR uses a feature called the garbage collector (GC) to perform the following steps:

1. The GC releases resources.
2. The memory that is allocated to the object is reclaimed.

The GC runs automatically in a separate thread. When the GC runs, other threads in the application are halted, because the GC may move objects in memory and therefore must update the memory pointers.

 **Additional Reading:** For more information about GC, refer to the Garbage Collection page at <http://go.microsoft.com/fwlink/?LinkId=267861>.

MCT USE ONLY. STUDENT USE PROHIBITED

## Implementing the Dispose Pattern

The dispose pattern is a design pattern that frees resources that an object has used. The .NET Framework provides the **IDisposable** interface in the **System** namespace to enable you to implement the dispose pattern in your applications.

The **IDisposable** interface defines a single parameterless method named **Dispose**. You should use the **Dispose** method to release all of the unmanaged resources that your object consumed. If the object is part of an inheritance hierarchy, the **Dispose** method can also release resources that the base types consumed by calling the **Dispose** method on the parent type. Invoking the **Dispose** method does not destroy an object. The object remains in memory until the final reference to the object is removed and the GC reclaims any remaining resources.

Many of the classes in the .NET Framework that wrap unmanaged resources, such as the **StreamWriter** class, implement the **IDisposable** interface. You should also implement the **IDisposable** interface when you create your own classes that reference unmanaged types.

### Implementing the IDisposable Pattern

To implement the **IDisposable** pattern in your application, perform the following steps:

1. Ensure that the **System** namespace is in scope.

```
using System;
```

2. Implement the **IDisposable** interface in your class definition.

```
...
public class ManagedWord : IDisposable
{
 public void Dispose()
 {
 throw new NotImplementedException();
 }
}
```

3. Add a private field to the class, which you can use to track the disposal status of the object, and check whether the **Dispose** method has already been invoked and the resources released.

```
public class ManagedWord : IDisposable
{
 bool _isDisposed;
 ...
}
```

#### Implement the **IDisposable** interface

```
public class ManagedWord : IDisposable
{
 bool _isDisposed;

 ~ManagedWord
 {
 Dispose(false);
 }

 public void Dispose()
 {
 Dispose(true);
 GC.SuppressFinalize(this);
 }

 protected virtual void Dispose(bool isDisposing) { ... }
}
```

4. Add code to any public methods in your class to check whether the object has already been disposed of. If the object has been disposed of, you should throw an **ObjectDisposedException**.

```
public void OpenWordDocument(string filePath)
{
 if (this._disposed)
 throw new ObjectDisposedException("ManagedWord");
 ...
}
```

5. Add an overloaded implementation of the **Dispose** method that accepts a Boolean parameter. The overloaded **Dispose** method should dispose of both managed and unmanaged resources if it was called directly, in which case you pass a Boolean parameter with the value **true**. If you pass a Boolean parameter with the value of **false**, the **Dispose** method should only attempt to release unmanaged resources. You may want to do this if the object has already been disposed of or is about to be disposed of by the GC.

```
public class ManagedWord : IDisposable
{
 ...
 protected virtual void Dispose(bool isDisposing)
 {
 if (this._disposed)
 return;
 if (isDisposing)
 {
 // Release only managed resources.
 ...
 }
 // Always release unmanaged resources.
 ...
 // Indicate that the object has been disposed.
 this._disposed = true;
 }
}
```

6. Add code to the parameterless **Dispose** method to invoke the overloaded **Dispose** method and then call the **GC.SuppressFinalize** method. The **GC.SuppressFinalize** method instructs the GC that the resources that the object referenced have already been released and the GC does not need to waste time running the finalization code.

```
public void Dispose()
{
 Dispose(true);
 GC.SuppressFinalize(this);
}
```

After you have implemented the **IDisposable** interface in your class definitions, you can then invoke the **Dispose** method on your object to release any resources that the object has consumed. You can invoke the **Dispose** method from a destructor that is defined in the class.

### Implementing a Destructor

You can add a destructor to a class to perform any additional application-specific cleanup that is necessary when your class is garbage collected. To define a destructor, you add a tilde (~) followed by the name of the class. You then enclose the destructor logic in braces.

The following code example shows the syntax for adding a destructor.

### Defining a Destructor

```
class ManagedWord
{
 ...
 // Destructor
 ~ManagedWord
 {
 // Destructor logic.
 }
}
```

When you declare a destructor, the compiler automatically converts it to an override of the **Finalize** method of the object class. However, you cannot explicitly override the **Finalize** method; you must declare a destructor and let the compiler perform the conversion.

If you want to guarantee that the **Dispose** method is always invoked, you can include it as part of the finalization process that the GC performs. To do this, you can add a call to the **Dispose** method in the destructor of the class.

The following code example shows how to invoke the **Dispose** method from a destructor.

### Calling the Dispose Method from a Destructor

```
class ManagedWord
{
 ...
 // Destructor
 ~ManagedWord
 {
 Dispose(false);
 }
}
```

 **Additional Reading:** For more information about the **IDisposable** interface, refer to the **IDisposable** Interface page at <https://aka.ms/moc-20483c-m11-pg3>

## Managing the Lifetime of an Object

Using types that implement the **IDisposable** interface is not sufficient to manage resources. You must also remember to invoke the **Dispose** method in your code when you have finished with the object. If you choose not to implement a destructor that invokes the **Dispose** method when the GC processes the object, you can do this in a number of other ways.

One approach is to explicitly invoke the **Dispose** method after any other code that uses the object.

#### • Explicitly invoke the **Dispose** method

```
var word = default(ManagedObject);
try
{
 word = new ManagedWord();
 // Code to use the ManagedWord object.
}
finally
{
 if(word!=null) word.Dispose();
}
```

#### • Implicitly invoke the **Dispose** method

```
using (var word = default(ManagedObject))
{
 // Code to use the ManagedWord object.
}
```

The following code example shows how you can invoke the **Dispose** method on an object that implements the **IDisposable** interface.

### Invoking the Dispose Method

```
var word = new ManagedWord();
// Code to use the ManagedWord object.
word.Dispose();
```

Invoking the **Dispose** method explicitly after code that uses the object is perfectly acceptable, but if your code throws an exception before the call to the **Dispose** method, the **Dispose** method will never be invoked.

A more reliable approach is to invoke the **Dispose** method in the **finally** block of a **try/catch/finally** or a **try/finally** statement. Any code in the scope of the **finally** block will always execute, regardless of any exceptions that might be thrown. Therefore, with this approach, you can always guarantee that your code will invoke the **Dispose** method.

The following code example shows how you can invoke the **Dispose** method in a **finally** block.

### Invoking the Dispose Method in a finally Block

```
var word = default(ManagedWord);
try
{
 word = new ManagedWord();
 // Code to use the ManagedWord object.
}
catch
{
 // Code to handle any errors.
}
finally
{
 word?.Dispose();
}
```

 **Note:** When explicitly invoking the **Dispose** method, it is good practice to check whether the object is not null beforehand, because you cannot guarantee the state of the object.

Alternatively, you can use a **using** statement to implicitly invoke the **Dispose** method. A **using** block is exception safe, which means that if the code in the block throws an exception, the runtime will still dispose of the objects that are specified in the **using** statement.

The following code example shows how to implicitly dispose of your object by using a **using** statement.

### Disposing Of an Object by Using a using Statement

```
using (var word = default(ManagedWord))
{
 // Code to use the ManagedWord object.
}
```

If your object does not implement the **IDisposable** interface, a **try/finally** block is an exception-safe approach to execute code to release resources. You should aim to use a **try/finally** block when it is not possible to use a **using** statement.



**Additional Reading:** For more information about **using** statements, refer to the using Statement (C# Reference) page at <https://aka.ms/moc-20483c-m11-pg4>.

## Demonstration: Upgrading the Grades Report Lab

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: Upgrading the Grades Report Lab** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD11\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_DEMO.md).

# Lab: Upgrading the Grades Report

## Scenario

You have been asked to upgrade the grades report functionality to generate reports in Word format. In Module 6, you wrote code that generates reports as an XML file; now you will update the code to generate the report as a Word document.

## Objectives

After completing this lab, you will be able to:

- Use dynamic types.
- Manage object lifetime and resources.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD11\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD11\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD11_LAK.md).

## Exercise 1: Generating the Grades Report by Using Word

### Scenario

In this exercise, you will update the reporting functionality to generate reports in Word format.

First you will review the existing code in the **WordWrapper** class that appends headings, text, breaks, and carriage returns to a document. You will write code to create an instance of Word, create a blank Word document, and save a Word document. You will then review the code in the **GenerateStudentReport** method to create a blank document, add a heading and grade data to the document, and save the document using the methods that you reviewed and wrote in the **WordWrapper** class. You will run this method as a separate task. Finally, you will build and test the application and verify that the reports are now generated in Word format.

## Exercise 2: Controlling the Lifetime of Word Objects by Implementing the Dispose Pattern

### Scenario

In this exercise, you will write code to ensure that Word is correctly terminated after generating a grades report.

You will begin by observing that currently the Word object remains in memory after the application has generated a report. You will verify this by observing the running tasks in Task Manager. You will update the code in the **WordWrapper** class to implement the dispose pattern to ensure correct termination of the Word instance. You will then update the code in the **GenerateStudentReport** method to ensure that the **WordWrapper** object is disposed of when the method finishes. Finally, you will build and test the application and verify that Word now closes after the report is generated.

## Module Review and Takeaways

In this module, you have learned how to interoperate with COM assemblies and how to ensure that your objects dispose of any resources they consume.

### Review Questions

#### Check Your Knowledge

Question
Which of the following statements best describes the dynamic keyword?
Select the correct answer.
<input type="checkbox"/> It defines an object of type object and instructs the compiler to perform type checking.
<input type="checkbox"/> It defines a nullable object and instructs the compiler to defer type checking.
<input type="checkbox"/> It defines an object of type object and instructs the compiler to defer type checking.
<input type="checkbox"/> It defines a nullable object and instructs the compiler to perform type checking.

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

Statement	Answer
You can use a <b>using</b> statement to implicitly invoke the <b>Dispose</b> method on an object that implements the <b>IDisposable</b> pattern.	

# Module 12

## Creating Reusable Types and Assemblies

### Contents:

Module Overview	12-1
<b>Lesson 1:</b> Examining Object Metadata	12-2
<b>Lesson 2:</b> Creating and Using Custom Attributes	12-10
<b>Lesson 3:</b> Generating Managed Code	12-16
<b>Lesson 4:</b> Versioning, Signing, and Deploying Assemblies	12-23
<b>Lab:</b> Specifying the Data to Include in the Grades Report	12-30
Module Review and Takeaways	12-32

## Module Overview

Systems often consist of many components, some of which may be shared with other applications in your organization's infrastructure. When you design applications, it is important to think about reuse and consider how the functionality you are implementing might be useful in other applications.

In this module, you will learn how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. You will also learn how to generate code at run time by using the Code Document Object Model (CodeDOM) and how to ensure that your assemblies are signed and versioned, and available to other applications, by using the global assembly cache (GAC).

### Objectives

After completing this module, you will be able to:

- Use reflection to inspect and execute assemblies.
- Create and consume custom attributes.
- Generate managed code at run time by using CodeDOM.
- Version, sign, and deploy your assemblies to the GAC.

## Lesson 1

# Examining Object Metadata

Sometimes applications need to load an existing compiled assembly, view its contents, and execute functionality. For example, a unit testing utility may require the ability to obtain a reference to a type and execute some of the public methods that it exposes. Reflection provides a way for you to examine the types in a compiled assembly.

In this lesson, you will learn how to use reflection to examine the metadata of objects and execute functionality that the object exposes.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of reflection.
- Load an assembly.
- Examine the metadata of an existing type.
- Invoke the members that an assembly exposes.

### What Is Reflection?

Reflection is a powerful feature that enables you to inspect and perform dynamic manipulation of assemblies, types, and type members at run time.

Reflection is used throughout the Microsoft® .NET Framework. Reflection is used by some of the classes that the base class library provides and some of the utilities that ship with Microsoft Visual Studio®. For example, the serialization classes in the **System.Runtime.Serialization** namespace use reflection to determine which type members should be serialized when serializing types.

- Reflection enables you to inspect and manipulate assemblies at run time
- The **System.Reflection** namespace contains:
  - **Assembly**
  - **TypeInfo**
  - **ParameterInfo**
  - **ConstructorInfo**
  - **FieldInfo**
  - **MemberInfo**
  - **PropertyInfo**
  - **MethodInfo**

### Reflection Usage Scenarios

The following table describes some of the possible uses for reflection in your applications.

Use	Scenario
Examining metadata and dependencies of an assembly.	You might choose to do this if you are consuming an unknown assembly in your application and you want to determine whether your application satisfies the unknown assembly's dependencies.
Finding members in a type that have been decorated with a particular attribute.	You might choose to do this if you are implementing a generic storage repository, which will inspect each type and determine which members it needs to persist.
Determining whether a type implements a specific interface.	You might choose to do this if you are creating a pluggable application that enables you to include new assemblies at run time, but you only want your application to load types that implement a specific interface.

Use	Scenario
Defining and executing a method at run time.	You might choose to do this if you are implementing a virtualized platform that can read types and methods that are implemented in a language such as JavaScript, and then creating managed implementations that you can execute in your .NET Framework application.

Executing code by using reflection is much slower than executing static Microsoft Visual C#® code, so you should only use reflection to create and execute code when you really have to and not just because reflection makes it possible.

## Reflection in the .NET Framework

The .NET Framework provides the **System.Reflection** namespace, which contains classes that enable you to take advantage of reflection in your applications. The following list describes some of these classes:

- **Assembly**. This class enables you to load and inspect the metadata and types in a physical assembly.
- **TypeInfo**. This class enables you to inspect the characteristics of a type.
- **ParameterInfo**. This class enables you to inspect the characteristics of any parameters that a member accepts.
- **ConstructorInfo**. This class enables you to inspect the constructor of the type.
- **FieldInfo**. This class enables you to inspect the characteristics of fields that are defined within a type.
- **MemberInfo**. This class enables you to inspect members that a type exposes.
- **PropertyInfo**. This class enables you to inspect the characteristics of properties that are defined within a type.
- **MethodInfo**. This class enables you to inspect the characteristics of the methods that are defined within a type.

The **System** namespace includes the **Type** class, which also exposes a selection of members that you will find useful when you use reflection. For example, the **GetFields** instance method enables you to get a list of **FieldInfo** objects, representing the fields that are defined within a type.

 **Additional Reading:** For more information about the **Type** class, refer to the Type Class page at <https://aka.ms/moc-20483c-m12-pg1>

 **Additional Reading:** For more information about reflection in the .NET Framework class, refer to the Reflection in the .NET Framework page at <http://go.microsoft.com/fwlink/?LinkId=267865>.

## Loading Assemblies by Using Reflection

The **System.Reflection** namespace provides the **Assembly** class, which enables you to encapsulate an assembly in your code so that you can inspect any metadata that is associated with that assembly.

The **Assembly** class provides a number of static and instance members that you can use to load and examine the contents of assemblies. There are two ways that you can load an assembly into your application by using reflection:

- Reflection-only context, in which you can view the metadata that is associated with the assembly and not execute code.
- Execution context, in which you can execute the loaded assembly.

Loading an assembly in reflection-only context can improve performance; however, if you do try to execute it, the Common Language Runtime (CLR) will throw an **InvalidOperationException** exception.

To load an assembly, the **Assembly** class provides a number of static methods, which include the following:

- **LoadFrom**. This method enables you to load an assembly in execution context by using an absolute file path to the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly.

```
var assemblyPath =
 "C:\\\\FourthCoffee\\\\Libs\\\\FourthCoffee.Service.ExceptionHandling.dll";
var assembly = Assembly.LoadFrom(assemblyPath);
```

- **ReflectionOnlyLoad**. This method enables you to load an assembly in reflection-only context from a binary large object (BLOB) that represents the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly from a byte array.

```
var assemblyPath =
 "C:\\\\FourthCoffee\\\\Libs\\\\FourthCoffee.Service.ExceptionHandling.dll";
var rawBytes = File.ReadAllBytes(assemblyPath);
var assembly = Assembly.ReflectionOnlyLoad(rawBytes);
```

- **ReflectionOnlyLoadFrom**. This method enables you to load an assembly in reflection-only context by using an absolute file path to the assembly. The following code example shows how to load the **FourthCoffee.Service.ExceptionHandling** assembly.

```
var assemblyPath =
 "C:\\\\FourthCoffee\\\\Libs\\\\FourthCoffee.Service.ExceptionHandling.dll";
var assembly = Assembly.ReflectionOnlyLoadFrom(assemblyPath);
```

After you have loaded an assembly and have created an instance of the **Assembly** class, you can use any of the instance members to examine the contents of the assembly. The following list describes some of the instance members that the **Assembly** class provides:

- **FullName**. This property enables you to get the full name of the assembly, which includes the assembly version and public key token. The following code example shows the full name of the **File** class in the **System.IO** namespace.

- The **Assembly.LoadFrom** method

```
var assemblyPath = "...";
var assembly = Assembly.LoadFrom(assemblyPath);
```

- The **Assembly.ReflectionOnlyLoad** method

```
var assemblyPath = "...";
var rawBytes = File.ReadAllBytes(assemblyPath);
var assembly = Assembly.ReflectionOnlyLoad(rawBytes);
```

- The **Assembly.ReflectionOnlyLoadFrom** method

```
var assemblyPath = "...";
var assembly = Assembly.ReflectionOnlyLoadFrom(assemblyPath);
```

```
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

- **GetReferencedAssemblies**. This method enables you to get a list of all of the names of any assemblies that the loaded assembly references.
- **GlobalAssemblyCache**. This property enables you to determine whether the assembly was loaded from the GAC.
- **Location**. This property enables you to get the absolute path to the assembly.
- **ReflectionOnly**. This property enables you to determine whether the assembly was loaded in a reflection-only context or in an execution context. If you load an assembly in reflection-only context, you can only examine the code.
- **GetType**. This method enables you to get an instance of the **Type** class that encapsulates a specific type in an assembly, based on the name of the type.
- **GetTypes**. This method enables you to get all of the types in an assembly in an array of type **Type**.

 **Additional Reading:** For more information about the **Assembly** class, refer to the Assembly Class page at <https://aka.ms/moc-20483c-m12-pg2>

## Examining Types by Using Reflection

Reflection enables you to examine the definition of any type in an assembly. Depending on whether you are looking for a type with a specific name or you want to iterate through each type in the assembly in sequence, you can use the **GetType** and **GetTypes** methods that the **Assembly** class provides.

The following code example shows how to load a type by using the **GetType** method, passing the fully qualified name of the type as a parameter to the method call.

- Get a type by name  

```
var assembly = FourthCoffeeServices.GetAssembly();
var type = assembly.GetType("...");
```
- Get all of the constructors  

```
var constructors = type.GetConstructors();
```
- Get all of the fields  

```
var fields = type.GetFields();
```
- Get all of the properties  

```
var properties = type.GetProperties();
```
- Get all of the methods  

```
var methods = type.GetMethods();
```

### Load a Type by Using the **GetType** Method

```
var assembly = FourthCoffeeServices.GetAssembly();
...
var type = assembly.GetType("FourthCoffee.Service.ExceptionHandling.HandleError");
```

If you use the **GetType** method and specify a name of a type that does not exist in the assembly, the **GetType** method returns **null**.

The following code example shows how to iterate through each of the types in an assembly by using the **GetTypes** method. The **GetTypes** method returns an array of **Type** objects.

### Iterate All Types in an Assembly by Using the **GetTypes** Method

```
var assembly = FourthCoffeeServices.GetAssembly();
...
foreach (var type in assembly.GetTypes())
{
 // Code to process each type.
 var typeName = type.FullName;
```

```
}
```

After you have created an instance of the **Type** class, you can then use any of its members to inspect the type's definition. The following list describes some of the key members that the **Type** class provides:

- **GetConstructors.** This method enables you to get all of the constructors that exist in a type. The **GetConstructors** method returns an array that contains **ConstructorInfo** objects. The following code example shows how to get each of the constructors that exists in a type and then get each of the parameters.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var constructors = type.GetConstructors();
foreach (var constructor in constructors)
{
 var parameters = constructor.GetParameters();
}
```

- **GetFields.** This method enables you to get all of the fields that exist in the current type. The **GetFields** method returns an array that contains **FieldInfo** objects. The following code example shows how to iterate through each field in a type and then determine whether the field is static or instance by using the **IsStatic** property that the **FieldInfo** class provides.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var fields = type.GetFields();
foreach (var field in fields)
{
 var isStatic = field.IsStatic;
}
```

- **GetProperties.** This method enables you to get all of the properties that exist in the current type. The **GetProperties** method returns an array that contains  **PropertyInfo** objects. The following code example shows how to iterate through each property and then get the property's name.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var properties = type.GetProperties();
foreach (var property in properties)
{
 var propertyName = property.Name;
}
```

- **GetMethods.** This method enables you to get all of the methods that exist in the current type. The **GetMethods** method returns an array that contains **MethodInfo** objects. The following code example shows how to iterate through each method and then get the method's name.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var methods = type.GetMethods();
foreach (var method in methods)
{
 var methodName = method.Name;
}
```

- **GetMembers.** This method enables you to get any members that exist in the current type, including properties and methods. The **GetMembers** method returns an array that contains **MemberInfo** objects. The following code example shows how to iterate through each member and then determine whether the member is a property or method.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var members = type.GetMembers();
foreach (var member in members)
{
```

```

 if (member.MemberType == MemberTypes.Method)
 {
 // Process the method.
 }
 if (member.MemberType == MemberTypes.Property)
 {
 // Process the property.
 }
 }
}

```

You can use these members to get a collection of **xxxxInfo** objects that represents the different members that a type exposes. If you loaded the type in execution context, you can then use the **xxxxInfo** objects to execute each member.



**Additional Reading:** For more information about the **Type** class, refer to the Type Class page at <https://aka.ms/moc-20483c-m12-pg1>

## Invoking Members by Using Reflection

The reflection application programming interface (API) in the .NET Framework enables you to invoke objects and use the functionality that they encapsulate. Invoking an object by using reflection follows the same pattern that you use to invoke an object in Visual C#, which typically involves the following steps:

1. Create an instance of the type.
2. Invoke methods on the instance.
3. Get or set property values on the instance.

When you invoke static members by using reflection, there is no need to explicitly create an instance of the type.

### Instantiating a Type

To create an instance of a type by using reflection, you need to get a reference to a constructor that the type exposes and then use the **Invoke** method. To instantiate a type by using the **ConstructorInfo** class, perform the following steps:

1. Create an instance of the **ConstructorInfo** class that represents the constructor you will use to initialize an instance of the type. The following code example shows how to initialize an object by using the default constructor.

```

var type = FourthCoffeeServices.GetHandleErrorType();
...
var constructor = type.GetConstructor(new Type[0]);

```

- **Instantiate a type**

```

var type = FourthCoffeeServices.GetHandleErrorType();
...
var constructor = type.GetConstructor(new Type[0]);
...
var initializedObject = constructor.Invoke(new object[0]);

```

- **Invoke methods on the instance**

```

var methodToExecute = type.GetMethod("LogError");
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var response = methodToExecute.Invoke(initializedObject,
 new object[] { "Error message" }) as string;

```

- **Get or set property values on the instance**

```

var property = type.GetProperty("LastErrorMessage");
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var lastErrorMessage = property.GetValue(initializedObject) as string;

```

**Note:** To get the default constructor, you pass an empty array of type **Type** to the **GetConstructor** method call. If you want to invoke a constructor that accepts parameters, you must pass a **Type** array that contains the types that the constructor accepts.

- Call the **Invoke** method on the **ConstructorInfo** object to initialize an instance of the type, passing an object array that represents any parameters that the constructor call expects. The following code example shows how to invoke the default constructor.

```
var initializedObject = constructor.Invoke(new object[0]);
```

You now have an instance of the type that you can use to invoke any instance methods that are defined in the type.

## Invoking Methods

To invoke an instance method, perform the following steps:

- Create an instance of the **MethodInfo** class that represents the method you want to invoke. The following code example shows how to get a method named **LogError**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var methodToExecute = type.GetMethod("LogError");
```

- Call the **Invoke** method on the **MethodInfo** object, passing the initialized object and an object array that represents any parameters that the method call expects. You can then cast the return value of the **Invoke** method to the type of value you were expecting from the method call. The following code example shows how to execute the **LogError** instance method that accepts a *string* parameter and returns a *string* value.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var response =
 methodToExecute.Invoke(initializedObject, new object[] { "Error message" })
 as string;
```

When you invoke static methods, there is no need to create an instance of the type. Instead, you just create an instance of the **MethodInfo** class that represents the method you want to invoke and then call the **Invoke** method. To invoke a static method, perform the following steps:

- Create an instance of the **MethodInfo** class that represents the method you want to invoke. The following code example shows how to get a method named **FlushLog**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var methodToExecute = type.GetMethod("FlushLog");
```

- Call the **Invoke** method on the **MethodInfo** object, passing a **null** value to satisfy the initialized object and an object array that represents any parameters that the method call expects. The following code example shows how to execute the **FlushLog** static method that accepts no parameters and returns a Boolean value.

```
var isFlushed = methodToExecute.Invoke(null, new object[0]) as bool;
```

## Getting and Setting Properties Values

To get or set the value of an instance property, you must first perform the following steps:

- Create an instance of the  **PropertyInfo** class that represents the property you want to get or set. The following code example shows how to get a property named **LastErrorMessage**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var property = type.GetProperty("LastErrorMessage");
```

2. If you want to get the value of a property, you must invoke the **GetValue** method on the  **PropertyInfo** object. The **GetValue** method accepts an instance of the type as a parameter. The following code example shows how to get the value of the **LastErrorMessage** property.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
var lastErrorMessage = property.GetValue(initializedObject) as string;
```

3. If you want to set the value of a property, you must invoke **SetValue** method on the  **PropertyInfo** object. The **SetValue** method accepts an instance of the type and the value you want to set the property to as parameters. The following code example shows how to set the **LastErrorMessage** property to the text **Database connection error**.

```
var initializedObject = FourthCoffeeServices.InstantiateHandleErrorType();
...
property.SetValue(initializedObject, "Database connection error");
```

When you get or set a static property, there is no need to create an instance of the type. Instead, you just create an instance of the  **PropertyInfo** class and then call either the **GetValue** or **SetValue** method. To get or set the value of a static property, perform the following steps:

1. Create an instance of the  **PropertyInfo** class that represents the property you want to get or set. The following code example shows how to get a property named **LastErrorMessage**.

```
var type = FourthCoffeeServices.GetHandleErrorType();
...
var property = type.GetProperty("LastErrorMessage");
```

2. If you want to get the value of a property, you must invoke the **GetValue** method on the  **PropertyInfo** object, passing a **null** value to satisfy the initialized object parameter. The following code example shows how to get the value of the **LastErrorMessage** static property.

```
var lastErrorMessage = property.GetValue(null) as string;
```

3. If you want to set the value of a property, you must invoke the **SetValue** method on the  **PropertyInfo** object, passing a **null** value to satisfy the initialized object parameter, and the value you want to set the property too. The following code example shows how to set the **LastErrorMessage** static property to the text **Database connection error**.

```
property.SetValue(null, "Database connection error");
```

## Demonstration: Inspecting Assemblies

In this demonstration, you will create a tool that you can use to inspect the contents of an assembly.

The application will use the **System.Reflection** classes to load an assembly, get all of the types in that assembly, and then for each type, get all of the properties and methods.

### Demonstration Steps

You will find the steps in the **Demonstration: Inspecting Assemblies** section on the following page:  
[https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md).

## Lesson 2

# Creating and Using Custom Attributes

The .NET Framework uses attributes to provide additional metadata about a type or type member. The .NET Framework provides many attributes out of the box that you can use in your applications.

In this lesson, you will learn how to create your own custom attributes and read the metadata that is encapsulated in custom attributes at run time by using reflection.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of attributes.
- Create and use custom attributes.
- Process custom attributes by using reflection

### What Are Attributes?

Attributes provide a mechanism that uses the declarative programming model to include additional metadata about elements, such as types, properties, and methods, in your application. Your application can use the additional information that attributes provide to control application behavior and how data is processed at run time. You can also use the information that attributes encapsulate in case tools and other utilities that aid the development process, such as unit test frameworks.

The .NET Framework makes extensive use of attributes throughout the base class library. The following list describes some of the attributes that the .NET Framework provides:

- The **Obsolete** attribute in the **System** namespace, which you can use to indicate that a type or a type member has been superseded and is only there to ensure backward compatibility.
- The **Serializable** attribute in the **System** namespace, which you can use to indicate that an **IFormatter** implementation can serialize and deserialize a type.
- The **NonSerialized** attribute in the **System** namespace, which you can use to indicate that an **IFormatter** implementation should not serialize or deserialize a member in a type.
- The **DataContract** attribute in the **System.Runtime.Serialization** namespace, which you can use to indicate that a **DataContractSerializer** object can serialize and deserialize a type.
- The **QueryInterceptor** attribute in the **System.Data.Services** namespace, which you can use to control access to an entity in Window Communication Foundation (WCF) Data Services.
- The **ConfigurationProperty** attribute in the **System.Configuration** namespace, which you can use to map a property member to a section in an application configuration file.

All attributes in the .NET Framework derive either directly from the abstract **Attribute** base class in the **System** namespace or from another attribute.

- Use attributes to provide additional metadata about an element
- Use attributes to alter run-time behavior

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
 [Obsolete("This property will be removed in the next release.")]
 [DataMember]
 public string Name { get; set; }

 ...
}
```

## Applying Attributes

To use an attribute in your code, perform the following steps:

1. Bring the namespace that contains the attribute you want to use into scope.
2. Apply the attribute to the code element, satisfying any parameters that the constructor expects.
3. Optionally set any of the named parameters that the attribute exposes.

The following code example shows how to apply the **DataContract** attribute to the **SalesPerson** class definition and set the *Name* and *IsReference* named parameters.

### Applying the **DataContract** Attribute

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
 ...
}
```

You can apply multiple attributes to a single element to create a hierarchy of metadata that describes the element.

The following code example shows how to apply the **Obsolete** and **DataMember** attributes to the **Name** property in the **SalesPerson** class, to indicate that the property should be serialized but will be removed from the type definition in the next release.

### Applying the **Obsolete** and **DataContract** Attributes

```
[DataContract(Name = "SalesPersonContract", IsReference=false)]
public class SalesPerson
{
 [Obsolete("This property will be removed in the next release. Use the FirstName and LastName properties instead.")]
 [DataMember]
 public string Name { get; set; }
}
```

 **Additional Reading:** For more information about the **Attribute** class, refer to the **Attribute Class** page at <https://aka.ms/moc-20483c-m12-pg4>

## Creating and Using Custom Attributes

The .NET Framework provides an extensive set of attributes that you can use in your applications. However, there will be a time when you need an attribute that the .NET Framework does not provide. For example, maybe you want to include information about the developer who authored the source code for an application, or maybe you need some additional data for a custom testing framework you are using to test the application.

Derive from the **Attribute** class or another attribute

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperInfoAttribute : Attribute
{
 private string _emailAddress;
 private int _revision;

 public DeveloperInfo(string emailAddress, int revision)
 {
 this._emailAddress = emailAddress;
 this._revision = revision;
 }

 [DeveloperInfo("holly@fourthcoffee.com", 3)]
 public class SalePerson
 {
 ...
 }
}
```

To create an attribute, perform the following steps:

1. Create a type that derives from the **Attribute** base class or another existing attribute type.
2. Apply the **AttributeUsage** attribute to your custom attribute class to describe which elements you can apply this attribute to.

 **Note:** If you apply an attribute to an element that conflicts with the value of the **AttributeUsage** attribute, the compiler will throw an error at build time.

3. Define a constructor to initialize the custom attribute.
4. Define any properties that you want to enable users of the attribute to optionally provide information. Any properties that you define that have a **get** accessor will be exposed through the attribute as a named parameter.

When creating attributes, the convention is to end the name with an attribute suffix. When the type is used as an attribute, the compiler removes that suffix.

The code example shows how to create an attribute that encapsulates metadata about the developer who creates the element.

### Creating a Custom Attribute

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperInfoAttribute : Attribute
{
 private string _emailAddress;
 private int _revision;

 public DeveloperInfo(string emailAddress, int revision)
 {
 this._emailAddress = emailAddress;
 this._revision = revision;
 }

 public string EmailAddress
 {
 get { return this._emailAddress; }
 }

 public int Revision
 {
 get { return this._revision; }
 }
}
```

Using a custom attribute is no different from using an attribute that the .NET Framework provides. You simply apply the attribute to an element and ensure that you pass the required information to the constructor.

The following code example shows how to apply the **DeveloperInfo** attribute to a type definition.

### Applying a Custom Attribute

```
[DeveloperInfo("holly@fourthcoffee.com", 3)]
public class SalePerson
{
 ...
 [DeveloperInfo("linda@fourthcoffee.com", 1)]
 public IEnumerable<Sale> GetAllSales()
 {
```

```

 ...
}

}

```

 **Additional Reading:** For more information about how to create custom attributes, refer to the Creating Custom Attributes (C# and Visual Basic) page at <https://aka.ms/moc-20483c-m12-pg5>.

## Processing Attributes by Using Reflection

If you use the existing attributes that the .NET Framework provides, they normally have a purpose other than just to exist in your code. For example, the **IFormatter** serializers use the **Serializable** attribute during the serializing and deserializing processes.

Similarly, you can use attributes to encapsulate metadata about an element in your code. For example, if you create an attribute that provides information about the developer who authored the element, you may want a way to extract this information so that you can produce a document listing all of the developers who were involved in developing the application.

Similar to accessing other type and member information, you can also use reflection to process attributes. The **System.Reflection** namespace provides a collection of extension methods that you can use to access attributes and the metadata they encapsulate. The following list describes some of these methods:

- **GetCustomAttribute**. This method enables you to get a specific attribute that was used on an element. The following code example shows how to get a **DeveloperInfo** attribute that has been applied to a type.

```

var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attribute =
 type.GetCustomAttribute(typeof(DeveloperInfo), getInheritedAttributes);

```

Use reflection to access the metadata that is encapsulated in custom attributes

```

var type = FourthCoffee.GetSalesPersonType();

var attributes = type.GetCustomAttributes(typeof(DeveloperInfo),
false);

foreach (var attribute in attributes)
{
 var developerEmailAddress = attribute.EmailAddress;
 var codeRevision = attribute.Revision;
}

```

 **Note:** The *getInheritedAttributes* parameter instructs the **GetCustomAttribute** method call to return either only attributes that have been explicitly applied to the current type, or attributes that have been either explicitly applied or inherited from any parent type.

- **GetCustomAttribute**. This generic method also enables you to get a specific attribute that was used on an element. This is a generic method, so you can specify the type of attribute you want the **GetCustomAttribute** method to return. This means that you do not have to write conditional logic to filter an array of objects to determine the type of an attribute and perform any casting. The following code example shows how to get a **DeveloperInfo** attribute that has been applied to a type.

```

var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attribute = type.GetCustomAttribute<DeveloperInfo>(getInheritedAttributes);

```

- **GetCustomAttributes**. This method enables you to get a list of specific attributes that were used on an element. Typically, you would use this method if more than one attribute of the same type has been applied to an element, in which case this method call would return all instances. The following code example shows how to get all of the **DeveloperInfo** attributes that were applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attributes =
 type.GetCustomAttributes(typeof(DeveloperInfo), getInheritedAttributes);
```

- **GetCustomAttributes**. This generic method enables you to get a list of specific attributes that were used on an element. Similar to the **GetCustomAttribute** generic method, because this is a generic method, the method call will only return attributes that match the generic type. The following code example shows how to get all of the **DeveloperInfo** attributes that were applied to a type.

```
var type = FourthCoffeeServices.GetHandleErrorType();
var getInheritedAttributes = false;
var attributes =
 type.GetCustomAttributes<DeveloperInfo>(getInheritedAttributes);
```

The following code example shows how to iterate through each of the custom attributes that have been applied to a type definition and then access the data that any **DeveloperInfo** attributes encapsulate.

#### Iterating Custom Attributes That Have Been Applied to a Type

```
var type = FourthCoffee.GetSalesPersonType();

var attributes = type.GetCustomAttributes(typeof(DeveloperInfo), false);

foreach (var attribute in attributes)
{
 var developerEmailAddress = attribute.EmailAddress;
 var codeRevision = attribute.Revision;
}
```

 **Note:** The **false** value that is passed to the **GetCustomAttributes** method instructs the method to only get custom attributes that have been explicitly applied to the current type, not attributes that have been inherited.

To access custom attributes that have been applied to a member, you must create an **xxxxInfo** object that represents the member and then invoke the **GetCustomAttributes** method.

The following code example shows how to iterate through each of the custom attributes that have been applied to a method called **GetAllSales** and then access the data that any **DeveloperInfo** attributes have encapsulated.

#### Iterate Custom Attributes That Have Been Applied to a Method

```
var type = FourthCoffee.GetSalesPersonType();

var methodToInspect = type.GetMethod("GetAllSales");

var attributes = methodToInspect.type.GetCustomAttributes(typeof(DeveloperInfo), false);

foreach (var attribute in attributes)
{
 var developerEmailAddress = attribute.EmailAddress;
 var codeRevision = attribute.Revision;
}
```



**Additional Reading:** For more information about how to process custom attributes, refer to the Accessing Custom Attributes page at <http://go.microsoft.com/fwlink/?LinkId=267869>.

## Demonstration: Consuming Custom Attributes by Using Reflection

In this demonstration, you will use reflection to read the **DeveloperInfo** attributes that have been used to provide additional metadata on types and type members.

The application uses the **GetCustomAttribute** generic method that the **Type** and **MethodInfo** classes provide to extract any additional information that a custom **DeveloperInfo** attribute encapsulates.

### Demonstration Steps

You will find the steps in the **Demonstration: Consuming Custom Attributes by Using Reflection** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md).

## Lesson 3

# Generating Managed Code

You can use Visual Studio to write managed code when you have clearly defined requirements upon which to base your implementation. However, sometimes you may want to generate code at run time based on a varying set of requirements. In this scenario, you need a framework that enables you to define instructions that a process can translate into executable code. The .NET Framework provides the CodeDOM feature for this very purpose.

In this lesson, you will learn how to generate managed code at run time by using CodeDOM.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the purpose of CodeDOM.
- Define a type by using CodeDOM.
- Compile a CodeDOM model into source code files.
- Compile source code files into an assembly.

### What Is CodeDOM?

CodeDOM is a feature of the .NET Framework that enables you to generate code at run time.

CodeDOM enables you to build a model that represents the source code that you want to create and then generate the source code by using one of the code generators that are included in CodeDOM. By default, CodeDOM includes code generators for Visual C#, Microsoft Visual Basic®, and Microsoft JScript®.

CodeDOM is a powerful feature that you can use for generating template source files that contain boilerplate code or even generating source code files that serve as a proxy between your application and a remote entity. To use CodeDOM in your application, use the following namespaces:

- Define a model that represents your code by using:
  - The **CodeCompileUnit** class
  - The **CodeNamespace** class
  - The **CodeTypeDeclaration** class
  - The **CodeMemberMethod** class
- Generate source code from the model:
  - Visual C# by using the **CSharpCodeProvider** class
  - JScript by using the **JScriptCodeProvider** class
  - Visual Basic by using the **VBCodeProvider** class
- Generate a .dll or a .exe that contains your code

- **System.CodeDom**. This namespace contains types that enable you to define a model that represents the source code you want to generate.
- **System.CodeDom.Compiler**. This namespace contains types that enable you to generate and manage compiled code.
- **Microsoft.CSharp.CSharpCodeProvider**. This namespace contains the Visual C# code compiler and generator.
- **Microsoft.JScript.JScriptCodeProvider**. This namespace contains the JScript code compiler and generator.
- **Microsoft.VisualBasic.VBCodeProvider**. This namespace contains the Visual Basic code compiler and generator.

You can use the classes in the **System.CodeDom** namespace to create a model that represents the code you want to create. The model can include anything from a complex class hierarchy to a single class with some members. For example, you can use the **CodeNamespace** class to represent a namespace, and you can use the **CodeMemberMethod** class to represent a method.

The following table describes some of the classes you can use to create your model.

Class	Description
<b>CodeCompileUnit</b>	Enables you to encapsulate a collection of types that ultimately will compile into an assembly.
<b>CodeNamespace</b>	Enables you to define a namespace that you can use to organize your class hierarchy.
<b>CodeTypeDeclaration</b>	Enables you to define a class, structure, interface, or enumeration in your model.
<b>CodeMemberMethod</b>	Enables you to define a method in your model and add it to a type, such as a class or an interface.
<b>CodeMemberField</b>	Enables you to define a field, such as an <b>int</b> variable, and add it to a type, such as a class or struct.
<b>CodeMemberProperty</b>	Enables you to define a property with <b>get</b> and <b>set</b> accessors and add it to a type, such as a class or struct.
<b>CodeConstructor</b>	Enables you to define a constructor so that you can create an instance type in your model.
<b>CodeTypeConstructor</b>	Enables you to define a static constructor so that you can create a singleton type in your model.
<b>CodeEntryPoint</b>	Enables you to define an entry point in your type, which is typically a static method with the name <b>Main</b> .
<b>CodeMethodInvokeExpression</b>	Enables you to create a set of instructions that represents an expression that you want to execute.
<b>CodeMethodReferenceExpression</b>	Enables you to create a set of instructions that detail a method in a particular type that you want to execute. Typically, you would use this class with the <b>CodeMethodInvokeExpression</b> class when you implement the body of method in a model.
<b>CodeTypeReferenceExpression</b>	Enables you to represent a reference type that you want to use as part of an expression in your model. Typically, you would use this class with the <b>CodeMethodInvokeExpression</b> class and the <b>CodeTypeReferenceExpression</b> class when you implement the body of method in a model.
<b>CodePrimitiveExpression</b>	Enables you to define an expression value, which you may want to pass as a parameter to a method or store in a variable.

After you have defined your model by using the classes in the **System.CodeDom** namespace, you can then use a code generator provider, such as the **CSharpCodeProvider** class in the **Microsoft.CSharp.CSharpCodeProvider** namespace, to compile your model and generate your code.



**Additional Reading:** For more information about CodeDOM, refer to the Dynamic Source Code Generation and Compilation page at <http://go.microsoft.com/fwlink/?LinkId=267870>.

## Defining a Type and Type Members

Defining a type by using CodeDOM follows the same pattern as defining a type in native Visual C#. The only difference is that when using CodeDOM, you write a set of instructions that a code generator provider will interpret to generate the source code that represents your model.

The **System.CodeDOM** namespace includes the types that you can use to write these instructions. The following steps describe how to use some of the **System.CodeDOM** types to define a type that contains an entry point method named **Main**:

1. Create a **CodeCompileUnit** object to represent the assembly that will contain the type. The following code example shows how to create a **CodeCompileUnit** object.

```
var unit = new CodeCompileUnit();
```

2. Create a **CodeNamespace** object to represent the namespace that the type will be scoped to and add the namespace to the **CodeCompileUnit** object. The following code example shows how to define the **FourthCoffee.Dynamic** namespace.

```
var dynamicNamespace = new CodeNamespace("FourthCoffee.Dynamic");
unit.Namespaces.Add(dynamicNamespace);
```

3. Import any additional namespaces that the types in the namespace will use. The following code example shows how to bring the **System** namespace into scope.

```
dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));
```

4. Create a **CodeTypeDeclaration** object that represents the type you want to add to the namespace. The following code example shows how to create a type named **Program**.

```
var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);
```

5. Create a **CodeEntryPointMethod** object to represent the static main method in the **Program** type. The following code example shows how to define an entry point method named **Main** and add it to the **Program** type.

```
var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);
```

### Defining a type with a **Main** method

```
var unit = new CodeCompileUnit();

var dynamicNamespace = new CodeNamespace("FourthCoffee.Dynamic");
unit.Namespaces.Add(dynamicNamespace);

dynamicNamespace.Imports.Add(new CodeNamespaceImport("System"));

var programType = new CodeTypeDeclaration("Program");
dynamicNamespace.Types.Add(programType);

var mainMethod = new CodeEntryPointMethod();
programType.Members.Add(mainMethod);

var expression = new CodeMethodInvokeExpression(
 new CodeTypeReferenceExpression("Console"), "WriteLine",
 new CodePrimitiveExpression("Hello Development Team..!!"));
```

6. Define the body of the **Main** method by using the **CodeMethodInvokeExpression**, **CodeTypeReferenceExpression**, and **CodePrimitiveExpression** classes. The parameters that you pass to the constructors of these objects enable you to define which method you want to invoke and the parameters that the method expects. The following code example shows how invoke the **Console.WriteLine** method to write the message **Hello Development Team..!!** to the console window.

```
var expression = new CodeMethodInvokeExpression(
 new CodeTypeReferenceExpression("Console"),
 "WriteLine",
 new CodePrimitiveExpression("Hello Development Team..!!"));
mainMethod.Statements.Add(expression);
```

After you have defined your model, you can then use a code generator provider to compile and generate your code.



**Additional Reading:** For more information about how to define a model by using CodeDOM, refer to the Using the CodeDOM page at <http://go.microsoft.com/fwlink/?LinkId=267871>.

## Compiling a CodeDOM Model

After you have defined the contents of your assembly by using the types in the **System.CodeDOM** namespace, you can then compile and generate an assembly. You can split the process for compiling and generating an assembly into the following parts:

1. Compiling the model and generating source code files for each type.
2. Generating an assembly that contains the necessary references and the types that are defined in the source code files.

Generate source code files from your CodeDOM model

```
var provider = new CSharpCodeProvider();
var fileName = "program.cs";
var stream = new StreamWriter(fileName);
var textWriter = new IndentedTextWriter(stream);

var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;

var compileUnit = FourthCoffee.GetModel();
provider.GenerateCodeFromCompileUnit(
 compileUnit,
 textWriter,
 options);

textWriter.Close();
stream.Close();
```

## Compiling a Model into a Source Code File

To compile your model and generate source code files, perform the following steps:

1. Create an instance of the code generator provider you want to use. The following code example shows how to create an instance of the **CSharpCodeProvider** class that will produce Visual C# code.

```
var provider = new CSharpCodeProvider();
```

2. Create a **StreamWriter** object that the code generator will use to write the compiled code to a file. The following code example shows how to create a **StreamWriter** object.

```
var fileName = "program.cs";
var stream = new StreamWriter(fileName);
```

3. Create an **IndentedTextWriter** object that will write the indented source code to a file. The following code example shows how to create an **IndentedTextWriter** object.

```
var textWriter = new IndentedTextWriter(stream);
```

4. Create a **CodeGeneratorOptions** object that encapsulates your code generation settings. The following code example shows how to create a **CodeGeneratorOptions** object and set the **BlankLinesBetweenMembers** property so that members are separated by a blank line.

```
var options = new CodeGeneratorOptions();
options.BlankLinesBetweenMembers = true;
```

5. Invoke the **GenerateCodeFromCompileUnit** method on the **CSharpCodeProvider** object to generate the source code. The following code example shows how to invoke the **GenerateCodeFromCompileUnit** method, passing the **CodeCompileUnit**, **IndentedTextWriter**, and **CodeGeneratorOptions** objects as parameters.

```
var compileUnit = FourthCoffee.GetModel();
provider.GenerateCodeFromCompileUnit(compileunit, textWriter, options);
```

6. Close the **IndentedTextWriter** and **StreamWriter** objects. The following code example shows how to close the **IndentedTextWriter** and **StreamWriter** objects, flushing the compiled code to the output file.

```
textWriter.Close();
stream.Close();
```

After you have executed the code to compile your model, you will have a source .cs file that contains the compiled Visual C# code.

The following code example shows the compiled Visual C# code for a model that contains the **Program** type with an entry point method called **Main**.

#### Compiled Visual C# Code

```
//--
// <auto-generated>
// This code was generated by a tool.
// Runtime Version:4.0.30319.17626
//
// Changes to this file may cause incorrect behavior and will be lost if
// the code is regenerated.
// </auto-generated>
//--

namespace FourthCoffee.Dynamic {
 using System;

 public class Program {

 public static void Main() {
 Console.WriteLine("Hello Development Team..!!!");
 }
 }
}
```

## Compiling Source Code into an Assembly

After you have compiled your CodeDOM model into one or more source code files, you can compile the files into an assembly.

To generate an executable assembly that contains the **FourthCoffee.Dynamic.Program** type from a source code file, perform the following steps:

1. Create an instance of the code generator provider you want to use. The following code example shows how to create an instance of the **CSharpCodeProvider** class that will produce the executable assembly.

```
var provider = new CSharpCodeProvider();
```

2. Create a **CompilerParameters** object that you will use to define the settings for the compiler, such as which assemblies to reference, and whether to generate a .dll or an .exe file. The following code example shows how to create a **CompilerParameters** object, add a reference to the System.dll file, and instruct the compiler to generate an executable file named FourthCoffee.exe.

```
var compilerSettings = new CompilerParameters();
compilerSettings.ReferencedAssemblies.Add("System.dll");
compilerSettings.GenerateExecutable = true;
compilerSettings.OutputAssembly = "FourthCoffee.exe";
```

3. Invoke the **CompileAssemblyFromFile** method on the **CSharpCodeProvider** object to generate the assembly. The following code example shows how to invoke the **CompileAssemblyFromFile** method, passing the **CompilerParameters** object and a string variable that contains the path to the source code file.

```
var sourceCodeFileName = "program.cs";
var compilationResults = provider.CompileAssemblyFromFile(
 compilerSettings,
 sourceCodeFileName);
```

 **Note:** The **CompileAssemblyFromFile** method also accepts an array of source file names, so you can compile several source code files into a single assembly.

4. You can then use the properties that the **CompilerResults** object provides to determine whether the compilation was successful. The following code example shows how to iterate the **CompilationErrorCollection** object by using the **Errors** property.

```
var buildFailed = false;
foreach (var error in compilationResults.Errors)
{
 var errorMessage = error.ToString();
 buildFailed = true;
}
```

You have now generated an assembly that displays the message **Hello Development Team..!!** when it is executed.

### Generate an assembly from your source code files

```
var provider = new CSharpCodeProvider();

var compilerSettings = new CompilerParameters();
compilerSettings.ReferencedAssemblies.Add("System.dll");
compilerSettings.GenerateExecutable = true;
compilerSettings.OutputAssembly = "FourthCoffee.exe";

var sourceCodeFileName = "program.cs";
var compilationResults = provider.CompileAssemblyFromFile(
 compilerSettings,
 sourceCodeFileName);

var buildFailed = false;
foreach (var error in compilationResults.Errors)
{
 var errorMessage = error.ToString();
 buildFailed = true;
}
```



**Additional Reading:** For more information about how to compile a CodeDOM model, refer to the Generating and Compiling Source Code from a CodeDOM Graph page at <http://go.microsoft.com/fwlink/?LinkId=267872>.

## Lesson 4

# Versioning, Signing, and Deploying Assemblies

When you finish developing an application, you should sign and version the assembly before you distribute it to users. You must also consider how and where the assembly is going to be installed.

In this lesson, you will learn how to version assemblies and install an assembly into the GAC.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe the key features of assemblies.
- Describe the GAC.
- Explain how to sign an assembly.
- Describe the key features of assembly versioning.
- Install an assembly in the GAC.

### What Is an Assembly?

An assembly is a collection of types and resources that form a unit of functionality. An assembly might consist of a single portable executable (PE) file, such as an executable (.exe) program or dynamic link library (.dll) file, or it might consist of multiple PE files and external resource files, such as bitmaps or data files. An assembly is the building block of a .NET Framework application because an application consists of one or more assemblies.

- An assembly is a collection of types and resources
- An assembly is a versioned deployable unit
- An assembly can contain:
  - IL code
  - Resources
  - Type metadata
  - Manifest

### Contents of Assemblies

An assembly consists of the following components:

- *Intermediate language (IL) code.* The compiler translates the source code into IL, which is the set of instructions that the just-in-time (JIT) compiler then translates to CPU-specific code before the application runs.
- *Resources.* These include images and assembly metadata. Assembly metadata exists in the form of the assembly manifest.
- *Type metadata.* Type metadata provides information about available classes, interfaces, methods, and properties, similar to the way that a type library provides information about COM components.
- *The assembly manifest.* This contains assembly metadata and provides information about the assembly such as the title, the description, and version information. The manifest also contains information about links to the other files in the assembly. The information in the manifest is used at run time to resolve references and validate loaded assemblies. The assembly manifest can be stored in a separate file but is often part of one of the PE files.

MCT USE ONLY. STUDENT USE PROHIBITED

## Boundaries of Assemblies

By arranging your code into assemblies, you create a set of boundaries that you can use to isolate configuration to a particular assembly. The following list describes some of the boundaries that assemblies provide:

- *Security boundary.* You set security permissions at an assembly level. You can use these permissions to request specific access for an application, for example, file I/O permissions if the application must write to a disk. When the assembly is loaded at run time, the permissions that are requested are entered into the security policy and used to determine whether permissions can be granted.
- *Type boundary.* An assembly provides a boundary for data types, because each type has the assembly name as part of its identity. As a result, two types can have the same name in different assemblies without any conflict.
- *Reference scope boundary.* An assembly provides a reference scope boundary by using the assembly manifest to resolve type and resource requests. This metadata specifies which types and resources are exposed outside the assembly.

## Benefits of Assemblies

Assemblies provide you with the following benefits:

- *Single units of deployment.* The client application loads assemblies when it needs them, which enables a minimal download strategy where appropriate.
- *Versioning.* An assembly is the smallest unit in a .NET Framework application that you can version. The assembly manifest describes the version information and any version dependencies that are specified for any dependent assemblies. You can only version assemblies that have strong names.



**Additional Reading:** For more information about assemblies, see the Assemblies in the Common Language Runtime page at <http://go.microsoft.com/fwlink/?LinkId=267873>.

## What Is the GAC?

When you create an assembly, by default you create a private assembly that a single application can use. If you need to create an assembly that multiple applications can share, you should give the assembly a strong name and install the assembly into the GAC.

A strong name is a unique name for an assembly that consists of the assembly's name, version number, culture information, and a digital signature that contains a public and private key.

The GAC stores the assemblies that you want to share between multiple applications. When you add an assembly to the GAC, the GAC performs integrity checks on all of the files that form the assembly. These checks ensure that nothing has tampered with an assembly. For example, the GAC checks for changes to a file that the manifest does not reflect.

You can examine the GAC by using File Explorer. Browse to C:\Windows\assembly to see the list of assemblies in the GAC. The information in the list of installed assemblies includes the following:

- The global assembly name

- The GAC provide a robust solution to share assemblies between multiple application on the same machine
- Find the contents of the GAC at C:\Windows\assembly
- Benefits:
  - Side-by-side deployment
  - Improved loading time
  - Reduced memory consumption
  - Improved search time
  - Improved maintainability

- The version number of the assembly
- The culture of the assembly, if applicable
- The public key token of the assembly in the strong name
- The type of assembly

### Benefits of Using the GAC

Although you can install strong-named assemblies in directories on the computer, the GAC offers several benefits, including the following:

- *Side-by-side deployment and execution.* Different versions of an assembly in the GAC do not affect each other. Therefore, applications that reference different versions of an assembly do not fail if a later incompatible version of the assembly is installed into the cache.
- *Improved loading time.* When you install a strong-named assembly in the GAC, it undergoes strong-name validation, which ensures that the digital signature is valid. The verification process occurs at installation time, so strong-named assemblies in the GAC load faster at run time than assemblies that are not installed in the GAC.
- *Reduced memory consumption.* If multiple applications reference an assembly, the operating system loads only one instance of the assembly, which can reduce the total memory that is used on the computer.
- *Improved search time.* The CLR can locate a strong-named assembly in the GAC faster than it can locate a strong-named assembly that is in a directory. This is because the runtime checks the GAC for a referenced assembly before it checks other locations.
- *Improved maintainability.* With a single file that multiple applications share, you can easily make fixes that affect all of the applications.

## Signing Assemblies

When you *sign* an assembly, you give the assembly a *strong name*. A strong name provides an assembly with a globally unique name that applications use when they reference your assembly. This ensures that no one else can compile an assembly with the same name as yours and impersonate your assembly. This helps to avoid malicious code overwriting one of your assemblies and then being run from an application that expects to be using your authentic code.

A strong name requires two cryptographic keys, a public key and a private key, known as a *key pair*. The compiler uses the key pair at build time to create the strong name. The strong name consists of the simple text name of the assembly, the version number, optional culture information, the public key, and a digital signature.

### Creating Key Pairs

You must have a key pair to sign an assembly with a strong name. To create a key pair, use the Strong Name tool (Sn.exe) that the .NET Framework provides. To create a key pair file, perform the following steps:

**Sign an assembly:**

- Create a key file
- Associate the key file with an assembly  
sn -k FourthCoffeeKeyFile.snk

**Delay the signing of an assembly:**

1. Open the properties for the project  
[assembly: AssemblyKeyFileAttribute("FourthCoffeeKeyFile.snk")]
2. Click the **Signing** tab
3. Select the **Sign the assembly** check box
4. Specify a key file
5. Select the **Delay sign only** check box

1. Open the Visual Studio 2012 command prompt.
2. In the **Command Prompt** window, use the Sn.exe tool with the **K** switch to create a new key file. The following code example shows how to create a new key file with the name **FourthCoffeeKeyFile**.

```
sn -k FourthCoffeeKeyFile.snk
```

After you have created a key file, you can then sign your assembly.

## **Signing an Assembly**

When you have created the key pair, you can then associate the key file with your assembly. You can achieve this using the **Signing** tab in the project properties pane. When you specify a key file in the properties pane, Visual Studio adds the **AssemblyKeyFileAttribute** attribute to the **AssemblyInfo** class in your application.

The following code example shows how to associate the FourthCoffeeKeyFile.snk file with your assembly by using the **AssemblyKeyFileAttribute** attribute.

### **The AssemblyKeyFileAttribute attribute**

```
[assembly: AssemblyKeyFileAttribute("FourthCoffeeKeyFile.snk")]
```

## **Delay-Signing an Assembly**

When you sign an assembly, you might not have access to a private key. For example, for security reasons, some organizations restrict access to their private key to just a few individuals. The public key will generally be available because as its name implies, it is publicly accessible. In this situation, you can use delayed signing at build time. You provide the public key and reserve space in the PE file for the strong-name signature. However, you defer the addition of the private key until a later stage, typically just before the assembly ships.

You can enable delay-signing on the **Signing** tab of the project properties window as follows:

1. In Solution Explorer, right-click the project, and then click **Properties**.
2. In the properties window of the project, click the **Signing** tab.
3. Select the **Sign the assembly** check box.
4. Specify a key file.
5. Select the **Delay sign only** check box.

You cannot run or debug a delay-signed project. You can, however, use the Sn.exe tool with the **-Vr** option to skip verification, which means that the identity of the assemblies will not be checked. However, you should only use this option at development time because it creates a security vulnerability.

The following code example turns off verification for an assembly called FourthCoffee.Core.dll.

### **Disabling Verification**

```
sn -Vr FourthCoffee.Core.dll
```

You can then submit the assembly to the signing authority of your organization for the actual strong-name signing. Use the **-R** option with the Sn.exe tool to resign a delay-signed assembly.

The following code example signs an assembly called FourthCoffee.Core.dll with a strong name by using the sgKey.snk key pair.

### **Signing an Assembly**

```
sn -R FourthCoffee.Core.dll sgKey.snk
```

 **Additional Reading:** For more information about delay signing, refer to the Delay Signing an Assembly page at <http://go.microsoft.com/fwlink/?LinkId=267874>.

## **Versioning Assemblies**

All assemblies are given a version number by Visual Studio, which is typically 1.0.0.0. It is the responsibility of the developer to increment the assembly's version number as the assembly evolves.

It is important to version assemblies so that you can keep track of which version of your application users are using. Without a version number, debugging and reproducing production issues can be difficult.

### **Assembly Version Number**

The version number of an assembly is a four-part string with the following format: *<major version>.<minor version>.<build number>.<revision>*. For example, version 1.2.3.0 indicates 1 as the major version, 2 as the minor version, 3 as the build number, and 0 as the revision number.

The assembly manifest stores the version number along with other identity information such as the assembly name and public key. The CLR uses the version number, in conjunction with the available configuration information, to load the proper version of a referenced assembly.

By default, applications only run with the version of an assembly with which they were built. To change an application to use a different version of an assembly, you can create a version policy in one of the configuration files: the application configuration file, the publisher policy file, or the computer's administrator configuration file.

### **Redirecting Binding Requests**

When you want to update a strong-named component without redeploying the client application that uses it, you can use a publisher policy file to redirect a binding request to a newer instance of the component.

When a client application makes a binding request, the runtime performs the following tasks:

- It checks the original assembly reference for the version to be bound.
- It checks the configuration files for version policy instructions.

A version number of an assembly is a four-part string:

```
<major version>.<minor version>.<build number>.<revision>
```

Applications reference particular versions of assemblies

```
<configuration>
 <runtime>
 <assemblyBinding xmlns="...">
 <dependentAssembly>
 <assemblyIdentity name="FourthCoffee.Core"
 publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
 <bindingRedirect oldVersion="1.0.0.0" newVersion="2.0.0.0"/>
 </dependentAssembly>
 </assemblyBinding>
 </runtime>
</configuration>
```

The following code example shows a publisher policy file. To redirect one version to another, use the **<bindingRedirect>** element. The **oldVersion** attribute can specify either a single version or a range of versions. For example, **<bindingRedirect oldVersion="1.1.0.0-1.2.0.0" newVersion="2.0.0.0"/>** specifies that the runtime should use version 2.0.0.0 instead of the assembly versions between 1.1.0.0 and 1.2.0.0.

### Assembly Binding Redirect

```
<configuration>
 <runtime>
 <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
 <dependentAssembly>
 <assemblyIdentity name="FourthCoffee.Core"
 publicKeyToken="32ab4ba45e0a69a1"
 culture="en-us" />
 <bindingRedirect oldVersion="1.0.0.0"
 newVersion="2.0.0.0"/>
 </dependentAssembly>
 </assemblyBinding>
 </runtime>
</configuration>
```



**Additional Reading:** For more information about versioning, refer to the Assembly Versioning page at <http://go.microsoft.com/fwlink/?LinkId=267875>.

## Installing an Assembly into the GAC

The GAC is a folder on file system where you can install your assemblies. You can install your assemblies into the GAC in a variety of ways, which include the following:

- *Global Assembly Cache tool (Gacutil.exe)*. You can use Gacutil.exe to add strong-named assemblies to the GAC and to view the contents of the GAC. Gacutil.exe is for development purposes. You should not use the tool to install production assemblies into the GAC.
- *Microsoft Windows Installer 2.0*. This is the recommended and most common way to add assemblies to the GAC. The installer provides benefits such as reference counting of assemblies in the GAC.

### Install an assembly in the GAC by using:

- Global Assembly Cache tool
- Microsoft Windows Installer

### Examples:

- Install an assembly by using Gacutil.exe:  
gacutil -i "<pathToAssembly>"
- View an assembly by using Gacutil.exe:  
gacutil -l "<assemblyName>"

### Installing an Assembly into the GAC by Using Gacutil.exe

To install an assembly into the GAC by using the Gacutil.exe command-line tool, perform the following steps:

1. Open the Visual Studio 2012 command prompt as an administrator.
2. In the **Command Prompt** window, type the following command:

```
gacutil -i "<pathToAssembly>"
```

## Viewing an Assembly in the GAC by Using Gacutil.exe

To view an assembly that is installed into the GAC by using the Gacutil.exe command-line tool, perform the following steps:

1. Open the Visual Studio 2012 command prompt as an administrator.
2. In the **Command Prompt** window, type the following command:

```
gacutil -l "<assemblyName>"
```



**Additional Reading:** For more information about the GAC, refer to the Global Assembly Cache page at <http://go.microsoft.com/fwlink/?LinkId=267876>.

## Demonstration: Signing and Installing an Assembly into the GAC

In this demonstration, you will use the Sn.exe and Gacutil.exe command-line tools to sign and install an existing assembly into the GAC.

### Demonstration Steps

You will find the steps in the **Demonstration: Signing and Installing an Assembly into the GAC** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md).

## Demonstration: Specifying the Data to Include in the Grades Report Lab

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

### Demonstration Steps

You will find the steps in the **Demonstration: Specifying the Data to Include in the Grades Report Lab** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_DEMO.md).

# Lab: Specifying the Data to Include in the Grades Report

## Scenario

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reusability of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

## Objectives

After completing this lab, you will be able to:

- Define custom attributes.
- Use reflection to examine metadata at run time.
- Sign an assembly and deploy it to the GAC.

## Lab Setup

Estimated Time: **75 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD12\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD12_LAK.md).

## Exercise 1: Creating and Applying the `IncludeInReport` attribute

### Scenario

In this exercise, you will create the **IncludeInReport** attribute to specify the fields and the format of each field that is included in the grades report.

First, you will write code for the **IncludeInReportAttribute** class and define the members that are contained in it. Next, you will apply the attribute to the appropriate properties in the **LocalGrade** class in the Data.cs file. Finally, you will build the application and then use the MSIL Disassembler utility (IL DASM) to examine the metadata that the attribute generates.

## Exercise 2: Updating the Report

### Scenario

In this exercise, you will update the grades report to include fields and properties only if they are tagged with the **IncludeInReport** attribute.

First, you will implement a method named **GetItemsToInclude** in a static helper class called **IncludeProcessor** that implements the logic that is necessary to discover the fields and properties that are tagged with the **IncludeInReport** attribute. You will then update the code for the **StudentProfile** view to include fields and properties in the report only if they are tagged with the **IncludeInReport** attribute and to format them appropriately.

## Exercise 3: Storing the Grades.Utilities Assembly Centrally (If Time Permits)

### Scenario

In this exercise, you will store the **Grades.Utilities** assembly in the GAC.

First, you will use Sn.exe to generate a key pair and then use the key pair to sign the **Grades.Utilities** assembly. Next, you will use Gacutil.exe to add the assembly to the GAC. You will then update the reference for the **Grades.Utilities** assembly in the Grades.WPF project to use the new signed assembly that is hosted in the GAC, and finally you will test the application to ensure that the reports are correctly generated.

## Module Review and Takeaways

In this module, you learned how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. You also learned how to generate code at run time by using CodeDOM and how you can ensure that your assemblies are versioned and available to other applications by using the GAC.

### Review Questions

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

Statement	Answer
You are developing an application that enables users to browse the object model of a compiled type. At no point will the application attempt to execute any code; it will merely serve as a viewer. You notice the code that loads the assembly uses the <b>Assembly.LoadFrom</b> static method. This is the most suitable method taking into account the requirements of the application.	

### Check Your Knowledge

Question	
You are developing a custom attribute. You want to derive your custom attribute class from the abstract base class that underpins all attributes. Which class should you use?	
Select the correct answer.	
	Attribute
	ContextAttribute
	ExtensionAttribute
	DataAttribute
	AddInAttribute

MCT USE ONLY. STUDENT USE PROHIBITED

## Check Your Knowledge

Question	
You are reviewing some code that uses CodeDOM to generate managed Visual C# at run time. What does the following line of code do?	
<pre>var method = new CodeEntryPointMethod();</pre>	
Select the correct answer.	
<input type="checkbox"/>	Defines an instance method with a random name.
<input type="checkbox"/>	Defines an instance method named EntryPoint.
<input type="checkbox"/>	Defines a static method named EntryPoint.
<input type="checkbox"/>	Defines an instance method named Main.
<input type="checkbox"/>	Defines a static method named Main.

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

Statement	Answer
The <b>FourthCoffee.Core.dll</b> assembly has 2.1.0.24 as its version number. The number 24 in the version number refers to the build number.	<input type="checkbox"/>

MCT USE ONLY. STUDENT USE PROHIBITED

# Module 13

## Encrypting and Decrypting Data

### Contents:

Module Overview	13-1
Lesson 1: Implementing Symmetric Encryption	13-2
Lesson 2: Implementing Asymmetric Encryption	13-8
Lab: Encrypting and Decrypting the Grades Report	13-16
Module Review and Takeaways	13-18

## Module Overview

It is a common requirement for applications to be able to secure information, whether it is a case of encrypting files saved to disk or web requests sent over an untrusted connection to other remote systems. The Microsoft® .NET Framework provides a variety of classes that enable you to secure your data by using encryption and hashing.

In this module, you will learn how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of your data. You will also learn how to create and manage X509 certificates and how to use them in the asymmetric encryption process.

### Objectives

After completing this module, you will be able to:

- Encrypt data by using symmetric encryption.
- Encrypt data by using asymmetric encryption.

## Lesson 1

# Implementing Symmetric Encryption

Symmetric encryption is the process of performing a cryptographic transformation of data by using a mathematical algorithm. Symmetric encryption is an established technique and is used by many applications to provide a robust way of protecting confidential data.

In this lesson, you will learn about several .NET Framework classes that enable applications to secure data by means of encryption and hashing.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe symmetric encryption.
- Encrypt and decrypt data by using symmetric encryption.
- Create digital fingerprints of data by using hashes.

### What Is Symmetric Encryption?

The name symmetric is derived from the fact that the same secret key is used to encrypt and decrypt the data. Therefore, when you use symmetric encryption, you must keep the secret key secure.

To help improve the effectiveness of symmetric encryption, many symmetric encryption algorithms also use an initialization vector (IV) in addition to a secret key. The IV is an arbitrary block of bytes that helps to randomize the first encrypted block of data. The IV makes it much more difficult for a malicious user to decrypt your data.

- Symmetric encryption is the cryptographic transformation of data by using a mathematical algorithm
- The same key is used to encrypt and decrypt the data
- The **System.Security.Cryptography** namespace includes:
  - **DESCryptoServiceProvider** class
  - **AesManaged** class
  - **RC2CryptoServiceProvider** class
  - **RijndaelManaged** class
  - **TripleDESCryptoServiceProvider** class

### Advantages and Disadvantages of Symmetric Encryption

The following table describes some of the advantages and disadvantages of symmetric encryption.

Advantage	Disadvantage
There is no limit on the amount of data you can encrypt.	The same key is used to encrypt and decrypt the data. If the key is compromised, anyone can encrypt and decrypt the data.
Symmetric algorithms are fast and consume far fewer system resources than asymmetric algorithms.	If you choose to use a different secret key for different data, you could end up with many different secret keys that you need to manage.

Symmetric algorithms are perfect for quickly encrypting large amounts of data.

## Symmetric Encryption Classes in the .NET Framework

The .NET Framework contains a number of classes in the **System.Security.Cryptography** namespace, which provide managed implementations of common symmetric encryption algorithms, such as Advanced Encryption Standard (AES), Data Encryption Standard (DES), and TripleDES. Each .NET Framework symmetric encryption class is derived from the abstract **SymmetricAlgorithm** base class.

The following table describes the key characteristics of the .NET Framework encryption classes.

Algorithm	.NET Framework Class	Encryption Technique	Block Size	Key Size
DES	<b>DESCryptoServiceProvider</b>	Bit shifting and bit substitution	64 bits	64 bits
AES	<b>AesManaged</b>	Substitution-Permutation Network (SPN)	128 bits	128, 192, or 256 bits
Rivest Cipher 2 (RC2)	<b>RC2CryptoServiceProvider</b>	Feistel network	64 bit	40-128 bits (increments of 8 bits)
Rijndael	<b>RijndaelManaged</b>	SPN	128-256 bits (increments of 32 bits)	128, 192, or 256 bits
TripleDES	<b>TripleDESCryptoServiceProvider</b>	Bit shifting and bit substitution	64 bit	128-192 bits

Each of the .NET Framework encryption classes are known as block ciphers, which means that the algorithm will chunk data into fixed-length blocks and then perform a cryptographic transformation on each block.



**Note:** You can measure the strength of an encryption algorithm by the key size. The higher the number of bits, the more difficult it is for a malicious user trying a large number of possible secret keys to decrypt your data.



**Additional Reading:** For more information about **symmetric encryption in the .NET Framework**, refer to the SymmetricAlgorithm Class **page** at <https://aka.ms/moc-20483c-m13-pg1>.

## Encrypting Data by Using Symmetric Encryption

You can encrypt data by using any of the symmetric encryption classes in the **System.Security.Cryptography** namespace. However, these classes only provide managed implementations of a particular encryption algorithm; for example, the **AesManaged** class provides a managed implementation of the AES algorithm. Aside from encrypting and decrypting data by using an algorithm, the encryption process typically involves the following tasks:

- Derive a secret key and an IV from a password or salt. A salt is a random collection of bits used in combination with a password to generate a secret key and an IV. A salt makes it much more difficult for a malicious user to randomly discover the secret key.
- Read and write encrypted data to and from a stream.

To encrypt and decrypt data symmetrically, perform the following steps:

1. Create an **Rfc2898DeriveBytes** object
2. Create an **AesManaged** object
3. Generate a secret key and an IV
4. Create a stream to buffer the transformed data
5. Create a symmetric encryptor or decryptor object
6. Create a **CryptoStream** object
7. Write the transformed data to the buffer stream
8. Close the streams

To help simplify the encryption logic in your applications, the .NET Framework includes a number of other cryptography classes that you can use.

### The **Rfc2898DeriveBytes** and **CryptoStream** Classes

The **Rfc2898DeriveBytes** class provides an implementation of the password-based key derivation function (PBKDF2), which complies with the Public-Key Cryptography Standards (PKCS). You can use the PBKDF2 functionality to derive your secret keys and your IVs from a password and a salt.

 **Additional Reading:** For more information about the **Rfc2898DeriveBytes** class, refer to the Rfc2898DeriveBytes Class [page](https://aka.ms/moc-20483c-m13-pg2) at <https://aka.ms/moc-20483c-m13-pg2>.

The **CryptoStream** class is derived from the abstract **Stream** base class in the **System.IO** namespace, and it provides streaming functionality specific to reading and writing cryptographic transformations.

 **Additional Reading:** For more information about the **CryptoStream** class, refer to the CryptoStream Class [page](https://aka.ms/moc-20483c-m13-pg3) at <https://aka.ms/moc-20483c-m13-pg3>.

### Symmetrically Encrypting and Decrypting Data

The following steps describe how to encrypt and decrypt data by using the **AesManaged** class:

1. Create an **Rfc2898DeriveBytes** object, which you will use to derive the secret key and the IV. The following code example shows how to create an instance of the **Rfc2898DeriveBytes** class, passing values for the password and salt into the constructor.

```
var password = "Pa$$w0rd";
var salt = "S@lt";
var rgb = new Rfc2898DeriveBytes(password, Encoding.Unicode.GetBytes(salt));
```

2. Create an instance of the encryption class that you want to use to encrypt the data. The following code example shows how to create an **AesManaged** object.

```
var algorithm = new AesManaged();
```

3. Generate the secret key and the IV from the **Rfc2898DeriveBytes** object. The following code example shows how to generate the secret key and the IV by using the algorithm's **KeySize** and **BlockSize** properties.

```
var rgbKey = rgb.GetBytes(algorithm.KeySize / 8);
var rgbIV = rgb.GetBytes(algorithm.BlockSize / 8);
```

 **Note:** You typically use the algorithm's **KeySize** and **BlockSize** properties when generating the secret key and the IV, so that the secret key and the IV that you generate are compatible with the algorithm.

4. Create a stream object that you will use to buffer the encrypted or unencrypted bytes. The following code example shows how to create an instance of the **MemoryStream** class.

```
var bufferStream = new MemoryStream();
```

5. Create either a symmetric encryptor or decryptor depending on whether you want to encrypt or decrypt data. The following code example shows how to invoke the **CreateEncryptor** method to create an encryptor and how to invoke the **CreateDecryptor** method to create a decryptor. Both methods accept the secret key and the IV as parameters.

```
// Create an encryptor object.
var algorithm = algorithm.CreateEncryptor(rgbKey, rgbIV);
...
// Create a decryptor object.
var algorithm = algorithm.CreateDecryptor(rgbKey, rgbIV);
```

6. Create a **CryptoStream** object, which you will use to write the cryptographic bytes to the buffer stream. The following code example shows how to create an instance of the **CryptoStream** class, passing the **bufferStream** object, the **algorithm** object, and the stream mode as parameters.

```
var cryptoStream = new CryptoStream(
 bufferStream,
 algorithm,
 CryptoStreamMode.Write)
```

7. Invoke the **Write** and **FlushFinalBlock** methods on the **CryptoStream** object, to perform the cryptographic transform. The following code example shows how to invoke the **Write** and **FlushFinalBlock** methods of the **CryptoStream** object.

```
var bytesToTransform = FourthCoffeeDataService.GetBytes();
cryptoStream.Write(bytesToTransform, 0, bytesToTransform.Length);
cryptoStream.FlushFinalBlock();
```

8. Invoke the **Close** method on the **CryptoStream** and the **MemoryStream** objects, so that the transformed data is flushed to the buffer stream. The following code example shows how to invoke the **Close** methods on both the **CryptoStream** and the **MemoryStream** objects.

```
cryptoStream.Close();
bufferStream.Close();
```

## Hashing Data

Hashing is the process of generating a numerical representation of your data. Typically, hash algorithms compute hashes by mapping the binary representation of your data to the binary values of a fixed-length hash. If you use a proven hash algorithm, it is considered unlikely that you could compute the same hash from two different pieces of data. Therefore, hashes are considered a reliable way to generate a unique digital fingerprint that can help to ensure the integrity of data.

- A hash is a numerical representation of a piece of data
- A hash can be computed by using the following code

```
public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
 using (var hashAlgorithm = new HMACSHA1(secretKey))
 {
 using (var bufferStream = new MemoryStream(dataToHash))
 {
 return hashAlgorithm.ComputeHash(bufferStream);
 }
 }
}
```

Consider the example of the

FourthCoffee.Beverage service, which sends messages to the FourthCoffee.Inventory service. When the FourthCoffee.Inventory service receives a message, how do the two services know that the message was not sabotaged during the transmission? You could use hashes, as the following steps describe:

1. Compute a hash of the message before the FourthCoffee.Beverage service sends the message.
2. Compute a hash of the message when the FourthCoffee.Inventory service receives the message.
3. Compare the two hashes. If the two hashes are identical, the data has not been tampered with. If the data has been modified, the two hashes will not match.

The .NET Framework provides a number of classes in the **System.Security.Cryptography** namespace, which encapsulate common hash algorithms.

### Hash Algorithms in the .NET Framework

The following table describes some of the hash classes that the .NET Framework provides.

.NET Framework Class	Description
<b>SHA512Managed</b>	The <b>SHA512Managed</b> class is an implementation of the Secure Hash Algorithm (SHA) and is able to compute a 512-bit hash. The .NET Framework also includes classes that implement the SHA1, SHA256, and SHA384 algorithms.
<b>HMACSHA512</b>	The <b>HMACSHA512</b> class uses a combination of the SHA512 hash algorithm and the Hash-Based Message Authentication Code (HMAC) to compute a 512-bit hash.
<b>MACTripleDES</b>	The <b>MACTripleDES</b> class uses a combination of the TripleDES encryption algorithm and a Message Authentication Code (MAC) to compute a 64-bit hash.
<b>MD5CryptoServiceProvider</b>	The <b>MD5CryptoServiceProvider</b> class is an implementation of the Message Digest (MD) algorithm, which uses block chaining to compute a 128-bit hash.
<b>RIPEMD160Managed</b>	The <b>RIPEMD160Managed</b> class is derived from the MD algorithm and is able to compute a 160-bit hash.

## Computing a Hash by Using the HMACSHA512 Class

To compute a hash by using the **HMACSHA512** class, perform the following steps:

1. Generate a secret key that the hash algorithm will use to hash the data. The sender would need access to the key to generate the hash, and the receiver would need access to the key to verify the hash.
2. Create an instance of the hash algorithm.
3. Invoke the **ComputeHash** method, passing in a stream that contains the data you want to hash. The **ComputeHash** method returns a byte array that represents the hash of your data.

The following code example shows how to compute a hash by using the **HMACSHA512** class.

### Hashing Data by Using the HMACSHA512 class

```
public byte[] ComputeHash(byte[] dataToHash, byte[] secretKey)
{
 using (var hashAlgorithm = new HMACSHA1(secretKey))
 {
 using (var bufferStream = new MemoryStream(dataToHash))
 {
 return hashAlgorithm.ComputeHash(bufferStream);
 }
 }
}
```

 **Additional Reading:** For more information about **hashing in the .NET Framework**, refer to the Hash Values section on the Cryptographic Services [page](#) at <http://go.microsoft.com/fwlink/?LinkId=267880>.

## Demonstration: Encrypting and Decrypting Data

In this demonstration, you will use symmetric encryption to encrypt and decrypt a message.

### Demonstration Steps

You will find the steps in the **Demonstration: Encrypting and Decrypting Data** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD13\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_DEMO.md).

## Lesson 2

# Implementing Asymmetric Encryption

Asymmetric encryption is the process of performing a cryptographic transformation of data by using an asymmetric encryption algorithm and a combination of public and private keys.

In this lesson, you will learn about the classes and tools that you can use to implement asymmetric encryption in your applications.

### Lesson Objectives

After completing this lesson, you will be able to:

- Describe asymmetric encryption.
- Encrypt and decrypt data by using asymmetric encryption.
- Create and manage X509 certificates.
- Manage encryption keys in your applications.

### What Is Asymmetric Encryption?

Unlike symmetric encryption, where one secret key is used to perform both the encryption and the decryption, asymmetric encryption uses a public key to perform the encryption and a private key to perform the decryption.

 **Note:** The public and private keys are mathematically linked, in that the private key is used to derive the public key. However, you cannot derive a private key from a public key. Also, you can only decrypt data by using the private key that is linked to the public key that was used to encrypt the data.

- Asymmetric encryption uses:
  - A public key to encrypt data
  - A private key to decrypt data
- The **System.Security.Cryptography** namespace includes:
  - The **RSACryptoServiceProvider** class
  - The **DSACryptoServiceProvider** class

In a system that uses asymmetric encryption, the public key is made available to any application that requires the ability to encrypt data. However, the private key is kept safe and is only distributed to applications that require the ability to decrypt the data. For example, HTTPS uses asymmetric encryption to encrypt and decrypt the browser's session key when establishing a secure connection between the browser and the server.

 **Note:** You can also use asymmetric algorithms to sign data. Signing is the process of generating a digital signature so that you can ensure the integrity of the data. When signing data, you use the private key to perform the signing and then use the public key to verify the data.

### Advantages and Disadvantages of Asymmetric Encryption

The following table describes some of the advantages and disadvantages of asymmetric encryption.

Advantage	Disadvantage
Asymmetric encryption relies on two keys, so it is easier to distribute the keys and to enforce who can encrypt and decrypt the data.	With asymmetric encryption, there is a limit on the amount of data that you can encrypt. The limit is different for each algorithm and is typically proportional with the key size of the algorithm. For example, an <b>RSACryptoServiceProvider</b> object with a key length of 1,024 bits can only encrypt a message that is smaller than 128 bytes.
Asymmetric algorithms use larger keys than symmetric algorithms, and they are therefore less susceptible to being cracked by using brute force attacks.	Asymmetric algorithms are very slow in comparison to symmetric algorithms.

Asymmetric encryption is a powerful encryption technique, but it is not designed for encrypting large amounts of data. If you want to encrypt large amounts of data with asymmetric encryption, you should consider using a combination of asymmetric and symmetric encryption.

 **Best Practice:** To encrypt data by using asymmetric and symmetric encryption, perform the following steps:

1. Encrypt the data by using a symmetric algorithm, such as the **AesManaged** class.
2. Encrypt the symmetric secret key by using an asymmetric algorithm.
3. Create a stream and write bytes for the following:
  - o The length of the IV
  - o The length of the encrypted secret key
  - o The IV
  - o The encrypted secret key
  - o The encrypted data

To decrypt, simply step through the stream extracting the data, decrypt the symmetric encryption key, and then decrypt the data.

### Asymmetric Encryption Classes in the .NET Framework

The .NET Framework contains a number of classes in the **System.Security.Cryptography** namespace, which enable you to implement asymmetric encryption and signing. Each .NET Framework asymmetric class is derived from the **AsymmetricAlgorithm** base class.

The following list describes some of these classes:

- **RSACryptoServiceProvider**. This class provides an implementation of the RSA algorithm, which is named after its creators, Ron Rivest, Adi Shamir, and Leonard Adleman. By default, the **RSACryptoServiceProvider** class supports key lengths ranging from 384 to 512 bits in 8-bit increments, but optionally, if you have the Microsoft Enhanced Cryptographic Provider installed, the **RSACryptoServiceProvider** class will support keys up to 16,384 bits in length. You can use the **RSACryptoServiceProvider** class to perform both encryption and signing.
- **DSACryptoServiceProvider**. This class provides an implementation of the Digital Signature Algorithm (DSA) algorithm and supports keys ranging from 512 to 1,024 bits in 64-bit increments.

Although the **RSACryptoServiceProvider** class supports both encryption and signing, the **DSCryptoServiceProvider** class only supports signing.



**Additional Reading:** For more information about **asymmetric encryption in the .NET Framework**, refer to the Public-Key Encryption section on the Cryptographic Services [page](#) at <http://go.microsoft.com/fwlink/?LinkId=267881>.

## Encrypting Data by Using Asymmetric Encryption

You can encrypt your data asymmetrically by using the **RSACryptoServiceProvider** class in the **System.Security.Cryptography** namespace.

### Encrypting Data by Using the RSACryptoServiceProvider Class

The **RSACryptoServiceProvider** class provides a number of members that enable you to implement asymmetric encryption functionality in your applications, including the ability to import and export key information and encrypt and decrypt data.

To encrypt and decrypt data asymmetrically

```
var rawBytes = Encoding.Default.GetBytes("hello world..");
var decryptedText = string.Empty;

using (var rsaProvider = new RSACryptoServiceProvider())
{
 var useOaepPadding = true;

 var encryptedBytes =
 rsaProvider.Encrypt(rawBytes, useOaepPadding);

 var decryptedBytes =
 rsaProvider.Decrypt(encryptedBytes, useOaepPadding);

 decryptedText = Encoding.Default.GetString(decryptedBytes);
}
// decryptedText == hello world..
```

You can create an instance of the **RSACryptoServiceProvider** class by using the default constructor. If you choose this approach, the **RSACryptoServiceProvider** class will generate a set of public and private keys.

The following code example shows how to create an instance of the **RSACryptoServiceProvider** class by using the default constructor.

#### Instantiating the RSACryptoServiceProvider Class

```
var rsaProvider = new RSACryptoServiceProvider();
```

After you have created an instance of the **RSACryptoServiceProvider** class, you can then use the **Encrypt** and **Decrypt** methods to protect your data.

The following code example shows how you can use the **Encrypt** and **Decrypt** methods to protect the contents of a string variable.

#### Encrypting and Decrypting Data by Using the RSACryptoServiceProvider Class

```
var plainText = "hello world..";
var rawBytes = Encoding.Default.GetBytes(plainText);
var decryptedText = string.Empty;
using (var rsaProvider = new RSACryptoServiceProvider())
{
 var useOaepPadding = true;
 var encryptedBytes = rsaProvider.Encrypt(rawBytes, useOaepPadding);
 var decryptedBytes = rsaProvider.Decrypt(encryptedBytes, useOaepPadding);
 decryptedText = Encoding.Default.GetString(decryptedBytes);
}
// The decryptedText variable will now contain " hello world..."
```



**Note:** You use the `useOaepPadding` parameter to determine whether the **Encrypt** and **Decrypt** methods use Optimal Asymmetric Encryption Padding (OAEP). If you pass **true**, the methods use OAEP, and if you pass **false**, the methods use PKCS#1 v1.5 padding.

Typically, applications do not encrypt and decrypt data in the scope of the same **RSACryptoServiceProvider** object. One application may perform the encryption, and then another performs the decryption. If you attempt to use different **RSACryptoServiceProvider** objects to perform the encryption and decryption, without sharing the keys, the **Decrypt** method will throw a **CryptographicException** exception. The **RSACryptoServiceProvider** class exposes members that enable you to export and import the public and private keys.

The following code example shows how to instantiate different **RSACryptoServiceProvider** objects and use the **ExportCspBlob** and **ImportCspBlob** methods to share the public and private keys.

### Importing and Exporting Keys

```
var keys = default(byte[]);
var exportPrivateKey = true;
using (var rsaProvider = new RSACryptoServiceProvider())
{
 keys = rsaProvider.ExportCspBlob(exportPrivateKey);
 // Code to perform encryption.
}

var decryptedText = string.Empty;
using (var rsaProvider = new RSACryptoServiceProvider())
{
 rsaProvider.ImportCspBlob(keys);
 // Code to perform decryption.
}
```



**Note:** The `exportPrivateKey` parameter instructs the **ExportCspBlob** method to include the private key in the return value. If you pass **false** into the **ExportCspBlob** method, the return value will not contain the private key. If you try to decrypt data without a private key, the Common Language Runtime (CLR) will throw a **CryptographicException** exception.

Instead of maintaining and persisting keys in your application, you can use the public and private keys in an X509 certificate, stored in the certificate store on the computer that is running your application.



**Additional Reading:** For more information about the **RSACryptoServiceProvider** class, refer to the **RSACryptoServiceProvider Class page** at <https://aka.ms/moc-20483c-m13-pg4>.

## Creating and Managing X509 Certificates

An X509 certificate is a digital document that contains information, such as the name of the organization that is supplying the data. X509 certificates are normally stored in certificate stores. In a typical Windows installation, there are user account, service account, and local computer machine certificate stores.

X509 certificates can also contain public and private keys, which you can use in the asymmetric encryption process. You can create and manage your X509 certificates by using the tools that Windows and the .NET Framework provide.

- Use MakeCert to create certificates

```
makecert -n "CN=FourthCoffee" -a sha1 -pe -r -sr LocalMachine -ss my -sky exchange
```

- Use the MMC Certificates snap-in to manage your certificate stores

### Creating a Certificate by Using MakeCert

MakeCert is a certificate creation tool that the .NET Framework provides. You can access the tool by using the Visual Studio command prompt. The MakeCert tool provides a number of command-line switches that enable you to configure the X509 certificate to meet the requirements of your application.

The following table describes some of the MakeCert command-line switches.

Switch	Description
<b>-n</b>	This enables you to specify the name of the certificate.
<b>-a</b>	This enables you to specify the algorithm that the certificate uses.
<b>-pe</b>	This enables you to create a certificate that allows you to export the private key.
<b>-r</b>	This enables you to create a self-signed certificate.
<b>-sr</b>	This enables you to specify the name of the certificate store where the MakeCert tool will import the generated certificate.
<b>-ss</b>	This enables you to specify the name of the container in the certificate store where the MakeCert tool will import the generated certificate.
<b>-sky</b>	This enables you to specify the type of key that the certificate will contain.

The following code example shows how you can use the MakeCert command-line tool to generate a self-signed certificate, which contains both a public and a private key.

#### MakeCert Example

```
makecert -n "CN=FourthCoffee" -a sha1 -pe -r -sr LocalMachine -ss my -sky exchange
```



**Additional Reading:** For more information about **MakeCert**, refer to the [Makecert.exe \(Certificate Creation Tool\) page](https://aka.ms/moc-20483c-m13-pg5) at <https://aka.ms/moc-20483c-m13-pg5>

## Managing X509 Certificates by Using the Microsoft Management Console Certificates Snap-in

The Microsoft Management Console (MMC) Certificates snap-in enables you to manage any X509 certificates installed in the context of your user account, service account, or local computer.

To open a certificate store by using the MMC snap-in, perform the following steps:

1. Log on as an administrator. If you log on without administrative privileges, you will only be able to view certificates in your user account certificate store.
2. In the Windows 8 **Start** window, use search to find mmc.exe, and then click **mmc.exe**.
3. In the MMC window, on the **File** menu, click **Add/Remove Snap-in**.
4. In the **Add or Remove Snap-ins** dialog box, in the **Available snap-ins** list, click **Certificates**, and then click **Add**.
5. In the **Certificates snap-in** dialog box, click either **My user account**, **Service account**, or **Computer account**, and then perform one of the following steps:
  - a. If you chose **My user account**, click **Finish**.
  - b. If you chose **Service account**, click **Next**, and then perform the following steps:
    - i. In the **Select Computer** dialog box, click **Next**.
    - ii. In the **Certificates snap-in** dialog box, in the **Service account** list, click the service account you want to manage, and then click **Finish**.
  - c. If you chose **Computer account**, click **Next**, and then click **Finish**.
6. In the **Add or Remove Snap-ins** dialog box, repeat steps 4 and 5 if you want to add additional certificate stores to your session, and then click **OK**.

After you have opened one or more certificate stores, you can perform any of the following tasks:

- View the properties that are associated with any X509 certificate in any of the certificate stores, such as Personal or Trusted Root Certificate Authorities stores.
- Export an X509 certificate from a certificate store to the file system.
- Manage the private keys that are associated with an X509 certificate.
- Issue a request to renew an existing X509 certificate.



**Additional Reading:** For more information about managing certificates, refer to the **Working with Certificates page** at <http://go.microsoft.com/fwlink/?LinkId=267884>.

## Managing Encryption Keys

The .NET Framework provides the **System.Security.Cryptography.X509Certificate**s namespace, which contains a number of classes that enable you to use X509 certificates and their keys in your applications. These classes include the following:

- **X509Store**. This class enables you to access a certificate store and perform operations, such as finding an X509 certificate with a particular name.
- **X509Certificate2**. This class enables you create an in-memory representation of an X509 certificate that currently exists in a certificate store. When you have instantiated an **X509Certificate2** object, you can then use its members to access information, such as the X509 certificate's public and private keys.
- **PublicKey**. This class enables you to manipulate the various pieces of metadata that are associated with an X509 certificate's public key, such as the public key's value.

The following code example shows how to use the **X509Store** and **X509Certificate2** classes to enumerate the personal certificate store on the local machine.

### Enumerating a Certificate Store

```
var store = new X509Store(StoreName.My, StoreLocation.LocalMachine);
var certificate = default(X509Certificate2);
var certificateName = "CN=FourthCoffee";

store.Open(OpenFlags.ReadOnly);

foreach (var storeCertificate in store.Certificates)
{
 if (storeCertificate.SubjectName.Name == certificateName)
 {
 certificate = storeCertificate;
 continue;
 }
}

store.Close();
```

After you have created an **X509Certificate2** object, you can use its members to determine whether the X509 certificate contains either a public or private key. The following list describes some of the members you can use:

- **HasPrivateKey**. This property enables you to determine whether the X509 certificate contains a private key.
- **FriendlyName**. This property enables you to get the friendly name that is associated with the X509 certificate.
- **GetPublicKeyString**. This method enables you to extract the public key that the X509 certificate contains as a string value.
- **PublicKey**. This property enables you to get the public key that the X509 certificate contains as a **PublicKey** object.

The **System.Security.Cryptography.X509Certificates** namespace contains classes that enable access to the certificate store and certificate metadata

```
var store = new X509Store(
 StoreName.My,
 StoreLocation.LocalMachine);

store.Open(OpenFlags.ReadOnly);

foreach (var storeCertificate in store.Certificates)
{
 // Code to process each certificate.
}

store.Close();
```

- **PrivateKey**. This property enables you to get the private key that the X509 certificate contains as an **AsymmetricAlgorithm** object.

You can use the **PublicKey** and **PrivateKey** properties of the **X509Certificate2** class to create an instance of the **RSACryptoServiceProvider** class.

The following code example shows how you can use the **PublicKey** and **PrivateKey** properties to create an instance of the **RSACryptoServiceProvider** class..

#### Instantiating the RSACryptoServiceProvider Class

```
var certificate = new X509Certificate2();
// Code to set the public and private keys.

// Create an RSA encryptor.
var rsaEncryptorProvider = (RSACryptoServiceProvider)certificate.PublicKey.Key;

// Create an RSA decryptor.
var rsaDecryptorProvider = (RSACryptoServiceProvider)certificate.PrivateKey;
```

 **Additional Reading:** For more information about managing encryption keys in your application, refer to the **System.Security.Cryptography.X509Certificates Namespace page** at <https://aka.ms/moc-20483c-m13-pg6>

### Demonstration: Encrypting and Decrypting Grade Reports Lab

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: Encrypting and Decrypting Grade Reports Lab** section on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD13\\_DEMO.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_DEMO.md).

# Lab: Encrypting and Decrypting the Grades Report

## Scenario

You have been asked to update the Grades application to ensure that reports are secure when they are stored on a user's computer. You decide to use asymmetric encryption to protect the report as it is generated, before it is written to disk. Administrative staff will need to merge reports for each class into one document, so you decide to develop a separate application that generates a combined report and prints it.

## Objectives

After completing this lab, you will be able to:

- Encrypt data by using asymmetric encryption.
- Decrypt data.

## Lab Setup

Estimated Time: **60 minutes**

You will find the high-level steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD13\\_LAB\\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_LAB_MANUAL.md).

You will find the detailed steps on the following page: [https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C\\_MOD13\\_LAK.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD13_LAK.md).

## Exercise 1: Encrypting the Grades Report

### Scenario

In this exercise, you will update the reporting functionality to encrypt the report as it is generated, but before it is saved to disk.

First, you will create an asymmetric certificate by using a prewritten batch file. The batch file uses the MakeCert tool that ships with the Windows Software Development Kit (SDK). You will create a self-signed certificate named Grades using the SHA-1 hash algorithm and store it in the LocalMachine certificate store. You will then write code in the Grades application to retrieve the certificate by looping through the certificates in the LocalMachine store and checking the name of the certificate against the name that is stored in the App.Config file. Next, you will use the classes that are provided in the

**System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces to write the **EncryptWithX509** method in the **Grades.Utilities.WordWrapper** class. You will get the public key from the certificate that you created and use it to create an instance of the

**RSAPKCS1KeyExchangeFormatter** class. You will use this to encrypt the data for the report and then return the encrypted buffered data to the calling method as a byte array. You will then write code in the **EncryptAndSaveToDisk** method to write the returned data to the file that the user specifies. Finally, you will build and test the application and verify that the reports are now encrypted.

## Exercise 2: Decrypting the Grades Report

### Scenario

In this exercise, you will create a separate utility to enable users to print reports. Users will be able to select a folder that contains encrypted reports, and the application will then generate one combined report and send it to the default printer.

First, you will use the classes that are provided in the **System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces to write the **DecryptWithX509** method in the **SchoolReports.WordWrapper** class. You will get the private key from the certificate and use it to create an instance of the **RSACryptoServiceProvider** class. You will use this to decrypt the data from the individual reports and then return the decrypted data to the calling method as a byte array. Finally, you will build and test the application and verify that a printed version of the composite report has been generated.

## Module Review and Takeaways

In this module, you learned how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of your data.

### Review Questions

#### Check Your Knowledge

Question
Fourth Coffee wants you to implement an encryption utility that can encrypt and decrypt large image files. Each image will be more than 200 megabytes (MB) in size. Fourth Coffee envisages that only a small internal team will use this tool, so controlling who can encrypt and decrypt the data is not a concern. Which of the following techniques will you choose?
Select the correct answer.
<input type="checkbox"/> Symmetric encryption
<input type="checkbox"/> Asymmetric encryption
<input type="checkbox"/> Hashing

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

Statement	Answer
Is the following statement true or false? Asymmetric encryption uses a public key to encrypt data.	

## 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.

MCT USE ONLY. STUDENT USE PROHIBITED

---

## **Notes**

---

## **Notes**