Microsoft®

MCT USE ONLY. STUDENT USE PROHIBITED

OFFICIAL MICROSOFT LEARNING PRODUCT

10266A

Programming in C# with Microsoft® Visual Studio® 2010



Be sure to access the extended learning content on your Course Companion CD enclosed on the back cover of the book.

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.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are property of their respective owners.

Download from www.eBookTM.com

Product Number: 10266A

Part Number: 01918

Released: 09/2010

MCT USE ONLY. STUDENT USE PROHIBITED

MICROSOFT LICENSE TERMS
OFFICIAL MICROSOFT LEARNING PRODUCTS - TRAINER EDITION – Pre-Release and Final Release Versions

MCT USE ONLY. STUDENT USE PROHIBITED

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the Licensed Content named above, which includes the media on which you received it, if any. The terms also apply to any Microsoft

- updates,
- supplements,
- Internet-based services, and
- support services

for this Licensed Content, unless other terms accompany those items. If so, those terms apply.

By using the Licensed Content, you accept these terms. If you do not accept them, do not use the Licensed Content.

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

1. **DEFINITIONS.**

   a. **"Academic Materials"** means the printed or electronic documentation such as manuals, workbooks, white papers, press releases, datasheets, and FAQs which may be included in the Licensed Content.

   b. **"Authorized Learning Center(s)"** means a Microsoft Certified Partner for Learning Solutions location, an IT Academy location, or such other entity as Microsoft may designate from time to time.

   c. **"Authorized Training Session(s)"** means those training sessions authorized by Microsoft and conducted at or through Authorized Learning Centers by a Trainer providing training to Students solely on Official Microsoft Learning Products (formerly known as Microsoft Official Curriculum or "MOC") and Microsoft Dynamics Learning Products (formerly know as Microsoft Business Solutions Courseware). Each Authorized Training Session will provide training on the subject matter of one (1) Course.

   d. "**Course**" means one of the courses using Licensed Content offered by an Authorized Learning Center during an Authorized Training Session, each of which provides training on a particular Microsoft technology subject matter.

   e. **"Device(s)"** means a single computer, device, workstation, terminal, or other digital electronic or analog device.

   f. **"Licensed Content"** means the materials accompanying these license terms. The Licensed Content may include, but is not limited to, the following elements: (i) Trainer Content, (ii) Student Content, (iii) classroom setup guide, and (iv) Software. There are different and separate components of the Licensed Content for each Course.

   g. **"Software"** means the Virtual Machines and Virtual Hard Disks, or other software applications that may be included with the Licensed Content.

   h. **"Student(s)"** means a student duly enrolled for an Authorized Training Session at your location.

i. **"Student Content"** means the learning materials accompanying these license terms that are for use by Students and Trainers during an Authorized Training Session. Student Content may include labs, simulations, and courseware files for a Course.

j. **"Trainer(s)"** means a) a person who is duly certified by Microsoft as a Microsoft Certified Trainer and b) such other individual as authorized in writing by Microsoft and has been engaged by an Authorized Learning Center to teach or instruct an Authorized Training Session to Students on its behalf.

k. **"Trainer Content"** means the materials accompanying these license terms that are for use by Trainers and Students, as applicable, solely during an Authorized Training Session. Trainer Content may include Virtual Machines, Virtual Hard Disks, Microsoft PowerPoint files, instructor notes, and demonstration guides and script files for a Course.

l. **"Virtual Hard Disks"** means Microsoft Software that is comprised of virtualized hard disks (such as a base virtual hard disk or differencing disks) for a Virtual Machine that can be loaded onto a single computer or other device in order to allow end-users to run multiple operating systems concurrently. For the purposes of these license terms, Virtual Hard Disks will be considered "Trainer Content".

m. **"Virtual Machine"** means a virtualized computing experience, created and accessed using Microsoft® Virtual PC or Microsoft® Virtual Server software that consists of a virtualized hardware environment, one or more Virtual Hard Disks, and a configuration file setting the parameters of the virtualized hardware environment (e.g., RAM). For the purposes of these license terms, Virtual Hard Disks will be considered "Trainer Content".

n. **"you"** means the Authorized Learning Center or Trainer, as applicable, that has agreed to these license terms.

2. **OVERVIEW.**

Licensed Content. The Licensed Content includes Software, Academic Materials (online and electronic), Trainer Content, Student Content, classroom setup guide, and associated media.

License Model. The Licensed Content is licensed on a per copy per Authorized Learning Center location or per Trainer basis.

3. **INSTALLATION AND USE RIGHTS.**

a. **Authorized Learning Centers and Trainers: For each Authorized Training Session, you may:**

i. either install individual copies of the relevant Licensed Content on classroom Devices only for use by Students enrolled in and the Trainer delivering the Authorized Training Session, provided that the number of copies in use does not exceed the number of Students enrolled in and the Trainer delivering the Authorized Training Session, **OR**

ii. install one copy of the relevant Licensed Content on a network server only for access by classroom Devices and only for use by Students enrolled in and the Trainer delivering the Authorized Training Session, provided that the number of Devices accessing the Licensed Content on such server does not exceed the number of Students enrolled in and the Trainer delivering the Authorized Training Session.

iii. and allow the Students enrolled in and the Trainer delivering the Authorized Training Session to use the Licensed Content that you install in accordance with (ii) or (ii) above during such Authorized Training Session in accordance with these license terms.

MCT USE ONLY. STUDENT USE PROHIBITED

i. Separation of Components. The components of the Licensed Content are licensed as a single unit. You may not separate the components and install them on different Devices.

ii. Third Party Programs. The Licensed Content may contain third party programs. These license terms will apply to the use of those third party programs, unless other terms accompany those programs.

b. **Trainers:**

i. Trainers may Use the Licensed Content that you install or that is installed by an Authorized Learning Center on a classroom Device to deliver an Authorized Training Session.

ii. Trainers may also Use a copy of the Licensed Content as follows:

A. Licensed Device. The licensed Device is the Device on which you Use the Licensed Content. You may install and Use one copy of the Licensed Content on the licensed Device solely for your own personal training Use and for preparation of an Authorized Training Session.

B. Portable Device. You may install another copy on a portable device solely for your own personal training Use and for preparation of an Authorized Training Session.

4. **PRE-RELEASE VERSIONS.** If this is a pre-release ("beta") version, in addition to the other provisions in this agreement, these terms also apply:

a. **Pre-Release Licensed Content.** This Licensed Content is a pre-release version. It may not contain the same information and/or work the way a final version of the Licensed Content will. We may change it for the final, commercial version. We also may not release a commercial version. You will clearly and conspicuously inform any Students who participate in each Authorized Training Session of the foregoing; and, that you or Microsoft are under no obligation to provide them with any further content, including but not limited to the final released version of the Licensed Content for the Course.

b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, you give to Microsoft, without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software, Licensed Content, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software or documentation to third parties because we include your feedback in them. These rights survive this agreement.

c. **Confidential Information.** The Licensed Content, including any viewer, user interface, features and documentation that may be included with the Licensed Content, is confidential and proprietary to Microsoft and its suppliers.

i. **Use.** For five years after installation of the Licensed Content or its commercial release, whichever is first, you may not disclose confidential information to third parties. You may disclose confidential information only to your employees and consultants who need to know the information. You must have written agreements with them that protect the confidential information at least as much as this agreement.

ii. **Survival.** Your duty to protect confidential information survives this agreement.

iii. **Exclusions**. You may disclose confidential information in response to a judicial or governmental order. You must first give written notice to Microsoft to allow it to seek a

MCT USE ONLY. STUDENT USE PROHIBITED

protective order or otherwise protect the information. Confidential information does not include information that

- becomes publicly known through no wrongful act;
- you received from a third party who did not breach confidentiality obligations to Microsoft or its suppliers; or
- you developed independently.

d. **Term**. The term of this agreement for pre-release versions is (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) the commercial release of the final release version of the Licensed Content, whichever is first ("beta term").

e. **Use**. You will cease using all copies of the beta version upon expiration or termination of the beta term, and will destroy all copies of same in the possession or under your control and/or in the possession or under the control of any Trainers who have received copies of the pre-released version.

f. **Copies**. Microsoft will inform Authorized Learning Centers if they may make copies of the beta version (in either print and/or CD version) and distribute such copies to Students and/or Trainers. If Microsoft allows such distribution, you will follow any additional terms that Microsoft provides to you for such copies and distribution.

5. ADDITIONAL LICENSING REQUIREMENTS AND/OR USE RIGHTS.

a. Authorized Learning Centers and Trainers:

i. Software.

ii. **Virtual Hard Disks**. The Licensed Content may contain versions of Microsoft XP, Microsoft Windows Vista, Windows Server 2003, Windows Server 2008, and Windows 2000 Advanced Server and/or other Microsoft products which are provided in Virtual Hard Disks.

A. If the Virtual Hard Disks and the labs are launched through the Microsoft Learning Lab Launcher, then these terms apply:

Time-Sensitive Software**.** If the Software is not reset, it will stop running based upon the time indicated on the install of the Virtual Machines (between 30 and 500 days after you install it). You will not receive notice before it stops running. You may not be able to access data used or information saved with the Virtual Machines when it stops running and may be forced to reset these Virtual Machines to their original state. You must remove the Software from the Devices at the end of each Authorized Training Session and reinstall and launch it prior to the beginning of the next Authorized Training Session.

B. If the Virtual Hard Disks require a product key to launch, then these terms apply:

Microsoft will deactivate the operating system associated with each Virtual Hard Disk. Before installing any Virtual Hard Disks on classroom Devices for use during an Authorized Training Session, you will obtain from Microsoft a product key for the operating system software for the Virtual Hard Disks and will activate such Software with Microsoft using such product key.

C. These terms apply to all Virtual Machines and Virtual Hard Disks:

MCT USE ONLY. STUDENT USE PROHIBITED

You may only use the Virtual Machines and Virtual Hard Disks if you comply with the terms and conditions of this agreement and the following security requirements:

- You may not install Virtual Machines and Virtual Hard Disks on portable Devices or Devices that are accessible to other networks.
- You must remove Virtual Machines and Virtual Hard Disks from all classroom Devices at the end of each Authorized Training Session, except those held at Microsoft Certified Partners for Learning Solutions locations.
- You must remove the differencing drive portions of the Virtual Hard Disks from all classroom Devices at the end of each Authorized Training Session at Microsoft Certified Partners for Learning Solutions locations.
- You will ensure that the Virtual Machines and Virtual Hard Disks are not copied or downloaded from Devices on which you installed them.
- You will strictly comply with all Microsoft instructions relating to installation, use, activation and deactivation, and security of Virtual Machines and Virtual Hard Disks.
- You may not modify the Virtual Machines and Virtual Hard Disks or any contents thereof.
- You may not reproduce or redistribute the Virtual Machines or Virtual Hard Disks.

ii. **Classroom Setup Guide**. You will assure any Licensed Content installed for use during an Authorized Training Session will be done in accordance with the classroom set-up guide for the Course.

iii. **Media Elements and Templates.** You may allow Trainers and Students to use images, clip art, animations, sounds, music, shapes, video clips and templates provided with the Licensed Content solely in an Authorized Training Session. If Trainers have their own copy of the Licensed Content, they may use Media Elements for their personal training use.

iv. **iv Evaluation Software**. Any Software that is included in the Student Content designated as "Evaluation Software" may be used by Students solely for their personal training outside of the Authorized Training Session.

b. **Trainers Only:**

i. **Use of PowerPoint Slide Deck Templates***.* The Trainer Content may include Microsoft PowerPoint slide decks. Trainers may use, copy and modify the PowerPoint slide decks only for providing an Authorized Training Session. If you elect to exercise the foregoing, you will agree or ensure Trainer agrees: (a) that modification of the slide decks will not constitute creation of obscene or scandalous works, as defined by federal law at the time the work is created; and (b) to comply with all other terms and conditions of this agreement.

ii. **Use of Instructional Components in Trainer Content***.* For each Authorized Training Session, Trainers may customize and reproduce, in accordance with the MCT Agreement, those portions of the Licensed Content that are logically associated with instruction of the Authorized Training Session. If you elect to exercise the foregoing rights, you agree or ensure the Trainer agrees: (a) that any of these customizations or reproductions will only be used for providing an Authorized Training Session and (b) to comply with all other terms and conditions of this agreement.

MCT USE ONLY. STUDENT USE PROHIBITED

iii. **Academic Materials.** If the Licensed Content contains Academic Materials, you may copy and use the Academic Materials. You may not make any modifications to the Academic Materials and you may not print any book (either electronic or print version) in its entirety. If you reproduce any Academic Materials, you agree that:

- The use of the Academic Materials will be only for your personal reference or training use
- You will not republish or post the Academic Materials on any network computer or broadcast in any media;
- You will include the Academic Material's original copyright notice, or a copyright notice to Microsoft's benefit in the format provided below:

  Form of Notice:

  © 2010 Reprinted for personal reference use only with permission by Microsoft Corporation. All rights reserved.

  Microsoft, Windows, and Windows Server are either registered trademarks or trademarks of Microsoft Corporation in the US and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

6. **INTERNET-BASED SERVICES.** Microsoft may provide Internet-based services with the Licensed Content. It may change or cancel them at any time. You may not use these services in any way that could harm them or impair anyone else's use of them. You may not use the services to try to gain unauthorized access to any service, data, account or network by any means.

7. **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 allow you to use it in certain ways. You may not

- install more copies of the Licensed Content on classroom Devices than the number of Students and the Trainer in the Authorized Training Session;
- allow more classroom Devices to access the server than the number of Students enrolled in and the Trainer delivering the Authorized Training Session if the Licensed Content is installed on a network server;
- copy or reproduce the Licensed Content to any server or location for further reproduction or distribution;
- disclose the results of any benchmark tests of the Licensed Content to any third party without Microsoft's prior written approval;
- work around any technical limitations in the Licensed Content;
- reverse engineer, decompile or disassemble the Licensed Content, except and only to the extent that applicable law expressly permits, despite this limitation;
- make more copies of the Licensed Content than specified in this agreement or allowed by applicable law, despite this limitation;
- publish the Licensed Content for others to copy;

MCT USE ONLY. STUDENT USE PROHIBITED

- transfer the Licensed Content, in whole or in part, to a third party;
- access or use any Licensed Content for which you (i) are not providing a Course and/or (ii) have not been authorized by Microsoft to access and use;
- rent, lease or lend the Licensed Content; or
- use the Licensed Content for commercial hosting services or general business purposes.
- Rights to access the server software that may be included with the Licensed Content, including the Virtual Hard Disks does not give you any right to implement Microsoft patents or other Microsoft intellectual property in software or devices that may access the server.

8. **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.**

9. **NOT FOR RESALE SOFTWARE/LICENSED CONTENT.** You may not sell software or Licensed Content marked as "NFR" or "Not for Resale."

10. **ACADEMIC EDITION.** You must be a "Qualified Educational User" to use Licensed Content marked as "Academic Edition" or "AE." If you do not know whether you are a Qualified Educational User, visit www.microsoft.com/education or contact the Microsoft affiliate serving your country.

11. **TERMINATION**. Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of these license terms. In the event your status as an Authorized Learning Center or Trainer a) expires, b) is voluntarily terminated by you, and/or c) is terminated by Microsoft, this agreement shall automatically terminate. Upon any termination of this agreement, you must destroy all copies of the Licensed Content and all of its component parts.

12. **ENTIRE AGREEMENT. This agreement, and the terms for supplements, updates, Internet-based services and support services that you use, are the entire agreement for the Licensed Content and support services.**

13. **APPLICABLE LAW.**

    a. **United States.** If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.

    b. **Outside the United States.** If you acquired the Licensed Content in any other country, the laws of that country apply.

14. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.

15. **DISCLAIMER OF WARRANTY. The Licensed Content is licensed "as-is." You bear the risk of using it. Microsoft 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 excludes the implied warranties of merchantability, fitness for a particular purpose and non-infringement.**

MCT USE ONLY. STUDENT USE PROHIBITED

16. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. YOU CAN RECOVER FROM MICROSOFT AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO U.S. $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, software, 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.

MCT USE ONLY. STUDENT USE PROHIBITED

Welcome!

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

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.

- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance[1]. 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

Microsoft | Learning

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

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Acknowledgement

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

John Sharp—Content Developer

John Sharp is a principal technologist at Content Master, part of CM Group Ltd, a technical authoring and consulting company. An expert on developing applications with the Microsoft® .NET Framework and interoperability issues, John has produced numerous courses, tutorials, white papers, and presentations on distributed systems, Web services, and the C# language. John is the author of several popular books, including five editions of Microsoft Visual C# Step by Step and Microsoft Windows Communication Foundation Step by Step.

Antony Norris—Content Developer

Antony Norris is a senior technologist at Content Master, part of CM Group Ltd, a technical authoring and consulting company. Antony is a Microsoft Visual C#® developer who specializes in various .NET Framework technologies, including ASP.NET, Windows® Communication Foundation, and Windows Mobile. Antony has worked on several other Microsoft Learning courses, including *Programming with the Microsoft .NET Framework Using Microsoft Visual Studio 2005 and Visual Studio 2008 Connected Systems: Windows Communication Foundation.*

Mike Sumsion—Content Developer

Mike Sumsion is a senior technologist at Content Master, part of CM Group Ltd, a technical authoring and consulting company. Mike is a developer who specializes in SharePoint Products and Technologies, .NET Framework client applications, and Windows Mobile. Mike has worked on several other Microsoft Learning courses, including *Developing Solutions with MS Windows SharePoint Services 3.0 and Visual Studio 2005* and *Core Web Application Technologies with Microsoft Visual Studio 2005.*

Chris Barker—Technical Reviewer

Chris Barker is an MCT working in the New Zealand market currently employed as a staff trainer at Auldhouse, one of New Zealand's major CPLS training centers in Wellington. Chris' background includes programming from the early 1970s–his first program was written in assembly language and debugged in binary (literally)! While focusing training on programming (mostly using the .NET Framework) and

MCT USE ONLY. STUDENT USE PROHIBITED

databases (mostly Microsoft SQL Server) Chris has also been an infrastructure trainer and has Microsoft networking qualifications.

Bill Chapman – Technical Reviewer

Bill is the Principal Architect at Chapman and Associates. He helps the Microsoft Certified Trainer Community with custom courses on how to succeed as a small business as a Microsoft Certified Trainer. He is now in his fourteenth year as an MCT. Before joining Microsoft in 2007 he spent 11 years specializing in developer and database training. He has taught as both a staff instructor and as an independent contractor throughout his career, and has taught courses all around the world.

Manish Sharma – Open Beta Facilitator

Manish Sharma is an MCT with more than 7 Years of experience in Software Technology Training. Apart from being MCT, MCTS, and MCP he has a Masters in Computer Applications, gained as part of his formal education. He conducts .NET Technology training courses on behalf of Microsoft for Microsoft Certified Partners and other Microsoft clients. He has a vast experience in conducting developer-centric training on various versions of the Microsoft .NET Framework, Visual Studio, and SharePoint technologies.

MCT USE ONLY. STUDENT USE PROHIBITED

Contents

Module 1: Introducing C# and the .NET Framework

Lesson 1: Introduction to the .NET Framework 4 1-4
Lesson 2: Creating Projects Within Visual Studio 2010 1-16
Lesson 3: Writing a C# Application 1-33
Lesson 4: Building a Graphical Application 1-44
Lesson 5: Documenting an Application 1-58
Lesson 6: Debugging Applications by Using Visual Studio 2010 1-66
Lab: Introducing C# and the .NET Framework 1-78

Module 2: Using C# Programming Constructs

Lesson 1: Declaring Variables and Assigning Values 2-4
Lesson 2: Using Expressions and Operators 2-23
Lesson 3: Creating and Using Arrays 2-36
Lesson 4: Using Decision Statements 2-49
Lesson 5: Using Iteration Statements 2-63
Lab: Using C# Programming Constructs 2-78

Module 3: Declaring and Calling Methods

Lesson 1: Defining and Invoking Methods 3-3
Lesson 2: Specifying Optional Parameters and Output Parameters 3-29
Lab: Declaring and Calling Methods 3-39

Module 4: Handling Exceptions

Lesson 1: Handling Exceptions 4-3
Lesson 2: Raising Exceptions 4-23
Lab: Handling Exceptions 4-34

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5: Reading and Writing Files

Lesson 1: Accessing the File System 5-3
Lesson 2: Reading and Writing Files by Using Streams 5-27
Lab: Reading and Writing Files 5-45

Module 6: Creating New Types

Lesson 1: Creating and Using Enumerations 6-3
Lesson 2: Creating and Using Classes 6-12
Lesson 3: Creating and Using Structures 6-33
Lesson 4: Comparing References to Values 6-41
Lab: Creating New Types 6-55

Module 7: Encapsulating Data and Methods

Lesson 1: Controlling Visibility of Type Members 7-4
Lesson 2: Sharing Methods and Data 7-15
Lab: Encapsulating Data and Methods 7-29

Module 8: Inheriting from Classes and Implementing Interfaces

Lesson 1: Using Inheritance to Define New Reference Types 8-3
Lesson 2: Defining and Implementing Interfaces 8-27
Lesson 3: Defining Abstract Classes 8-45
Lab: Inheriting from Classes and Implementing Interfaces 8-56

Module 9: Managing the Lifetime of Objects and Controlling Resources

Lesson 1: Introduction to Garbage Collection 9-4
Lesson 2: Managing Resources 9-21
Lab: Managing the Lifetime of Objects and Controlling Resources 9-35

Module 10: Encapsulating Data and Defining Overloaded Operators

Lesson 1: Creating and Using Properties 10-4
Lab A: Creating and Using Properties 10-26
Lesson 2: Creating and Using Indexers 10-38
Lab B: Creating and Using Indexers 10-50

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3: Overloading Operators 10-60
Lab C: Overloading Operators 10-79

Module 11: Decoupling Methods and Handling Events

Lesson 1: Declaring and Using Delegates 11-4
Lesson 2: Using Lambda Expressions 11-14
Lesson 3: Handling Events 11-22
Lab: Decoupling Methods and Handling Events 11-38

Module 12: Using Collections and Building Generic Types

Lesson 1: Using Collections 12-4
Lab A: Using Collections 12-22
Lesson 2: Creating and Using Generic Types 12-28
Lesson 3: Defining Generic Interfaces and Understanding Variance 12-42
Lesson 4: Using Generic Methods and Delegates 12-56
Lab B: Building Generic Types 12-69

Module 13: Building and Enumerating Custom Collection Classes

Lesson 1: Implementing a Custom Collection Class 13-3
Lesson 2: Adding an Enumerator to a Custom Collection Class 13-21
Lab: Building and Enumerating Custom Collection Classes 13-37

Module 14: Using LINQ to Query Data

Lesson 1: Using the LINQ Extension Methods and Query Operators 14-3
Lesson 2: Building Dynamic LINQ Queries and Expressions 14-28
Lab: Using LINQ to Query Data 14-47

Module 15: Integrating Visual C# Code with Dynamic Languages and COM Components

Lesson 1: Integrating Visual C# Code with Ruby and Python 15-4
Lesson 2: Accessing COM Components from Visual C# 15-19
Lab: Integrating Visual C# Code with Dynamic Languages and COM Components 15-36

MCT USE ONLY. STUDENT USE PROHIBITED

Appendix: Lab Answer Keys

Module 1 Lab: Introducing C# and the .NET Framework L1-1
Module 2 Lab: Using C# Programming Constructs L2-1
Module 3 Lab: Declaring and Calling Methods L3-1
Module 4 Lab: Handling Exceptions L4-1
Module 5 Lab: Reading and Writing Files L5-1
Module 6 Lab: Creating New Types L6-1
Module 7 Lab: Encapsulating Data and Methods L7-1
Module 8 Lab: Inheriting from Classes and Implementing Interfaces L8-1
Module 9 Lab: Managing the Lifetime of Objects and Controlling Resources L9-1
Module 10 Lab A: Creating and Using Properties L10A-1
Module 10 Lab B: Creating and Using Indexers L10B-1
Module 10 Lab C: Overloading Operators L10C-1
Module 11 Lab: Decoupling Methods and Handling Events L11-1
Module 12 Lab A: Using Collections L12A-1
Module 12 Lab B: Building Generic Types L12B-1
Module 13 Lab: Building and Enumerating Custom Collection Classes L13-1
Module 14 Lab: Using LINQ to Query Data L14-1
Module 15 Lab: Integrating Visual C# Code with Dynamic Languages and COM Components L15-1

MCT USE ONLY. STUDENT USE PROHIBITED

About This Course

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

Course Description

This course teaches you C# language syntax, program structure, and implementation by using Microsoft® Visual Studio® 2010 and the Microsoft .NET Framework 4.

This course provides a solid foundation in C# to the level necessary to enable students to attend other courses in the Technical Specialist tracks.

Audience

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

This course is not designed for new programmers; it is targeted at professional developers with at least 12 months experience of programming in an object-oriented environment.

Student Prerequisites

This course requires that you meet the following prerequisites:

- C++, Java, or experience with another programming language and knowledge of the following items:
  - Creating classes
  - Inheritance and abstraction
  - Polymorphism
  - Interfaces
  - Exceptions
- Knowledge of the Visual Studio integrated development environment (IDE).

MCT USE ONLY. STUDENT USE PROHIBITED

Course Objectives

After completing this course, students will be able to:

- Describe the purpose of the .NET Framework, and explain how to use Microsoft Visual C#® and Visual Studio 2010 to build .NET Framework applications.
- Describe the syntax of basic C# programming constructs.
- Describe how to create and call methods.
- Describe how to catch, handle, and throw exceptions.
- Describe how to perform basic file I/O operations in a Visual C# application.
- Describe how to create and use new types (enumerations, classes, and structures), and explain the differences between reference types and value types.
- Describe how to control the visibility and lifetime of members in a type.
- Describe how to use inheritance to create new reference types.
- Describe how to manage the lifetime of objects and control the use of resources.
- Describe how to create properties and indexers to encapsulate data, and explain how to define operators for this data.
- Describe how to decouple an operation from the method that implements it, and explain how to use these decoupled operations to handle asynchronous events.
- Describe the purpose of collections, and explain how to use generics to implement type-safe collection classes, structures, interfaces, and methods.
- Describe how to implement custom collection classes that support enumeration.
- Describe how to query in-memory data by using Language-Integrated Query (LINQ) queries.
- Describe how to integrate code written by using a dynamic language such as Ruby and Python, or technologies such as Component Object Model (COM), into a C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Course Outline

This section provides an outline of the course:

Module 1, "Introducing C# and the .NET Framework," provides an overview of the .NET Framework and shows how you can start to build your own .NET Framework applications by using C# and Visual Studio 2010.

Module 2, "Using C# Programming Constructs," provides an introduction to C# programming language syntax and introduces many of the basic C# language data types and programming constructs.

Module 3, "Declaring and Calling Methods," introduces the concept of methods and describes how, in object-oriented languages such as C#, a method is a unit of code that is designed to perform a discrete piece of work. This module shows you how to declare and call methods by using C#.

Module 4, "Handling Exceptions," introduces the importance of exception handling and explains why applications should be designed with exception handling in mind. This module explains how you can implement effective exception handling in your applications and describes how to use exceptions in your methods to indicate an error condition to the code that calls your methods.

Module 5, "Reading and Writing Files," explains how the ability to access and manipulate files on the file system is a common requirement for many applications. This module shows you how to read and write to files by using the classes in the .NET Framework. It also describes the different approaches that you can take and explains how to read and write different formats of data.

Module 6, "Creating New Types," explains how to build your own types that model items in the real world and describes how to implement the business logic for these items that your applications require. This module explains the differences between reference types and value types.

Module 7, "Encapsulating Data and Methods," describes how to use the access modifiers that C# provides to enable you to implement encapsulation. This module also introduces the **static** modifier, which enables you to define members that can be shared over multiple instances of the same type.

Module 8, "Inheriting from Classes and Implementing Interfaces," explains that inheritance is a key concept in an object-oriented language and describes how you can use inheritance, interfaces, and abstract classes to develop object hierarchies. This module also explains how you can use these object hierarchies to help reduce bugs by defining clear contracts for the functionality that a class should expose and providing default implementations where you can sensibly abstract code into a base type.

Module 9, "Managing the Lifetime of Objects and Controlling Resources," introduces the concept of resource management and discusses its importance. This module explains how the .NET Framework simplifies resource management by automatically reclaiming the resources for a managed object when an application no longer references it. This module also explains that the garbage collector does not control unmanaged resources and describes the steps that you can take to dispose of such resources.

Module 10, "Encapsulating Data and Defining Overloaded Operators," introduces properties and indexers. These are elements of C# that enable you to encapsulate data and expose data appropriately and efficiently. This module also describes how to implement operators for your types by using overloading.

Module 11, "Decoupling Methods and Handling Events," explains how to decouple an operation from the method that implements it and describes how to use anonymous methods to implement decoupled operations. This module also explains how to use events to inform consuming applications of a change or notable occurrence in a type.

Module 12, "Using Collections and Building Generic Types," introduces the concept of collection classes and explains that you can use them with greater flexibility than a simple array. This module also introduces generics and explains how to use generic classes to maintain type integrity and avoid the issues that are associated with a lack of type safety.

Module 13, "Building and Enumerating Custom Collection Classes," explains how to use the collection classes that the .NET Framework base class library includes. This module also describes how to build custom collection classes.

Module 14, "Using LINQ to Query Data," explains how you can use LINQ to abstract the mechanism that an application uses to query data from the application code. This module describes built-in C# LINQ extension methods and LINQ query operators. This module also describes how to build LINQ queries dynamically by using expression trees.

Module 15, "Integrating Visual C# Code with Dynamic Languages and COM Components," explains how the .NET Framework 4 enables you to invoke code and components that were written by using other languages from your C# code. It describes how the dynamic language runtime (DLR) enables you to reuse code built by using a wide range of scripting languages, such as Ruby and Python. This module also describes how to invoke COM components from a C# application.

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 all the critical technical information in a crisp, tightly-focused format, which is just right 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 improved on-the-job reference material to boost knowledge and skills retention.
    - Lab Answer Keys: Provide step-by-step lab solution guidance at your finger tips when it's needed.
- *Course Companion CD*. Searchable, easy-to-navigate digital content with integrated premium on-line resources designed to supplement the Course Handbook.
    - Labs: Include complete lab exercise information and answer keys in digital form to use during lab time.
    - Resources: Include well-categorized additional resources that give you immediate access to the most up-to-date premium content on TechNet, MSDN®, Microsoft Press®.
    - Student Course Files: Include the Allfiles.exe, a self-extracting executable file that contains all the files required for the labs and demonstrations.



Note: To access the full course content, insert the Course Companion CD into the CD-ROM drive, and then in the root directory of the CD, double-click StartCD.exe.

- *Course evaluation*. At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.

To provide additional comments or feedback on the course, send e-mail to support@mscourseware.com. To inquire about the Microsoft Certification Program, send e-mail to mcphelp@microsoft.com.

MCT USE ONLY. STUDENT USE PROHIBITED

Virtual Machine Environment

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

Virtual Machine Configuration

In this course, you will use Windows Server® 2008 with Hyper-V™ to perform the labs.

The following table shows the role of each virtual machine used in this course.

Virtual machine	Role
10266A-GEN-DEV	Windows 7 Client

Software Configuration

The following software is installed on each VM:

- Visual Studio 2010 Professional Edition
- IronRuby
- IronPython
- SandCastle
- HTML Help Workshop
- The 2007 Microsoft Office system

Course Files

There are files associated with the labs in this course. The lab files are located in the folder E:\Labfiles\ on the student computers.

Classroom Setup

Each classroom computer will have the same virtual machine configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all

MCT USE ONLY. STUDENT USE PROHIBITED

Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware are taught.

This course requires that you have a computer that meets or exceeds hardware level 6, which prescribes the following:

- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120-GB hard disks, 7,200 RM SATA or better (configured as a stripe array)
- 4 gigabytes (GB) of RAM expandable to 8 GB or higher
- DVD drive
- Network adapter
- Super VGA (SVGA) 17-inch monitor
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

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

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1

Introducing C# and the .NET Framework

Contents:

Lesson 1: Introduction to the .NET Framework 4 1-4

Lesson 2: Creating Projects Within Visual Studio 2010 1-16

Lesson 3: Writing a C# Application 1-33

Lesson 4: Building a Graphical Application 1-44

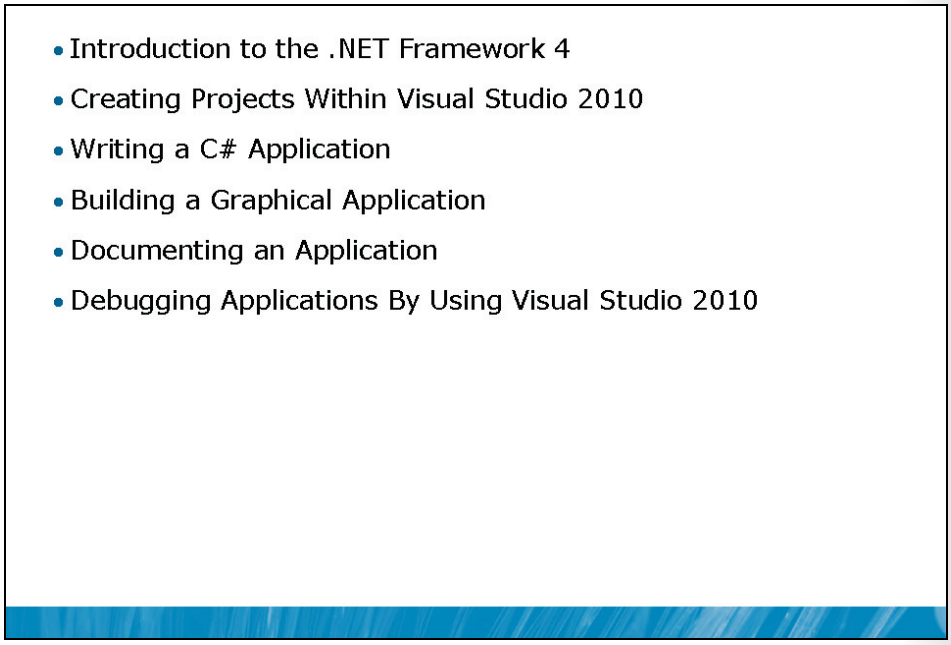
Lesson 5: Documenting an Application 1-58

Lesson 6: Debugging Applications by Using Visual Studio 2010 1-66

Lab: Introducing C# and the .NET Framework 1-78

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



Microsoft® Visual Studio® 2010 and the Microsoft .NET Framework 4 provide a comprehensive development platform to enable you to build, debug, deploy, and manage applications.

This module describes the purpose of the .NET Framework 4, and how to build applications by using Visual Studio 2010.

Objectives

After completing this module, you will be able to:

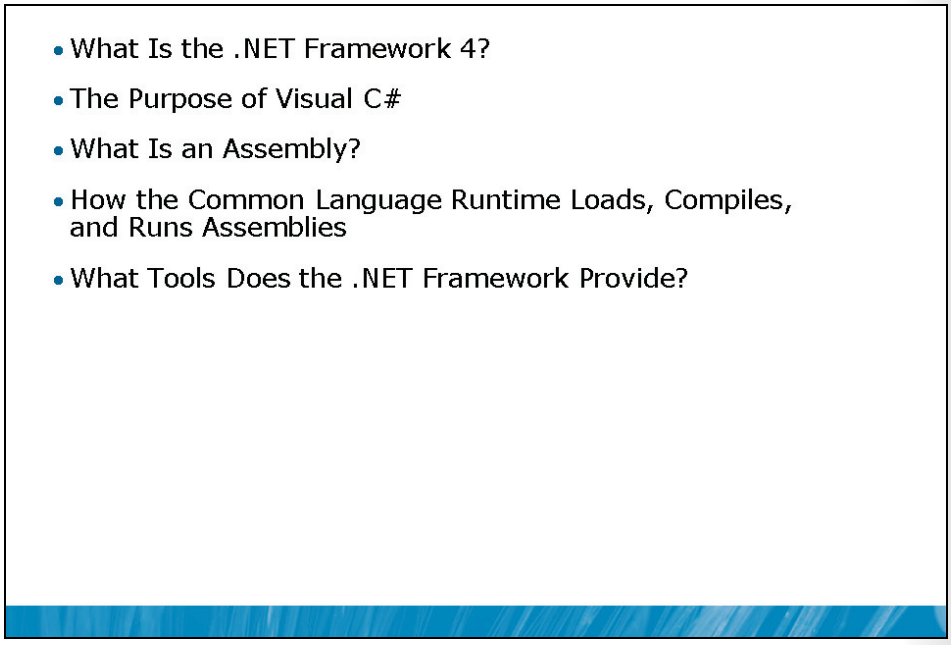
- Explain the purpose of the .NET Framework 4.
- Create Microsoft Visual C#® projects by using Visual Studio 2010.
- Explain the structure of a Visual C# application.
- Use the Windows® Presentation Foundation (WPF) Application template to build a simple graphical application.

MCT USE ONLY. STUDENT USE PROHIBITED

- Use XML comments to document an application.
- Use the debugger to step through a program.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Introduction to the .NET Framework 4



This lesson introduces the .NET Framework 4, and describes the key concepts of .NET and some of the tools that are provided to help simplify development.

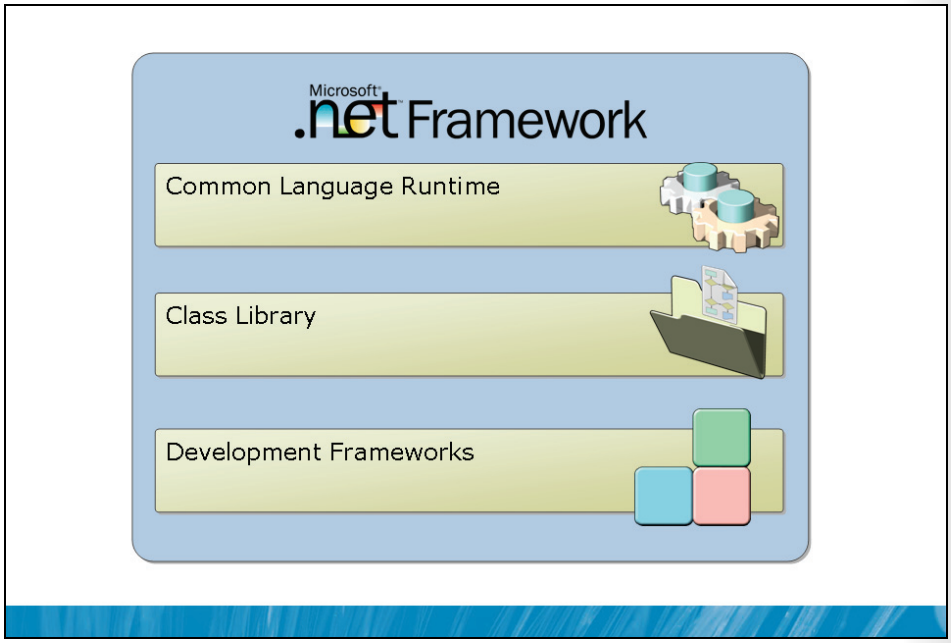
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of the .NET Framework 4.
- Describe the role of Visual C# for writing the code for .NET Framework 4 applications.
- Describe the purpose of an assembly.
- Explain how the common language runtime (CLR) compiles and runs assemblies.
- Describe the tools that the .NET Framework 4 provides.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the .NET Framework 4?



Key Points

The .NET Framework 4 provides a comprehensive development platform that offers a fast and efficient way to build applications and services. Using Visual Studio 2010, developers can utilize the .NET Framework 4 to create a wide range of solutions that operate across a broad range of computing devices.

The .NET Framework 4 provides three principal elements: the CLR, the .NET Framework class library, and a collection of development frameworks.

The Common Language Runtime

The .NET Framework 4 provides an environment called the CLR. The CLR manages the execution of code and simplifies the development process by providing a robust and secure execution environment that provides common services such as memory management, transactions, interprocess communications, multithreading, and many other features.

MCT USE ONLY. STUDENT USE PROHIBITED

The .NET Framework Class Library

The .NET Framework 4 provides a library of reusable classes that developers can use to build applications. The classes provide a foundation of common functionality and constructs that help simplify application development and remove the requirement for developers to constantly reinvent logic. For example, the **System.IO.File** class contains functionality that enables developers 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 4 provides several development frameworks that you can use to build common types of applications. These frameworks provide the necessary components and infrastructure to get you started. The development frameworks include:

- *ASP.NET*. Enables you to build server-side Web applications.
- *WPF*. Enables you to build rich client applications.
- *Windows Communication Foundation (WCF)*. Enables you to build secure and reliable service-oriented applications.
- *Windows Workflow Foundation (WF)*. Enables you to build workflow solutions to fulfill the complex business requirements of modern organizations.

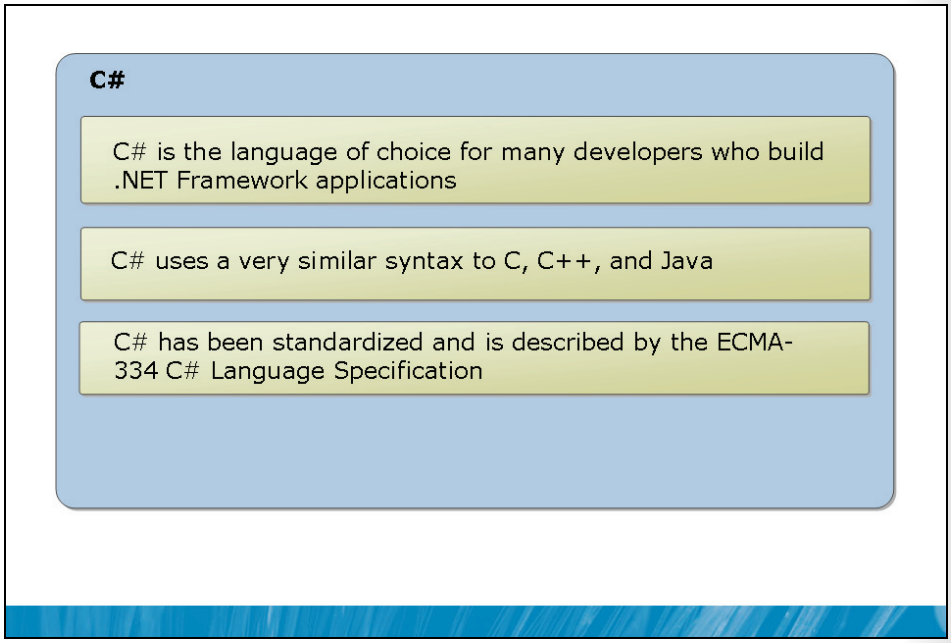
Question: What is the purpose of the .NET Framework 4, and the three main components that it provides?

Additional Reading

For more information about the .NET Framework, see the Microsoft .NET page at http://go.microsoft.com/fwlink/?LinkId=192876.

MCT USE ONLY. STUDENT USE PROHIBITED

The Purpose of Visual C#



Key Points

The CLR runs executable code that is generated by using a compiler. You can build applications for the .NET Framework by using any language that has a compiler that can generate executable code in the format that the CLR recognizes. Visual Studio 2010 provides compilers for C++, Visual Basic, F#, and C#. Compilers for other languages are available from a variety of third-party vendors.

C# is the language of choice for many developers. It uses a syntax that is very similar to C, C++, and Java, and has several extensions and features that are designed for operation with the .NET Framework. Because of its heritage, many developers who are familiar with other programming languages find C# easy to learn and can be productive very quickly.

The C# language has been standardized and is described by the ECMA-334 C# Language Specification. Several vendors apart from Microsoft produce C# compilers. The Microsoft implementation is called Visual C#, and is integrated into Visual Studio. Visual Studio supports Visual C# with a full-featured code editor, compiler, project templates, designers, code wizards, a powerful and easy-to-use debugger, and other tools. C# is also available from Microsoft as Visual C# Express

MCT USE ONLY. STUDENT USE PROHIBITED

Edition, which provides a subset of the features that are provided with Visual Studio.



Note: C# is an evolving language. Visual C# 2010 uses C# 4.0, which contains several extensions to the C# language that are not yet part of the ECMA standard.

Question: Which programming languages have you used?

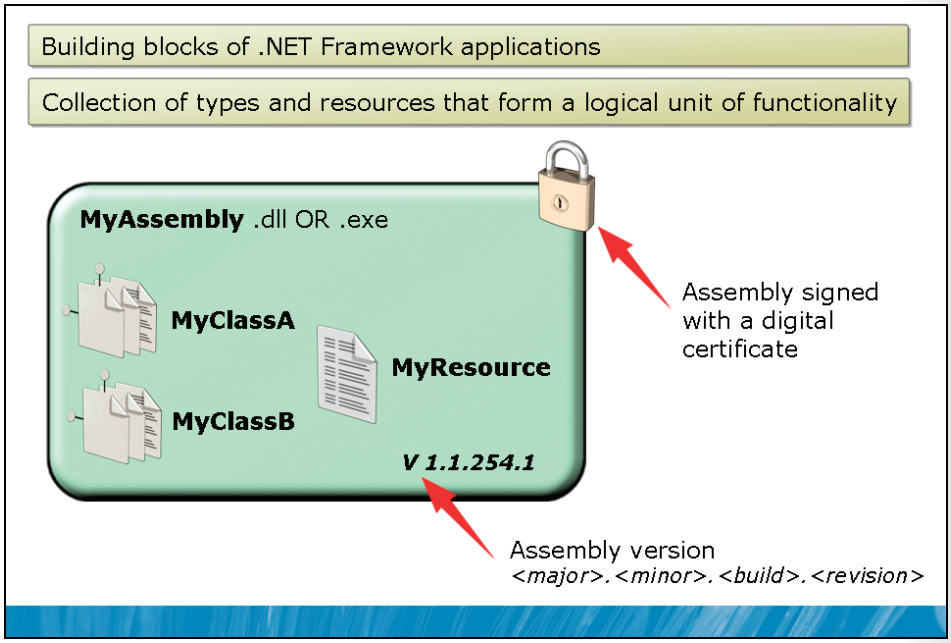
Additional Reading

For more information about the Microsoft implementation of Visual C# 2010, see the Visual C# page http://go.microsoft.com/fwlink/?LinkId=192877.

For more information about the new features of C# 4.0, see the What's New in Visual C# 2010 page at http://go.microsoft.com/fwlink/?LinkId=192878.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Assembly?



Key Points

When you compile a Visual C# application by using Visual Studio 2010, the compiler generates an executable file that the CLR can run. This file is called an assembly. An assembly contains code in an intermediate format called Microsoft intermediate language (MSIL). All compilers for the .NET Framework generate code in this format, regardless of the programming language that was used to write an application. This enables the CLR to run code in the same way, regardless of the language that the developer used.

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, and security.

You can think of an assembly as a collection of types and resources that work together and form a logical unit of functionality. An assembly provides the CLR with the information that it needs to be aware of type implementations.

An assembly can be of two types: an executable program, or a library that contains executable code that other programs can reuse. By using a library, developers can modularize the development of their applications into logical components.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Typically, when you are distributing assemblies to customers as part of your application, you will want to ensure that the assembly contains versioning information, and that the assembly is signed.

Versioning your assemblies is important because ultimately, any applications that you build will have multiple releases. Versioning information can help you identify which versions customers already have and enable you to perform the necessary steps to upgrade the application. Similarly versioning information can also help when documenting and fixing bugs.

Signing your assemblies is equally important because it ensures that your assembly cannot easily be modified or replaced by an alternative implementation from a malicious source, and because it gives the assembly a strong name.

Information such as the assembly version and security identity is stored as metadata in an assembly manifest. The manifest also contains metadata that describes the scope of the assembly, and any references to classes and resources. The manifest is typically stored in a portable executable (PE) file.

Assembly Versioning

Assembly version information is stored in the assembly manifest and is used with the assembly name and culture to derive the assembly's identity. An assembly version number consists of the following:

- Major version number
- Minor version number
- Build number
- Revision number

Assembly Signing

Assembly signing is an important step that developers should include in their build process because it provides the following benefits:

- It protects assemblies from modification.
- It enables you to include the signed assembly in the Global Assembly Cache (GAC), so you can share the assembly with multiple applications.
- It guarantees that the name of the assembly is unique.
- To sign your assembly, you can use the Sign Tool that is provided with the .NET Framework, or you can use the assembly-signing functionality in Visual Studio 2010.

Question: Why would you choose to distribute an assembly rather than distribute the source code?

Additional Reading

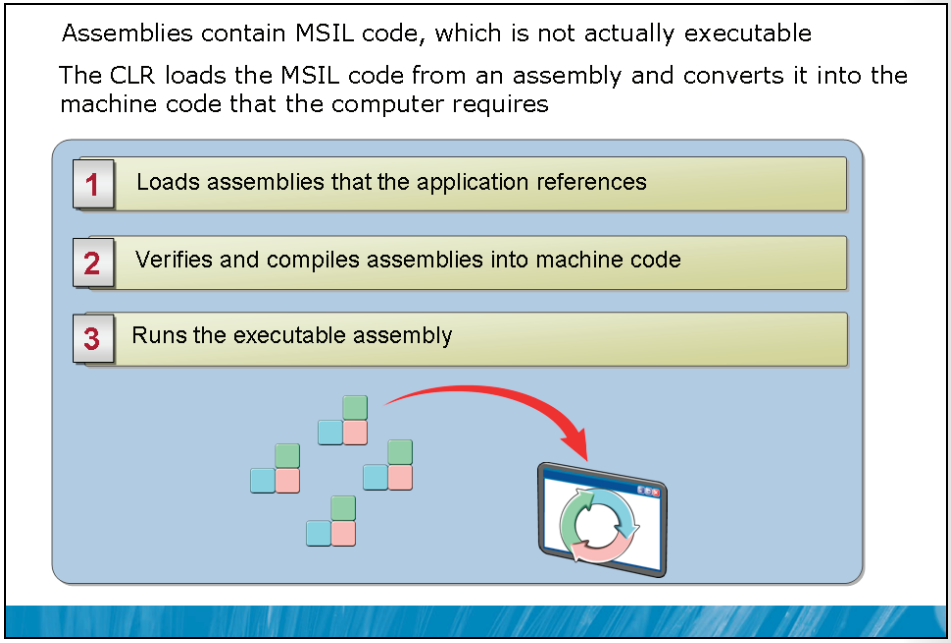
For more information about the purpose and features of assemblies, see the Assemblies in the Common Language Runtime page at http://go.microsoft.com/fwlink/?LinkId=192879.

For more information about assembly versioning, see the Assembly Versioning page at http://go.microsoft.com/fwlink/?LinkId=192880.

For more information about assembly signing, see the SignTool.exe (Sign Tool) page at http://go.microsoft.com/fwlink/?LinkId=192881.

MCT USE ONLY. STUDENT USE PROHIBITED

How the Common Language Runtime Loads, Compiles, and Runs Assemblies



Key Points

Assemblies contain MSIL code, which is not executable. When you run a .NET Framework application, the CLR loads the MSIL code from an assembly and converts it into the machine code that the computer requires.

The CLR is a fundamental component of the .NET Framework. It handles code execution and provides useful services for application development. The CLR contains several components that perform the following tasks when you run a .NET Framework application:

1. The Class Loader locates and loads all assemblies that the application requires. The assemblies will already be compiled into MSIL.
2. The MSIL-to-native compiler verifies the MSIL code and then compiles all assemblies into machine code ready for execution.

MCT USE ONLY. STUDENT USE PROHIBITED



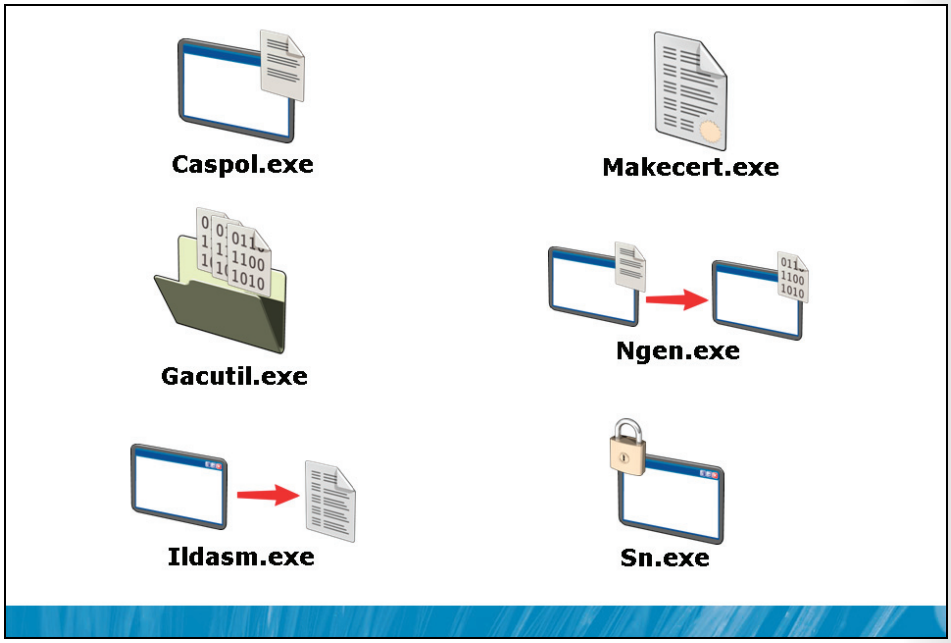
Note: The CLR performs the verification step because it is possible to write your own MSIL code. If you use a C# compiler, the MSIL code will be valid, but the CLR cannot make any assumptions.

3. The Code Manager loads the executable assembly and runs the **Main** method.
4. The Garbage Collector provides automatic lifetime memory management of all objects that your application creates. The Garbage Collector disposes of any objects that your application is no longer using.
5. The Exception Manager provides structured exception handling for .NET applications, which is integrated with Windows structured exception handling.

Question: What steps does the CLR perform when you run your application?

MCT USE ONLY. STUDENT USE PROHIBITED

What Tools Does the .NET Framework Provide?



Key Points

The .NET Framework provides several tools to help simplify the development of .NET applications. The following table describes some of the key tools.

Tool	Description
Code Access Security Policy Tool (Caspol.exe)	Enables users to modify the machine, user, and enterprise security policy. This can include defining a custom permission set and adding assemblies to the full trust list.
Certificate Creation Tool (Makecert.exe)	Enables users to create x.509 certificates for use in their development environment. Typically, you can use these certificates to sign your assemblies and define Secure Sockets Layer (SSL) connections.
Global Assembly Cache Tool (Gacutil.exe)	Enables users to manipulate the assemblies in the GAC. This can include installing and uninstalling assemblies in the GAC so that multiple applications can access them.

MCT USE ONLY. STUDENT USE PROHIBITED

Tool	Description
Native Image Generator (Ngen.exe)	Enables users to improve the performance of .NET applications. The Native Image Generator improves performance by precompiling assemblies into images that contain processor-specific machine code. The CLR can then run the precompiled images instead of using just-in-time (JIT) compilation. Alternatively, if you use JIT compilation, your code is compiled just before it is executed.
MSIL Disassembler (Ildasm.exe)	Enables users to manipulate assemblies, such as determining whether an assembly is managed, or disassembling an assembly to view the compiled MSIL code.
Strong Name Tool (Sn.exe)	Enables users to sign assemblies with strong names. The Strong Name Tool includes commands to create a new key pair, extract a public key from a key pair, and verify assemblies.

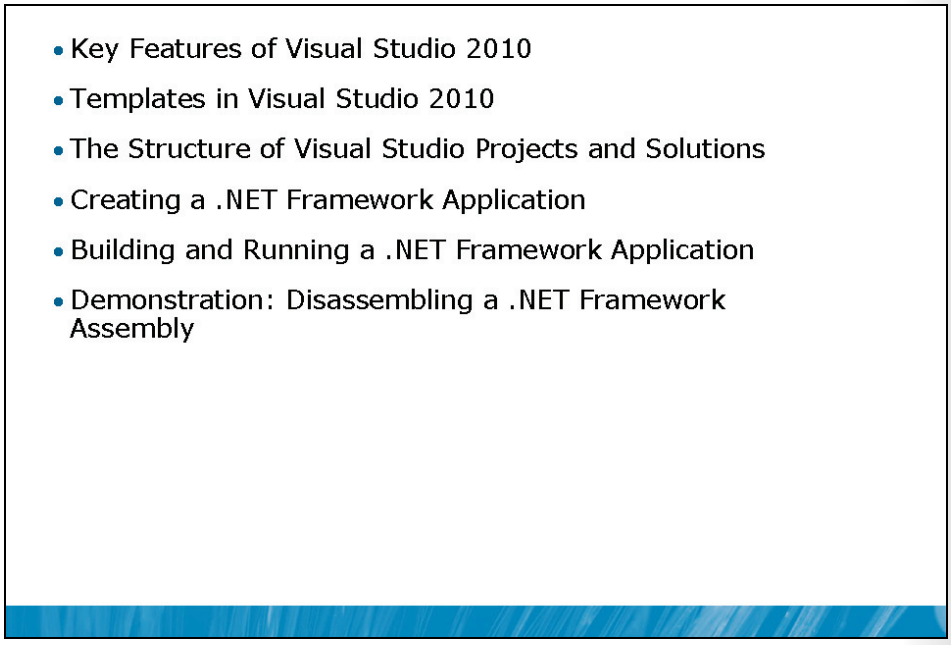
Question: You have created two applications that both use an assembly called **Contoso.ReportGenerator.dll**. Both applications will run on the same machine. What is the best approach to share the **Contoso.ReportGenerator.dll** assembly and which tool would you use?

Additional Reading

For more information about the tools that the .NET Framework provides, see the .NET Framework Tools page at http://go.microsoft.com/fwlink/?LinkId=192882.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Creating Projects Within Visual Studio 2010



This lesson introduces you to Visual Studio 2010 and describes how it can help simplify the development of .NET applications through the use of predefined application templates, and features of the integrated development environment (IDE).

Objectives

After completing this lesson, you will be able to:

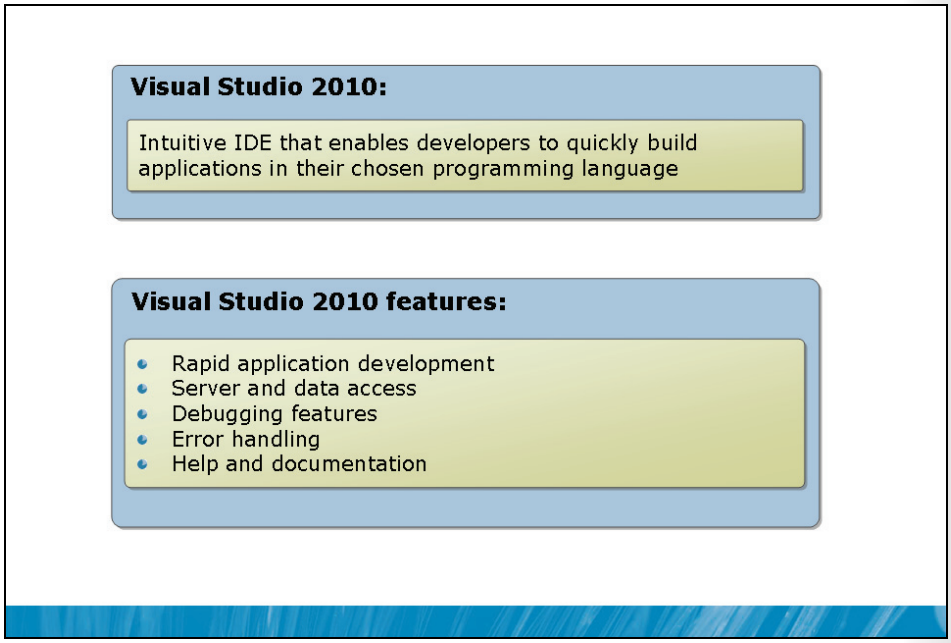
- Describe the features that are available in Visual Studio 2010 that aid programming productivity.
- Describe the various project types that Visual Studio 2010 supports and when to use them.
- Describe the primary files that are found in most Visual Studio solutions.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain how to create a console application by using the Console Application template in Visual Studio 2010.
- Use Visual Studio to compile and run an application.

MCT USE ONLY. STUDENT USE PROHIBITED

Key Features of Visual Studio 2010



Key Points

Visual Studio 2010 presents 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 2010 are:

- *Intuitive integrated development environment.* The Visual Studio 2010 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 2010 provides design views for graphical components that enable you to build complex user interfaces easily. Alternatively, you can use the Code Editor views, which provide more control. Visual Studio 2010 also provides wizards that help speed up the development of particular components.

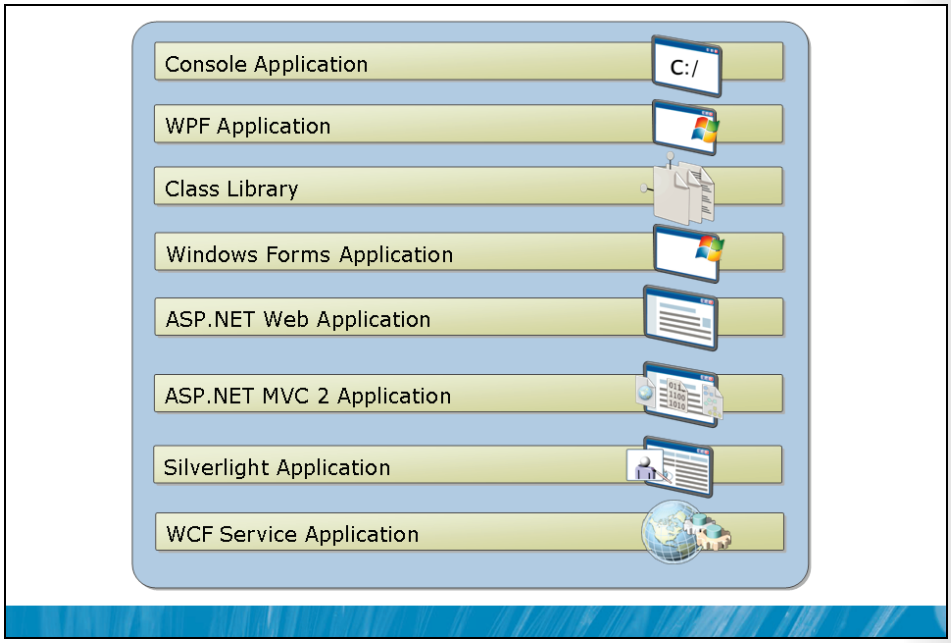
MCT USE ONLY. STUDENT USE PROHIBITED

- *Server and data access.* Visual Studio 2010 provides the Server Explorer, which enables you to log on to servers and explore their databases and system services. It provides a familiar way to create, access, and modify databases that your application uses.
- *Debugging features.* Visual Studio 2010 provides a debugger, which enables you to step through local or remote code, pause at breakpoints, and follow execution paths.
- *Error handling.* Visual Studio 2010 provides the Error List window, which displays any errors, warnings, or messages that are produced as you edit and build your code.
- *Help and documentation.* Visual Studio 2010 also provides help and guidance through Microsoft IntelliSense®, code snippets, and the integrated help system, which contains documentation and samples.

Question: What are the main reasons why you may choose Visual Studio 2010 over a text editor such as Notepad++?

MCT USE ONLY. STUDENT USE PROHIBITED

Templates in Visual Studio 2010



Key Points

Visual Studio 2010 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 2010 provides several 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 a functioning application.
- Include supporting components and controls that are relevant to the project type.
- Configure the Visual Studio 2010 IDE to the type of application that you are developing.
- Add references to any initial assemblies that this type of application usually requires.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Types of Templates

The following table describes some of the common application templates that you can use when you develop .NET Framework applications by using Visual Studio 2010.

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 compared to the Windows Forms application template 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.
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.
Windows Forms Application	Provides the environment settings, tools, project references, and starter code to build a graphical Windows Forms application.
ASP.NET Web Application	Provides the environment settings, tools, project references, and starter code to create a server-side, compiled ASP.NET Web application.
ASP.NET MVC 2 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.
Silverlight Application	Provides the environment settings, tools, project references, and starter code to build a rich, graphical Web application.

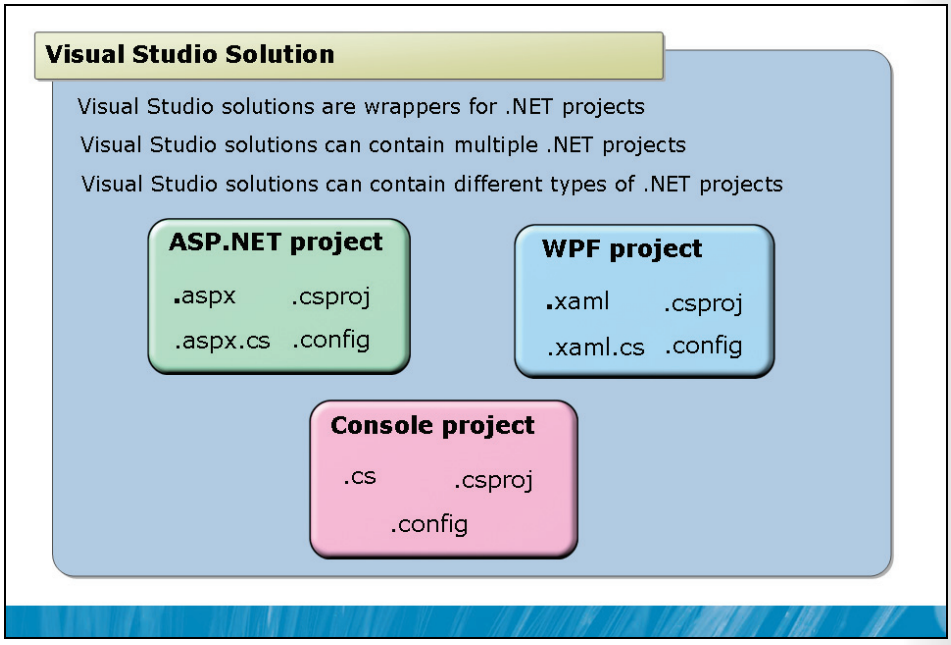
Template	Description
WCF Service Application	Provides the environment settings, tools, project references, and starter code to build Service Orientated Architecture (SOA) services.

Question: What project templates would you use for each of the following:

- A client application that will run on a Windows-based computer.
- A library of functionality that you want to use in other applications.
- A Web site that you will host on an Internet Information Services (IIS) Web server.

MCT USE ONLY. STUDENT USE PROHIBITED

The Structure of Visual Studio Projects and Solutions



Key Points

Visual Studio 2010 uses solutions and projects as conceptual containers to organize your source files during development. Categorizing your source files in this way simplifies the build and deployment process for your .NET Framework applications.

Visual Studio Projects

A project is used to organize source files, references, and project-level configuration settings that make up a single .NET Framework application or library. When you create a project in Visual Studio, the project is automatically organized into a solution.

The following table describes some of the common file types that you will find in a Visual Studio project.

MCT USE ONLY. STUDENT USE PROHIBITED

File	Description
.cs	Code files that can belong to a single project solution. This type of file can represent any of the following: • Modules • Windows Forms files • Class files
.csproj	Project files that can belong to multiple project solutions. The .csproj file also stores settings for the project, such as the output path for the build output and the target platform.
.aspx	Files that represent ASP.NET Web pages. An ASP.NET file can contain your Visual C# code or you can use an accompanying .aspx.cs file to store your code in addition to the page markup.
.config	Configuration files are XML-based files that you can use to store application-level settings such as database connection strings, which you can then modify without recompiling your application.
.xaml	XAML files are used in WPF and Microsoft Silverlight® applications to define user interface elements.

Visual Studio Solutions

A single Visual Studio solution is a container for one or more projects. By default, when you create a new project, Visual Studio automatically creates a solution for the project. You can add additional projects to a solution. This is useful if, for example, you are building a library assembly and an application that tests this library. You can build and compile both projects as part of the same solution rather than having to run multiple instances of Visual Studio.

A solution can also contain project-independent items that any of the projects in the solution can use. For example, an ASP.NET solution can contain a single cascading style sheet (.css) file that applies a standard look and feel to any of the included ASP.NET projects.

Categorizing multiple projects into a single Visual Studio solution provides the following advantages:

- It enables you to work on multiple projects within a single Visual Studio 2010 session.
- It enables you to apply configuration settings globally to multiple projects.

MCT USE ONLY. STUDENT USE PROHIBITED

- It enables you to deploy multiple projects within a single solution.

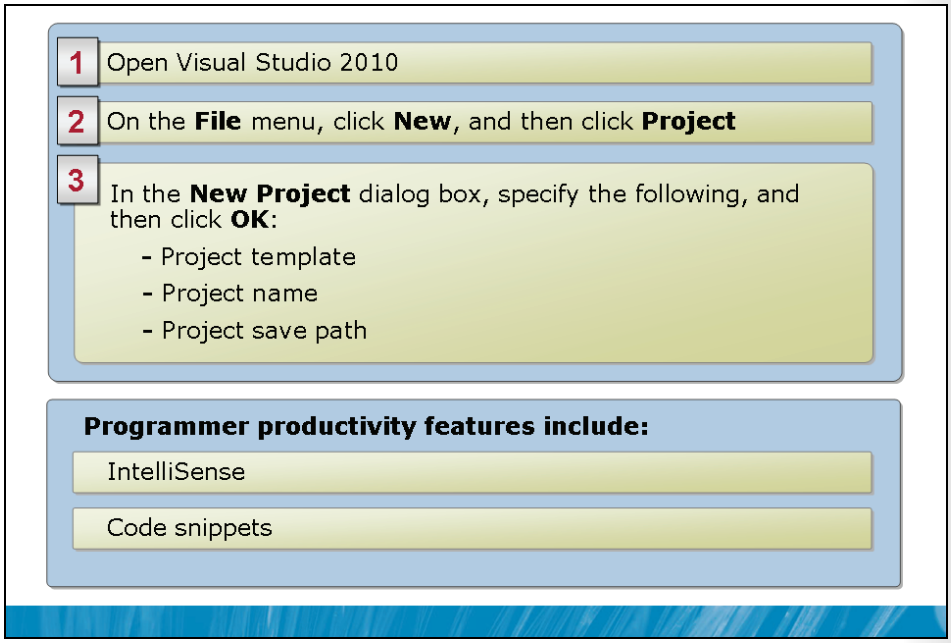
The following table describes the solution definition files.

File	Description
.sln	A Visual Studio 2010 solution file that provides a single point of access to multiple projects, project items, and solution items. The .sln file is a standard text file, but it is not recommended to change it outside Visual Studio 2010.
.suo	A solution user options file that stores any settings that you have changed to customize the Visual Studio 2010 IDE.

Question: What role does the .sln file play in Visual Studio solutions?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating a .NET Framework Application



Key Points

The application templates that Visual Studio 2010 provides 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.

▶ Create a new console project by using the Console Application template in Visual Studio 2010

1. Open Visual Studio 2010.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **New Project** dialog box, specify the following settings for the project, and then click **OK**:
   a. In the **Installed Templates** list, under **Visual C#**, click **Windows**.
   b. In the center pane, click **Console Application**.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Programmer Productivity Features

Visual Studio 2010 provides a host of features that can help you to write code. When writing code, developers 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 developers need directly from the editor. IntelliSense provides the following features:

- *Quick Info.* The Quick Info option displays the complete declaration for any identifier in your code. Move the mouse so that the cursor rests on an identifier to display Quick Info for that identifier, which appears in a yellow pop-up box.
- *Complete Word.* The Complete Word option types the rest of a variable, command, or function name after you have entered 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.

Often, when you are building a .NET Framework application, you will need to repeat common constructs in your code. Examples might be a loop, or code to handle exceptions. Code snippets are designed to ease the burden of having to implement such common code by providing boilerplate code templates that can be readily inserted into your code and amended to suit your needs. You can access these code snippets by using the Code Snippet Picker.

You can manage code snippets by using the **Code Snippet Manager** dialog box, which is available on the **Tools** menu. The Code Snippet Manager enables you to add new code snippets by specifying new folders that the Code Snippet Picker will look in for code snippets; by importing code snippets; or by searching for code snippets online. The Code Snippets Manager is also useful for discovering the shortcut key sequence that is associated with a code snippet.

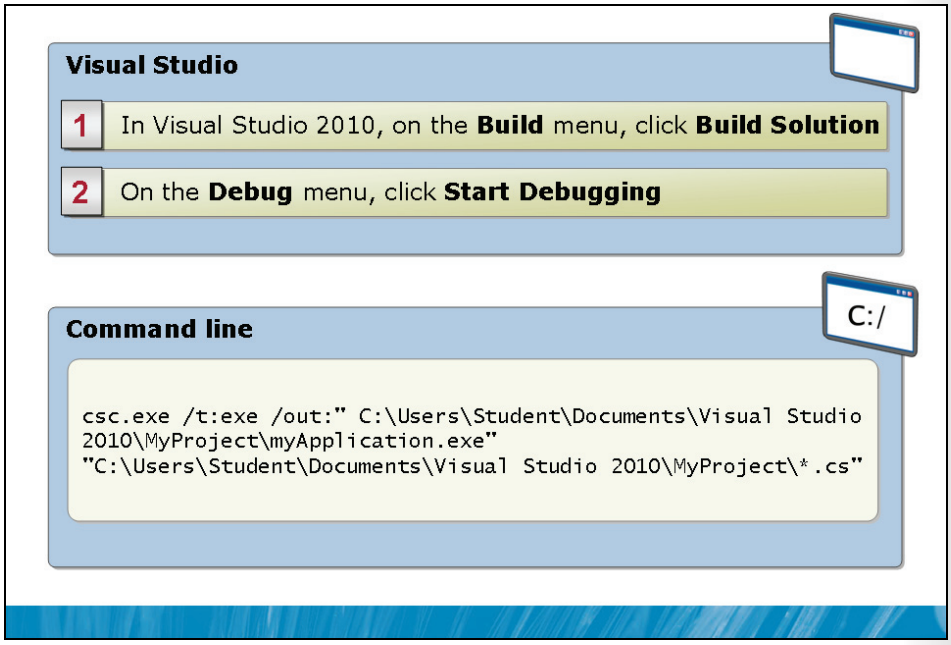
Finally, Visual Studio 2010 provides a host of other features on the shortcut menu that appears when you right-click a code statement. These include Refactor, Organize, Create Unit Tests, Go To Definition, Find All References, and Outline. These features will be covered in more detail in later modules.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What is the purpose of code snippets?

MCT USE ONLY. STUDENT USE PROHIBITED

Building and Running a .NET Framework Application



Key Points

Visual Studio provides an integrated environment that enables you to quickly compile and run your applications.

You can also build and run an application from the command line if you do not have Visual Studio available.

The following steps describe how to build and run an application.

▶ Build and run an application in Visual Studio 2010

The following steps assume that you have created a new console application.

1. In Visual Studio 2010, on the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Debugging**.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Build an application from the command line

The following steps assume that you have created a new console application called MyProject, which is saved in the C:\Users\Student\Documents\Visual Studio 2010\MyProject\ folder.

1. Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt (2010)**.
2. In the Visual Studio Command Prompt window, type the text in the following code example, and then press ENTER.

```
csc.exe /t:exe /out:"C:\Users\Student\Documents\Visual Studio
2010\MyProject\myApplication.exe" "C:\Users\Student\Documents\Visual
Studio 2010\MyProject\*.cs"
```

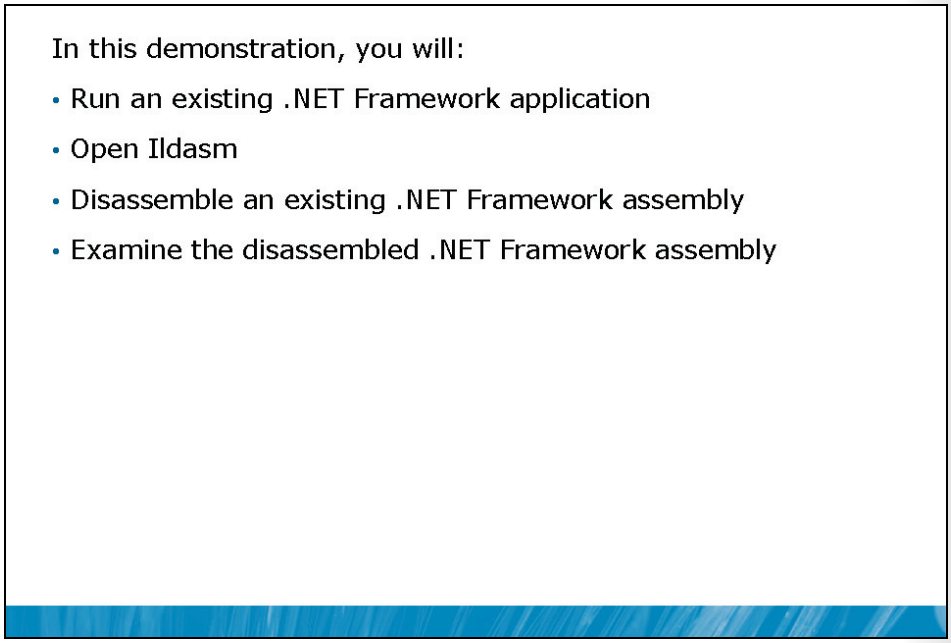
3. Right-click the **Start** menu, click **Open Windows Explorer**, and then move to **C:\Users\Student\Documents\Visual Studio 2010\MyProject**.

   The MyProject folder should now contain the myApplication.exe executable assembly, which you can run.

Question: Describe two ways to build and run a .NET Framework application.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Disassembling a .NET Framework Assembly



Key Points

- Run an existing .NET Framework application.
- Open Ildasm.
- Disassemble an existing .NET Framework assembly.
- Examine the disassembled .NET Framework assembly.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Run MyFirstApplication.exe in the **E:\Demofiles\Mod1\Demo1** folder, and examine the applications output.
3. Close MyFirstApplication.exe.
4. Run ildasm.exe in the **C:\Program Files\Microsoft SDKs \Windows\v7.0A\bin** folder.

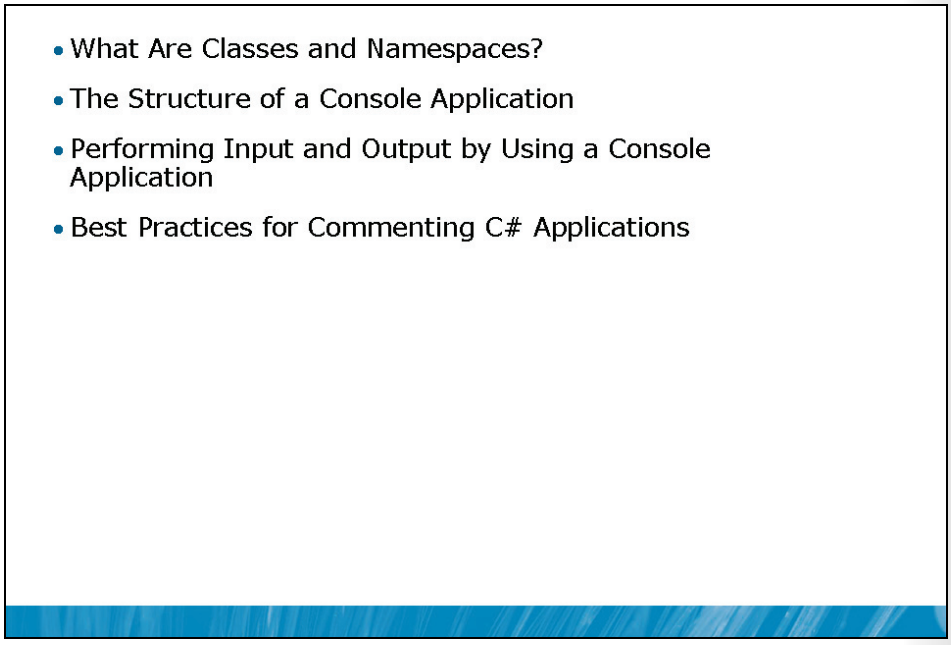
MCT USE ONLY. STUDENT USE PROHIBITED

5. Using ildasm, open the MyFirstApplication.exe in the **E:\Demofiles\Mod1\Demo1** folder, and then inspect the contents of the **MyFirstApplication** assembly. Examine the following items:
   - The public key token and the version number in the assembly **Manifest**.
   - The constructor and **Main** method in the **MyFirstApplication.Program** node.
6. Close ildasm.exe.

Question: When developing a .NET Framework application, how would you find Ildasm useful?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Writing a C# Application



This lesson describes the structure of a simple C# application, and how a C# application contains one or more classes. This lesson describes how to reference functionality that is defined in classes in other assembles and libraries, and how you can use the **Console** class in the .NET Framework class library to perform simple input and output operations. Finally, this lesson explains how and why you should add comments to your applications.

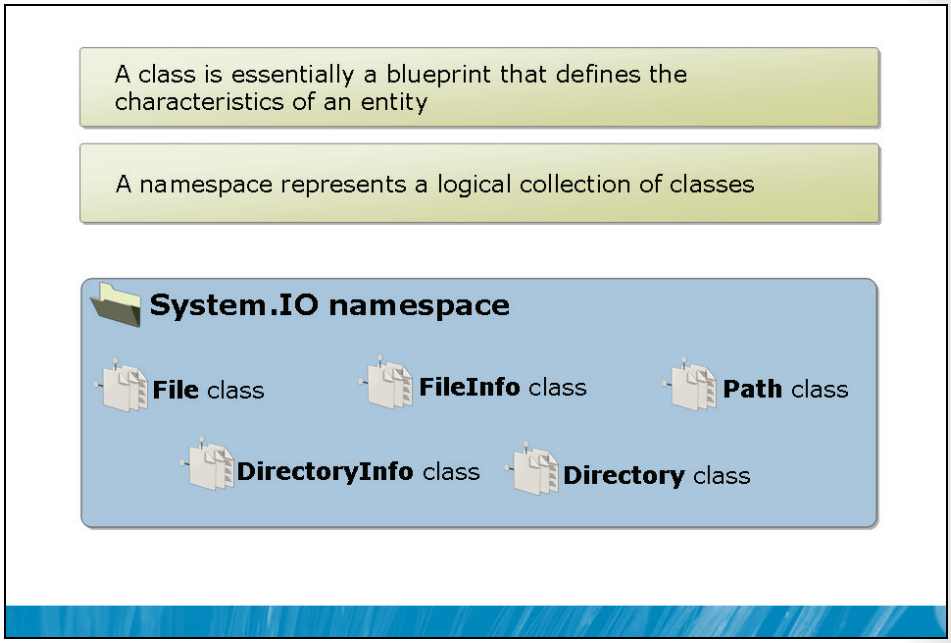
Objectives

After completing this lesson, you will be able to:

- Describe how Visual C# uses namespaces and classes.
- Describe the structure of an application.
- Perform input and output operations by using methods that the **Console** class provides.
- Apply best practices commenting a Visual C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Classes and Namespaces?



Key Points

Visual C# is an object-oriented language that uses classes and namespaces to modularize .NET Framework applications into logical components.

A class is essentially a blueprint that defines the characteristics of an entity, and includes properties that define the types of data that the object can contain and methods that describe the behavior of the object. A namespace represents a logical collection of classes. Classes are stored in assemblies, and a namespace is simply a device to disambiguate classes that might have the same name in different assemblies.

For example, the **System.IO** namespace includes the following classes that enable you to manipulate the Windows file system. However, you could create classes with the same name under your own namespace:

- **File**
- **FileInfo**
- **Directory**

MCT USE ONLY. STUDENT USE PROHIBITED

- **DirectoryInfo**
- **Path**

To use a class that is defined in the .NET Framework, perform the following tasks:

1. Add a reference to the assembly that contains the compiled code for the class.
2. Bring the namespace that contains the class into scope.

If you are writing a .NET Framework application to write text to a new file on the file system, you can bring the **System.IO** namespace into scope and then use the **WriteAllText** method of the **File** class.

To bring a namespace into scope in a Visual C# application, you can use the **using** statement. The following code example shows how to bring the **System**, **System.IO**, and **System.Collections** namespaces into scope.

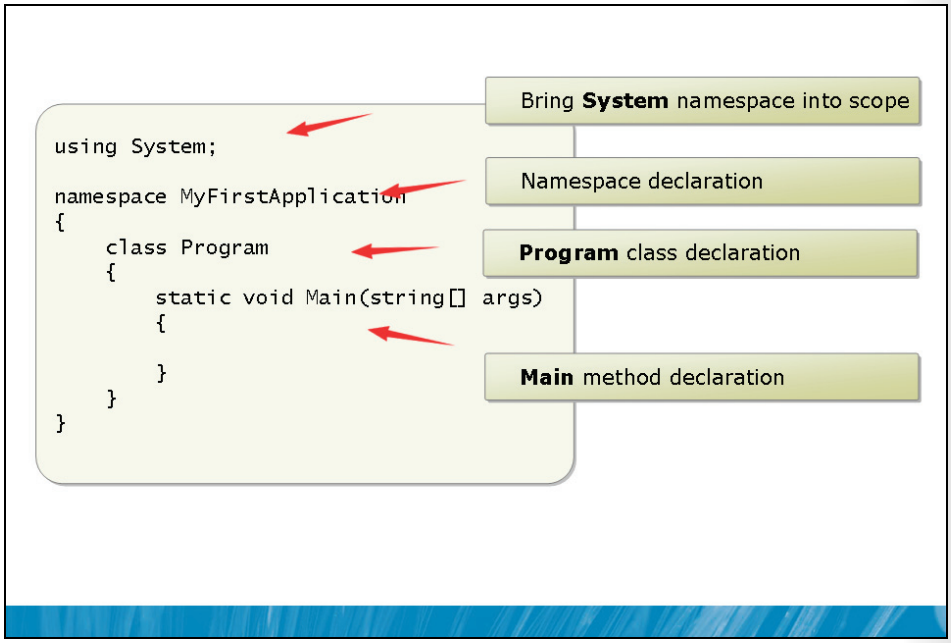
```
using System;
using System.IO;
using System.Collections;
```

The **using** statement is simply a convenience and you can manage without it. For example, you can use **System**.**Console** rather than **Console**.

Question: In your console application, you want to use the **Console** class, which is part of the **System** namespace. How do you bring the **System** namespace into scope?

MCT USE ONLY. STUDENT USE PROHIBITED

The Structure of a Console Application



Key Points

When you create a new console application by using the Console Application template, Visual Studio 2010 performs the following tasks:

- It creates a new .csproj file to represent the console project and structure all of the default components in a console project.
- It adds references to the assemblies in the .NET Framework class library that console applications most commonly require. This set of assemblies includes the **System** assembly.
- It creates the Program.cs file with a **Main** method, which provides an entry point into the console application.

The Program.cs file that Visual Studio 2010 creates resembles the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
using System;

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

        }
    }
}
```

The following table describes the code items in the Program.cs file.

Code item	Description
`using System;`	Brings the **System** namespace into scope.
`namespace MyFirstApplication { ... }`	Defines a new namespace called **MyFirstApplication**. Typically, in a new project, this defaults to the project name.
`class Program { ... }`	Defines a new **internal** class called **Program**.
`static void Main(string[] args) { ... }`	Defines a new **private static Main** method with a **void** return type that accepts a parameter of type *string array*.

What Is the Main Method?

Every .NET Framework application that compiles into an executable file must have a **Main** method. This method provides the CLR with an entry point into the application. When you run a .NET Framework application, the **Main** method is the first method that the CLR executes.

MCT USE ONLY. STUDENT USE PROHIBITED

When you develop your .NET Framework applications, it is good practice to keep the **Main** method lightweight, and let it serve as just an entry point, not a container for most of the logic in your application.

The **Main** method has the following significant characteristics:

- It is **private**. This means that it is not visible to other classes outside the **Program** class.
- It uses the **static** key, so it can be called without creating an instance of the **Program** class.
- It uses the **void** return type, so it is a method that does not return data.
- It accepts data in the form of a *string array*. Therefore, when you run the console application, any command-line arguments that you provide will be available in the *args* parameter.

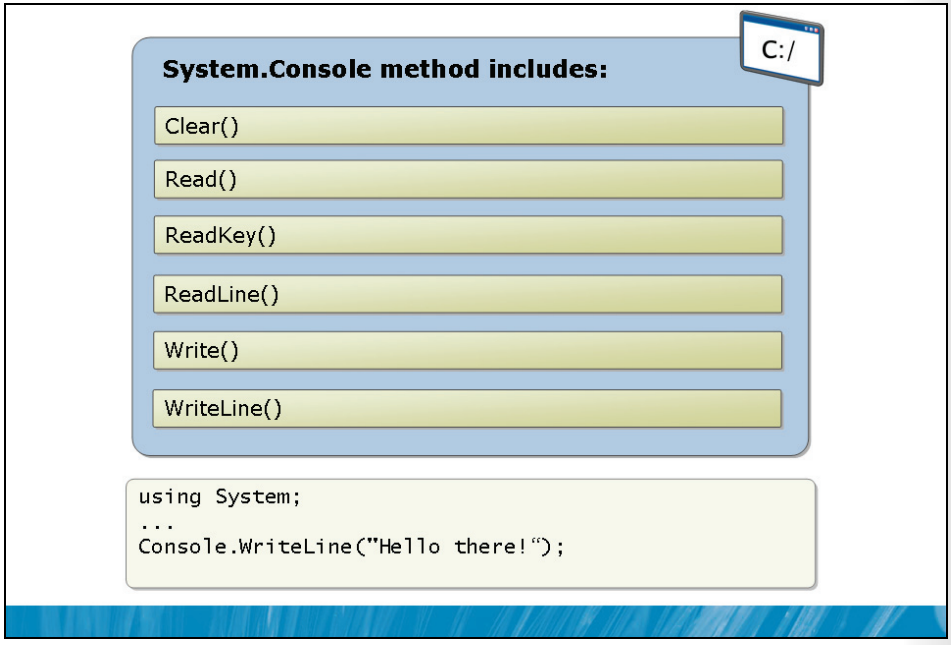
Question: In your console application, you have a method called **Main**. What is the purpose of the **Main** method?

Additional Reading

For more information about command-line arguments, see the Main() and Command-Line Arguments (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192889.

MCT USE ONLY. STUDENT USE PROHIBITED

Performing Input and Output by Using a Console Application



Key Points

The **System** namespace provides the **Console** class, which contains several methods that enable you to add basic console I/O functionality to an application, such as accepting input and displaying data.

The following table describes some of the key methods that the **Console** class provides.

Method	Description

Clear()	Clears the console window and console buffer of any data. The following code example provides an example of this.
	`using System;`
	`...`
	`Console.Clear(); // clears the console display`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description

Read()	Reads the next character from the console window. The following code example provides an example of this.
	`using System;`
	`...`
	`int nextCharacter = Console.Read();`

ReadKey()	Reads the next character or key press from the console window. The following code example provides an example of this.
	`using System;`
	`...`
	`ConsoleKeyInfo key = Console.ReadKey();`

ReadLine()	Reads the next line of characters from the console window. The following code example provides an example of this.
	`using System;`
	`...`
	`string line = Console.ReadLine();`

Write()	Writes the text to the console window. The following code example provides an example of this.
	`using System;`
	`...`
	`Console.Write("Hello there!");`

WriteLine()	Writes the text followed by a line break to the console window. The following code example provides an example of this.
	`using System;`
	`...`
	`Console.WriteLine("Hello there!");`

Question: Which two methods would you use to do the following:

- Display the message "**Please press any key**" on a new line.
- Capture the key that the user pressed.

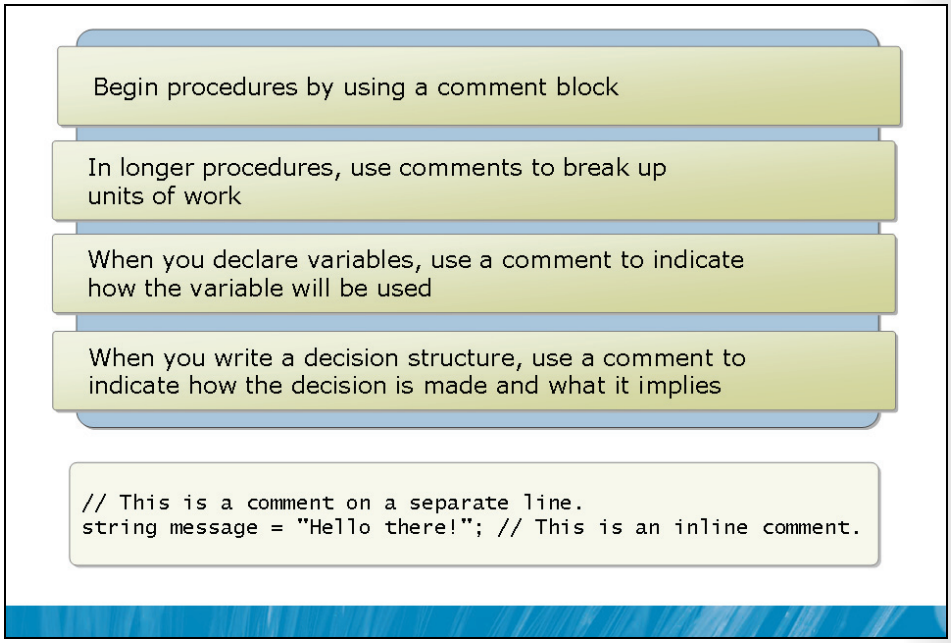
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about the **Console** class, see the Console Class page at http://go.microsoft.com/fwlink/?LinkId=192883.

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices for Commenting C# Applications



Key Points

It is good programming practice to begin all procedures with a brief comment that describes the functional characteristics of the procedure. This is for your own benefit and the benefit of anyone else who examines the code.

In Visual C#, comments begin with two slash marks (//). Comments can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code example.

```
// This is a comment on a separate line.
string message = "Hello there!"; // This is an inline comment.
```

The Comment and Uncomment Toolbar Buttons

You can add or remove comment symbols for a block of code by selecting the lines of code and choosing the **Comment** or **Uncomment** buttons on the **Text Editor** toolbar.

MCT USE ONLY. STUDENT USE PROHIBITED

Commenting Guidelines

As your code becomes more complex, use comments to make your code more readable and easier to maintain. You should use comments to explain the purpose of a section of code in natural language, especially when the purpose might not be obvious or clear.

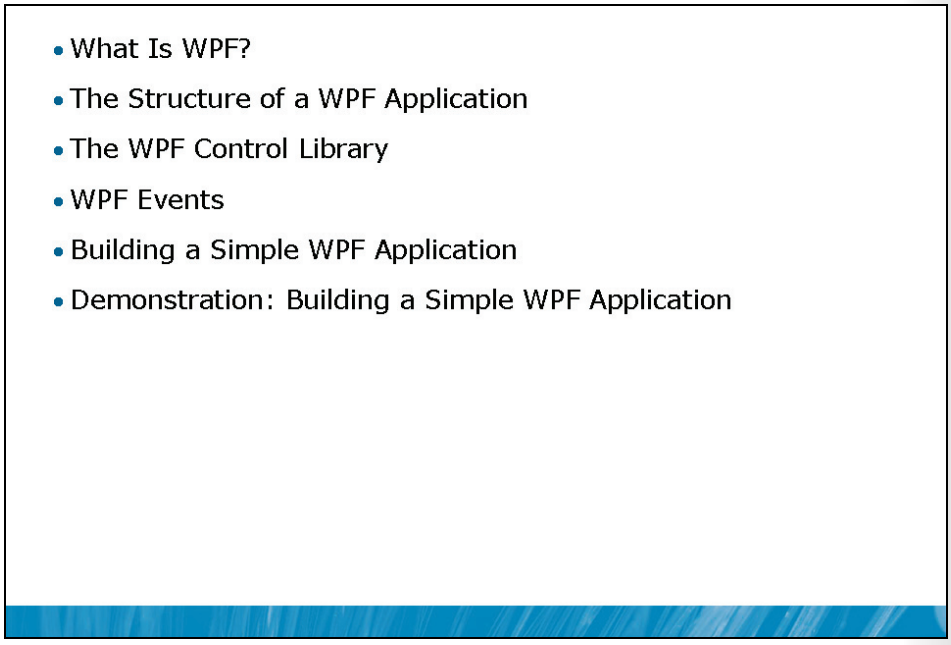
The following list provides some guidelines regarding when you should comment your code:

- Begin procedures with a comment block. This block should include information such as the purpose of the procedure, the value returned, the arguments, and so on.
- In longer procedures, use comments to break up units of work within the procedure.
- When you declare variables, use a comment to indicate how the variable will be used.
- When you write a decision structure, use a comment to indicate how the decision is made and what it implies.

Question: Why is it important for you to comment your code?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Building a Graphical Application



This lesson introduces you to applications that have a graphical user interface, and provides the example of a WPF application.

This lesson also explains what WPF is, how WPF applications are structured, and how you can create your own WPF applications by using Visual Studio 2010.

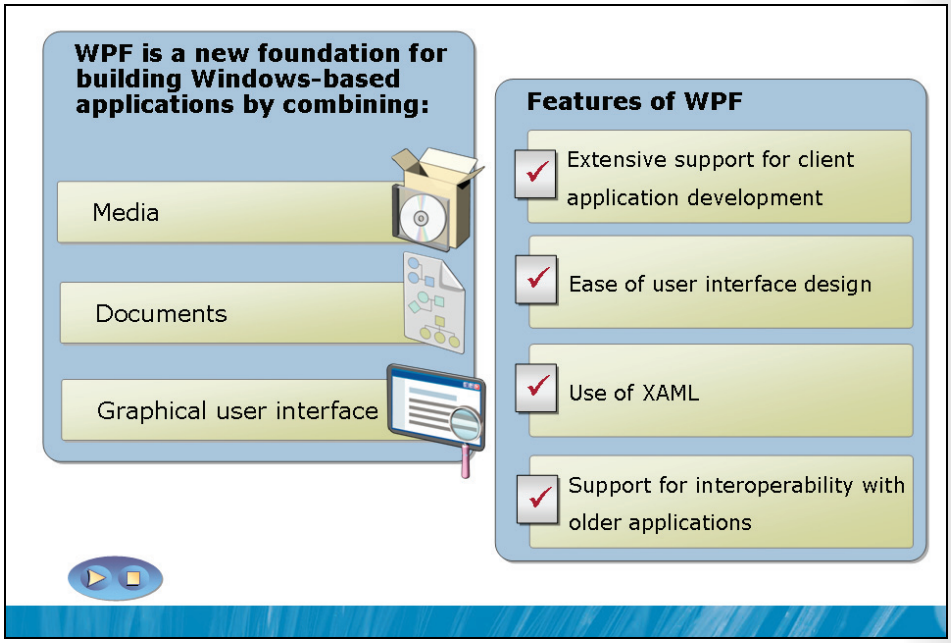
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of WPF.
- Describe the structure of a WPF application.
- Describe the controls that WPF provides, and how to set control properties.
- Describe the concept of events, and how WPF controls use events.
- Explain how to build a simple WPF application by using Visual Studio 2010.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is WPF?



Key Points

Windows Presentation Foundation is the unified graphical subsystem for Windows that provides the foundation for building applications and high-fidelity experiences. It unifies how Windows creates, displays, and manipulates documents, media, and user interfaces. This enables you to create visually stunning user experiences.

Features of Windows Presentation Foundation

The main features of Windows Presentation Foundation are:

- *Extensive support for client application development.* Developers can create eye-catching, highly functional applications. WPF includes several text-rendering features such as OpenType and TrueType.
- *Ease of user interface design.* WPF provides a set of built-in controls. It uses the concept that there is a logical separation of a control from its appearance, which is generally considered to be a good architectural principle.

MCT USE ONLY. STUDENT USE PROHIBITED

- *Use of XAML.* XAML enables developers to use an XML-based model to declaratively manipulate the object model. XAML is faster and easier to implement than procedural code. XAML is used to define the user interface in a WPF application.
- *Support for interoperability with older applications.* Developers can use WPF inside existing Win32 code or existing Win32 code inside WPF.

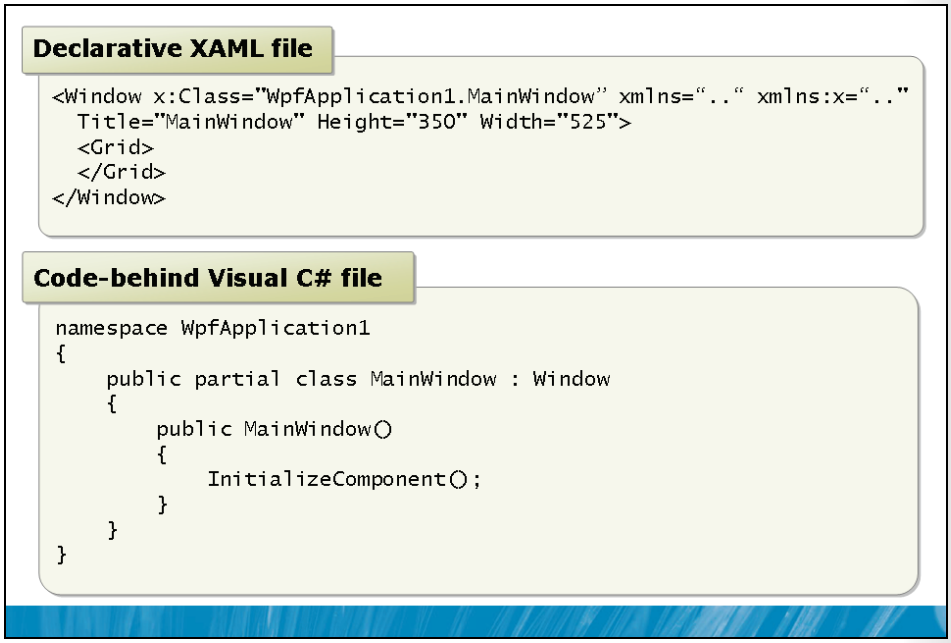
Question: Why would you choose to use WPF to create an application instead of Windows Forms?

Additional Reading

For more information about what WPF is, see the Introduction to WPF page at http://go.microsoft.com/fwlink/?LinkId=192884.

MCT USE ONLY. STUDENT USE PROHIBITED

The Structure of a WPF Application



Key Points

When you create a new WPF application by using the WPF Application template, Visual Studio 2010 performs the following tasks:

- It creates a new .csproj file to represent the WPF project and structure all of the default components in a WPF project.
- It adds references to the necessary assemblies, which include the **PresentationCore**, **PresentationFramework**, **System**, **System.Core**, and **System.Xaml** assemblies.
- It creates the App.xaml markup file and an App.xaml.cs code-behind file, which you can use to define application-level resources and functionality.
- It creates the MainWindow.xaml markup file and the MainWindow.xaml.cs code-behind file, which you use as a starting point to building your first WPF window.

The default markup that is generated in the MainWindow.xaml markup file is shown in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
<Window x:Class="WpfApplication1.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

This markup defines a simple window with a default title, width, and height. You can change these properties by editing the XAML code, or by using the Properties window in Visual Studio. You can also change these properties dynamically, by using code when the application runs. The **Grid** control governs the layout of controls that you add to the window. If you want to use an alternative layout, you can replace the markup for the **Grid** control with a different layout control.

The default markup that is generated in the App.xaml markup file is shown in the following code example.

```
<Application x:Class="WpfApplication1.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

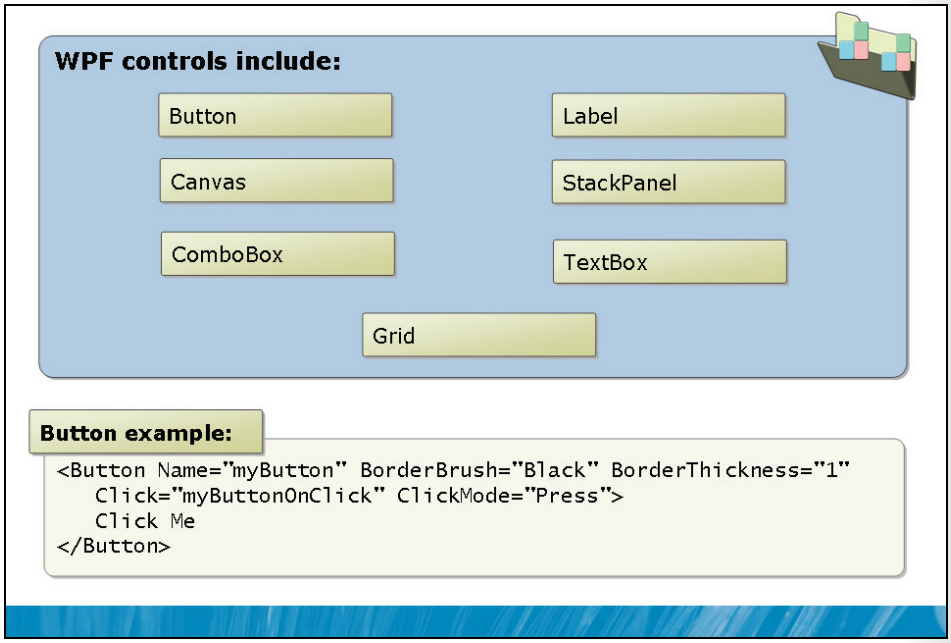
Note that the **Application** element contains a **StartupUri** attribute that points to the window that you want to open when the application runs.

Both the App.xaml and MainWindow.xaml markup files use XAML to represent resources and user interface elements. XAML is a markup language for declarative application programming. Using the XAML markup at design time enables you to separate the user interface design from the application logic, which is stored in code-behind files. XAML directly represents the instantiation of managed objects.

Question: Can you think of any other markup languages that behave in a similar way to XAML?

MCT USE ONLY. STUDENT USE PROHIBITED

The WPF Control Library



Key Points

WPF includes a rich library of controls that you can use to build your WPF applications. The controls that are included in the library are common user interface components that you would typically find in every Windows-based application, such as the button and the text box. You can also define your own custom controls.

WPF Common Controls

The following table describes some of the commonly used controls in the WPF control library. It also provides a simple XAML example for each, showing the common properties that you can set at design time.

MCT USE ONLY. STUDENT USE PROHIBITED

Control	Description	XAML example

Button	The **Button** control represents a typical clickable button that you would find in most Windows applications.	`<Button Name="myButton"`
		`  BorderBrush="Black"`
		`  BorderThickness="1"`
		`  Click="myButtonOnClick"`
		`  ClickMode="Press">`
		
		`  Click Me`
		`</Button>`

Canvas	The **Canvas** control represents a layout panel that enables you to position child controls absolutely.	`<Canvas Background="Black"`
		`  Height="200" Width="200">`
		`  <!-- Child controls -->`
		`</Canvas>`

ComboBox	The **ComboBox** control represents a drop-down list that a user can scroll through and make a selection from.	`<ComboBox Name="myComboBox">`
		`  <ComboBoxItem>`
		`  Item a`
		`  </ComboBoxItem>`
		`  <ComboBoxItem>`
		`  Item b`
		`  </ComboBoxItem>`
		`</ComboBox>`

Grid	The **Grid** control represents a flexible table that can contain multiple columns and rows. You typically use the **Grid** control to position child controls.	`<Grid ShowGridLines="True"`
		`  Width="200" Height="200">`
		`  <Grid.ColumnDefinitions>`
		`  <ColumnDefinition />`
		`  <ColumnDefinition />`
		`  </Grid.ColumnDefinitions>`
		`  <Grid.RowDefinitions>`
		`  <RowDefinition />`
		`  </Grid.RowDefinitions>`
		`  <!-- Child controls -->`
		`</Grid>`

Label	The **Label** control represents a read-only text block that you could use to display some static text.	`<Label Name="myLabel">`
		`  Hello`
		`</Label>`

MCT USE ONLY. STUDENT USE PROHIBITED

Control	Description	XAML example

StackPanel	The **StackPanel** control enables you to stack child controls horizontally or vertically.	`<StackPanel`
		`  Name="myStackPanel"`
		`  Orientation="Vertical">`
		`  <Label>Item 1</Label>`
		`  <Label>Item 2</Label>`
		`  <Label>Item 3</Label>`
		`</StackPanel>`

TextBox	The **TextBox** control represents an editable field that you can use to display and capture text.	`<TextBox Name="myTextBox">` `</TextBox>`

Note that you can also define controls dynamically by using Visual C# in your code-behind file.

WPF Control Properties

Each control in WPF has an associated set of properties that you can use to define the appearance and behavior of a control. For example, most controls have a **Height** property and a **Width** property that specify the dimensions of the control, and a **Margin** property that indicates where the control should appear relative to the layout control it is contained within.

You can set control properties:

- In the XAML window declaratively by editing the XAML directly.
- In the Properties window. This approach modifies the XAML definition of a control on your behalf.
- At run time, by using Visual C# code. This approach does not change the XAML definition of any controls.

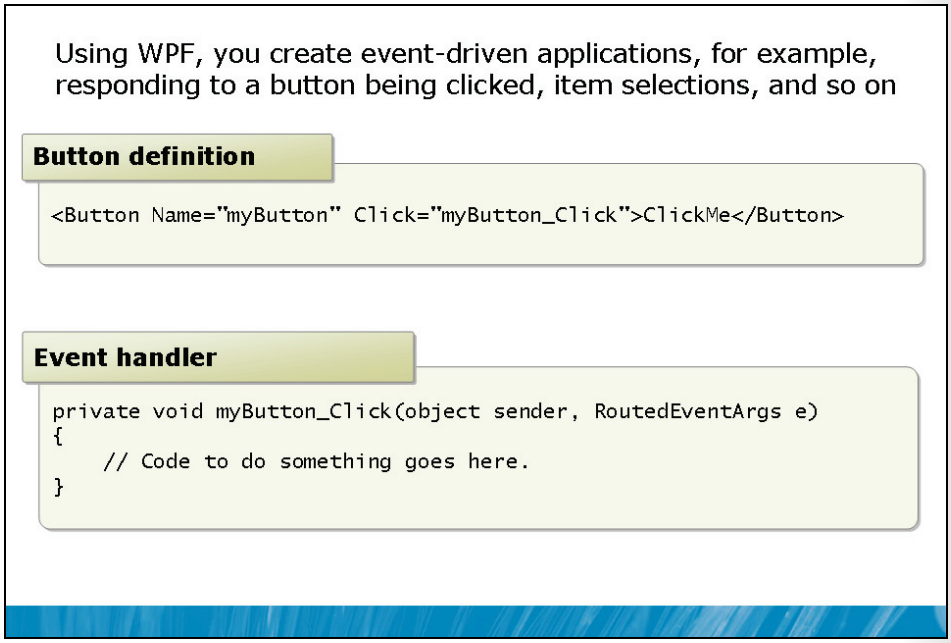
Question: You are building a simple form to capture user credentials and enable users to log on. Which controls could you use to build this form?

Additional Reading

For more information about the controls in the WPF control library, see the Control Library page at http://go.microsoft.com/fwlink/?LinkId=192886.

MCT USE ONLY. STUDENT USE PROHIBITED

WPF Events



Key Points

When you create a WPF, ASP.NET, or Windows Forms application in Visual Studio 2010, you create an event-driven application. Event-driven applications execute code in response to an event. Each form and control that you create exposes a predefined set of events. When one of these events occurs, and there is code in the associated event handler, that code is invoked.

Handling Events

You can specify the events that a control responds to at design time by editing the XAML definition of a control (you specify the event and the name of an event-handling method to run when the event occurs). Alternatively, you can use the **Events** tab in the Properties window (this technique modifies the XAML definition of a control automatically).

You must provide the methods that handle the events by using code in the code-behind file.

The following code examples show the XAML markup for a **Button** control with a **Click** event handler, and the C# code that defines the event handler. When the

MCT USE ONLY. STUDENT USE PROHIBITED

user clicks the button, the **myButton_Click** method is called. The parameters to the **myButton_Click** method are defined by WPF, and they are populated with information about the button and the event at run time.

```
[XAML control declaration]
<Button Name="myButton" Click="myButton_Click">ClickMe</Button>
```

```
[Visual C# event handler]
private void myButton_Click(object sender, RoutedEventArgs e)
{
    // Code to do something goes here.
}
```

The following code examples show how you can define a closing event handler for a **Window** control.

```
[XAML control declaration]
<Window x:Class="WpfApplication.MainWindow" Name="myWindow"
    xmlns="..."
    xmlns:x="..."
    Title="MainWindow" Height="350" Width="525"
    Closing="myWindow_Closing">
</Window>
```

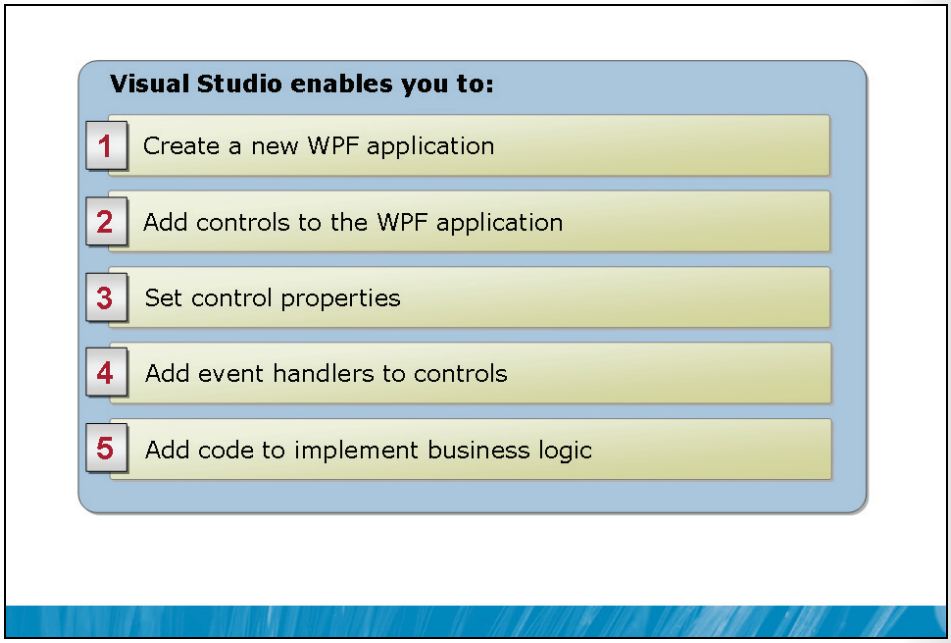
```
[Visual C# event handler]
private void myWindow_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Code to do something goes here.
}
```

Question: When you develop your WPF applications, what two ways can you use to specify events for controls?

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

Building a Simple WPF Application



Key Points

You can create a WPF application in Visual Studio 2010 by using the WPF Application template.

▶ Create a new WPF application

1. Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
2. In Visual Studio 2010, on the **File** menu, click **New**, and then click **Project**.
3. In the **New Project** dialog box, perform the following, and then click **OK**:
   - In the center pane, click **WPF Application**.
   - In the **Name** box, type a name for your WPF application.
   - In the **Location** box, type a path where you would like to save your project.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Add controls to the WPF application

1. On the **View** menu, click **Toolbox**.
2. In the Toolbox window, double-click the control that you want to add to your application.
3. You can then use the Design window or the XAML window to customize the control.

▶ Set control properties

1. In the Design window, click the control that you want to customize.
2. You can then set the properties as follows:
   - Switch to the XAML window, and then edit the XAML directly.
   - Switch to the Properties window, and then set the predefined properties.



Note: You can also set properties in Visual C# by using the Code Editor window.

▶ Add event handlers to controls

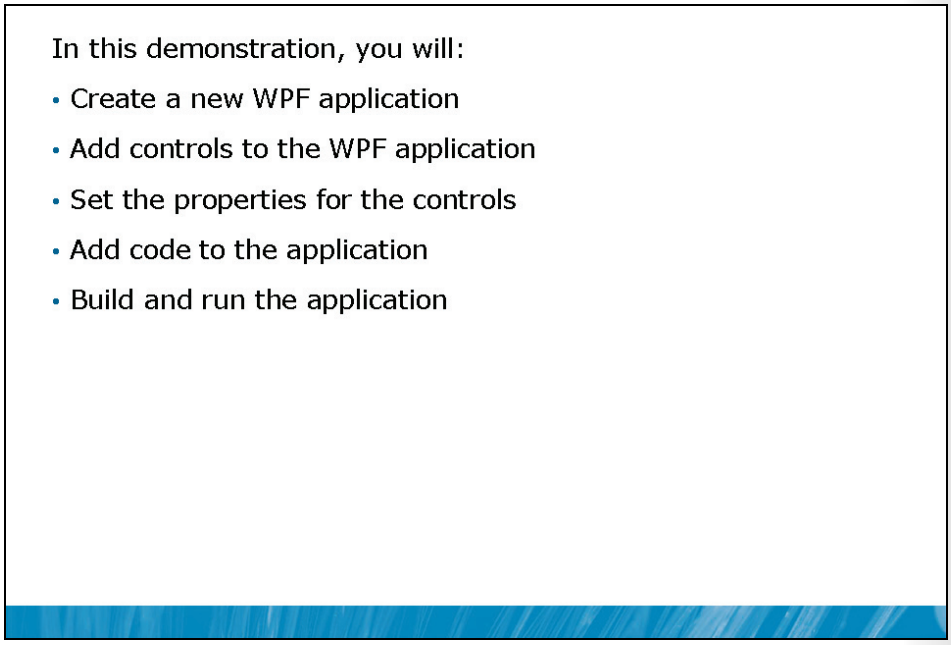
1. In the Design window, click the control that you want to add an event handler to.
2. In the Properties window, on the **Events** tab, double-click the event that you want to add, for example, a **Click** event handler for a button.

▶ Add code to the WPF application

1. In the Solution Explorer window, right-click the XAML file that you want to add code to, and then click **View Code**.
2. You can then use the Code Editor window to define the logic behind your controls.

Question: What windows in Visual Studio 2010 do you typically use when you are building your applications?

Demonstration: Building a Simple WPF Application



Key Points

- Create a new WPF application.
- Add controls to the WPF application.
- Set the properties for the controls.
- Add code to the application.
- Build and run the application.

Demonstration Steps

1. Open **Microsoft Visual Studio 2010**.
2. In Visual Studio 2010, create a new project with the following characteristics:
   - **Type**: WPF Application
   - **Name**: MyFirstWpfApp

MCT USE ONLY. STUDENT USE PROHIBITED

- **Location**: E:\Demofiles\Mod1\Demo2\Starter

3. Use the Toolbox to add a button control to the application.
4. Examine the XAML mark-up generated by Visual Studio 2010.
5. Use the Properties window to set the following properties for the button control:
   - **FontSize**: 20
   - **Height**: 50
   - **Width**: 150
6. Use the XAML window to perform the following:
   - In the **Button** element, set the **Content** attribute to **Click Me**.
   - In the **Window** element, set the **Height** attribute to **150**.
   - In the **Window** element, set the **Width** attribute to **190**.
7. Use the **Events** tab in the Properties window to generate a **Click** event handler for the button control.
8. Open the **MainWindow.xaml.cs** file, and in the **ClickMeButton_Click** method add the following code:

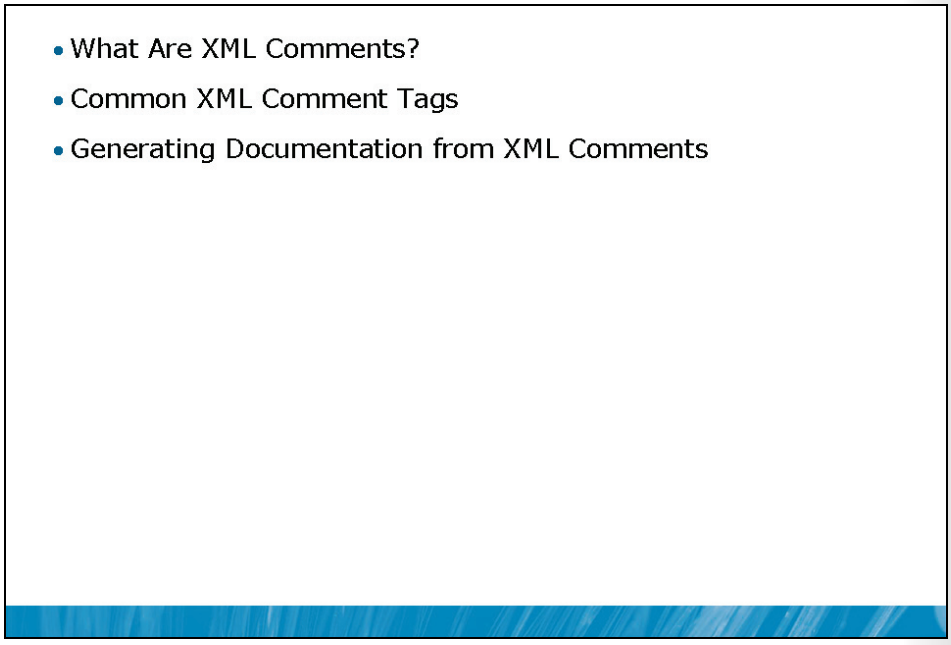
```
...
private void ClickMeButton_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("You clicked me!!");
}
...
```

9. Build and run the application.

Question: When you are developing a WPF application in Visual Studio 2010, what are the two main ways in which you can set properties for WPF controls?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 5
Documenting an Application



This lesson introduces XML comments and explains how you can use them when you are developing your .NET applications. This lesson also shows how to build a formatted help file by using the Sandcastle tool.

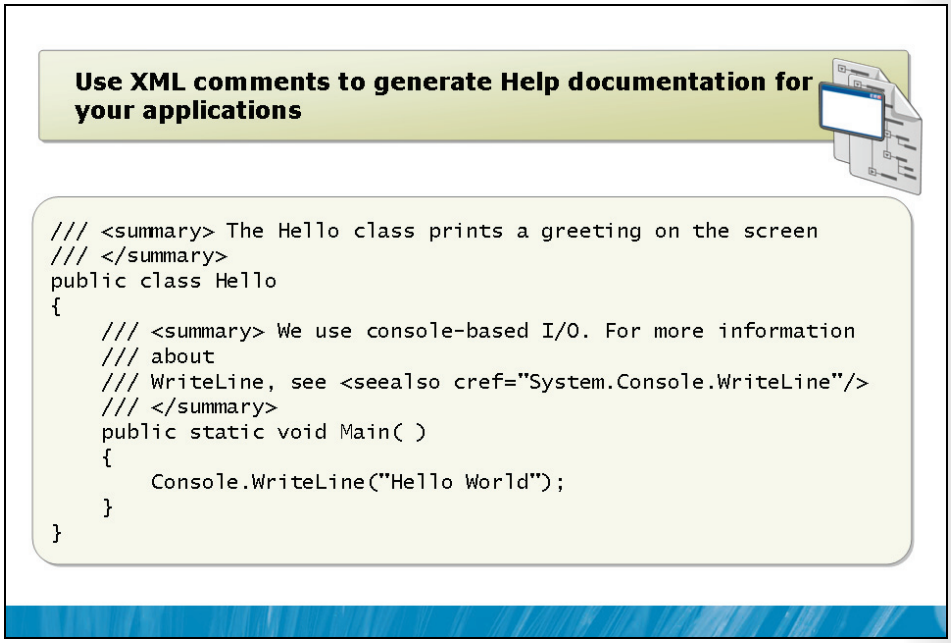
Objectives

After completing this lesson, you will be able to:

- Describe what XML comments are and how you can use them in .NET applications.
- Describe some of the commonly used XML comment tags.
- Explain how to generate an XML documentation file and how to use Sandcastle to generate a formatted help file by using this XML documentation file.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are XML Comments?



Key Points

In Visual Studio 2010, you can add comments to your source code that will be processed to an XML file. This file can then be the input to a process that creates Help documentation for the classes in your code. You can also use an XML file to support IntelliSense on your component.

Inline comments are part of the Visual C# standard, whereas XML comments are a Microsoft extension and are typically used by third-party tools such as Sandcastle Help File Builder.

XML Documentation Comments

Documentation comments in Visual C# begin with three slash marks (///) followed by an XML documentation tag.

In the following code example, the **Hello** class contains <summary> and <seealso> documentation tags.

MCT USE ONLY. STUDENT USE PROHIBITED

```
/// <summary> The Hello class prints a greeting on the screen
/// </summary>
public class Hello
{
    /// <summary> We use console-based I/O. For more information about
    /// WriteLine, see <seealso cref="System.Console.WriteLine()"/>
    /// </summary>
    public static void Main( )
    {
        Console.WriteLine("Hello World");
    }
}
```

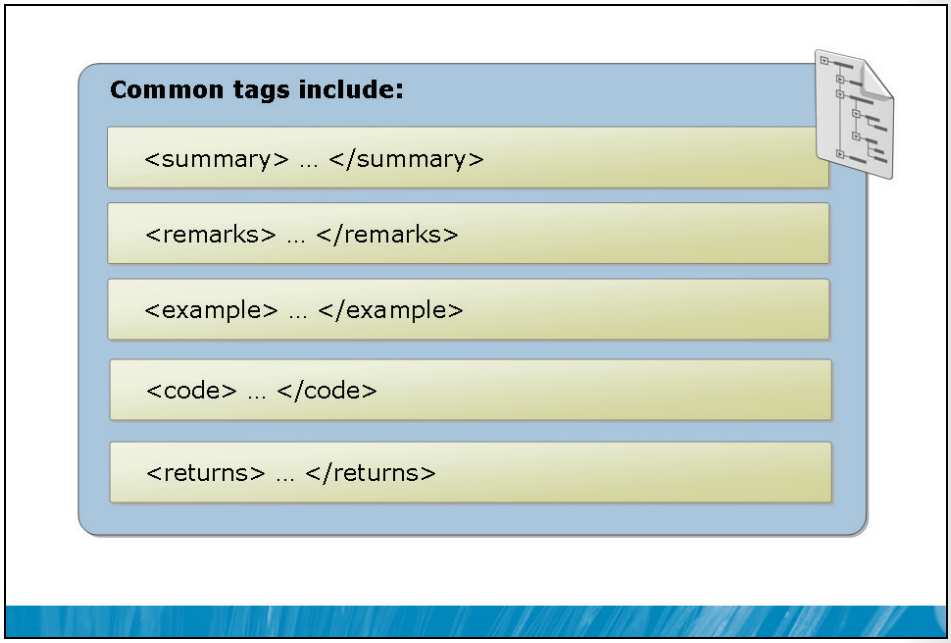
Question: Why would you use XML comments rather than standard comments?

Additional Reading

For more information about XML comments, see the XML Documentation Comments (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192887.

MCT USE ONLY. STUDENT USE PROHIBITED

Common XML Comment Tags



Key Points

There are several suggested XML tags that you can use. You can also create your own custom tags. The following table shows some XML tags and their uses.

Tag	Purpose
<summary> ... </summary>	Provides a brief description. Use the <remarks> tag for a longer description.
<remarks> ... </remarks>	Provides a detailed description. This tag can contain nested paragraphs, lists, and other types of tags.
<example> ... </example>	Provides an example of how a method, property, or other library member should be used. It often involves the use of a nested <code> tag.
<code> ... </code>	Indicates that the enclosed text is application code.
<returns> ... </returns>	Documents the return value and type of a method.

MCT USE ONLY. STUDENT USE PROHIBITED

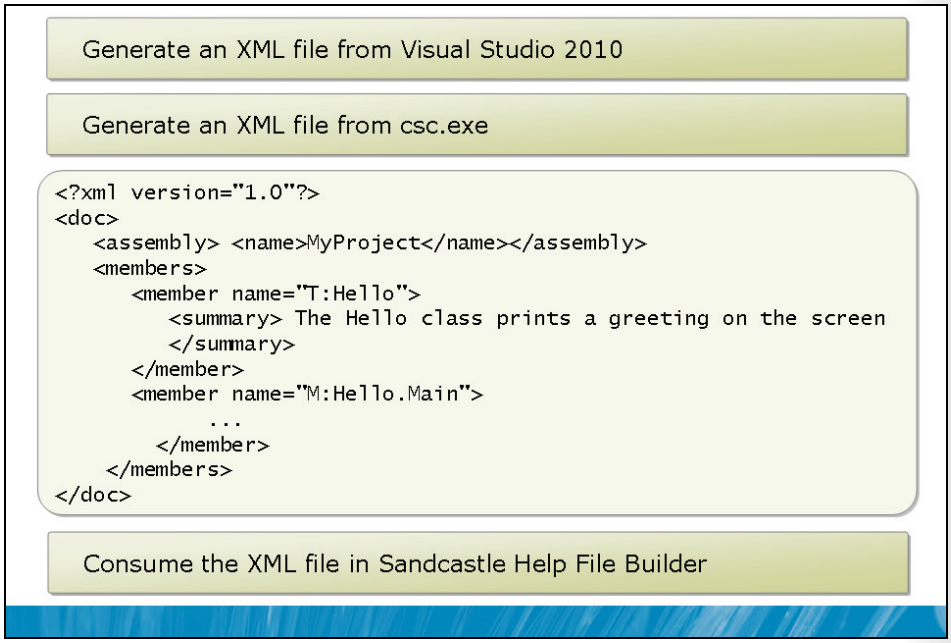
Question: Which tag would you use to provide a detailed description of a method?

Additional Reading

For more information about XML comment tags, see the Recommended Tags for Documentation Comments (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192888.

MCT USE ONLY. STUDENT USE PROHIBITED

Generating Documentation from XML Comments



Key Points

You can compile the XML tags and documentation into an XML file by selecting the **XML documentation file** check box in the Properties window for a project, or by using the /**doc** command-line switch when you build an application that has embedded XML comments.

If there are no errors, you can view the XML file that is generated by using an application such as Windows Internet Explorer®, and you can generate a help file by using a tool such as Sandcastle.

Note: Sandcastle is not provided as part of Visual Studio, but it is available separately from the CodePlex Web site.

▶ Generate an XML file by using Visual Studio 2010

1. In Solution Explorer, right-click a project, and then click **Properties**.
2. In the Properties window, on the **Build** tab, select the **XML documentation file** check box.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Generate an XML file by using csc.exe

1. Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt (2010)**.
2. In the Visual Studio Command Prompt (2010) window, type the command in the following code example.

```
csc.exe /t:exe /doc:"C:\Users\Student\Documents\Visual Studio
2010\MyProject\myComments.xml" /out:"C:\Users\Student\Documents\Visual
Studio 2010\MyProject\myApplication.exe"
"C:\Users\Student\Documents\Visual Studio 2010\MyProject\*.cs"
```



Note: The **/doc** switch instructs the compiler to generate an XML file that contains the XML comments.

The XML that the compiler generates should resemble the following code example.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>MyProject</name>
    </assembly>
    <members>
        <member name="T:Hello">
            <summary> The Hello class prints a greeting on the screen
            </summary>
        </member>
        <member name="M:Hello.Main">
            <summary> We use console-based I/O. For more information
                      about WriteLine,
                      see <seealso  cref="M:System.Console.WriteLine"
                      />
            </summary>
        </member>
    </members>
</doc>
```

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Generate a .chm file by using Sandcastle Help File Builder

Now that you have an XML file that contains the comments that were extracted from your project, you can create a .chm file by using a tool such as Sandcastle Help File Builder.

1. Click **Start**, point to **All Programs**, click **Sandcastle Help File Builder**, and then click **Sandcastle Help File Builder GUI**.
2. In Sandcastle Help File Builder, on the **File** menu, click **New Project**.
3. In the **Save New Help Project As** dialog box, perform the following, and then click **Save**:
   a. Browse to the path where you want to save the project.
   b. Specify a name for the Sandcastle project.
4. In the Project Explorer window, right-click **Documentation Sources**, and then click **Add Documentation Source**.
5. In the **Select the documentation source(s)** dialog box, browse to the XML file folder, and then click **Open**.
6. On the **Documentation** menu, click **Build Project**.

   Wait for the project to successfully build. This will take a minute.

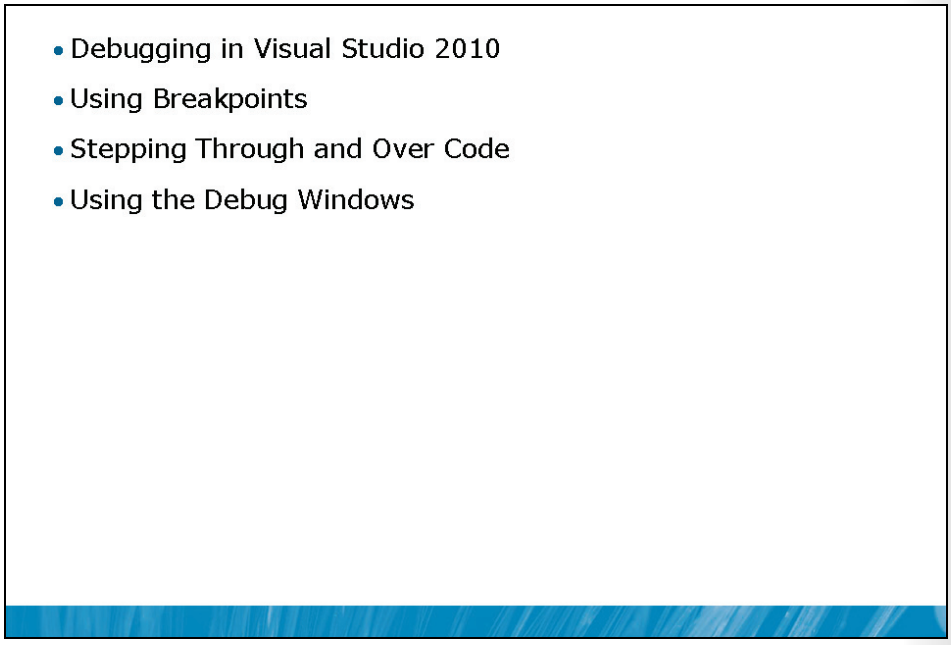
Question: Which switch do you need to provide to get csc.exe to produce XML output?

Additional Reading

For more information about Sandcastle Help File Builder, see the Sandcastle Help File Builder page at http://www.codeplex.com/SHFB.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 6
Debugging Applications by Using Visual Studio 2010



In this lesson, you will learn how to use Visual Studio 2010 to help you debug your applications. You will learn how to use the **Debug** toolbar, breakpoints, and debug windows to examine your application and step through application code at run time.

Objectives

After completing this lesson, you will be able to:

- Describe the functions that Visual Studio 2010 provides to aid debugging.
- Explain how to set, disable, enable, and remove breakpoints.
- Explain how to step into, step over, and step out of code.
- Describe how to use the debug windows to examine information about an application.

MCT USE ONLY. STUDENT USE PROHIBITED

Debugging in Visual Studio 2010



Key Points

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 specific circumstances that you do not test for. Users may report these errors to you, and you will have to correct them.

Visual Studio 2010 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. To access the numerous debug functions, including the ability to step through code line by line, you can use the controls on the **Debug** menu, the controls on the **Debug** toolbar, and keyboard shortcuts.

Debug Controls

The following table lists the main debug controls on the **Debug** menu and the **Debug** toolbar, and the corresponding keyboard shortcuts.

MCT USE ONLY. STUDENT USE PROHIBITED

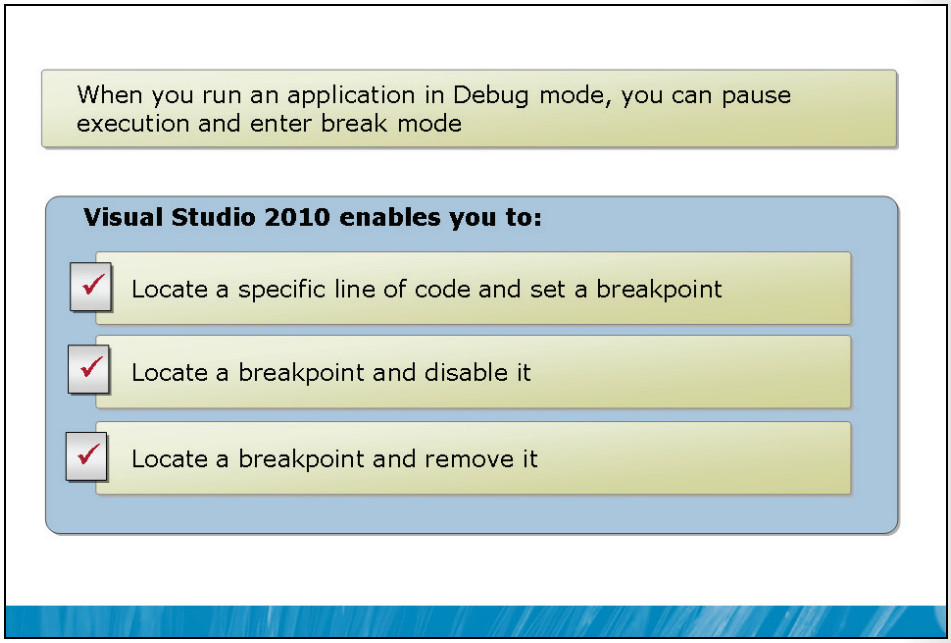
Menu option	Toolbar button	Keyboard shortcut	Description
Start Debugging	**Start/continue**	F5	This button is available when your application is not running and when you are in break mode. It will start your application in Debug mode or resume the application if you are in break mode.
Break All	**Break 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 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 in break mode.
Step Into	**Step into**	F11	This button is used for stepping through code. See the next topic in this lesson.
Step Over	**Step over**	F10	This button is used for stepping through code. See the next topic in this lesson.
Step Out	**Step out**	SHIFT+F11	This button is used for stepping through code. See the next topic in this lesson.
Windows	**Windows**	Various	This button enables access to various debug windows, each of which has its own shortcut key.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What are some of the debug functions that Visual Studio 2010 provides?

MCT USE ONLY. STUDENT USE PROHIBITED

Using Breakpoints



Key Points

When you run an application in Debug mode, you can pause execution and enter break mode. In break mode, no further execution takes place until you restart the application or step through the code line by line. You can also view and change variable values, execute additional code or evaluate expressions, and more. When you are in break mode, the current line of code is indicated by a yellow arrow in the gray bar to the left of the code and by a yellow background for the next statement due to be executed.

The **Break All** debug function enables you to enter break mode. However, this function does not give you much control over exactly where code execution pauses.

Breakpoints enable you to choose exactly where code execution will pause. If you place a breakpoint on a line of code, the application will enter break mode as soon as that line of code is reached, before it executes that line of code.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Set a breakpoint

1. Locate the line of code where you want to set a breakpoint.
2. Add a breakpoint by using one of the following steps:
   a. Click the gray bar to the left of the line of code.
   b. Position the cursor on the line of code, and then press F9.
   c. Position the cursor on the line of code, and then, on the **Debug** menu, click **Toggle Breakpoint**.
   d. Right-click the line of code, point to **Breakpoint**, and then click **Insert Breakpoint**.

   The breakpoint is indicated by a solid red circle in the gray bar to the left of the code and by a red background for the line of code that contains the breakpoint.

▶ Disable or enable a breakpoint

1. Locate a line of code that has an enabled or disabled breakpoint.
2. Disable or enable the breakpoint by using one of the following steps:
   a. Right-click the solid red circle in the gray bar to the left of the line of code, and then click **Disable Breakpoint** or **Enable Breakpoint**.
   b. Right-click the line of code that contains the breakpoint, point to **Breakpoint**, and then click **Disable Breakpoint** or **Enable Breakpoint**.
   c. If the breakpoint is disabled, click the solid red circle to the left of the code to enable it.

   Disabled breakpoints are indicated by a red circle outline in the gray bar to the left of the code and a red outline around the code that contains the breakpoint.

▶ Remove a breakpoint

1. Locate a line of code that has a breakpoint.
2. Remove the breakpoint by using one of the following steps:
   a. If the breakpoint is enabled, click the solid red circle in the code to the left of the code to remove it.
   b. Position the cursor on the line of code, and then press F9.

c. Position the cursor on the line of code, and then, on the **Debug** menu, click **Toggle Breakpoint**.

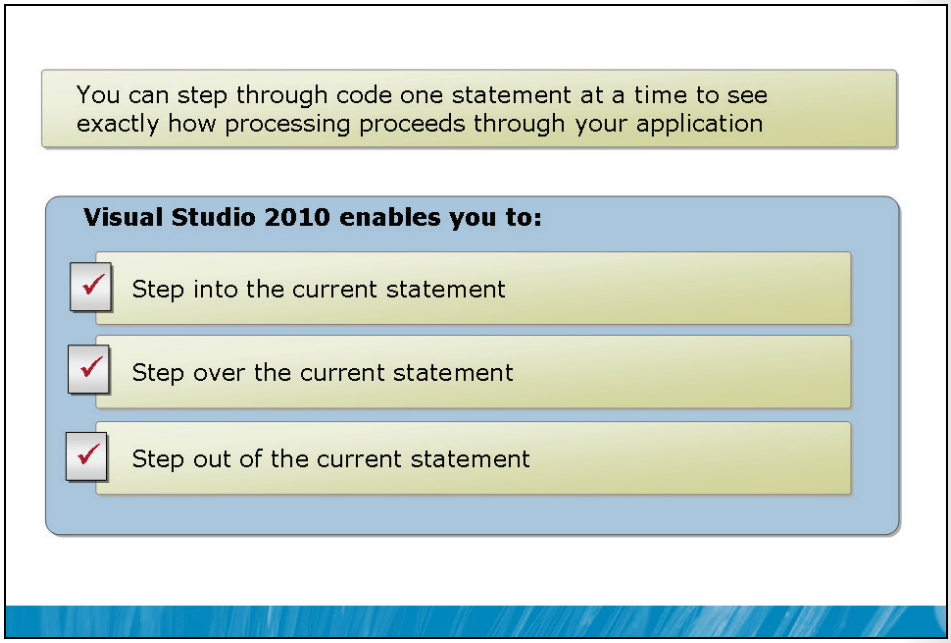
d. Right-click the line of code, point to **Breakpoint**, and then click **Delete Breakpoint**.

e. Right-click the solid red circle in the gray bar to the left of the line of code, and then click **Delete Breakpoint**.

Question: How would you use the debug functions in Visual Studio 2010 to debug your application and pause on a specific line of code?

MCT USE ONLY. STUDENT USE PROHIBITED

Stepping Through and Over Code



Key Points

You can step through code one statement at a time to see exactly how processing proceeds through your application. This is an extremely useful debugging technique because it enables you to test the logic that your application uses. Between statement executions, you can view and edit variable values. Each time your code reaches a branching statement such as a conditional statement, you can verify that the correct code executes and modify the code if it does not.

The various tools that you use to step through code enable you to step through code in exactly the way you want to. You can, for example, step through each line in each method that is executed, or you can ignore the statements inside a method that you know is working correctly. You can also skip over code completely, which prevents some statements from execution.

Step Into, Step Over, and Step Out

There are three debug functions that are essential for stepping through code. These are as follows:

MCT USE ONLY. STUDENT USE PROHIBITED

- **Step into**. This function 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. This also applies to properties. In addition, you can use the **Step into** function to start an application in Debug mode. If you do this, the application will enter break mode as soon as it starts.
- **Step over**. As with **Step into**, the **Step over** function executes the statement at the current execution position. However, this function does not step into code inside a method or property. Instead, the code inside the method or property is executed and the executing position moves to the statement after the method call or property access. The exception to this is where the code for the method or property contains a breakpoint. If this is the case, execution will continue up to the breakpoint.
- **Step out**. The **Step out** function enables you to execute the remaining code in a method, property accessor, or loop. Execution will continue to the statement that called the method or accessed the property, or to the statement following the loop code. Execution will pause at this point.

Skipping Code

In break mode, the next statement to be executed is indicated by a yellow arrow in the gray bar to the left of the code and a yellow background for the statement. You can override this and set a different statement as the next one to execute. To do this, right-click the statement that you want to be executed next, and then click **Set next statement**. The arrow and yellow background will move to the statement that you have chosen.

If you use this technique, you should be aware that you will change the way in which your application works. If you skip important code such as variable assignments or critical method calls, you risk introducing errors that would not otherwise occur. You should skip statements with caution.

Continuing and Restarting

When you have finished stepping through your code, you can return to Debug mode with the **start/continue** functions. Execution will then continue until you

MCT USE ONLY. STUDENT USE PROHIBITED

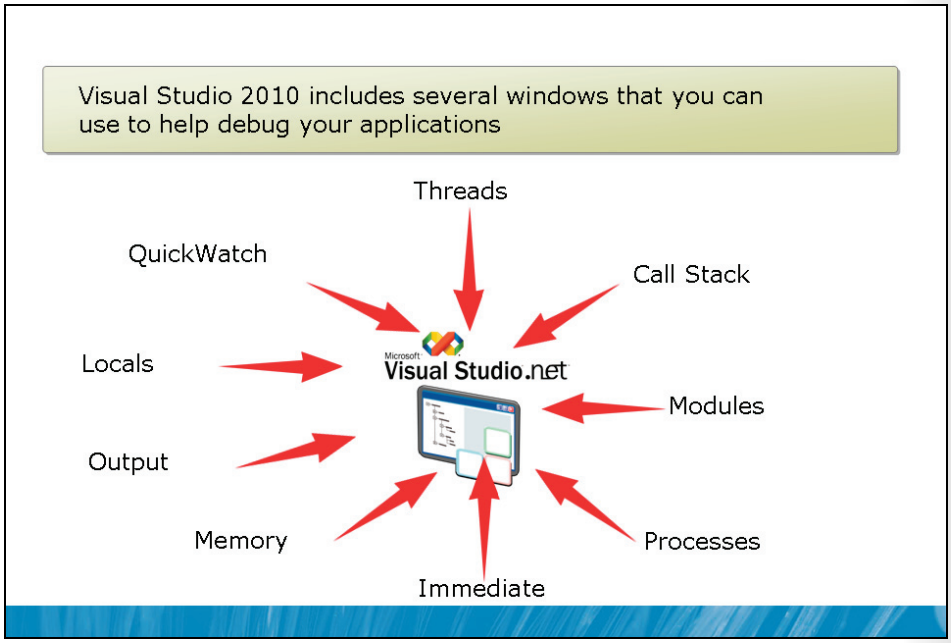
enter break mode again, either with the **Break all** button or if the code encounters a breakpoint.

If you want to terminate the application and then run it again in Debug mode, you can use the **Restart** function. This is useful if you want to test the code that executes when an application first runs or any code that is only executed once when an application is used.

Question: Why would you use the **Step into** and **Step over** debug functions?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Debug Windows



Key Points

Visual Studio 2010 includes several windows that you can use to help debug your applications. These windows are available at run time, mostly in break mode.

The following table describes some of the commonly used debug windows in Visual Studio 2010.

Window	Description
QuickWatch	This is a modal window that enables you to evaluate variables and expressions. Type variable names or expressions in **Expression**, and then click **Reevaluate** to view the value and type of the variable or the result of the expression. Click **Close** to exit the QuickWatch window.
Locals	This window enables you to view and edit local (in-scope) variables. You can expand variables, view members, and edit the contents of some variables in the **Value** column.

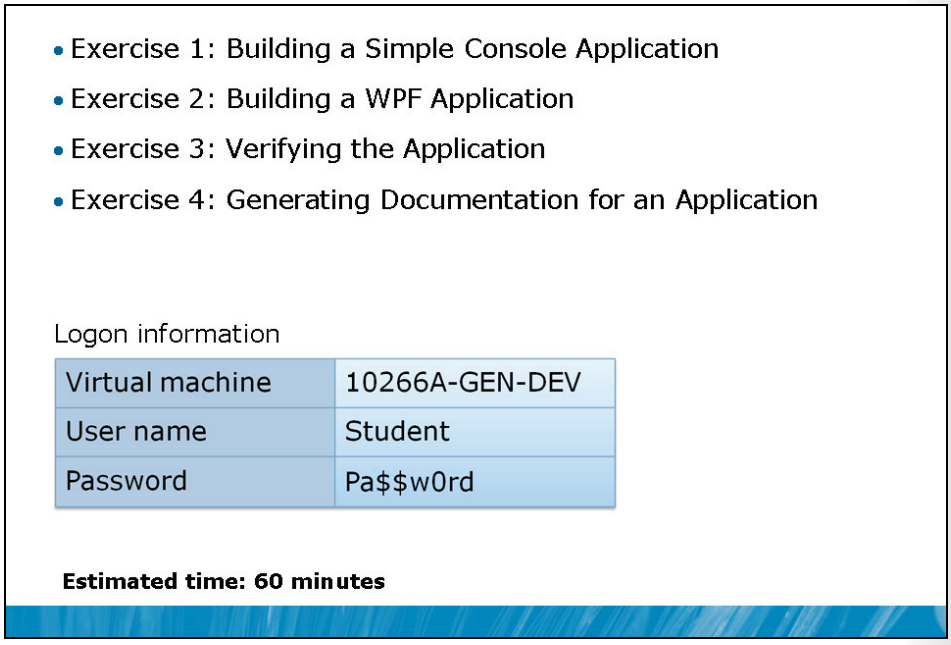
MCT USE ONLY. STUDENT USE PROHIBITED

Window	Description
Immediate	This window enables you to evaluate expressions, execute statements, and print out variable values. You can use this window to issue Visual Studio 2010 commands such as **Debug.Print?** to print the value of a variable or expression.
Output	In this window, you can view error and information messages. One of the main uses of this window is to view traces from your applications by using the **System.Diagnostics.Debug.WriteLine()** method.
Memory	This window enables you to examine and edit the contents of the memory that an application uses. This is an advanced function and can cause your application to behave unpredictably if you do not use this window carefully.
Call Stack	This window enables you to view the stack of method calls that are used to reach the current code location. The current position is shown at the top of the window, and the series of calls that the application has processed to reach this location is shown below.
Modules	This window enables you to view information about the modules (assemblies and executable files) that an application uses. Each module is listed along with its location, version, and other information.
Processes	In this window, you can view information about the processes that the debugger is attached to.
Threads	In this window, you can examine and control threads in an application.

Question: Why would you use the Locals and Immediate windows when developing your application?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Introducing C# and the .NET Framework



Objectives

After completing this lab, you will be able to:

- Create, build, and run a simple console application by using Visual Studio 2010 and C# 4.0.
- Create, build, and run a basic WPF application by using Visual Studio 2010.
- Use the Visual Studio 2010 debugger to set breakpoints, step through code, and examine the values of variables.
- Generate documentation for an application.

Introduction

In this lab, you will create simple console and WPF solutions to get started with using Visual Studio 2010 and C#. You will also configure projects, use code-editing features, and create comments. You will become familiar with the debugger

MCT USE ONLY. STUDENT USE PROHIBITED

interface. You will compile, run, and use the debugger to step through a program. Finally, you will generate documentation for an application.

Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

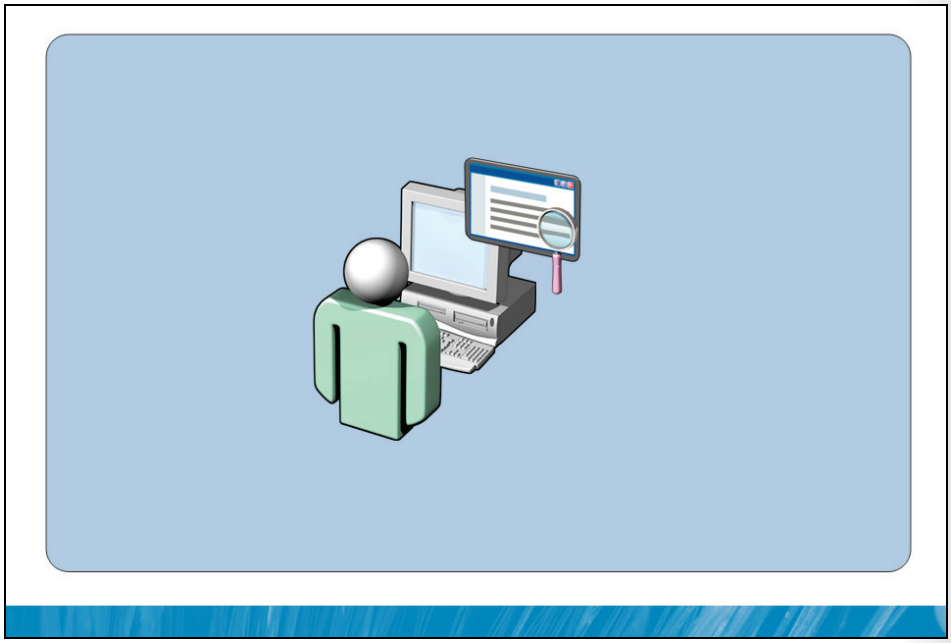
- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**



Note: Step-by-step instructions for completing the labs in this course are available in the lab answer keys provided. Completed, working code is available in the Solution folders under the Labfiles folder for each lab exercise on the virtual machine.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data. You have been asked to write a C# application to read a small set of input data that a measuring device has generated, format this data to make it more readable, and then display the formatted results.

The data consists of text data that contains pairs of numbers representing x-coordinates and y-coordinates of the location of an object. Each line of text contains one set of coordinates. The following code example resembles a typical dataset.

```
23.8976,12.3218
25.7639,11.9463
24.8293,12.2134
```

You have been asked to format the data like the following code example.

```
x:23.8976 y:12.3218
x:25.7639 y:11.9463
x:24.8293 y:12.2134
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 1: Building a Simple Console Application

In this exercise, you will initially build and test the application by using console I/O. You will then use I/O redirection to run the application by using data that is held in a file and verify that the results are as expected.

Scenario

As a prototype, you have decided to implement a console application to read input from the keyboard and format it. When you are happy that your code is working, you will then run the code and redirect input to come from a file that contains the data that you want to format.

The main tasks for this exercise are as follows:

1. Create a new Console Application project.
2. Add code to read user input and write output to the console.
3. Modify the program to read and echo text until end-of-file is detected.
4. Add code to format the data and display it.
5. Test the application by using a data file.

▶ Task 1: Create a new Console Application project

1. Log on to the 10266A-GEN-DEV machine as **Student** with the password **Pa$$w0rd**.
2. Open Visual Studio 2010.
3. Create a new console application project called ConsoleApplication in the E:\Labfiles\Lab 1\Ex1\Starter folder.

▶ Task 2: Add code to read user input and write output to the console

1. In the **Main** method, add the statements shown in bold in the following code example, which read a line of text from the keyboard and store it in a string variable called line.

```
static void Main(string[] args)

{
    // Buffer to hold a line as it is read in
    string line;
```

```
    // Read a line of text from the keyboard
    line = Console.ReadLine();
}
```

This code uses the **Console.ReadLine** method to read the input, and includes comments with each line of code that indicates its purpose.

2. Add the statement and comment shown in bold in the following code example, which echo the text back to the console by using the **Console.WriteLine** method.

```
static void Main(string[] args)
{
    // Buffer to hold a line as it is read in
    string line;

    // Read a line of text from the keyboard
    line = Console.ReadLine();

    // Write the results out to the console window
    Console.WriteLine(line);
}
```

3. Build the application.
4. Run the application and verify that it works as expected. You should be able to enter a line of text and see that line echoed to the console.

▶ Task 3: Modify the program to read and echo text until end-of-file is detected

1. In the **Main** method, modify the statement and comment shown in bold in the following code example, which read a line of text from the keyboard.

```
static void Main(string[] args)
{
    // Buffer to hold a line as it is read in
    string line;

    // Loop until no more input (Ctrl-Z in a console, or end-of-file)
    while ((line = Console.ReadLine()) != null)
    {

    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Write the results out to the console window
    Console.WriteLine(line);
}
```

This code incorporates the statement into a **while** loop that repeatedly reads text from the keyboard until the **Console.ReadLine** method returns a **null** value (this happens when the **Console.ReadLine** method detects the end of a file, or the user types CTRL+Z).

2. Move the **Console.WriteLine** statement into the body of the **while** loop as shown in bold in the following code example. This statement echoes each line of text that the user has entered.

```
static void Main(string[] args)
{
    // Buffer to hold a line as it is read in
    string line;

    // Loop until no more input (Ctrl-Z in a console, or end-of-file)
    while ((line = Console.ReadLine()) != null)
    {
        // Write the results out to the console window
        Console.WriteLine(line);
    }
}
```

3. Build the application.
4. Run the application and verify that it works as expected. You should be able to repeatedly enter lines of text and see those lines echoed to the console. The application should only stop when you press CTRL+Z.

▶ Task 4: Add code to format the data and display it

1. In the body of the **while** loop, add the statement and comment shown in bold before the **Console.WriteLine** statement in the following code example.

```
static void Main(string[] args)
{
    // Buffer to hold a line as it is read in
    string line;

    // Loop until no more input (Ctrl-Z in a console, or end-of-file)
    while ((line = Console.ReadLine()) != null)
    {
        // Format the data
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        line = line.Replace(",", " y:");

        // Write the results out to the console window
        Console.WriteLine(line);
    }
}
```

This code replaces each occurrence of the comma character, "," in the input read from the keyboard and replaces it with the text " y:". It uses the **Replace** method of the line string variable. The code then assigns the result back to the line variable.

2. Add the statement shown in bold in the following code example to the code in the body of the **while** loop.

```
static void Main(string[] args)
{
    // Buffer to hold a line as it is read in
    string line;

    // Loop until no more input (Ctrl-Z in a console, or end-of-file)
    while ((line = Console.ReadLine()) != null)
    {
        // Format the data
        line = line.Replace(",", " y:");
        line = "x:" + line;

        // Write the results out to the console window
        Console.WriteLine(line);
    }
}
```

This code adds the prefix "x:" to the line variable by using the string concatenation operator, **+**, before the **Console.WriteLine** statement. The code then assigns the result back to the line variable.

3. Build the application.
4. Run the application and verify that it works as expected.

   The application expects input that looks like the following code example.

```
23.54367,25.6789
```

Your code should format the output to look like the following code example.

```
x:23.54367 y:25.6789
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Test the application by using a data file

1. Perform the following steps to add the DataFile.txt file that contains the sample data to the project. This file is located in the E:\Labfiles\Lab 1\Ex1 \Starter folder. These steps specify that the file should be copied to the folder that holds the compiled application when the project is built:

   a. In Solution Explorer, right-click the **ConsoleApplication** project, point to **Add**, and then click **Existing Item**.

   b. In the **Add Existing Item – ConsoleApplication** dialog box, move to the **E:\Labfiles\Lab 1\Ex1\Starter** folder, select **All Files (*.*)** in the drop-down list box adjacent to the **File name** text box, click **DataFile.txt**, and then click **Add**.

   c. In Solution Explorer, select **DataFile.txt**. In the Properties window, change the **Build Action** property to **None**, and then change the **Copy to Output** property to **Copy Always**.

2. Rebuild the application.

3. Open a Visual Studio Command Prompt window, and then move to the **E:\Labfiles\Lab 1\Ex1\Starter\ConsoleApplication\bin\Debug** folder.

4. Run the ConsoleApplication application and redirect input to come from DataFile.txt.

   Verify that the output that is generated looks like the following code example.

```
x:23.8976 y:12.3218
x:25.7639 y:11.9463
x:24.8293 y:12.2134
```

   In the Command Prompt window, type the command in the following code example.

```
ConsoleApplication < DataFile.txt
```

5. Close the Command Prompt window, and then return to Visual Studio.

6. Modify the project properties to redirect input from the DataFile.txt file when the project is run by using Visual Studio.

7. Run the application in Debug mode from Visual Studio.

   The application will run, but the console window will close immediately after the output is generated. This is because Visual Studio only prompts the user to close the console window when a program is run without debugging. When a

program is run in Debug mode, Visual Studio automatically closes the console window as soon as the program finishes.

8. Set a breakpoint on the closing brace at the end of the **Main** method.
9. Run the application again in Debug mode. Verify that the output that is generated is the same as the output that is generated when the program runs from the command line.

Exercise 2: Building a WPF Application

In this exercise, you will build a simple WPF application that provides similar functionality to the console application that you developed in Exercise 1. You will initially test the display formatting by providing fields that the user can type data into. When you are satisfied that the display format is correct, you will modify the application to read input from the console and modify the **Debug** properties of the application to redirect this input to come from the same file as before.

Scenario

You have been asked to change the application to generate the data in a more helpful manner. The application should perform the same task as the console application except that the output is displayed in a WPF window.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Add code to format the data that the user enters.
4. Modify the application to read data from a file.

▶ Task 1: Create a new WPF Application project

- Create a new project called WpfApplication in the E:\Labfiles\Lab 1\Ex2 \Starter folder by using the WPF Application template.

▶ Task 2: Create the user interface

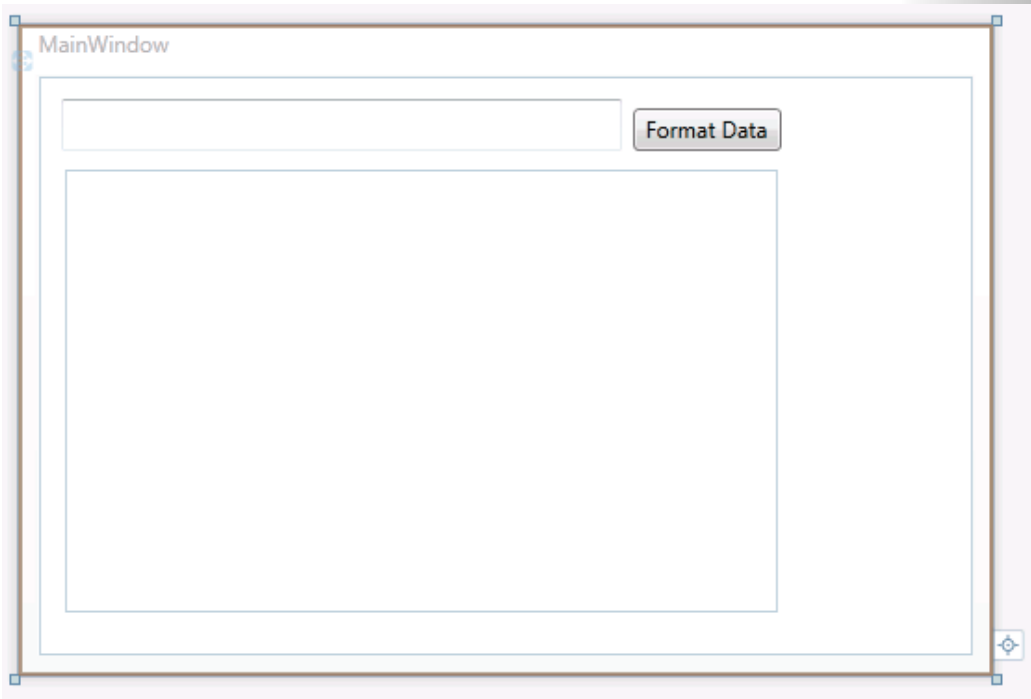
1. Add **TextBox**, **Button**, and **TextBlock** controls to the MainWindow window. Place them anywhere in the window.
2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

MCT USE ONLY. STUDENT USE PROHIBITED

Control	Property	Value
TextBox	**Name**	**testInput**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,12,0,0**
	VerticalAlignment	**Top**
	Width	**302**
Button	**Name**	**testButton**
	Content	**Format Data**
	Height	**23**
	HorizontalAlignment	**Left**
	Margin	**320,17,0,0**
	VerticalAlignment	**Top**
	Width	**80**
TextBlock	**Name**	**formattedText**
	Height	**238**
	HorizontalAlignment	**Left**
	Margin	**14,50,0,0**
	Text	***blank***
	VerticalAlignment	**Top**
	Width	**384**

MCT USE ONLY. STUDENT USE PROHIBITED

The MainWindow window should look like the following screen shot.



▶ Task 3: Add code to format the data that the user enters

1. Create an event handler for the **Click** event of the button.
2. Add the code shown in bold in the following code example to the event-handler method.

```
private void testButton_Click(object sender, RoutedEventArgs e)
{
      // Copy the contents of the TextBox into a string
      string line = testInput.Text;
      // Format the data in the string
      line = line.Replace(",", " y:");
      line = "x:" + line;

      // Store the results in the TextBlock
      formattedText.Text = line;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

This code reads the contents of the **TextBox** control into a string variable called line, formats this string in the same way as the console application in Exercise 1, and then displays the formatted result in the **TextBlock** control. Notice that you can access the contents of a **TextBox** control and a **TextBlock** control by using the **Text** property.

3. Build the solution, and then correct any errors.
4. Run the application and verify that it works in a similar manner to the original console application in Exercise 1.
5. Close the MainWindow window, and then return to Visual Studio.

▶ Task 4: Modify the application to read data from a file

1. Create an event handler for the **Window_Loaded** event. This event occurs when the window is about to be displayed, just after the application has started up.
2. In the event-handler method, add the code shown in bold in the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Buffer to hold a line read from the file on standard input
    string line;

    // Loop until the end of the file
    while ((line = Console.ReadLine()) != null)
    {
        // Format the data in the buffer
        line = line.Replace(",", " y:");
        line = "x:" + line + "\n";

        // Put the results into the TextBlock
        formattedText.Text += line;
    }
}
```

This code reads text from the standard input, formats it in the same manner as Exercise 1, and then appends the results to the end of the **TextBlock** control. It continues to read all text from the standard input until end-of-file is detected.

Notice that you can use the **+=** operator to append data to the **Text** property of a **TextBlock** control, and you can add the newline character ("\n") between

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

lines for formatted output to ensure that each item appears on a new line in the **TextBlock** control.

3. Perform the following steps to modify the project settings to redirect standard input to come from the DataFile.txt file. A copy of this file is available in the E:\Labfiles\Lab 1\Ex2\Starter folder:

   a. In Solution Explorer, right-click the **WpfApplication** project, point to **Add**, and then click **Existing Item**.

   b. In the **Add Existing Item – WpfApplication** dialog box, move to the **E:\Labfiles\Lab 1\Ex2\Starter** folder, select **All Files (*.*)** in the drop-down list box adjacent to the **File name** text box, click **DataFile.txt**, and then click **Add**.

   c. In Solution Explorer, select **DataFile.txt**. In the Properties window, change the **Build Action** property to **None**, and then change the **Copy to Output** property to **Copy Always**.

   d. In Solution Explorer, right-click the **WpfApplication** project, and then click **Properties**.

   e. On the **Debug** tab, in the **Command line arguments**: text box, type **< DataFile.txt**

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

   g. Close the WpfApplication properties window.

4. Build and run the application in Debug mode. Verify that, when the application starts, it reads the data from DataFile.txt and displays in the **TextBlock** control the results in the following code example.

```
x:23.8976 y:12.3218
x:25.7639 y:11.9463
x:24.8293 y:12.2134
```

5. Close the MainWindow window, and then return to Visual Studio.

Exercise 3: Verifying the Application

In this exercise, you will create some additional test data and use it as input to your application. You will use the Visual Studio 2010 debugger to step through your code and examine it as it runs.

MCT USE ONLY. STUDENT USE PROHIBITED

Scenario

You want to verify that the code for your WPF application is operating exactly as you require. You decide to create some additional test data and use the Visual Studio 2010 debugger to step through the application.

The main tasks for this exercise are as follows:

1. Modify the data in the DataFile.txt file.
2. Step through the application by using the Visual Studio 2010 debugger.

▶ Task 1: Modify the data in the DataFile.txt file

- Modify the contents of the DataFile.txt file as the following code example shows.

```
1.2543,0.342
32525.7639,99811.9463
24.8293,12.2135
23.8976,12.3218
25.7639,11.9463
24.8293,12.2135
```



Note: There must be a blank line at the end of DataFile.txt.

▶ Task 2: Step through the application by using the Visual Studio 2010 debugger

1. Set a breakpoint at the start of the **Window_Loaded** event handler.
2. Start the application running in Debug mode.

   When the application runs the **Window_Loaded** event handler, it reaches the breakpoint and drops into Visual Studio. The opening brace of the method is highlighted.
3. Step into the first statement in the **Window_Loaded** method that contains executable code.

   The **while** statement should be highlighted. This is because the statement that declares the line variable does not contain any executable code.

4. Examine the value of the line variable. It should be **null** because it has not yet been assigned a value.
5. Step into the next statement.

   The cursor moves to the opening brace at the start of the body of the **while** loop.
6. Examine the value of the line variable. It should be **1.2543,0.342**. This is the text from the first line of the DataFile.txt file. The **Console.ReadLine** statement in the **while** statement reads this text from the file.
7. Step into the next statement.

   The cursor moves to the line in the following code example.

```
line = line.Replace(",", " y:");
```

8. Step into the next statement.
9. Examine the value of the line variable. It should now be **1.2543 y:0.342**. This is the result of calling the **Replace** method and assigning the result back to line.
10. Step into the next statement.
11. Examine the value of the line variable. It should now be **x:1.2543 y:0.342\n**. This is the result of prefixing the text "x:" to line and suffixing a newline character.
12. Step into the next statement.

    The cursor moves to the closing brace at the end of the **while** loop.
13. In the Immediate window, examine the value of the **Text** property of the **formattedText TextBlock** control. It should contain the same text as the line variable.



Note: If the Immediate window is not visible, press CTRL+ALT+I.

14. Set another breakpoint at the end of the **while** loop.
15. Continue the programming running for the next iteration of the **while** loop. It should stop when it reaches the breakpoint at the end of the loop.

MCT USE ONLY. STUDENT USE PROHIBITED

16. Examine the value of the line variable. It should now be **x:32525.7639 y:99811.9463\n**. This is the data from the second line of DataFile.txt.
17. In the Immediate window, examine the value of the **Text** property of the **formattedText TextBlock** control again. It should now contain the formatted results from the first two lines of DataFile.txt.
18. Remove the breakpoint from the end of the **while** loop.
19. Continue the programming running. The **Window_Loaded** method should now run to completion and display the MainWindow window. The **TextBlock** control should contain all of the data from DataFile.txt, formatted correctly.
20. Close the MainWindow window, and then return to Visual Studio.

Exercise 4: Generating Documentation for an Application

In this exercise, you will add XML comments to your application, and use the Sandcastle tool to generate documentation for the application.

Scenario

You must ensure that your application is fully documented so that it can be maintained easily. You decide to add XML comments to the methods that you have added to the WPF application, and generate a help file.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add XML comments to the application.
3. Generate an XML comments file.
4. Generate a .chm file.

▶ Task 1: Open the starter project

- In Visual Studio, open the WpfApplication solution located in the E:\Labfiles\Lab 1\Ex4\Starter folder. This solution is a working copy of the solution from Exercise 2.

▶ Task 2: Add XML comments to the application

1. Display the MainWindow.xaml.cs file.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Add the XML comment in the following code example before the **MainWindow** class declaration.

```
/// <summary>
/// WPF application to read and format data
/// </summary>
```

3. Add the XML comment in the following code example before the **MainWindow** constructor.

```
/// <summary>
/// Constructor for MainWindow
/// </summary>
```

4. Add the XML comment in the following code example before the **testButton_Click** method.

```
/// <summary>
/// Read a line of data entered by the user.
/// Format the data and display the results in the
/// formattedText TextBlock control.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

5. Add the XML comment in the following code example before the **Windows_Loaded** method.

```
/// <summary>
/// After the Window has loaded, read data from the standard input.
/// Format each line and display the results in the
/// formattedText TextBlock control.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
```

6. Save MainWindow.xaml.cs.

▶ Task 3: Generate an XML comments file

1. Set the project properties to generate an XML documentation file when the project is built.
2. Build the solution, and then correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

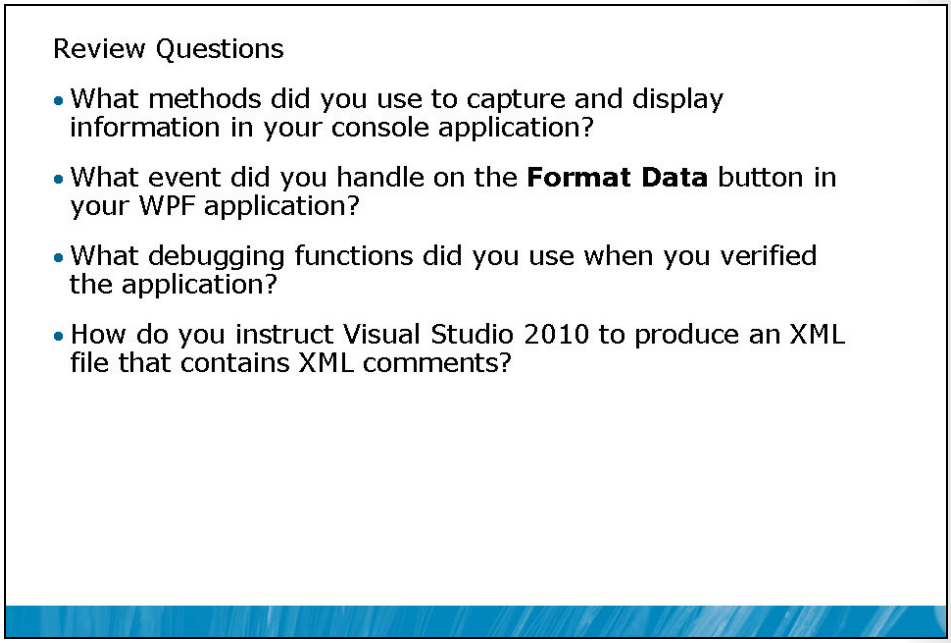
3. Verify that an XML comments file called comments.xml has been generated in the E:\Labfiles\Lab 1\Ex4\Starter\WpfApplication\bin\Debug folder, and then examine it.
4. Copy the comments.xml file to the E:\Labfiles\Lab 1\Ex4\Helpfile folder.

▶ Task 4: Generate a .chm file

1. Open a Windows Command Prompt window as **Administrator**. The **Administrator** password is **Pa$$w0rd**.
2. Move to the **E:\Labfiles\Lab 1\Ex4\HelpFile** folder.
3. Use Notepad to edit the builddoc.cmd script, and then verify that the input variable is set to "E:\Labfiles\Lab 1\Ex4\Starter\WpfApplication\bin\Debug \WpfApplication.exe".
4. Run the builddoc.cmd script.
5. Open the test.chm file that the builddoc.cmd script generates.
6. Browse documentation that is generated for your application, and then close test.chm.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

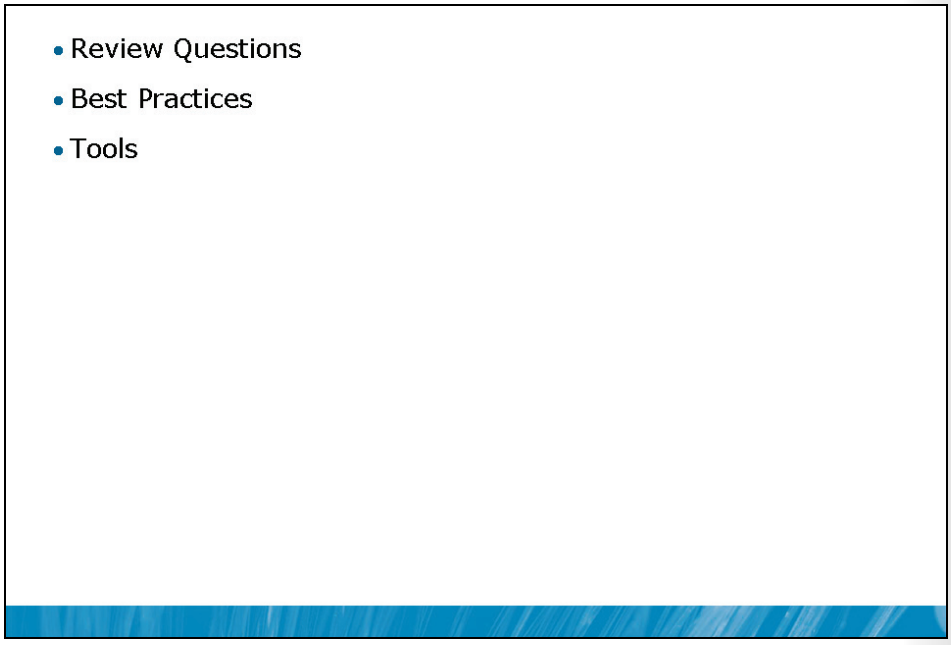


Review Questions

1. What methods did you use to capture and display information in your console application?
2. What event did you handle on the **Format Data** button in your WPF application?
3. What debugging functions did you use when you verified the application?
4. How do you instruct Visual Studio 2010 to produce an XML file that contains XML comments?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What is the purpose of the .NET Framework and the role of Visual C#?
2. What is the purpose of Visual Studio 2010 templates?
3. What is the purpose of Visual Studio projects and solutions?
4. What is the purpose of a **Main** method?
5. List some of the controls that WPF provides.
6. What is the purpose of XML comments?
7. What is the purpose of the Visual Studio 2010 debugger?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices Related to Writing a C# Application

Supplement or modify the following best practices for your own work situations:

- Keep the **Main** method small and lightweight.
- Declare variables by using meaningful names and avoid reference to the underlying data type, for example, **nameString**.
- Define controls by using meaningful names and avoid reference to the underlying control type, for example, **labelName**.
- Add comments to your code that describe your thought process.

Tools

Tool	Use for	Where to find it
Caspol.exe	Enables users to modify the machine, user, and enterprise security policy. This can include defining a custom permission set and adding assemblies to the full trust list.	C:\Windows\Microsoft.NET \Framework\v4.0.30319
Gacutil.exe	Enables users to manipulate the assemblies in the GAC. This can include installing and uninstalling assemblies in the GAC so that multiple applications can access them.	C:\Program Files \Microsoft SDKs\Windows \v7.0A\bin
Ildasm.exe	Enables users to manipulate assemblies, such as determining whether an assembly is managed, or disassembling an assembly to view the compiled MSIL code.	C:\Program Files \Microsoft SDKs\Windows \v7.0A\bin
Makecert.exe	Enables users to create x.509 certificates for use in their development environment. Typically, you can use these certificates to sign your assemblies and define SSL connections.	C:\Program Files \Microsoft SDKs\Windows \v7.0A\bin

Tool	Use for	Where to find it
Ngen.exe	Enables users to improve the performance of .NET applications. The Native Image Generator improves performance by precompiling assemblies into images that contain processor-specific machine code. The CLR can then run the precompiled images instead of using JIT compilation.	C:\Windows\Microsoft.NET \Framework\v4.0.30319
Sn.exe	Enables users to sign assemblies with strong names. The Strong Name Tool includes commands to create a new key pair, extract a public key from a key pair, and verify assemblies.	C:\Program Files \Microsoft SDKs\Windows \v7.0A\bin

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 2

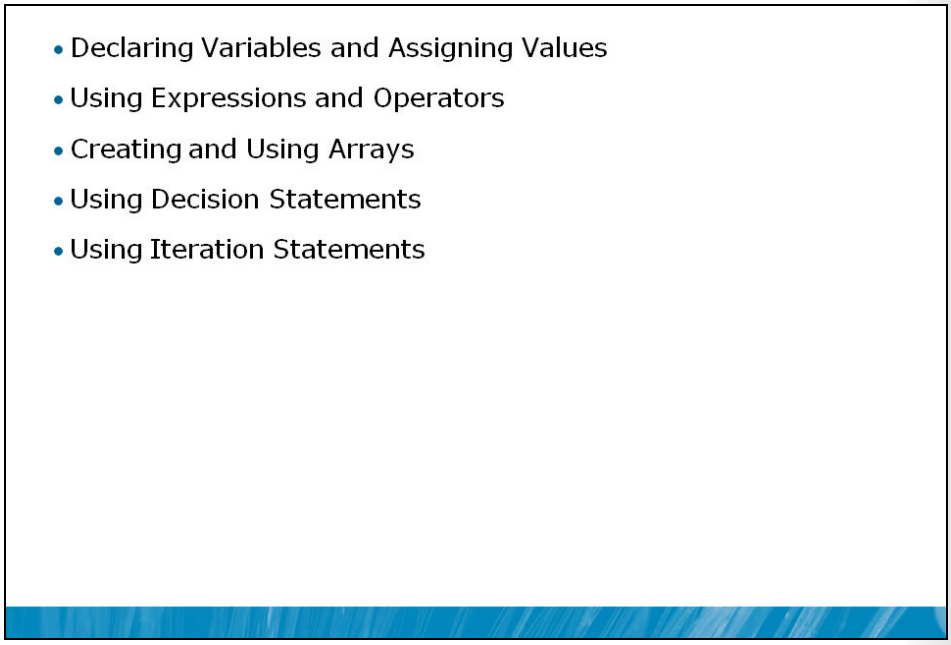
Using C# Programming Constructs

Contents:

Lesson 1: Declaring Variables and Assigning Values 2-4
Lesson 2: Using Expressions and Operators 2-23
Lesson 3: Creating and Using Arrays 2-36
Lesson 4: Using Decision Statements 2-49
Lesson 5: Using Iteration Statements 2-63
Lab: Using C# Programming Constructs 2-78

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



To make the best use of a programming language, it is essential that you understand the constructs that the language provides. C# is a procedural programming language that shares many features with other procedural programming languages that you may be familiar with. For example, you can declare variables, assign values to them, and make decisions based on the values of these variables.

This module introduces many of the basic C# language data types and programming constructs, and describes the syntax and semantics of these constructs.

Objectives

After completing this module, you will be able to:

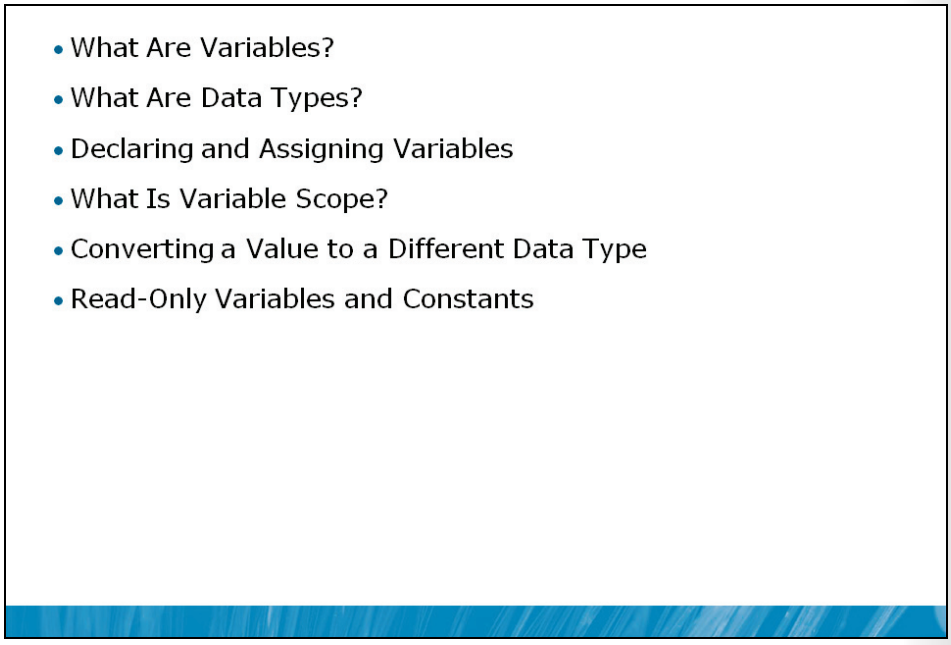
- Explain how to declare variables and assign values.
- Use operators to construct expressions.
- Create and use arrays.

MCT USE ONLY. STUDENT USE PROHIBITED

- Use decision statements.
- Use iteration statements.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Declaring Variables and Assigning Values



All applications use data. This data might be supplied via 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 data types in C#.

This lesson describes how C# uses variables and the built-in data types that C# provides. This lesson also explains how to convert the data that is held in a variable from one data type to another.

Objectives

After completing this lesson, you will be able to:

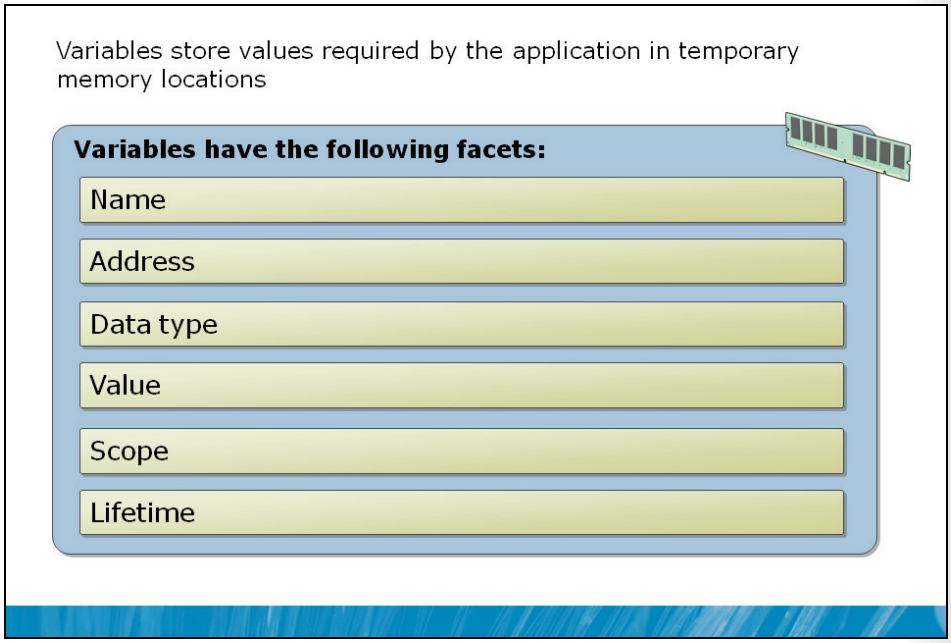
- Describe the purpose of variables.
- Describe the purpose of data types.
- Explain how to declare and assign variables.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain how variable scope determines where a variable is accessible in an application.
- Explain how to convert data in a variable to a different data type.
- Describe best practices for using read-only variables and constants.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Variables?



Key Points

A variable represents a named location in memory for a piece of data. An application can access a piece of data by using the variable it has been assigned to.

Variables store values that an application can change while it is running. You often need to store values temporarily when you perform calculations or pass data between the user, an application, and a database. For example, you might want to retrieve several values from a database, compare them, and perform different operations on them depending on the result of the comparison.

A variable has the following six facets:

- *Name*. Unique identifier that refers to the variable in code.
- *Address*. Memory location of the variable.
- *Data type*. Type and size of data that the variable can store.
- *Value*. Value at the address of the variable.
- *Scope*. Defined areas of code that can access and use the variable.

MCT USE ONLY. STUDENT USE PROHIBITED

- *Lifetime*. Period of time that a variable is valid and available for use.

Examples of Variables

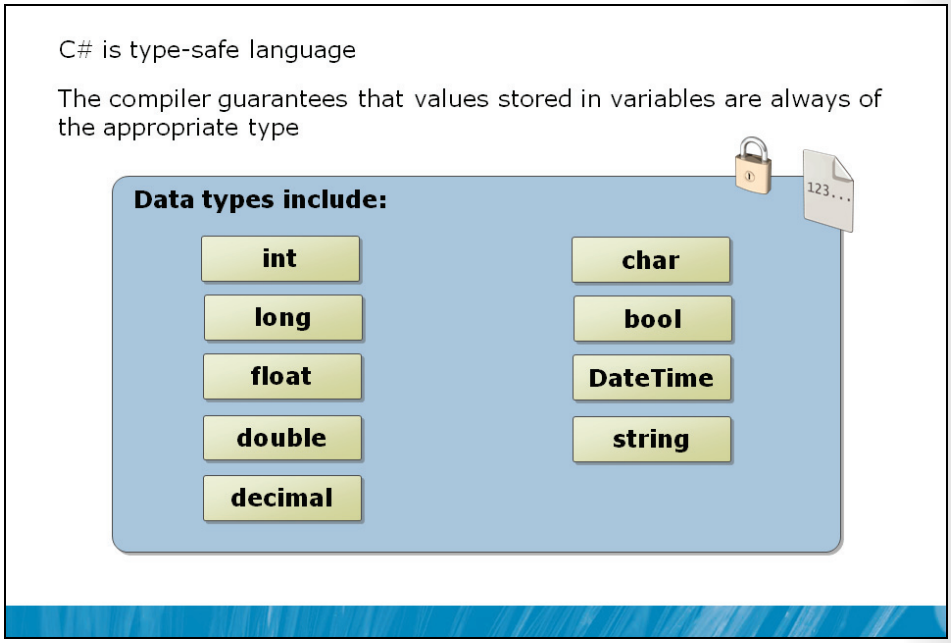
You can use variables in many ways, including:

- As a counter for loop structures.
- As temporary storage for property values.
- As a container to store a value that was returned from a function.

Question: What is a variable and how are variables used in Microsoft® .NET Framework applications?

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Data Types?



Key Points

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

Commonly Used Data Types

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

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

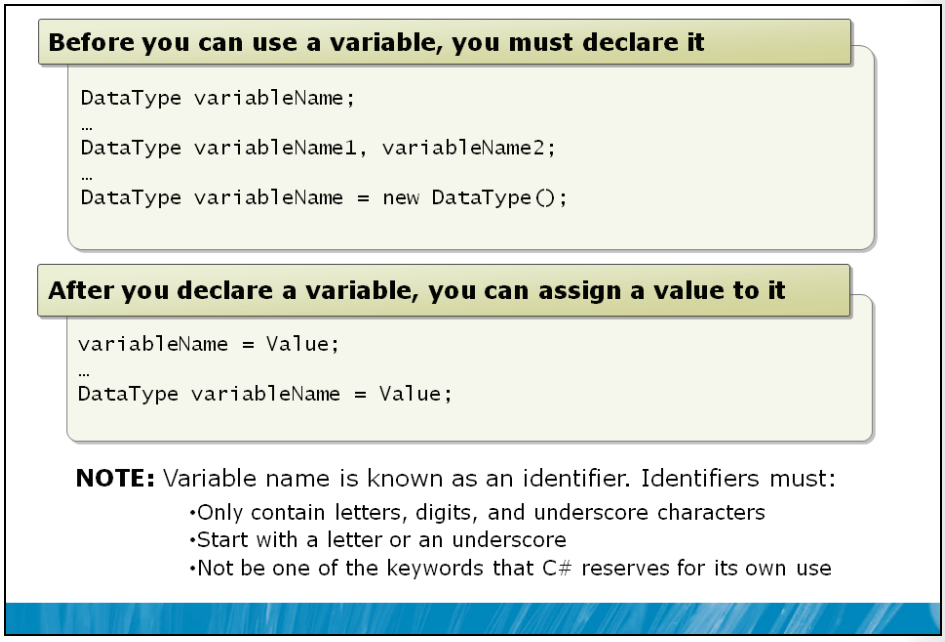
MCT USE ONLY. STUDENT USE PROHIBITED

Type	Description	Size (bytes)	Range
float	Floating-point numbers	4	+/–3.4 × 10^38
double	Double precision (more accurate) floating-point numbers	8	+/–1.7 × 10^308
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 01/01/0001 to 23:59:59 on 12/31/9999
string	Sequence of characters	2 per character	N/A

Question: What type would you use to store a sequence of alphanumeric characters?

MCT USE ONLY. STUDENT USE PROHIBITED

Declaring and Assigning Variables



Key Points

Before you can use a variable, you must declare it so that you can specify its name and characteristics.

Identifiers

The name of a variable is referred to as an identifier. 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 C# reserves for its own use. A full list of C# keywords is provided in the CD content for this topic.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: 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 C# will not confuse them, although this is not good practice.

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 stick to it.



Note: Different organizations may have different naming conventions. Some common conventions are described in the CD content for this topic. If your organization does not currently follow any specific naming style, you may want to adopt these conventions.

Declaring a Variable

When you declare a variable, you reserve some storage space for that variable in memory. You must specify 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 syntax for declaring variables is shown in the following code example.

```
DataType variableName;
// OR
DataType variableName1, variableName2;
```

Assigning a Value to a Variable

After you declare a variable, you can assign a value to it for later use in the application by using an assignment statement. You can change the value in a variable as many times as you want during the application.

The assignment operator (=) assigns a value to a variable. The syntax of a variable assignment is shown in the following code example.

```
variableName = value;
```

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

The following code example declares an integer called **price** and assigns the number **10** to the integer.

```
int price = 10;
```

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example assigns the number **20** to an existing integer variable called **price**.

```
price = 20;
```

You can also assign variables when you declare them. The following code example shows the syntax of a variable declaration and assignment.

```
DataType variableName = value;
```

The type of the expression must match the type of the variable, otherwise your program will not compile. For example, the code in the following code example will not work because you cannot assign a string value to an integer variable.

```
int numberOfEmployees;
numberOfEmployees = "Hello";
```



Note: 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. 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. Consequently, you must initialize a variable that is defined in this way when it is defined, as shown in the following code example.

```
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.

Question: What is the syntax for declaring and assigning a variable?

MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

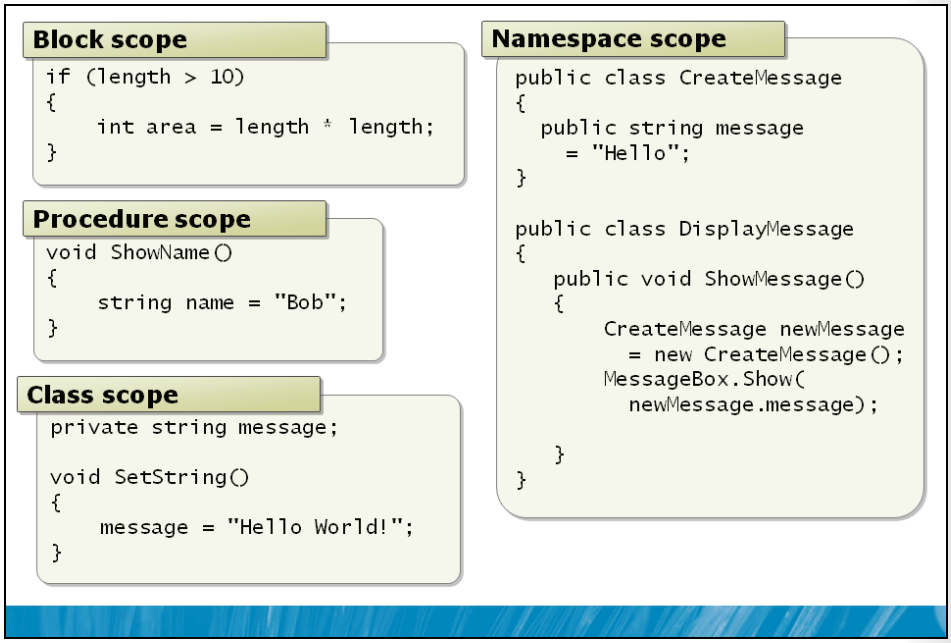
For more information about the keyword in C#, see the C# Keywords page at http://go.microsoft.com/fwlink/?LinkId=192890.

For more information about naming conventions, see the General Naming Conventions page at http://go.microsoft.com/fwlink/?LinkId=192891.

For more information about capitalization conventions, see the Capitalization Conventions page at http://go.microsoft.com/fwlink/?LinkId=192892.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Variable Scope?



Key Points

The scope of a variable determines the parts of a program that can access that variable. If you attempt to reference a variable outside its scope, the compiler will generate an error.

Levels of Scope

Variables can have one of the following levels of scope:

- Block
- Procedure
- Class
- Namespace

These levels of scope progress from the narrowest (block) to the widest (namespace). The following sections describe these different scopes.

MCT USE ONLY. STUDENT USE PROHIBITED

Block Scope

A block is a set of statements that is enclosed within initiating and terminating declaration statements, such as a loop. If you declare a variable within a block, you can use it only within that block. The lifetime of the variable is still that of the entire block. The following code example shows how to declare a local variable called area with block-level scope.

```
if (length > 10)
{
    int area = length * length;
}
```

Procedure Scope

Variables that are declared within a procedure are not available outside that procedure. Only the procedure that contains the declaration can use the variable. When you declare variables in a block or procedure, they are known as local variables. The following code example shows how to declare a local variable called name with procedure-level scope.

```
void ShowName()
{
    string name = "Bob";
    MessageBox.Show("Hello " + name);
}
```

Class Scope

If you want the lifetime of a local variable to extend beyond the lifetime of the procedure, declare the variable at class-level scope. When you declare variables in a class or structure, but not inside a procedure, they are known as class variables. You can assign a scope to class variables by using an access modifier. The following code example shows how to declare a local variable called message with class-level scope.

```
private string message;
void SetString()
{
    message = "Hello World!";
}

void ShowString()
{
    MessageBox.Show(message);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Namespace Scope

When you declare variables at class level by using the **public** keyword, they are available to all procedures within the namespace. The following code example shows you how to declare a variable called message in one class that you can access in another class.

```
public class CreateMessage
{
    public string message = "Hello";
}

public class DisplayMessage
{
    public void ShowMessage()
    {
        CreateMessage newMessage = new CreateMessage();
        MessageBox.Show(newMessage.message);
    }
}
```

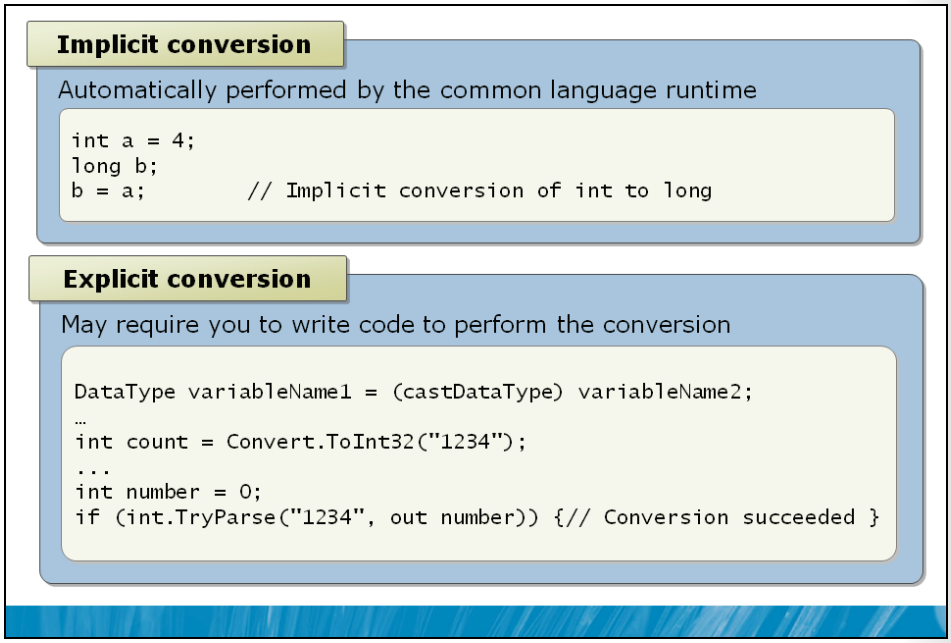
Question: You are developing an application and you need to declare a variable that is accessible to two methods in the same class. What is the easiest way to achieve this?

Additional Reading

For more information about scopes, see the 3.7 Scopes page at http://go.microsoft.com/fwlink/?LinkId=192893.

MCT USE ONLY. STUDENT USE PROHIBITED

Converting a Value to a Different Data Type



Key Points

When you are designing applications, you may need to convert data from one type to another. Conversions are necessary when a value of one type must be assigned to a variable of a different type. For example, you might need to convert the string value "99" that you have read from a text file into the integer value 99 that you can store in an integer variable. The process of converting a value of one data type to another is called conversion or casting.

Implicit and Explicit Conversions

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

- *Implicit conversion*. Automatically performed by the common language runtime (CLR) on operations that are guaranteed to succeed without losing information.
- *Explicit conversion*. Requires you to write code to perform a conversion that otherwise could lose information or produce an error.

MCT USE ONLY. STUDENT USE PROHIBITED

Explicit conversion reduces the possibility of some bugs in your code and makes your code more efficient. 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. C# only allows safe implicit conversions, such as widening of integers.

The following code example shows how data is converted implicitly from an integer to a **long** type.

```
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, the converse conversion is not true; you cannot implicitly convert a **long** value to an **int** type because this conversion risks losing information (the **long** value might be outside the range that the **int** type supports). The following table shows the implicit type conversions that are supported in 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**

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Explicit Conversions

In C#, you can use a cast operator to perform explicit conversions. A cast specifies the type to convert to, in round brackets. The syntax for performing an explicit conversion is shown in the following code example.

```
DataType variableName1 = (castDataType) variableName2;
```

You can only perform meaningful conversions in this way, such as converting a **long** to an **int** type. 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 because Microsoft IntelliSense® helps you locate the conversion method that you need. The following code example converts a **string** to an **int** type.

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

In addition to the **Convert.ToString** method, many types implement their own **ToString** method. The following code example converts an **int** to a **string** type.

```
int number = 1234;
string numberString = count.ToString();
```

Some of the built-in data types in C# 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** type by using the **int.TryParse**() method.

```
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
}
```

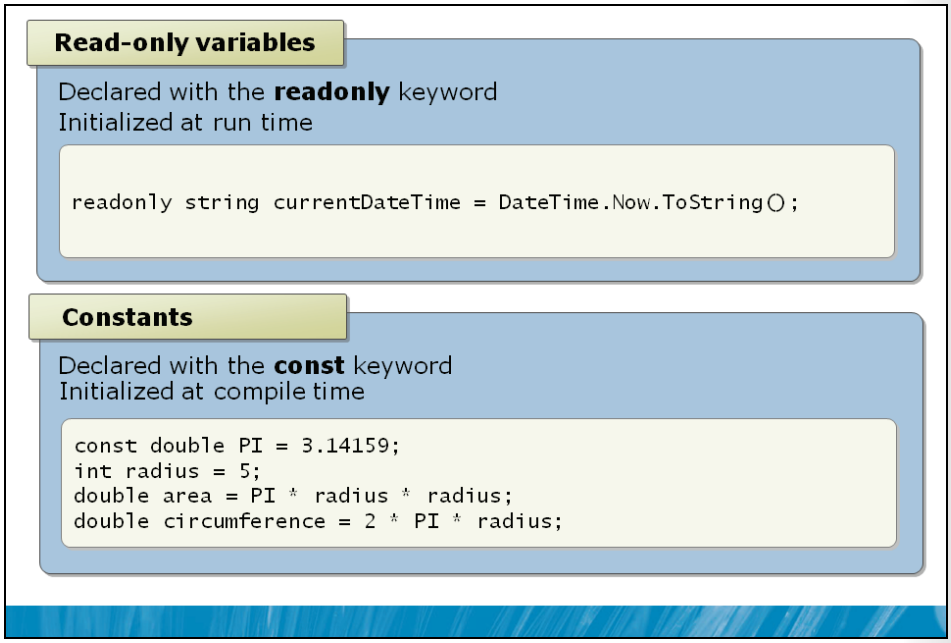
Question: You are converting a **string** to an **int** type, but you are unsure whether the **string** will contain a valid **int** value. Which conversion approach should you use?

Additional Reading

For more information about the **System.Convert** class, see the Convert Class page at http://go.microsoft.com/fwlink/?LinkId=192894.

MCT USE ONLY. STUDENT USE PROHIBITED

Read-Only Variables and Constants



Key Points

Read-only variables and constants enable you to store data just like you can with any other variables in C#. However, these variables have some subtle differences.

You can use read-only variables and constants to store data that does not change. You can use read-only variables or constants for many values such as:

- The number of hours in a day.
- The speed of light.
- The number of degrees in a circle.

Comparing Read-Only Variables and Constants

There is a subtle difference between using a read-only variable and using a constant. When you use a constant in an application, you can only initialize the constant when it is declared. However, you can initialize a read-only variable in its declaration or in the constructor of the class that contains the read-only variable.

MCT USE ONLY. STUDENT USE PROHIBITED

Therefore, you can only define and initialize constants at design time and you cannot assign a different value to the constant when your application runs.

Syntax

You declare read-only variables by using the **readonly** keyword, as the following code example shows.

```
readonly DataType variableName = Value;
```

You declare constants by using the **const** keyword, as the following code example shows.

```
const DataType variableName = Value;
```

Examples

The following code example declares a constant to store the current date and time. This example uses the **DateTime** class and the **Now** property, which enables you to compute the current date and time at run time. If you tried to use this approach with a constant, you would get a compile error.

```
readonly string currentDateTime = DateTime.Now.ToString();
```

The following code example declares a **PI** constant to calculate the area and circumference of a circle with a radius of 5.

```
const double PI = 3.14159;
int radius = 5;
double area = PI * radius * radius;
double circumference = 2 * PI * radius;
```

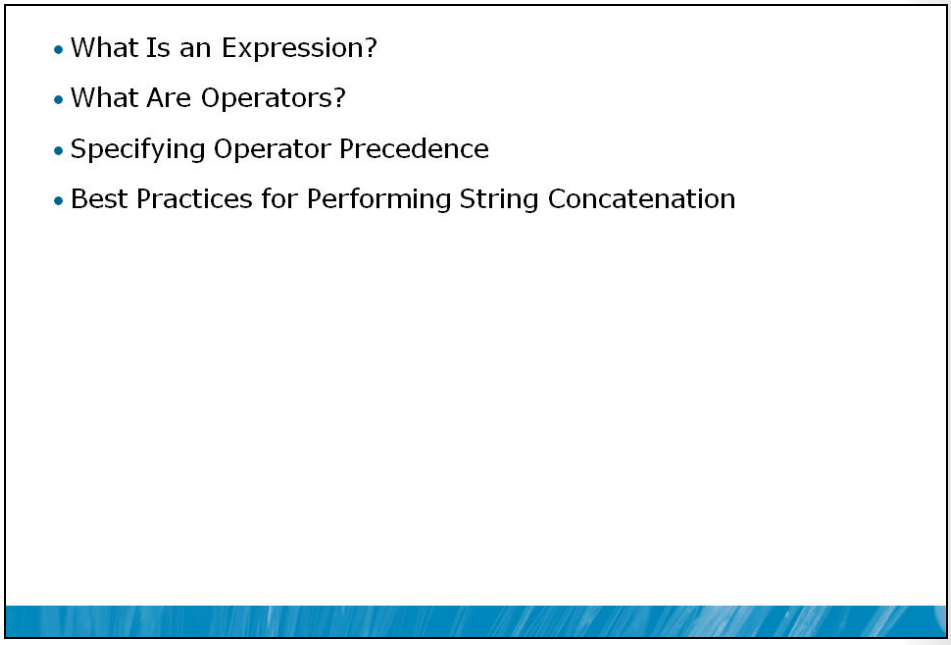
Question: What are the main differences between a constant and a read-only variable?

Additional Reading

For more information about constants, see the const (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192895.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using Expressions and Operators



The value that you assign to a variable can be a simple constant value, but more frequently, it is a value that is the result of an expression that is evaluated at run time. This lesson describes how to build an expression by using the various operators that C# provides. This lesson also describes operator precedence and how to control the order in which the elements in an expression are evaluated by using parentheses. Finally, this lesson explains the best practices for dynamically constructing string values.

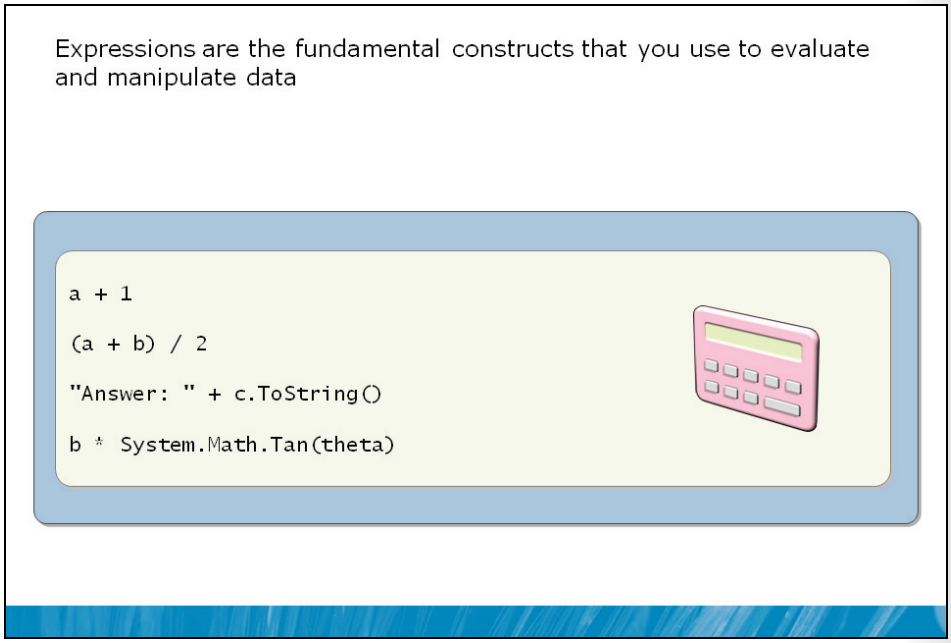
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of an expression.
- Describe the purpose of operators.
- Explain how to specify operator precedence.
- Explain the best practices for concatenating string values.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Expression?



Key Points

Expressions are a central component of practically every C# application. This is because expressions are the fundamental constructs that you use to evaluate and manipulate data.

Expressions are collections of operands and operators. These terms are defined as follows:

- *Operands*. Operands are values, for example, numbers and strings. They can be constant (literal) values, variables, properties, or method-call results.
- *Operators*. Operators define operations to perform on operands, for example, addition or multiplications. Operators exist for all of the basic mathematical operations in addition to some more advanced operations, such as logical comparison or the manipulation of the bits of data that constitutes a value.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

There is no limit to the length of expressions in 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, in addition to making it easier to debug your code when things don't work as you expect them to.

Examples

You can combine the basic building blocks of operators and operands to make expressions as simple or as complex as you like. At the simplest end of the scale, you can use a single operand for an expression, as the following code example shows.

```
a
```

This may not seem very useful, but is, in fact, essential. For example, if you wanted to assign a value to a variable, you would require an expression of this type.

You can build more complicated expressions by using operators, as the following code example shows.

```
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 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. This is important, because the expression in the following code example contains two integer operands, so the result is an integer.

```
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 C# compiler will convert the other operand to a double, and the result will be a double. Consequently, the expression in the following code example yields the double value 2.5.

MCT USE ONLY. STUDENT USE PROHIBITED

```
5.0 / 2
```

You can continue building up expressions with additional values and operators, as the following code example shows.

```
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. For example, the expression in the following code example uses the **+** operator to concatenate two strings.

```
"Answer: " + c.ToString()
```

The **+** operator uses an operand that is a result of a method call, **ToString()**. This 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. Later in this module, you will see how you can create your own. The **System.Math** namespace in particular contains several useful methods that you can use in expressions, as the following code example shows.

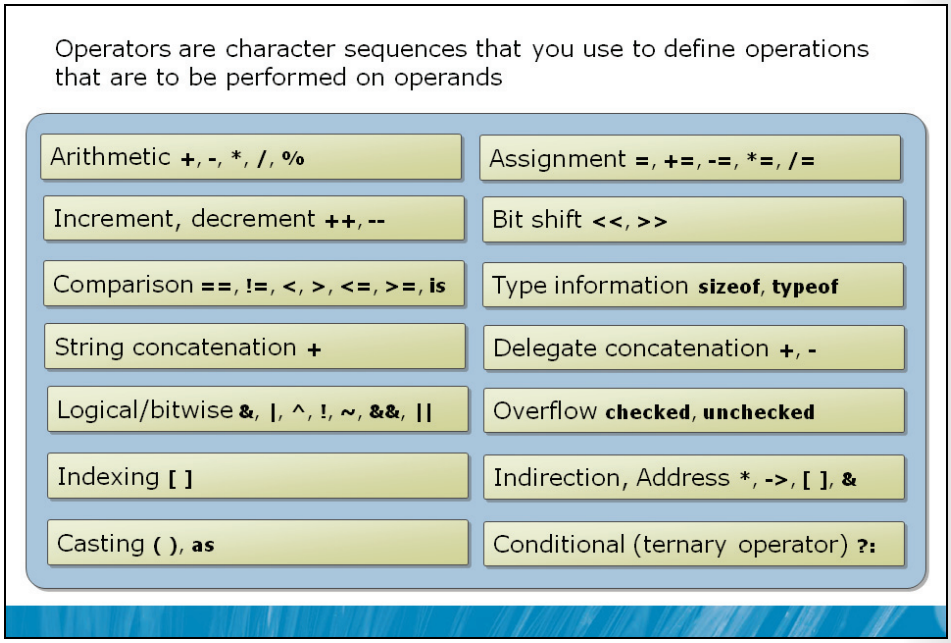
```
b * System.Math.Tan(theta)
```

This expression evaluates to the product of the variable b and the tangent of the variable theta.

Question: What is the value of the expression "99" + "1"?

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Operators?



Key Points

Operators combine operands together into expressions. C# provides a wide range of operators that you can use to perform most fundamental mathematical and logical operations.

Operator Types

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 C#. This is the **? :** operator and it is used in conditional expressions.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

C# Operators

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

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

Incrementing and Decrementing Variables

If you want to add 1 to a variable, you can use the **+** operator, as the following code example shows.

```
count = count + 1;
```

However, adding 1 to a variable is so common that C# provides its own operator just for this purpose: the **++** operator. To increment the variable count by 1, you can write the statement in the following code example.

```
count++;
```

Similarly, C# provides the -- operator that you can use to subtract 1 from a variable, as the following code example shows.

```
count--;
```

The ++ and -- operators are unary operators.

Using Compound Assignment Operators

If you want to add 42 to the value of a variable, you can combine the assignment operator and the addition operator. For example, the statement in the following code example adds 42 to a variable called answer. After this statement runs, the value of answer is 42 more than it was before.

```
answer = answer + 42;
```

However, adding a value to a variable is so common that C# lets you perform this task in a shorthand manner by using the operator **+=**. To add 42 to answer, you can write the statement in the following code example.

```
answer += 42;
```

You can use this shortcut to combine any arithmetic operator with the assignment operator, as the following table shows. These operators are collectively known as the compound assignment operators.

Replace this	With this
`variable = variable * number;`	`variable *= number;`
`variable = variable / number;`	`variable /= number;`
`variable = variable % number;`	`variable %= number;`
`variable = variable + number;`	`variable += number;`

MCT USE ONLY. STUDENT USE PROHIBITED

Replace this	With this
`variable = variable - number;`	`variable -= number;`

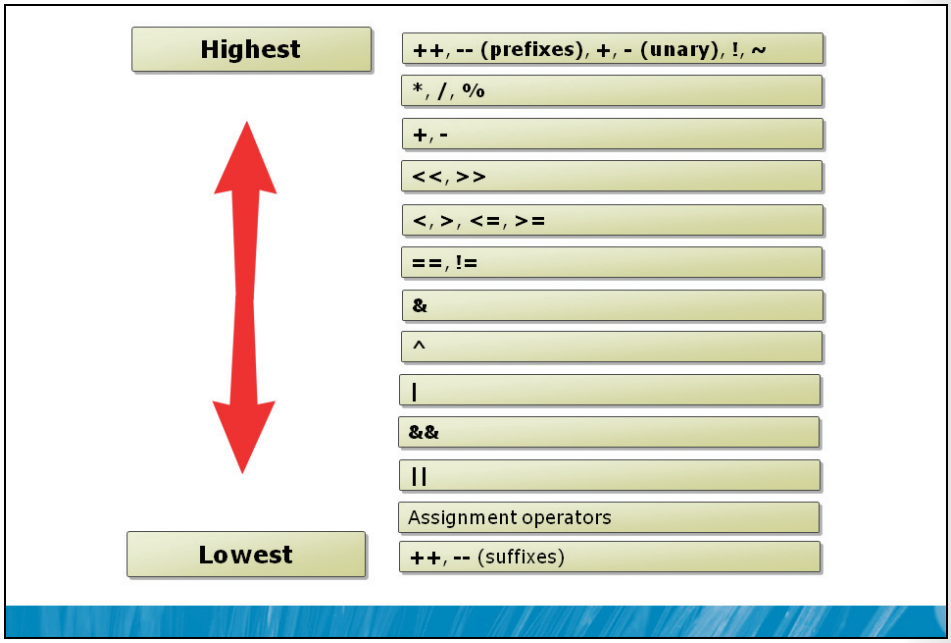
Question: Which operator would you use to calculate the remainder after dividing one integer value by another?

Additional Reading

For more information about the operator in C#, see the C# Operators page at http://go.microsoft.com/fwlink/?LinkId=192896, and the Operators (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192897

MCT USE ONLY. STUDENT USE PROHIBITED

Specifying Operator Precedence



Key Points

An expression can contain a complex series of operators and operands. The order in which the operators are processed and the operands are evaluated depends on the operators themselves. In many cases, there is not always a simple left-to-right flow of an expression.

The operators that you use to build an expression each have an associated precedence that determines the order in which they are processed. Also, operators have a particular associativity, which determines the order in which they are processed in relation to operators with a matching precedence. To make expressions work in exactly the way you want them to, you can control processing order by using parentheses.

Operator Precedence

Some operators have a higher precedence than others, which means that they are processed before other operators. For example, in the following code example, the division is performed before the addition.

MCT USE ONLY. STUDENT USE PROHIBITED

```
a = b + 1 / 2;
```

The following table shows the precedence of operators from highest at the top to lowest at the bottom.

Precedence	Operator
Highest	**++**, **--** (prefixes), **+**, **-** (unary), **!**, **~**
	*****, **/**, **%**
	+, **-**
	<<, **>>**
	<, **>**, **<=**, **>=**
	==, **!=**
	&
	^
	|
	&&
	||
	Assignment operators
Lowest	**++**, **--** (suffixes)

Operator Associativity

When you use operators of the same precedence, the operator associativity is used to determine the order of processing. Operators are either right-associative or left-associative. Left-associative operators are processed from left to right, for example, the / operator, as the following code example shows.

```
a / 5 / b
```

Here, **a** is divided by 5 and then the result of that division is divided by **b**. All binary operators are left-associative apart from assignment operators, which are right-associative, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
a = b = c
```

Here, the value of **c** is assigned to **b**, and then the value of **b** is assigned to **a**. In practice, this rarely has an effect. Also, it is worth noting that for many operators, associativity is not always important, as the following code example shows.

```
a + 5 + b
```

In this code example, there is no difference to the result if you process the expression from left to right or right to left. However, the **+** operator is still defined as left-associative, which may have an effect in more advanced situations, for example, when you overload operators.

Using Parentheses

You can use parentheses to control the order of processing and change the precedence in an expression. Any part of an expression that you surround with parentheses is processed before the part of the expression that is not inside the parentheses, as the following code example shows.

```
a = (b + 1) / 2;
```

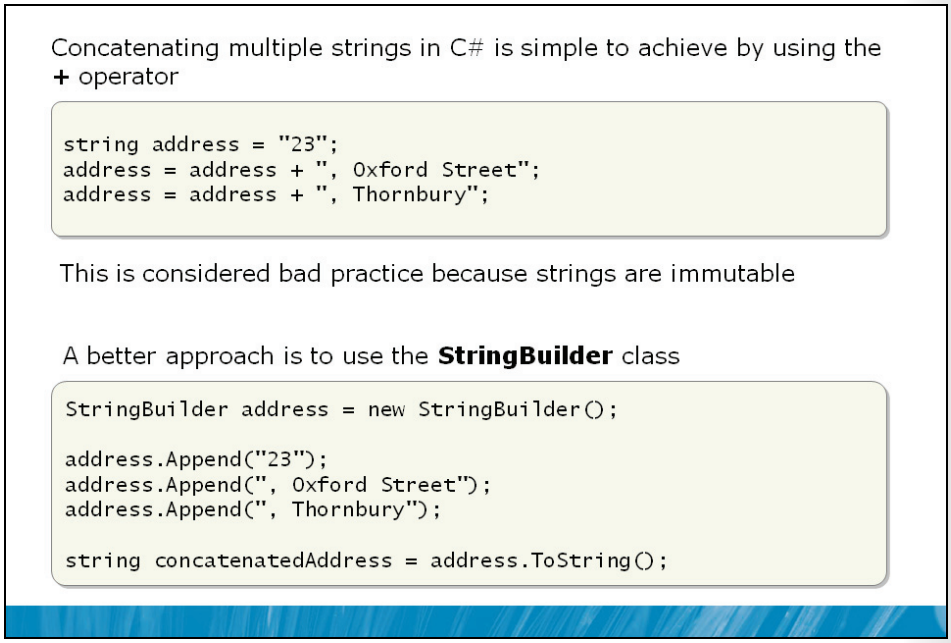
Here, the **(b + 1)** part of the expression is processed first, and the result of that operation is divided by 2 to determine the value that is assigned to **a**.

You can nest parentheses to further control the order of expression execution.

Question: How can you control the order of processing in an expression?

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices for Performing String Concatenation



Key Points

Concatenating multiple strings in C# is simple to achieve by using the **+** operator. However, this is considered bad 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. The following code example creates five string values as it runs.

```
string address = "23";

address = address + ", Oxford Street";

address = address + ", Thornbury";
```

An alternative approach would be 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.

```
StringBuilder address = new StringBuilder();

address.Append("23");
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
address.Append(", Oxford Street");

address.Append(", Thornbury");

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



Note: The **StringBuilder** class is in the **System.Text** namespace.

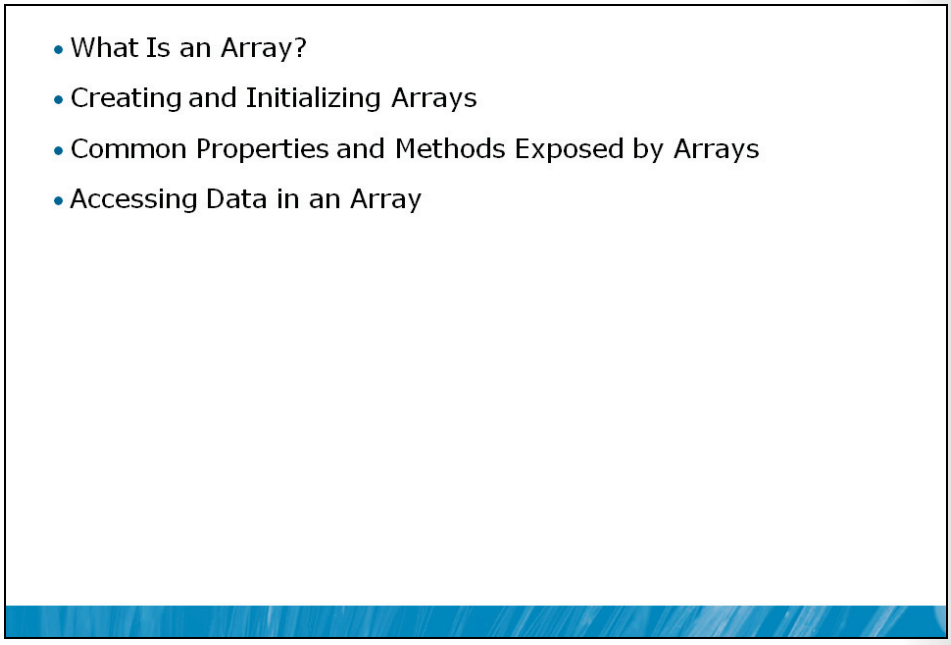
Question: Why is concatenating strings considered bad practice, and how can you avoid it?

Additional Reading

For more information about the **StringBuilder** class, see the StringBuilder Class page at http://go.microsoft.com/fwlink/?LinkId=192898.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Creating and Using Arrays



Variables hold a single value. Sometimes you need to be able to store and process a set of values, and you often do not know in advance how big this set is going to be. For example, you may have a list of customers in a database that you want to retrieve and process. Arrays enable you to read and process a variable number of related data items.

This lesson introduces arrays and explains how you can use them to store and manipulate data.

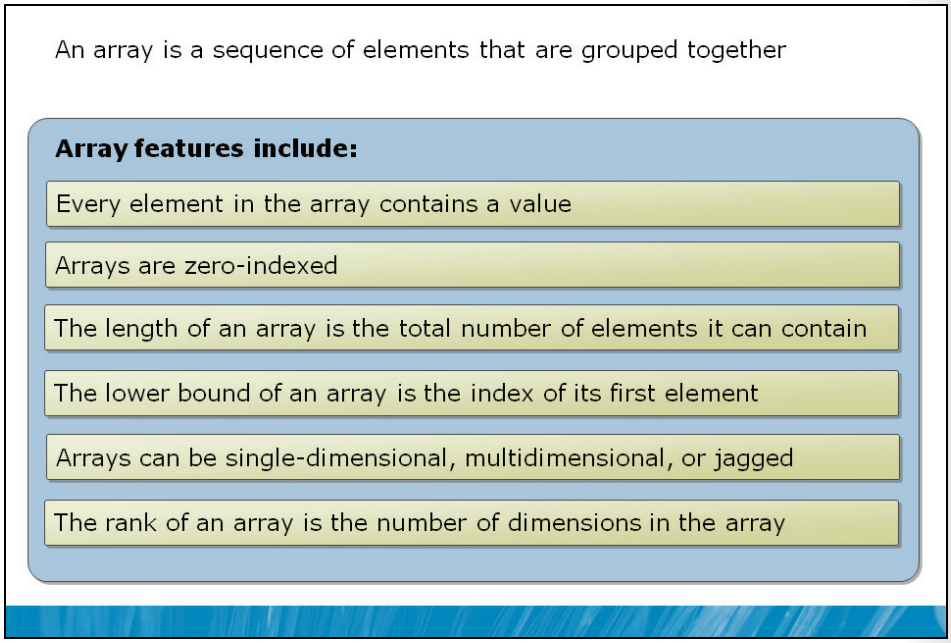
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of an array.
- Explain how to create and initialize an array.
- Describe the common properties and methods that arrays expose.
- Explain how to access data in an array.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Array?



Key Points

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 elements in an array have 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 have the following features:

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

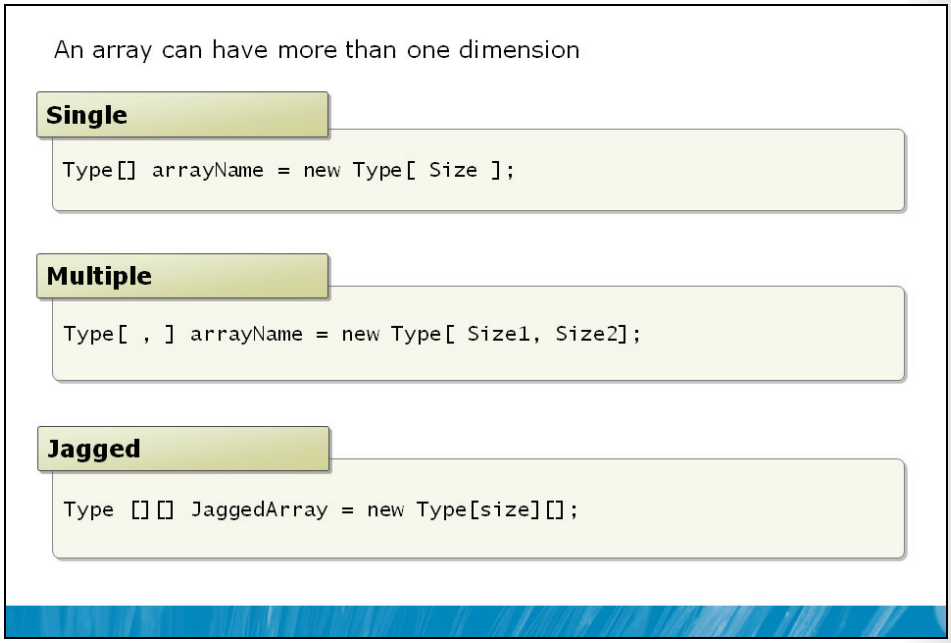
MCT USE ONLY. STUDENT USE PROHIBITED

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.

Question: What is an array, and why would you want to use arrays in a C# application?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Initializing Arrays



Key Points

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.

Single-Dimensional Arrays

To declare an array, you specify the type of elements in the array and use brackets, [], to indicate that a variable is an array. You specify the size of the array when you allocate memory for the array later by using the **new** keyword. The size of an array can be any integer expression. Alternatively, you can initialize an array and specify a set of values in braces, {}. In this case, the compiler uses the number of items in the set to determine the size of the array.

Note: If you do not initialize the elements in an array, the C# compiler initializes them for you automatically when the array is created by using the **new** keyword. The values that are used depend on the type of the elements in the array. For example, if the array

MCT USE ONLY. STUDENT USE PROHIBITED

contains numeric data, each element will be initialized to zero. If the array contains strings, each element will be initialized to the value **null**.

The following code example shows the syntax for declaring and initializing a single-dimensional array.

```
Type[] arrayName1 = new Type[ Size ];

Type[] arrayName2 = new Type{element1, element2, …, elementN};
```

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 can declare a multidimensional array variable just as you declare a single-dimensional array, but you separate the dimensions by using commas. As with a single-dimensional array, you can also specify sets of data for each dimension and the compiler will use the number of elements in a set to size the corresponding dimension. You nest sets inside braces when you initialize a multidimensional array.

The following code example shows the syntax for declaring and initializing a multidimensional array.

```
Type[ , ,   . . . ] arrayName1 = new Type[ Size1, Size2 ,   . . . ];

Type[ , ,   . . . ] arrayName2 = { {element1, element2, element3},
                                   {element4, element5, element6},
                                   ...
                                   {elementN-2, elementN-1, element}};
```

When you add dimensions to an array, the total storage of the array increases dramatically. Therefore, you should avoid declaring an array that is larger than your requirements.

Jagged Arrays

Multidimensional arrays in C# must be rectangular; the number of elements in each dimension must be the same. However, C# also supports 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.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

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

Implicitly Typed Arrays

Similar to implicitly typed variables where you declare a **var** type, and the compiler infers the type from the initializer, you can have implicitly typed arrays. When you use implicitly typed arrays, the type is inferred from the type of elements that are specified in the initializer. The elements that are specified must all be of the same type, otherwise the compiler will display an error. The following code example shows how to create an implicitly typed array.

```
var numbers = new[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

The following code example shows an example that does not compile because the initializer contains multiple elements of different data types.

```
var mixed = new[]{1, DateTime.Now, true, false, 1.2};
```

Question: How do you declare a multidimensional array?

Additional Reading

For more information about arrays, see the Multidimensional Arrays section on the Harness the Features of C# to Power Your Scientific Computing Projects page at http://go.microsoft.com/fwlink/?LinkId=192899.

For more information about single-dimensional arrays, see the Single-Dimensional Arrays (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192900.

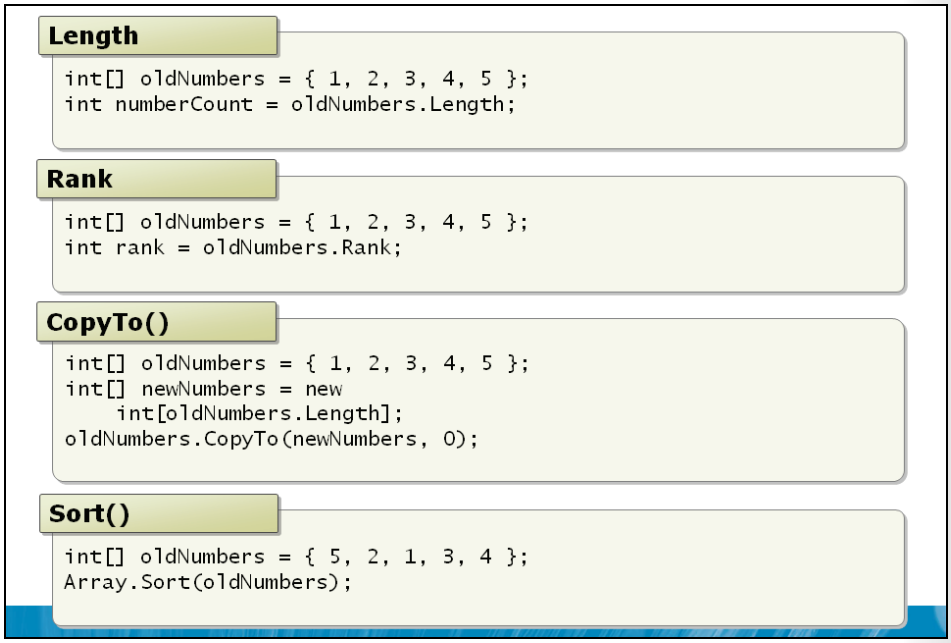
For more information about multidimensional arrays, see the Multidimensional Arrays (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192901.

MCT USE ONLY. STUDENT USE PROHIBITED

For more information about jagged arrays, see the Jagged Arrays (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192902.

MCT USE ONLY. STUDENT USE PROHIBITED

Common Properties and Methods Exposed by Arrays



Key Points

Arrays in C# are very useful for storing data and provide some useful functionality that enables you to manipulate data.

All arrays in C# are actually instances of another type called **System.Array**. The **System.Array** type provides common functionality that you can use from your own arrays. The following table describes some of the main properties and methods that arrays provide.

Member	Type	Description

BinarySearch()	Method	Enables you to search a sorted single-dimensional array for a particular value by using a binary search algorithm.
		`int[] numbers = { 1, 2, 3, 4, 5 };`
		`object searchTerm = 3;`
		`int result =`
		`Array.BinarySearch(oldNumbers,`

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Type	Description
		`searchTerm);`

Clone()	Method	Enables you to create a shallow copy of an array, which only copies the elements in the array, but does not copy objects that those elements might reference.
		`int[] numbers = { 1, 2, 3, 4, 5 };`
		`object numbersClone = numbers.Clone();`

CopyTo()	Method	Enables you to copy all elements and element references in an array to a new array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		`int[] newNumbers = new`
		`  int[oldNumbers.Length];`
		`oldNumbers.CopyTo(newNumbers, 0);`

GetEnumerator()	Method	Enables you to iterate through each of the items in sequence in an array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		
		`IEnumerator results =`
		`  oldNumbers.GetEnumerator();`
		
		`// OR`
		
		`foreach (int number in oldNumbers)`
		`{`
		`}`

GetLength()	Method	Enables you to get the length of a specific dimension in an array. `int[] oldNumbers = { 1, 2, 3, 4, 5 };`

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Type	Description
		`int count = oldNumbers.GetLength(0);`

GetValue()	Method	Enables you to get a value at a specific index in an array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		`object number = oldNumbers.GetValue(2);`
		`// returns the value 3`

Length	Property	Enables you to get the number of items in the array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		`int numberCount = oldNumbers.Length;`
		`// Returns the value 5`

Rank	Property	Enables you to get the number of dimensions in an array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		`int rank = oldNumbers.Rank;`
		`// Returns the value 1`

SetValue()	Method	Enables you to set a value at a specific index in an array.
		`int[] oldNumbers = { 1, 2, 3, 4, 5 };`
		`oldNumbers.SetValue(5000, 4);`
		`// Changes the value 5 to 5000`

Sort()	Method	Enables you to sort the elements in a single-dimensional array.
		`int[] oldNumbers = { 5, 2, 1, 3, 4 };`
		`Array.Sort(oldNumbers);`
		`// Sorted values: 1 2 3 4 5`

MCT USE ONLY. STUDENT USE PROHIBITED

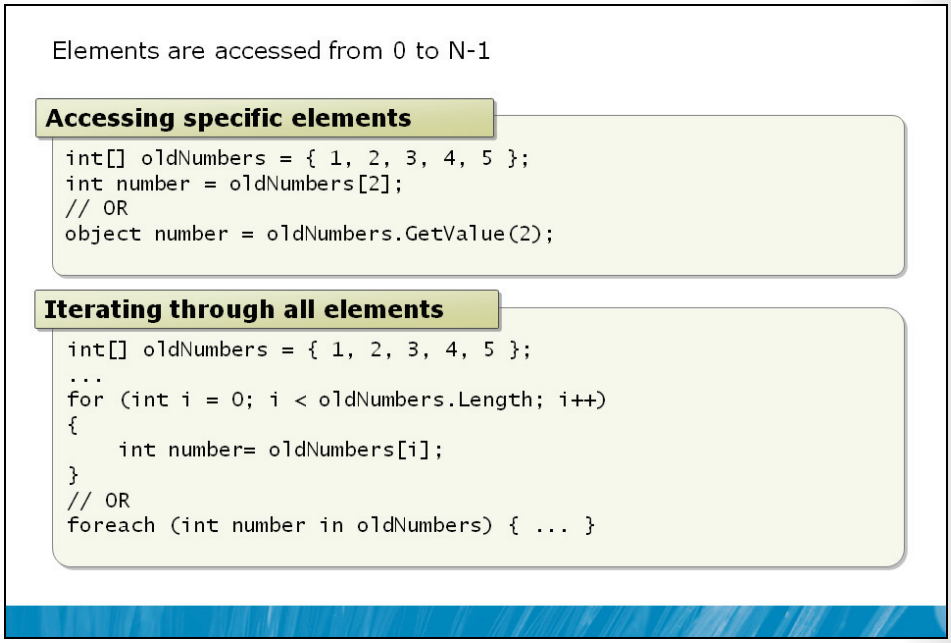
Question: What members would you use to locate the last element in an array, and then change that element's value?

Additional Reading

For more information about the **System.Array** class, see the Array Class page at http://go.microsoft.com/fwlink/?LinkId=192903.

MCT USE ONLY. STUDENT USE PROHIBITED

Accessing Data in an Array



Key Points

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

Accessing Specific Elements

You can access specific elements in two ways by using an index that specifies the element that you want to return. Note that 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.

The following code example uses an index to access the element at index two. Remember that arrays use zero-based indexes, so this example returns the value **3**.

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

MCT USE ONLY. STUDENT USE PROHIBITED

Iterating Through All Elements

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, as the following code example shows.



Note: The **for** statement is described in more detail later in this module.

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

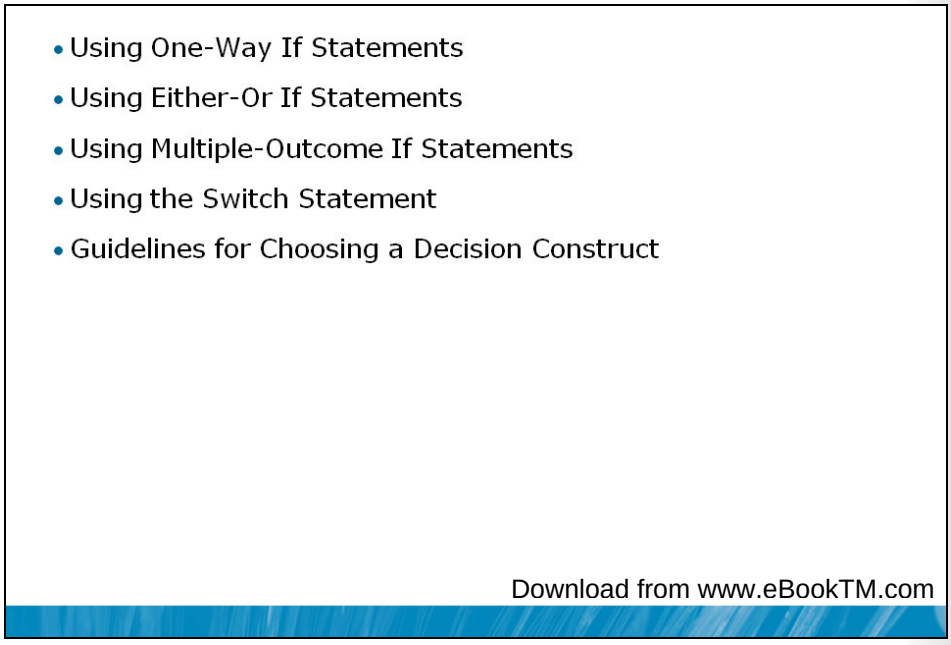
An alternative approach is to use the **foreach** loop (which is covered in more detail in Module 12). The **foreach** statement automatically retrieves all of the elements from the array in index order and assigns them to a variable that is specified in the **foreach** construct.

```
int[] oldNumbers = { 1, 2, 3, 4, 5 };
foreach (int number in oldNumbers)
{
    ...
}
```

Question: Explain two approaches to accessing data in an array.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Using Decision Statements



By default, C# performs the statements in a program in a sequential manner. However, you frequently need to specify that alternative statements should run depending on the value of an expression or a Boolean condition. To achieve this, C# provides conditional decision statements.

This lesson introduces the different types of decision statements, and explains how you can use them in your .NET Framework applications.

Objectives

After completing this lesson, you will be able to:

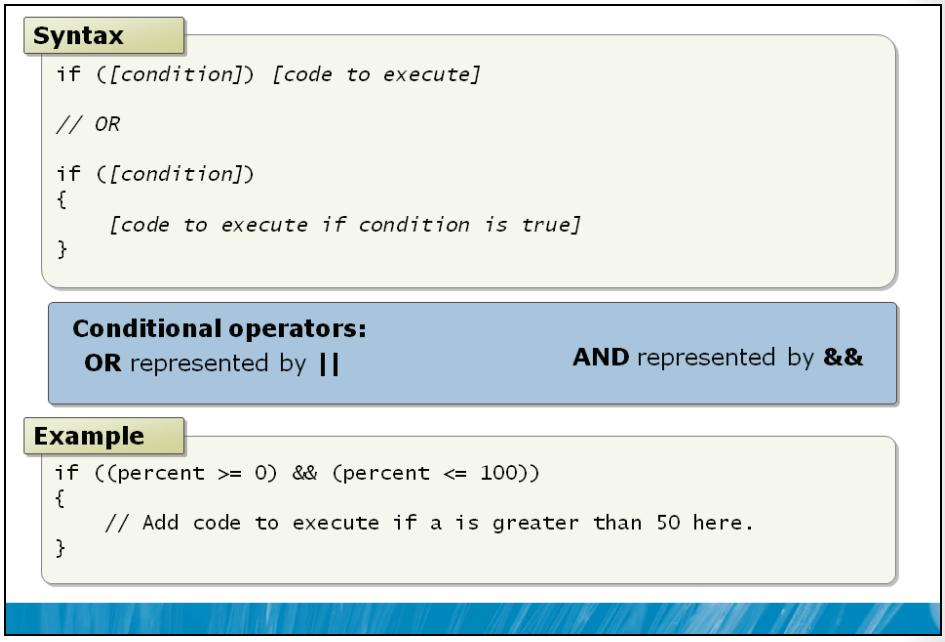
- Explain how to use the **if else** statement.
- Explain how to use the **?:** operator.
- Explain how to use the **switch** statement.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe when you should use each of the different decision constructs that are available in C#.
- Describe the guidelines that will help you decide when to choose a particular decision construct.

MCT USE ONLY. STUDENT USE PROHIBITED

Using One-Way If Statements



Key Points

One-way **if** statements are very useful when you want to execute a code statement based on a condition. The basic syntax for a one-way **if** statement is shown in the following code example.

```
if ([condition]) [code to execute]
```

In this code, if the expression ***[condition]*** evaluates to a **Boolean** true value, ***[code to execute]*** is executed. Notice that the condition must be enclosed in parentheses.

You can execute more than one code statement. To do this, you delimit the code to run by using braces. This extends the syntax as the following code example shows.

```
if ([condition])
{
    [code to execute if condition is true]
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

It is standard practice to use this format even if you only execute a single line of code when ***[condition]*** is true, because it makes your code both easier to read and to extend.

For example, if you want to execute code when a variable a has a value of more than 50, you can use the code in the following code example.

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
```

Using the Conditional Logical Operators

C# also provides two Boolean operators: the logical **AND** operator, which is represented by **&&**, and the logical **OR** operator, which is represented by **||**. Collectively, these are known as the conditional logical operators. Their purpose is to combine two Boolean expressions or values into a single Boolean result. These binary operators are similar to the equality and relational operators in that the value of the expressions in which they appear is either true or false. However, they differ in that the values on which they operate must be either true or false.

The outcome of the **&&** operator is true only if both of the Boolean expressions on which it operates are true. For example, the statement in the following code example assigns the value **true** to **validPercentage** only if the value of **percent** is greater than or equal to 0 and the value of **percent** is less than or equal to 100.

```
bool validPercentage;
if (percent >= 0) && (percent <= 100)
{
  validPercentage = true;
}
```



Note: You can achieve the same result by assigning the value of the Boolean expression directly to the validPercentage variable, as the following code example shows.

```
validPercentage = (percent >= 0) && (percent <= 100);
```

The outcome of the **||** operator is true if either of the Boolean expressions on which it operates is true. You use the **||** operator to determine whether any one of a combination of Boolean expressions is true. For example, the statement in the following code example assigns the value true to **invalidPercentage** if the value of **percent** is less than 0 or the value of **percent** is greater than 100.

MCT USE ONLY. STUDENT USE PROHIBITED

```
bool invalidPercentage;
if ((percent < 0) || (percent > 100))
{
  invalidPercentage = true;
}
```

Sometimes, when you evaluate an expression that uses the **&&** and **||** operators, it is not necessary to evaluate both operands to determine the overall result. For example, in the following code example, if the value of the age variable is greater than or equal to 20, the value of the entire expression is false, regardless of whether the value of the height variable is greater than 180.

```
(age < 20) && (height > 180)
```

Similarly, in the following code example, if the price variable has a value greater than or equal to 25, the value of the entire expression is true, regardless of whether the value of the weight variable is greater than 100.

```
 (price >= 25) || (weight > 100)
```

The **&&** and **||** operators in C# recognize these situations, and in cases such as this, evaluation of the operands stops as soon as the result can be determined. What this means is that the expression **(height > 180)** in the first case and the expression **(weight > 100)** in the second case will not be evaluated. This behavior is known as short-circuiting.

Question: When must you enclose the code in the body of an **if** statement in braces?

MCT USE ONLY. STUDENT USE PROHIBITED

Using Either-Or If Statements

Provide an additional code block to execute if *[condition]* evaluates to a Boolean false value

Example

```
if (a > 50)
{
    // Greater than 50 here
}
else
{
    // less than or equal to 50 here
}

// OR

string carColor = "green";

string response = (carColor == "red") ?
    "You have a red car" :
    "You do not have a red car";
```

Key Points

To provide an additional code block to execute only if ***[condition]*** evaluates to false, you use the **else** keyword, as the following code examples show.

```
if ([condition])
{
    [code to execute if condition is true]
}

else
{
    [code to execute if condition is false]
}
```

```
if (a > 50)

{
    // Add code to execute if a is greater than 50 here.
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
else
{
    // Add code to execute if a is less than or equal to 50 here.
}
```

Using the ?: Operator

As an alternative to using the **if else** statements, in some simple cases, you can use the **?:** ternary operator.

The basic syntax to use the **?:** operator is shown in the following code example.

```
Type result = [condition] ? [true expression] : [false expression]
```

In this code, if the expression ***[condition]*** evaluates to true, ***[true expression]*** is executed, but if the ***[condition]*** evaluates to false, ***[false expression]*** is executed.

The following code example shows an example of using the **?:** operator to check the value of a string, and then return a response.

```
string carColor = "green";

string response = (carColor == "red") ?
    "You have a red car" :
    "You do not have a red car";
```

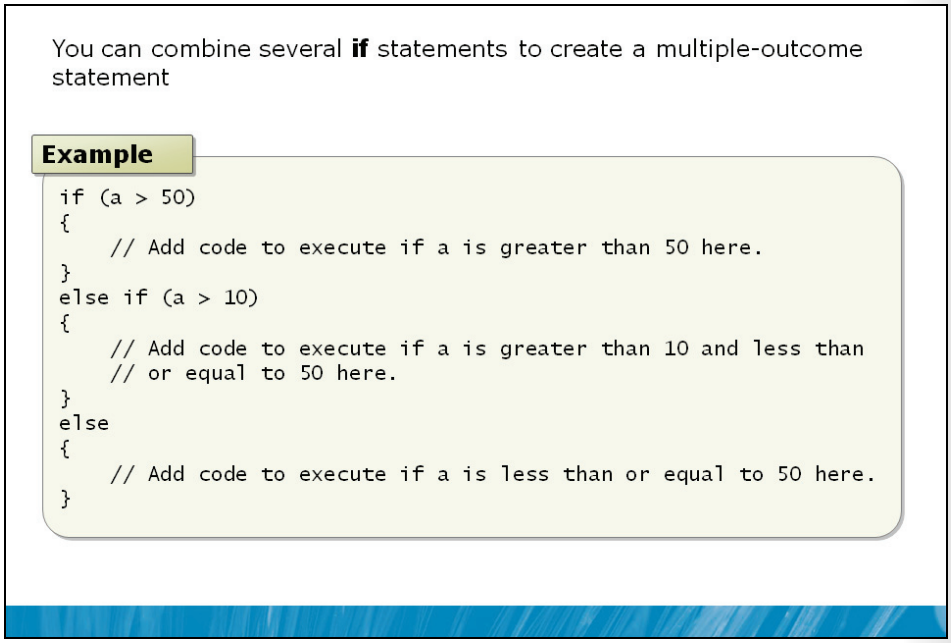
Question: Think of a scenario where you may want to use the **if else** statement and discuss.

Additional Reading

For more information about the **?:** operator, see the ?: Operator (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192904.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Multiple-Outcome If Statements



Key Points

You can combine several **if** statements to create a multiple-outcome statement. The following code example shows an example of the syntax.

```
if ([condition])
{
    [code to execute if condition is true]
}
else if ([condition2])
{
    [code to execute if condition is false and condition2 is true]
}
else
{
    [code to execute if condition and condition2 are both false]
}
```

It is important to note that if ***[condition]*** is true, the first block of code is executed, regardless of the value of ***[condition2]***. If this is the case, the remaining code is skipped, and ***[condition2]*** is not evaluated. This has performance consequences

MCT USE ONLY. STUDENT USE PROHIBITED

because it takes time for each condition to be evaluated. You can streamline your code by ensuring that the most commonly fulfilled condition or the condition that takes least processing to evaluate is tested first.

The following code example shows an example of a multiple-outcome statement that uses this structure.

```
if (a > 50)
{
    // Add code to execute if a is greater than 50 here.
}
else if (a > 10)
{
    // Add code to execute if a is greater than 10 and less than or
    // equal to 50 here.
}
else
{
    // Add code to execute if a is less than or equal to 50 here.
}
```

Note that the **else** statement must always come after all of the **else if** statements.

Question: What is the purpose of the **else** statement in an **else if** construct?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Switch Statement

The **switch** statement enables you to execute one of several blocks of code depending on the value of a variable or expression

Example

```
switch (a)
{
    case 0:
        // Executed if a is 0.
        break;

    case 1:
    case 2:
    case 3:
        // Executed if a is 1, 2, or 3.
        break;

    default:
        // Executed if a is any other value.
        break;
}
```

Key Points

The **switch** statement enables you to execute one of several blocks of code depending on the value of a variable or expression. These code blocks provide a very simple, easy-to-read structure and offer an alternative approach to using **if else** statements.

Switch Statement Syntax

The basic syntax for the **switch** statement is shown in the following code example.

```
switch ([expression to check])
{

    case [test1]:
        ...
        [exit case statement]

    case [test2]:
        ...
        [exit case statement]
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    default:
        ...
        [exit case statement]
}
```

In a **switch** statement, you specify the expression to check in ***[expression to check]***, and supply values to compare with the variable in ***[testX]***. Each comparison is tested in turn, so if ***[expression to check]*** equals ***[test1]***, the first code block is executed, if ***[expression to check]*** equals ***[test2]***, the second code block is executed, and so on. There is no limit to the number of comparisons that you can include here, other than the memory of your computer. If no match is made, the block of code that is specified by **default:** is executed. The **default** block is optional.

The type of value that ***[expression to check]*** returns must be an integer, string, or Boolean, and the values that are specified by the case statements must match this type.

Each comparison (***[testX]***) is a single value. You can also check for multiple values by using multiple consecutive **case** statements, as the following code example shows.

```
switch (a)
{
    case 0:
        // Executed if a is 0.
        break;

    case 1:
    case 2:
    case 3:
        // Executed if a is 1, 2, or 3.
        break;

    default:
        // Executed if a is any other value.
        break;
}
```

Every block of code in a **switch** statement must end with a statement that explicitly terminates the construct (shown as ***[exit case statement]*** in the earlier example). If you omit this statement, your code will not compile. You can use the following statements:

- **break;**. This statement terminates processing of the selection statement.

MCT USE ONLY. STUDENT USE PROHIBITED

- **goto case** ***[testX]***;. This statement causes execution to jump to the specified block of code in the **switch** statement.
- **return**;. This statement causes the **switch** statement and its containing method to terminate. You can pass return values with this statement.

The recommended approach is to use the **break** statement whenever possible. Using **goto** or **return** can lead to code that is difficult to maintain.

The following code example shows an example of using a **switch** statement to check the value of a string.

```
switch (carColor.ToLower())
{
    case "red":
        // Red car
        break;

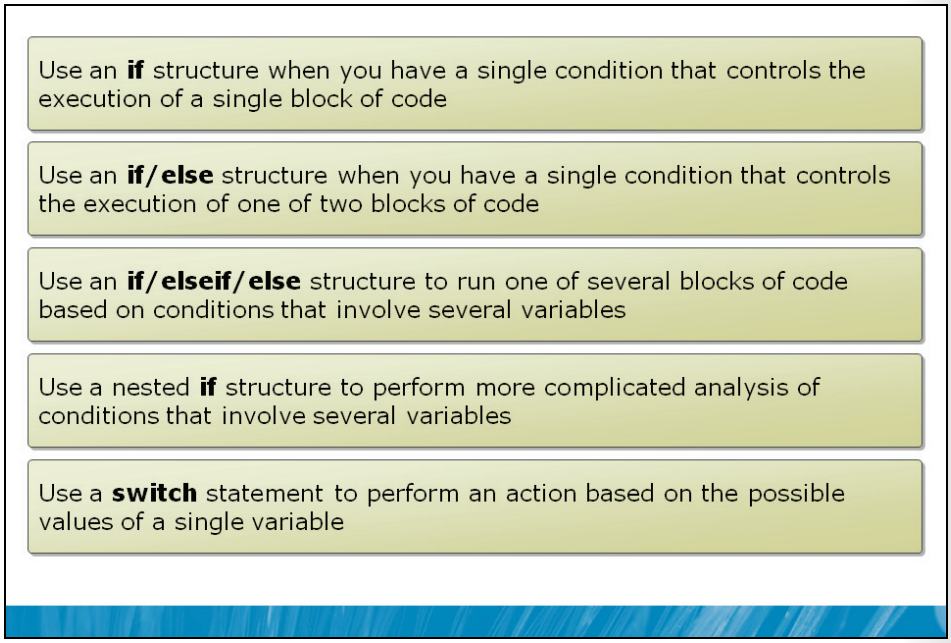
    case "blue":
        // Blue car
        break;

    default:
        // Unknown car
        break;
}
```

Question: With the exception of the default case, is the order of the cases in a **switch** statement important?

MCT USE ONLY. STUDENT USE PROHIBITED

Guidelines for Choosing a Decision Construct



Key Points

In this lesson, you have seen several structures that you can use to implement conditional statements. You should choose which structure to use based on the functionality that you want to implement.

The guidelines for choosing a decision structure are as follows:

- Use an **if** structure when you have a single condition that controls the execution of a single block of code.

  A typical example of this is after you receive a user response to a yes/no question. Your code can use an **if** structure to execute a block of code if the user responds with "yes."

- Use an **if/else** structure when you have a single condition that controls the execution of one of two blocks of code.

  If you prompt a user to choose between two alternatives, this is the structure to use.

MCT USE ONLY. STUDENT USE PROHIBITED

- Use an **if/elseif/else** structure to run one of several blocks of code based on conditions that involve several variables.

  A good example of this is to check x-coordinates and y-coordinates for points that are in defined rectangular areas of a surface. You can use an expression for each rectangular area in each condition. You can also use this structure to check whether a single variable has a value in a certain range or ranges.

- Use a nested **if** structure to perform more complicated analysis of conditions that involve several variables.

  This structure gives you the greatest flexibility, but often leads to code that is difficult to read, with several levels of indentation. You can use this structure to test multiple variables and conditions to provide a multiple outcome structure.

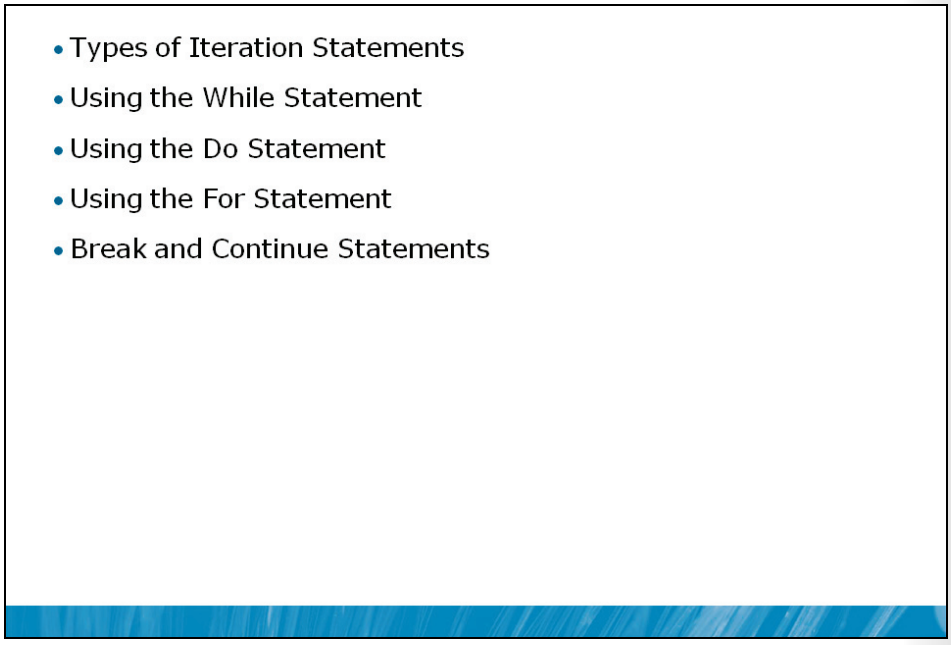
- Use a **switch** statement to perform an action based on the possible values of a single variable.

  You can use this structure instead of a nested **if** structure to make your code clearer when you are testing the value of a single variable.

Question: Which statement would you use to perform an action based on the possible values of a single variable?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 5
Using Iteration Statements



When you are writing the logic for your .NET Framework applications, it is common for you to want to repeatedly execute a section of logic either a set amount of times, or until a condition is met. To achieve this, you can use the iteration statements that C# provides.

This lesson introduces the three main iteration statements that are available in C# and explains how you can use them in your applications.

Objectives

After completing this lesson, you will be able to:

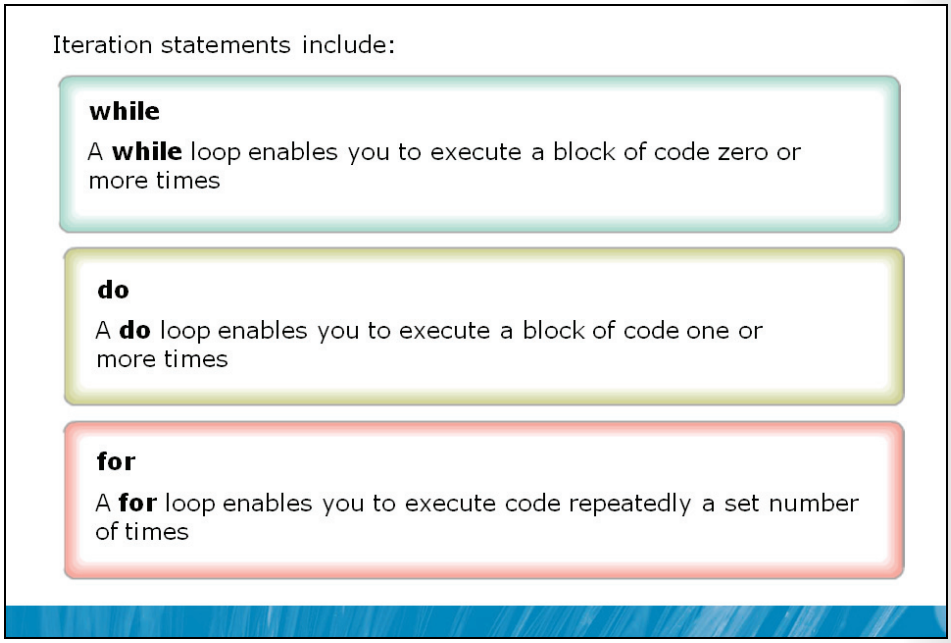
- Describe the types of iteration statement that are available in C#.
- Explain how to use the **while** statement.
- Explain how to use the **do** statement.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain how to use the **for** statement.
- Describe the difference between the **break** and **continue** statements in C#.

MCT USE ONLY. STUDENT USE PROHIBITED

Types of Iteration Statements



Key Points

There are three types of iteration statement that you can use in C# applications. Each of these statements works in a slightly different way and has a distinct purpose.

While Loops

A **while** loop enables you to execute a block of code zero or more times. **While** loops do not use a counter variable, although you can implement a counter variable by defining it outside the loop and manipulating it for each iteration.

At the start of each iteration of a **while** loop, a Boolean condition is checked. If this condition evaluates to **true**, an iteration begins. If the condition evaluates to **false**, the loop terminates.

While loops can be very useful if you do not know in advance whether you must perform iterative processing on a variable.

MCT USE ONLY. STUDENT USE PROHIBITED

Do Loops

Do loops are exactly like **while** loops apart from one detail. In a **do** loop, the condition is evaluated at the end of the iteration instead of at the start. This means that a **do** loop always executes at least once, unlike a **while** loop, which might not execute at all.

Do loops are very useful when you do not know in advance how many times your code needs to execute. For example, you can use a **do** loop to prompt a user repeatedly until the user provides valid input.

For Loops

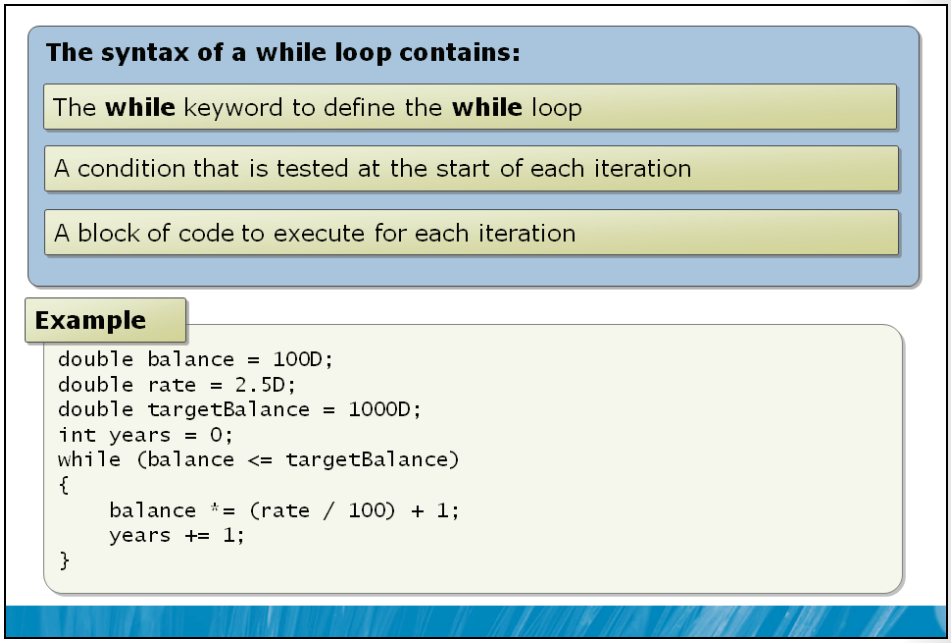
A **for** loop enables you to execute code repeatedly a set number of times. To achieve this, you define a counter variable for the loop, the value of which is changed for each iteration. When the counter variable reaches a limit value that you define, the loop terminates.

The code in the body of a **for** loop can use the value of the counter variable. This means, for example, that you can use a **for** loop to process each member of an array. You can also nest **for** loops with different counters so that you can process multidimensional arrays or examine pixels at specified coordinates.

Question: Which iteration statement would you use to prompt a user for a valid response?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the While Statement



Key Points

A **while** loop enables you to execute a block of code zero or more times. At the beginning of each iteration, the **while** loop evaluates an expression. If this expression is true, the next iteration begins. If it is false, the loop terminates.

While Loop Syntax

The syntax of a **while** loop contains the following elements:

- The **while** keyword to define the **while** loop.
- A condition that is tested at the start of each iteration.
- A block of code to execute for each iteration.

The following code example shows the syntax of a **while** loop.

```
while ([condition])
{
    // Code to loop.
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

[condition] can be any expression that evaluates to a Boolean value. Each time a iteration begins, including the first time that the **while** loop is encountered, the expression is evaluated. If the expression is true, the iteration executes; otherwise, the loop is terminated.



Note: The condition is evaluated once for each iteration, before the iteration begins. The condition is not monitored while the iteration executes, so the last iteration is always completed before the loop terminates.

Examples

The following code example shows a simple calculation that you could use to determine how many years it would take a bank balance to exceed a specified value with a specified interest rate.

```
double balance = 100D;
double rate = 2.5D;
double targetBalance = 1000D;
int years = 0;
while (balance <= targetBalance)
{
    balance *= (rate / 100) + 1;
    years += 1;
}
```

In this code, the condition **balance <= targetBalance** is checked before each iteration. If **balance** is more than **targetBalance** before the loop starts, no iterations will execute and **years** will remain at its default value of 0.

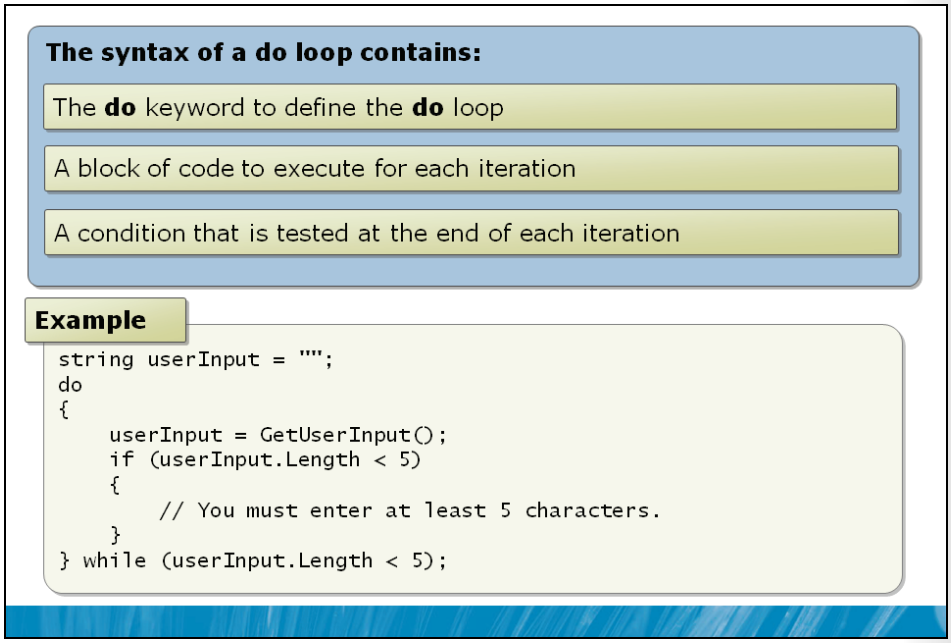


Note: When you use the **while** loop, your code must change the Boolean condition; otherwise, your loop will iterate an infinite number of times.

Question: When using the **while** loop, what type must the condition expression evaluate to?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Do Statement



Key Points

A **do** loop enables you to execute a block of code one or more times. At the end of each iteration, the **do** loop evaluates a Boolean expression. If this expression is true, another iteration begins. If it is false, the loop is terminated.

Do Loop Syntax

The syntax of a **do** loop contains the following elements:

- The **do** keyword to define the **do** loop.
- A block of code to execute for each iteration.
- A condition that is tested at the end of each iteration.

The following code example shows the syntax of a **do** loop.

```
do
{
    // Code to loop.
} while ([condition]);
```

MCT USE ONLY. STUDENT USE PROHIBITED

As with **while** loops, ***[condition]*** can be any expression that evaluates to a Boolean value. Each time an iteration ends, the expression is evaluated. If the expression is true, the next iteration executes; otherwise, the loop ends.

Examples

A typical use of a **do** loop is to prompt a user for input and then continue to prompt the user if the input is invalid. The following code example illustrates this with code that requires a string that is at least five characters long.

```
string userInput = "";
do
{
    userInput = GetUserInput();
    if (userInput.Length < 5)
    {
        // You must enter at least 5 characters.
    }
} while (userInput.Length < 5);
```

In this code, a method called **GetUserInput()** obtains the user input and returns it as a string. The code for this method is not shown here.

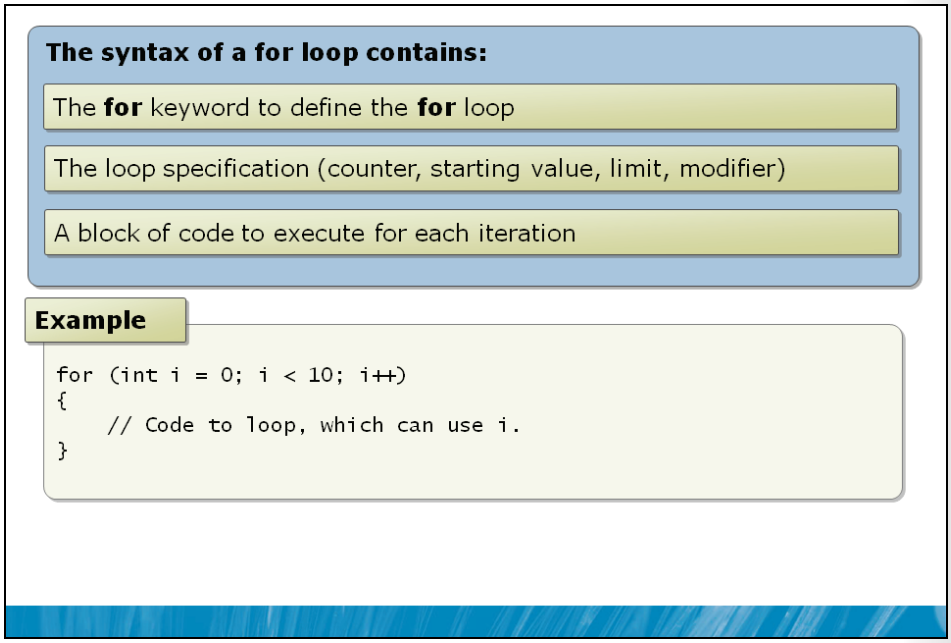


Note: When you use the **do** loop, your code must change the Boolean condition; otherwise, your loop will iterate an infinite number of times.

Question: What is the minimum number of iterations that a **do** loop will perform?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the For Statement



Key Points

A **for** loop enables you to execute a block of code repeatedly and track the number of iterations that are performed by using a counter variable.

For Loop Syntax

The syntax of a **for** loop contains the following elements:

- The **for** keyword to define the **for** loop.
- The loop specification, which consists of the following elements:
  a. A numeric variable to use for the counter (this can be a variable that is already defined or a variable that is defined as part of the loop specification).
  b. A starting value for the counter variable.
  c. A limit for the counter variable.
  d. Instructions for how to modify the counter variable at the end of each iteration.

MCT USE ONLY. STUDENT USE PROHIBITED

- A block of code to execute for each iteration.

The syntax is shown in the following code example.

```
for ([counter variable] = [starting value]; [limit]; [counter
modification])
{
    // Code to loop.
}
```

The following table explains the purpose of the placeholders in this code.

Placeholder	Usage
[counter variable]	The identifier of an existing numeric variable, or a definition for a new numeric variable.
[starting value]	A number to assign to the counter variable for the first iteration.
[limit]	A condition to be tested at the start of each iteration. If the condition evaluates to **true**, the loop continues. If it evaluates to **false**, the loop ends.
[counter modification]	An operation to perform at the end of each iteration.

Examples

The following code example shows a simple **for** loop that performs 10 iterations. The variable i is created and set to 0 for the first iteration, and incremented at the end of each iteration. When the value of i reaches 10, the loop terminates (the loop does not run when i is 10).

```
for (int i = 0; i < 10; i++)
{
    // Code to loop, which can use i.
}
```

The following code example extends this code to use a step of 2 instead of 1.

```
for (int i = 0; i < 10; i += 2)
{
    // Code to loop, which can use i.
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

This code loops five times, with values of i for each iteration of 0, 2, 4, 6, and 8.

In these code examples, the control variable, i, is created as part of the **for** construct. The scope of i is the body of the **for** loop. When the loop finishes, i is no longer available. If you need to examine the value of the control variable outside the loop, you can declare a variable before the loop starts and use that, as the following code example shows.

```
int j;
for (j = 0; j < 10; j++)
{
    // Code to loop, which can use j.
}
// j is also available here
```

You can use nested **for** loops that each define their own counter variable. This idiom is useful if you need to process multidimensional arrays. The following code example shows how to use two nested **for** loops to process the characters in an array of strings in reverse order.

```
string[] strings = new string[] {"One", "Two", "Three", "Four",
"Five"};
string result = "";
for (int stringIndex = 0; stringIndex < strings.Length; stringIndex++)
{
    for (int charIndex = strings[stringIndex].Length - 1;
        charIndex >= 0; charIndex--)
    {
        result += strings[stringIndex][charIndex];
    }
}
```

After the variables are initialized, the outer **for** loop iterates with counter values of 0, 1, 2, 3, and 4 (**strings.Length** is 5). For each value of this counter, the corresponding string in the array is used to determine the starting value for the counter of the inner loop.

The inner loop iterates through the string character by character, starting at the end of the string and working backwards (the charIndex control variable is set to the length of the string and decremented at the end of each iteration. The loop stops when charIndex is less than zero). The body of the inner loop retrieves the character that charIndex indexed from the string referenced by **stringIndex** in the array. Note that you can retrieve individual characters from a string by using array-like index access. When every character in the string has been processed, the first iteration of the outer loop finishes, and the outer loop begins its second iteration. This process continues until every character in every string is processed.

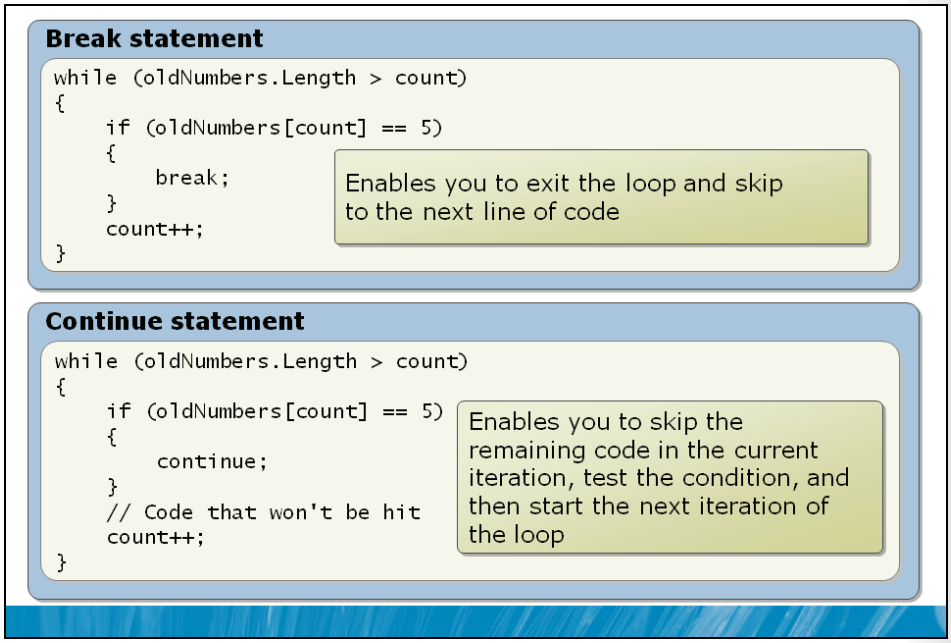
MCT USE ONLY. STUDENT USE PROHIBITED

The value of **result** when this code finishes is the string **enOowTeerhTruoFeviF**.

Question: What are the four components of a **for** loop?

MCT USE ONLY. STUDENT USE PROHIBITED

Break and Continue Statements



Key Points

When you use the **while**, **do**, and **for** loop constructs, you can also use the **break** and **continue** statements to modify the behavior of the loop.

Note: Use **break** and **continue** with caution. They can lead to code that is difficult to understand and maintain.

The Break Statement

The **break** statement enables you to exit the loop entirely, and skip to the next line of code outside the loop. The **break** statement is particularly useful if you are iterating through an array looking for a record, and you want to exit the loop when you have found the record.

Note: Do not confuse the use of **break** in a loop with **break** in a **switch** statement.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows how to exit a **while** loop if the value **5** is found in an array.

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };

int count = 0;

while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        break;
    }
    count++;
}
```

The **break** statement produces identical behavior when used with the **while**, **do**, and **for** loops.

The Continue Statement

The **continue** statement is similar to the **break** statement except that, instead of exiting the loop entirely, you skip the remaining code in the current iteration, test the condition, and then start the next iteration of the loop. The following code example shows how to add additional logic to a **while** loop that will not execute when the value **5** is found.

```
int[] oldNumbers = { 1, 2, 3, 4, 5, 6, 7, 8 };

int count = 0;

while (oldNumbers.Length > count)
{
    if (oldNumbers[count] == 5)
    {
        continue;
    }
    // Code that won't be hit when the value 5 is found
    count++;
}
```

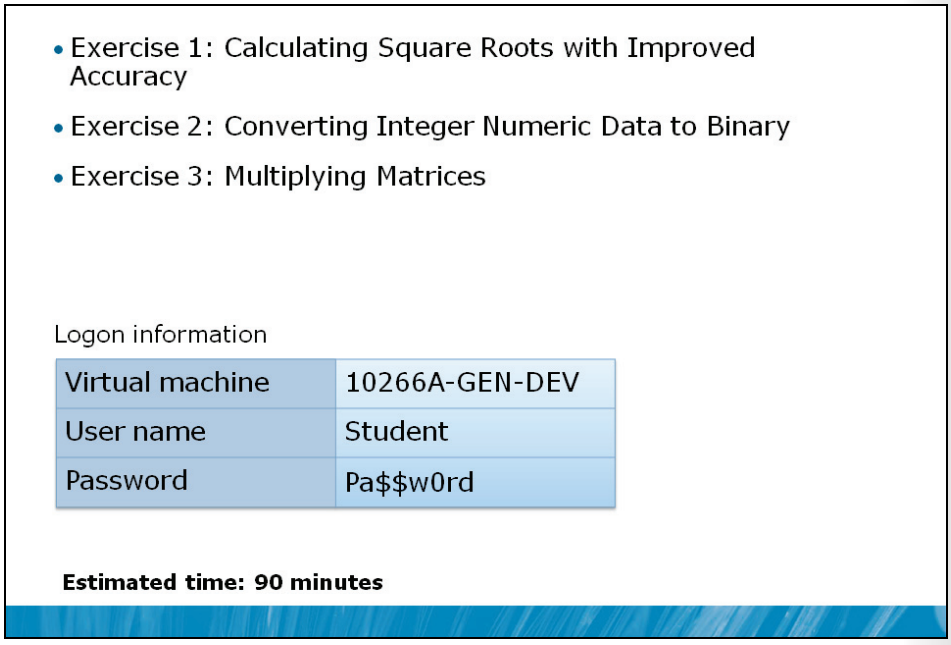
The **continue** statement produces identical behavior when it is used with the **while** and **do** loops. The only subtle difference is that, when it is used with the **for** loop, the remaining code in the current iteration is skipped as with the other loops, but the modifier in the **for** specification is incremented before the condition is tested, and the next iteration begins.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What is the difference between the **break** and **continue** statements?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Using C# Programming Constructs



Objectives

After completing this lab, you will be able to:

- Use C# data types and expressions to help implement a numeric algorithm.
- Use C# programming constructs to perform common programming tasks.
- Use arrays to store and process data.

Introduction

In this lab, you will create several applications that implement some common algorithms. This will help you to become familiar with using the C# syntax and learn many of the core C# programming constructs.

Important: The purpose of these exercises, and the remaining exercises throughout this course, is not to make you familiar with mathematical algorithms or engineering

MCT USE ONLY. STUDENT USE PROHIBITED

processes. Rather, the aim is to enable you to take a description of a problem or algorithm and use C# to implement a solution.

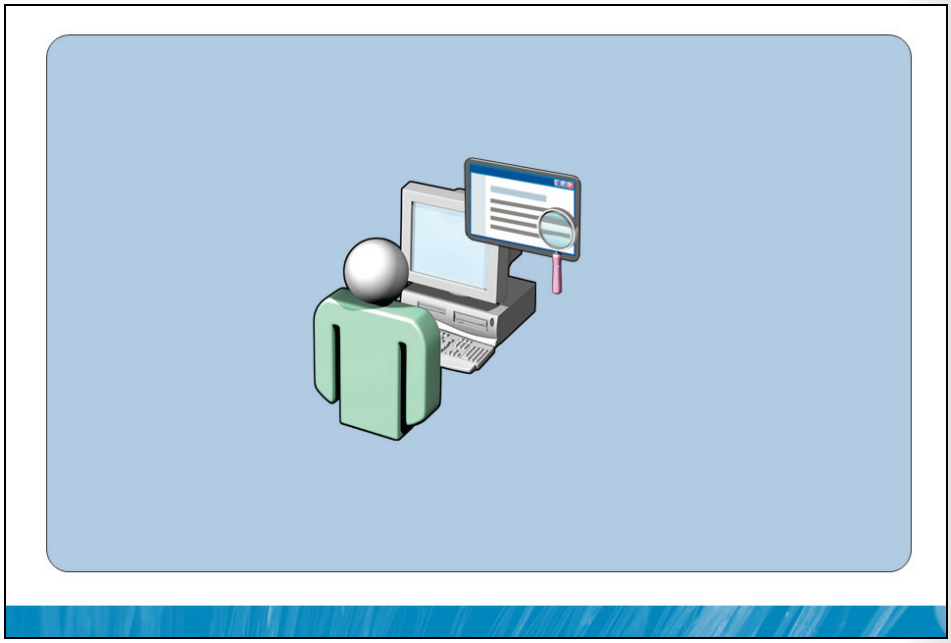
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data. You have been asked to implement some embedded functionality that several scientific instruments require. You will write C# applications to build and test your implementations.

Exercise 1: Calculating Square Roots with Improved Accuracy

In this exercise, you will write a program that prompts the user for a numeric value and then uses Newton's method to calculate the square root of this number. You will display the result, and compare it to the **double** value that is calculated by using the **Math.Sqrt** method in the .NET Framework class library.

Scenario

Some of the software that is being developed to support devices that perform scientific analysis requires applications to perform calculations with a high degree of accuracy. The .NET Framework uses the **double** type to perform many of its calculations. The **double** type has a very large range, but the accuracy is not always sufficient. The **decimal** type provides a higher degree of accuracy at the cost of a

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

smaller range and increased memory requirements. However, this accuracy is important. One scientific calculation requires the ability to calculate square roots to a high degree of accuracy. You decide to implement Newton's algorithm for estimating and successively refining square roots, but generate the result by using the **decimal** type.

The process that Newton used for calculating the square root of 10 is as follows:

1. Start with an initial guess: use the value that you want to find the square root of and divide by 2.

   In this case, 10 / 2, has the value 5.

2. Refine the guess by dividing the original number by the previous guess, adding the value of the previous guess, and dividing the entire result by 2: calculate ((number / guess) + guess) / 2.

   In this example, calculate ((10 / 5) + 5) / 2 = 3.5

   The answer 3.5 then becomes the next guess.

3. Perform the calculation ((number / guess) + guess) / 2 again, with the new guess

   In this example, calculate ((10 / 3.5) + 3.5) / 2 = 3.17857

   3.17857 is then the next guess.

4. Repeat this process until the difference between subsequent guesses is less than some predetermined amount. The final guess is the square root of 10 to the accuracy that was specified by this predetermined amount.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Calculate square roots by using the **Math.Sqrt** method of the .NET Framework.
4. Calculate square roots by using Newton's method.
5. Test the application.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Create a new WPF Application project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Microsoft Visual Studio® 2010.
3. Create a new project called SquareRoots by using the Windows® Presentation Foundation (WPF) Application template in the E:\Labfiles\Lab 2\Ex1\Starter folder.

▶ Task 2: Create the user interface

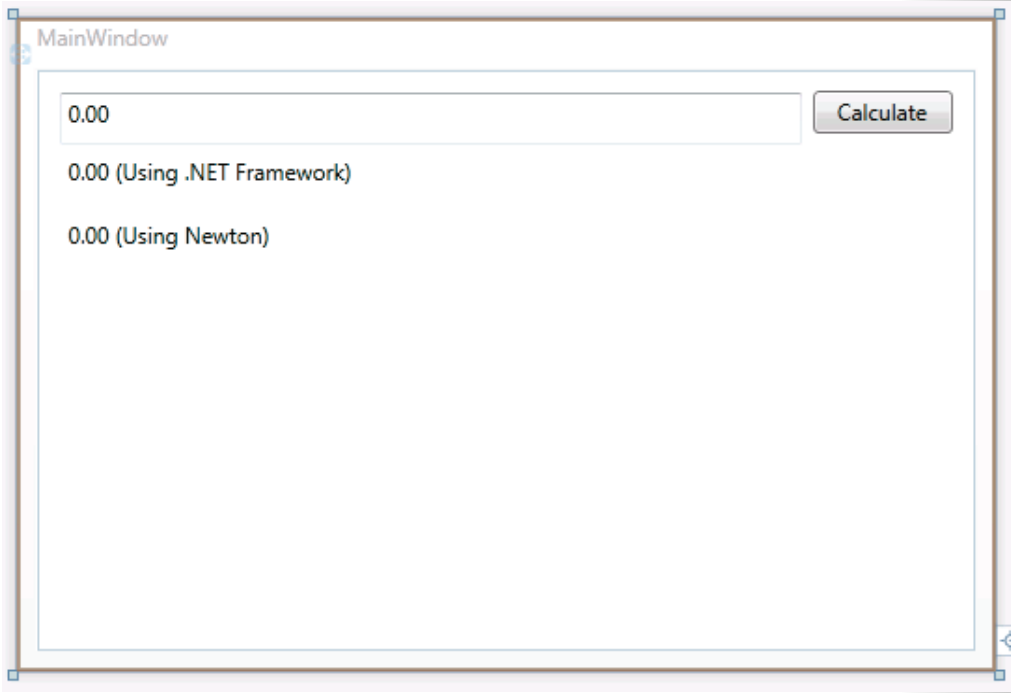
1. Add **TextBox**, **Button**, and two **Label** controls to the MainWindow window. Place them anywhere in the window.
2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

Control	Property	Value
TextBox	**Name**	**inputTextBox**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,12,0,0**
	Text	**0.00**
	VerticalAlignment	**Top**
	Width	**398**
Button	**Name**	**calculateButton**
	Content	**Calculate**
	Height	**23**
	HorizontalAlignment	**Right**
	Margin	**0,11,12,0**

Control	Property	Value
	VerticalAlignment	**Top**
	Width	**75**
Label	**Name**	**frameworkLabel**
	Content	**0.00 (Using .NET Framework)**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,41,0,0**
	VerticalAlignment	**Top**
	Width	**479**
Label	**Name**	**newtonLabel**
	Content	**0.00 (Using Newton)**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,75,0,0**
	VerticalAlignment	**Top**
	Width	**479**

MCT USE ONLY. STUDENT USE PROHIBITED

The MainWindow window should look like the following screen shot.



▶ Task 3: Calculate square roots by using the Math.Sqrt method of the .NET Framework

1. Create an event handler for the **Click** event of the button.
2. In the **calculateButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into a **double** value. Store the **double** value in a variable called numberDouble. Use the **TryParse** method of the **double** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "Please enter a double," and then execute a **return** statement to quit the method.

Note: You can display a message in a message box by using the **MessageBox.Show** method.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Check that the value that the user enters is a positive number. If it is not, display a message box with the text "Please enter a positive number," and then return from the method.
4. Calculate the square root of the value in the numberDouble variable by using the **Math.Sqrt** method. Store the result in a double variable called squareRoot.
5. Format the value in the squareRoot variable by using the layout shown in the following code example, and then display it in the **frameWorkLabel Label** control.

```
99.999 (Using the .NET Framework)
```

   Use the **string.Format** method to format the result. Set the **Content** property of a **Label** control to display the formatted result.

6. Build and run the application to test your code. Use the test values that are shown in the following table, and then verify that the correct square roots are calculated and displayed (ignore the "Using Newton" label for the purposes of this test).

Test value	Expected result
25	5
625	25
0.00000001	0.0001
–10	Message box appears with the message "Please enter a positive number"
Fred	Message box appears with the message "Please enter a double"
10	3.16227766016838
8.8	2.96647939483827
2.0	1.4142135623731
2	1.4142135623731

7. Close the application and return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Calculate square roots by using Newton's method

1. In the **calculateButton_Click** method, after the code that you added in the previous task, create a decimal variable called numberDecimal. Initialize this variable with the data that the user enters in the **inputTextBox TextBox** control, but convert it into a **decimal** this time (previously, you read it as a **double**). If the text that the user enters is not valid, display a message box with the text "Please enter a decimal," and then execute a **return** statement to quit the method.



Note: This step is necessary because the **decimal** and **double** types have different ranges. A number that the user enters that is a valid **double** might be out of range for the **decimal** type.

2. Declare a decimal variable called delta, and initialize it to the value of the expression **Math.Pow(10, −28)**. This is the smallest value that the **decimal** type supports, and you will use this value to determine when the answer that is generated by using Newton's method is sufficiently accurate. When the difference between two successive estimates is less than this value, you will stop.



Note: The **Math.Pow** method returns a double. You will need to use the **Convert.ToDecimal** method to convert this value to a decimal before you assign it to the delta variable.

3. Declare another decimal variable called guess, and initialize it with the initial guess at the square root. This initial guess should be the result of dividing the value in **numberDecimal** by 2.
4. Declare another decimal variable called result. You will use this variable to generate values for each iteration of the algorithm, based on the value from the previous iteration. Initialize the result variable to the value for the first iteration by using the expression **((numberDecimal / guess) + guess) / 2**.
5. Add a **while** loop to generate further refined guesses. The body of the **while** loop should assign result to guess, and generate a new value for result by using the expression **((numberDecimal / guess) + guess) / 2**. The **while** loop should terminate when the difference between result and guess is less than or equal to delta.



Note: Use the **Math.Abs** method to calculate the absolute value of the difference between result and guess. Using Newton's algorithm, it is possible for the difference between the two variables to alternate between positive and negative values as it diminishes. Consequently, if you do not use the **Math.Abs** method, the algorithm might terminate early with an inaccurate result.

6. When the **while** loop has terminated, format and display the value in the result variable in the **newtonLabel Label** control. Format the data in a similar manner to the previous task.

▶ Task 5: Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the correct square roots are calculated and displayed. Compare the value in the two labels, and then verify that the square roots that are calculated by using Newton's method are more accurate than those calculated by using the **Math.Sqrt** method.

Test value	.NET Framework	Newton's algorithm
25	5	5.000000000000000000000000000
625	25	25.000000000000000000000000000
0.00000001	0.0001	0.000100000000000000000000000000
10	3.16227766016838	3.162277660168379331998893544
8.8	2.96647939483827	2.966479394838265179484558976
2.0	1.4142135623731	1.414213562373095048801688724
2	1.4142135623731	1.414213562373095048801688724

2. As a final test, try the value 0.0000000000000000000000000001 (27 zeroes after the decimal point). Can you explain the result?
3. Close the application and return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Converting Integer Numeric Data to Binary

In this exercise, you will create another application that enables the user to enter an integer value, generate a string that holds the binary representation of this value, and then display the result.

Scenario

Another device has the requirement to display decimal numeric data in a binary format. You have been asked to develop some code that can convert a non-negative decimal integer value into a string that contains the binary representation of this value.

The process for converting the decimal value 6 into its binary representation is as follows:

1. Divide the integer by 2, save the integer result, and use the remainder as the first binary digit.

   In this example, 6 / 2 is 3 remainder 0. Save the character "0" as the first character of the binary representation.

2. Divide the result of the previous division by 2, save the result, and use the remainder as the next binary digit.

   In this example, 3 / 2 is 1 remainder 1. Save the character "1" as the next character of the binary representation.

3. Repeat the process until the result of the division is zero.

   In this example, 1 / 2 is zero remainder 1. Save the character "1" as the final character of the binary representation.

4. Display the characters saved in reverse order.

   In this example, the characters were generated in the sequence "0", "1", 1", so display them in the order "1", "1", "0". The value 110 is the binary representation of the decimal value 6.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Add code to generate the binary representation of an integer value.
4. Test the application.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Create a new WPF Application project

- Create a new project called IntegerToBinary by using the WPF Application template in the E:\Labfiles\Lab 2\Ex2\Starter folder.

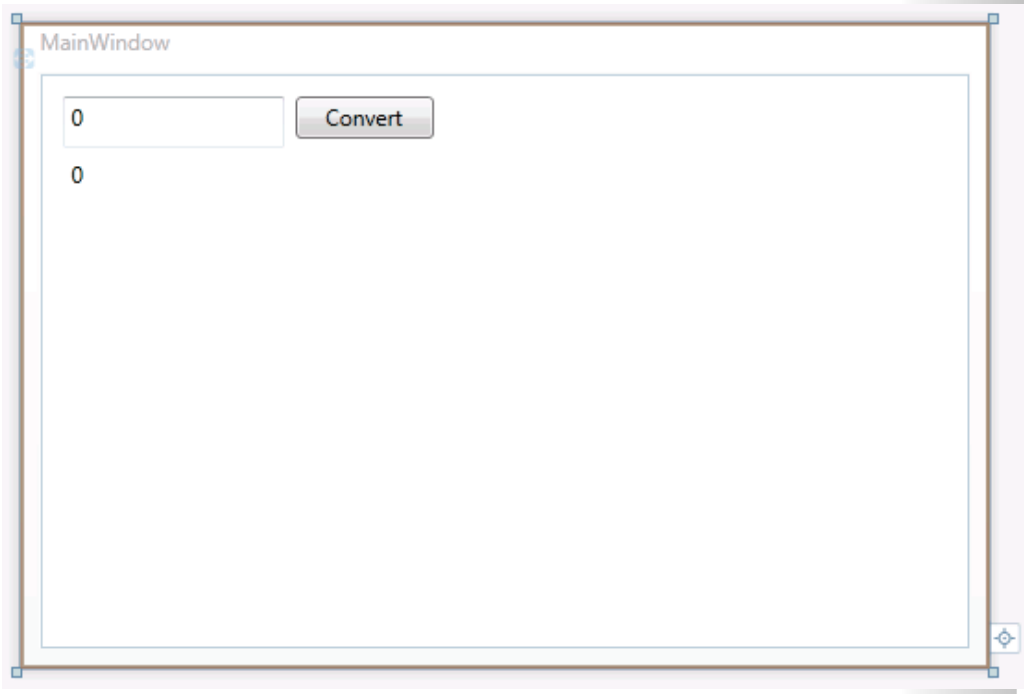
▶ Task 2: Create the user interface

1. Add a **TextBox**, **Button**, and **Label** control to the MainWindow window. Place them anywhere in the window.
2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

Control	Property	Value
TextBox	**Name**	**inputTextBox**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,12,0,0**
	Text	**0**
	VerticalAlignment	**Top**
	Width	**120**
Button	**Name**	**convertButton**
	Content	**Convert**
	Height	**23**
	HorizontalAlignment	**Left**
	Margin	**138,12,0,0**
	VerticalAlignment	**Top**
	Width	**75**

Control	Property	Value
Label	**Name**	**binaryLabel**
	Content	**0**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,41,0,0**
	VerticalAlignment	**Top**
	Width	**120**

The MainWindow window should look like the following screen shot.



MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Add code to generate the binary representation of an integer value

1. Create an event handler for the **Click** event of the button.
2. In the **convertButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into an **int** type. Store the integer value in a variable called i. Use the **TryParse** method of the **int** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "TextBox does not contain an integer," and then execute a **return** statement to quit the method.
3. Check that the value that the user enters is not a negative number (the integer-to-binary conversion algorithm does not work for negative numbers). If it is negative, display a message box with the text "Please enter a positive number or zero," and then return from the method.
4. Declare an integer variable called remainder and initialize it to zero. You will use this variable to hold the remainder after dividing i by 2 during each iteration of the algorithm.
5. Declare a StringBuilder variable called binary and instantiate it. You will use this variable to construct the string of bits that represent i as a binary value.
6. Add a **do** loop that performs the following tasks:
   a. Calculate the remainder after dividing i by 2, and then store this value in the remainder variable.
   b. Divide i by 2.
   c. Prefix the value of remainder to the start of the string being constructed by the binary variable.

   Terminate the **do** loop when i is less than or equal to zero.



Note: To prefix data into a **StringBuilder** object, use the **Insert** method of the **StringBuilder** class, and then insert the value of the data at position 0.

7. Display the value in the binary variable in the **binaryLabel Label** control.



Note: Use the **ToString** method to retrieve the string that a **StringBuilder** object constructs. Set the **Content** property of the **Label** control to display this string.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the binary representations are generated and displayed.

Test value	Expected result
0	0
1	1
–1	Message box appears with the message "Please enter a positive number or zero"
10.5	Message box appears with the message "TextBox does not contain an integer"
Fred	Message box appears with the message "TextBox does not contain an integer"
4	100
999	1111100111
65535	1111111111111111
65536	10000000000000000

2. Close the application and return to Visual Studio.

Exercise 3: Multiplying Matrices

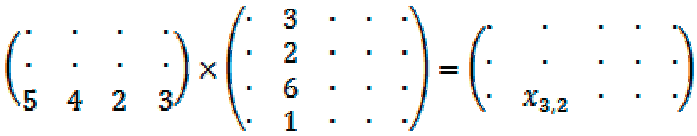
In this exercise, you will create another WPF application. This WPF application will provide a user interface that enables the user to provide the data for two matrices and store this data in rectangular arrays. The application will calculate the product of these two arrays and display them.

Scenario

Some of the devices that Fabrikam, Inc. has developed perform calculations that involve sets of data that are held as matrices. You have been asked to implement code that performs matrix multiplication. You decide to test your code by building

a WPF application that enables a user to specify the data for two matrices, calculate the product of these matrices, and then view the result.

Multiplying matrices is an iterative process that involves calculating the sum of the products of the values in each row in one matrix with the values in each column in the other, as the following screen shot shows.



This screen shot shows a 3×4 matrix multiplying a 4×5 matrix. This will result in a 3×5 matrix.

Note: The number of columns in the first matrix must match the number of rows in the second matrix. The starter code that is provided for you in this lab ensures that this is always the case.

To calculate each element xa,b in the result matrix, you must calculate the sum of the products of every value in row a in the first matrix with every value in column b in the second matrix. For example, to calculate the value placed at x3,2 in the result matrix, you calculate the sum of the products of every value in row 3 in the first matrix with every value in column 2 in the second matrix:

(5×3)+(4×2)+(2×6)+(3×1) = 38

You perform this calculation for every element in the result matrix.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication project and examine the starter code.
2. Define the matrix arrays and populate them with the data in the **Grid** controls.
3. Multiply the two input matrices and calculate the result.
4. Display the results and test the application.

▶ Task 1: Open the MatrixMultiplication project and examine the starter code

1. Open the MatrixMultiplication project located in the E:\Labfiles\Lab 2\Ex3\Starter folder.

MCT USE ONLY. STUDENT USE PROHIBITED

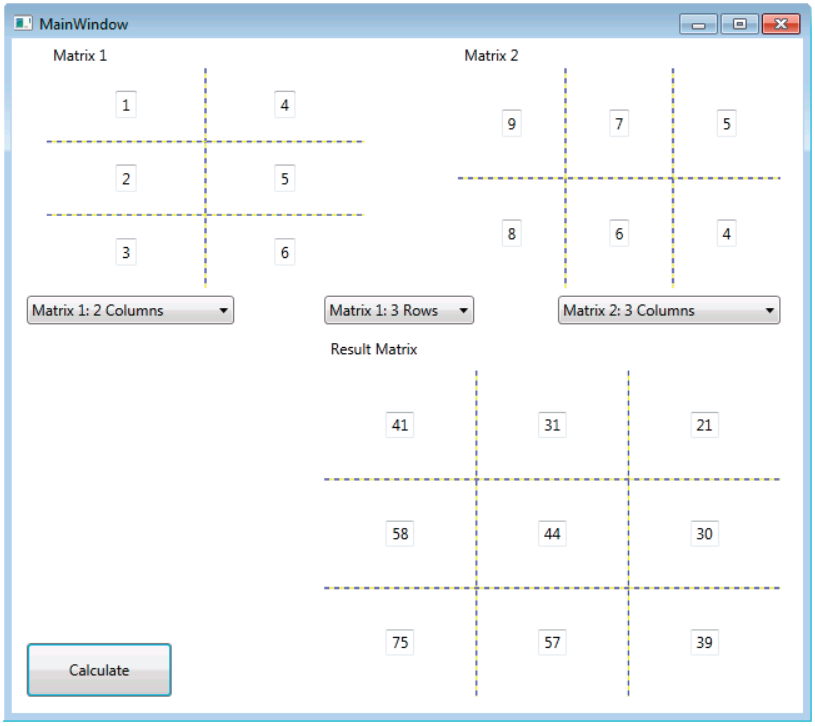
2. Examine the user interface that the MainWindow window defines.

   The user interface contains three **Grid** controls, three **ComboBox** controls, and a **Button** control.

   When the application runs, the first **Grid** control, labeled **Matrix 1**, represents the first matrix, and the second **Grid** control, labeled **Matrix 2**, represents the second matrix. The user can specify the dimensions of the matrices by using the **ComboBox** controls, and then enter data into each cell in them. There are several rules that govern the compatibility of matrices to be multiplied together, and Matrix 2 is automatically configured to have an appropriate number of rows based on the number of columns in Matrix 1.

   When the user clicks the **Calculate** button, Matrix 1 and Matrix 2 are multiplied together, and the result is displayed in the **Grid** control labeled **Result Matrix**. The dimensions of the result are determined by the shapes of Matrix 1 and Matrix 2.

   The following screen shot shows the completed application running. The user has multiplied a 2×3 matrix with a 3×2 matrix, and the result is a 3×3 matrix.



MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Define the matrix arrays and populate them with the data in the Grid controls

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file.
3. At the top of the **MainWindow** class, remove the comment **TODO Task 2 declare variables**, and then add statements that declare three two-dimensional arrays called **matrix1**, **matrix2**, and **result**. The type of the elements in these arrays should be **double**, but the size of each dimension should be omitted because the arrays will be dynamically sized based on the input that the user provides. The first dimension will be set to the number of columns, and the second dimension will be set to the number of rows.
4. In the task list, double-click the task **TODO Task 2 Copy data from input Grids**. This task is located in the **buttonCalculate_Click** method.
5. In the **buttonCalculate_Click** method, remove the comment **TODO Task 2 Copy data from input Grids**. Add two statements that call the **getValuesFromGrid** method. This method (provided in the starter code) expects the name of a **Grid** control and the name of an array to populate with data from that **Grid** control. In the first statement, specify that the method should use the data in **grid1** to populate **matrix1**. In the second statement, specify that the method should use the data from **grid2** to populate **matrix2**.
6. Remove the comment **TODO Task 2 Get the matrix dimensions**. Declare three integer variables called m1columns_m2rows, m1rows, and m2columns. Initialize m1columns_m2rows with the number of columns in the **matrix1** array (this is also the same as the number of rows in the **matrix2** array) by using the **GetLength** method of the first dimension of the array. Initialize m1rows with the number of rows in the **matrix1** array by using the **GetLength** method of the second dimension of the array. Initialize m2columns with the number of columns in the **matrix2** array.

▶ Task 3: Multiply the two input matrices and calculate the result

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 3 Calculate the result**. Define a **for** loop that iterates through all of the rows in the **matrix1** array. The dimensions of an array are integers, so use an integer variable called row as the control variable in this **for** loop. Leave the body of the **for** loop blank; you will add code to this loop in the next step.

2. In the body of the **for** loop, add a nested **for** loop that iterates through all of the columns in the **matrix2** array. Use an integer variable called column as the control variable in this **for** loop. Leave the body of this **for** loop blank.
3. The contents of each cell in the **result** array are calculated by adding the product of each item in the row identified by the row variable in **matrix1** with each item in the column identified by the column variable in **matrix2**. You will require another loop to perform this calculation, and a variable to store the result as this loop calculates it.

   In the inner **for** loop, declare a double variable called accumulator, and then initialize it to zero.
4. Add another nested **for** loop after the declaration of the accumulator variable. This loop should iterate through all of the columns in the current row in the **matrix1** array. Use an integer variable called cell as the control variable in this **for** loop. Leave the body of this **for** loop blank.
5. In the body of this **for** loop, multiply the value in **matrix1[cell, row]** with the value in **matrix2[column, cell]**, and then add the result to accumulator.
6. After the closing brace of the innermost **for** loop, store the value in accumulator in the **result** array. The value should be stored in the cell that the column and row variables have identified.

▶ Task 4: Display the results and test the application

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 4 Display the result**. The starter code contains a method called **initializeGrid** that displays the contents of an array in a **Grid** control in the WPF window. Add a statement that calls this method. Specify that the method should use the **grid3 Grid** control to display the contents of the **result** array.
2. Build the solution and correct any errors.
3. Run the application in Debug mode.
4. In the MainWindow window, define **Matrix 1** as a 3×2 matrix and define **Matrix 2** as a 3×3 matrix.



Note: The number of rows in the **Matrix 2** matrix is determined by the number of columns in the **Matrix 1** matrix.

5. Specify the values for the cells in the matrices as shown in the following tables.

MCT USE ONLY. STUDENT USE PROHIBITED

Matrix 1		
1	5	–9
3	–7	11

Matrix 2		
2	–8	14
4	–10	16
6	–12	18

6. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

Result		
–32	50	–68
44	–86	128

7. Change the data in **Matrix 2** as shown in the following table.

Matrix 2		
1	0	0
0	1	0
0	0	1

8. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

Result		
1	5	–9
3	–7	11

MCT USE ONLY. STUDENT USE PROHIBITED

Matrix 2 is an example of an identity matrix. When you multiply a matrix by an identity matrix, the result is the same data as defined by the original matrix (it is the matrix equivalent of multiplying a value by 1 in regular arithmetic). In this case, the values in the **Result** matrix are the same as those in **Matrix 1**.

9. Change the data in **Matrix 2** again, as shown in the following table.

Matrix 2		
–1	0	0
0	–1	0
0	0	–1

10. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

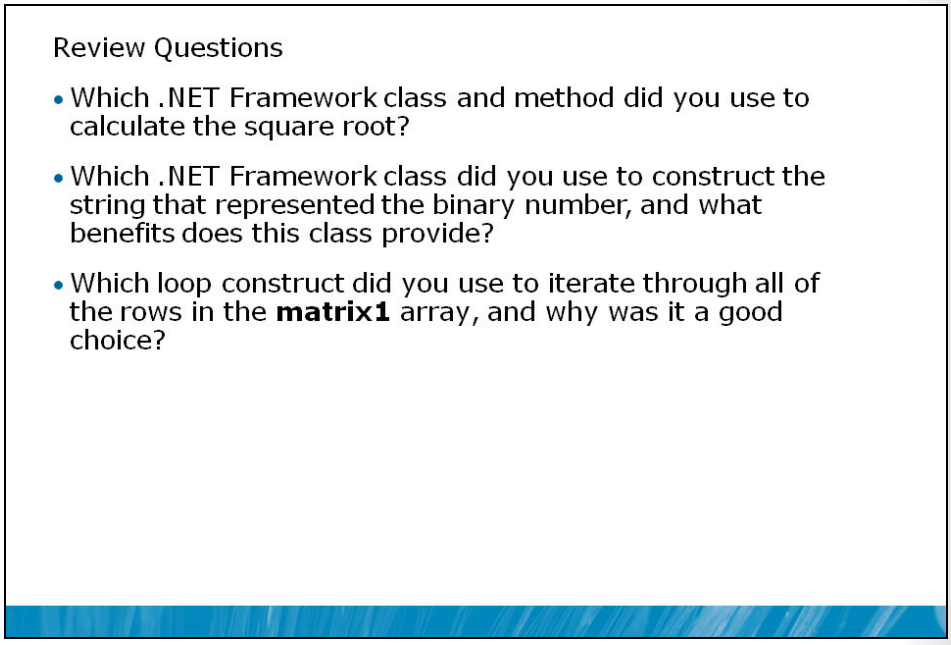
Result		
–1	–5	9
–3	7	–11

This time, the values in **Result** are the same as those in **Matrix 1** except that the sign of each element is inverted (**Matrix 2** is the matrix equivalent of −1 in regular arithmetic).

11. Close the MainWindow window.
12. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

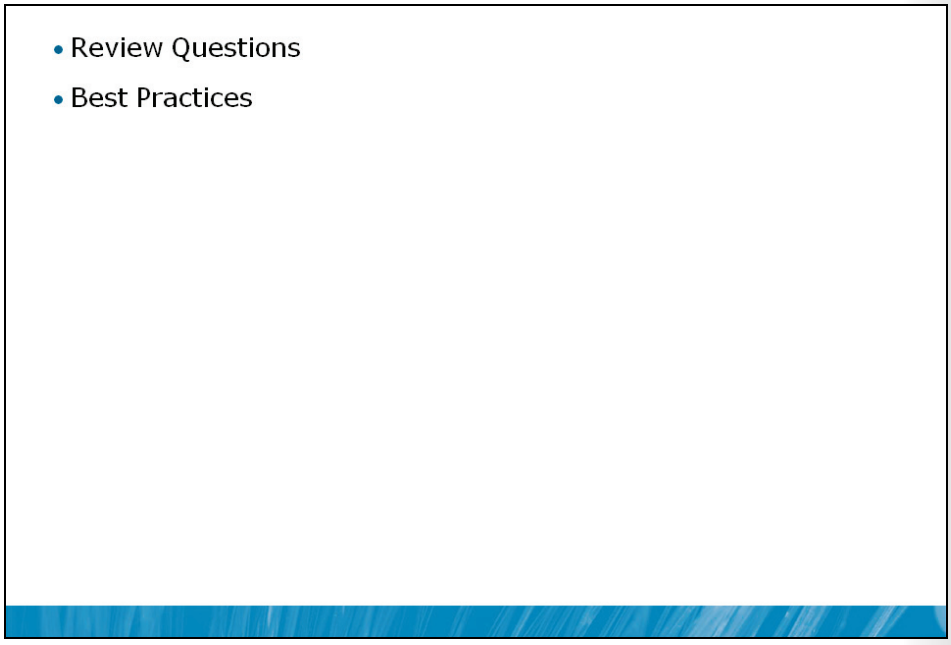


Review Questions

1. Which .NET Framework class and method did you use to calculate the square root?
2. Which .NET Framework class did you use to construct the string that represented the binary number, and what benefits does this class provide?
3. Which loop construct did you use to iterate through all of the rows in the **matrix1** array, and why was it a good choice?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. If you declare a variable with the type **var**, what does it mean?
2. How can you control the order of processing in an expression?
3. What is the purpose of arrays?
4. Name an alternative approach to using the **if else** statements.
5. Which loop construct should you use to execute a block of code one or more times?

Best Practices Related to Using C# Constructs

Supplement or modify the following best practices for your own work situations:

- When you choose a data type, ensure that you select one that is appropriate to the type of data that you are processing. For example, do not create a double variable for processing integer data because this requires that the compiler generates additional code to convert your integer data into **double** values.

MCT USE ONLY. STUDENT USE PROHIBITED

- Instead of concatenating strings by using the **+** operator, use the **StringBuilder** class or use the static **Format** method of the **String** class.
- When you access elements in an array by using the index of an element, make sure that you test to see whether the index exists. If the index doesn't exist, you will get an **IndexOutOfRange** exception.
- Avoid too many nested **if else** and loop statements because they can make debugging your applications complicated.
- Avoid using **break** and **continue** statements in loops unless you really need them.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 3

Declaring and Calling Methods

Contents:

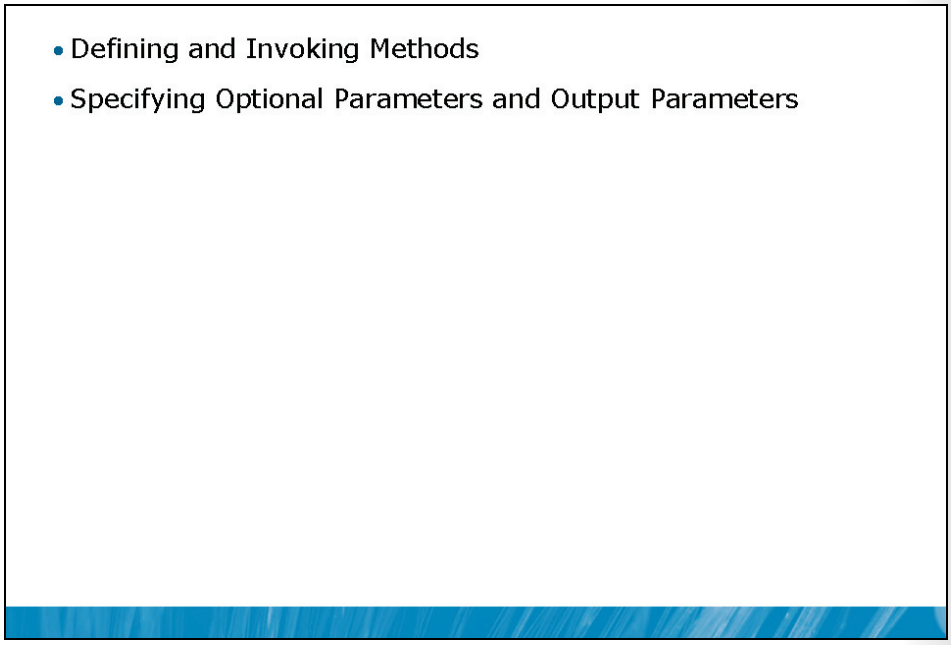
Lesson 1: Defining and Invoking Methods 3-3

Lesson 2: Specifying Optional Parameters and Output Parameters 3-29

Lab: Declaring and Calling Methods 3-39

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



A key part of developing any application is dividing the solution into logical components. In object-oriented languages such as C#, a method is a unit of code that is designed to perform a discrete piece of work. This module introduces methods and describes how to define and use them.

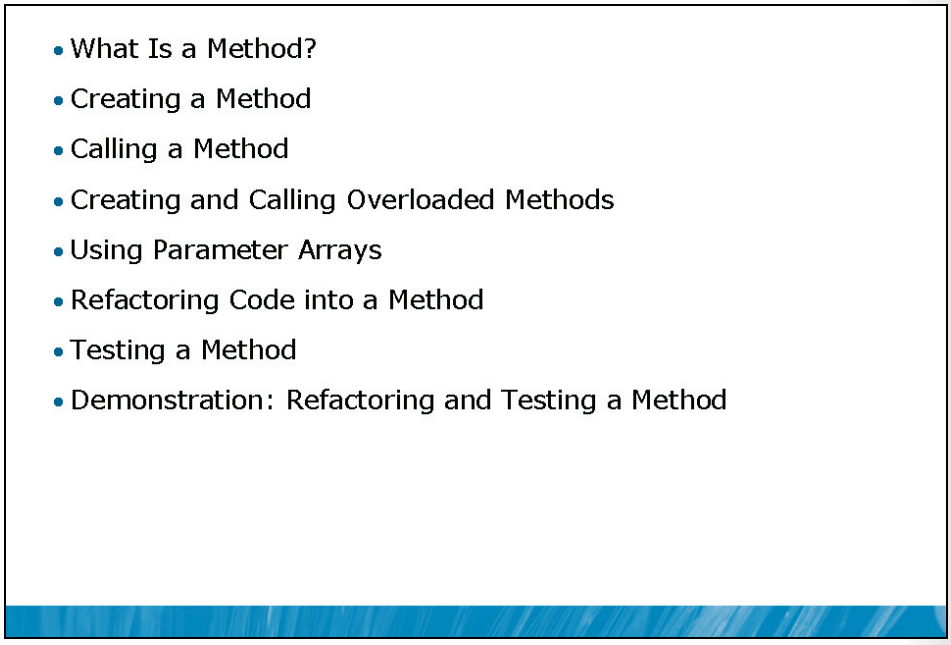
Objectives

After completing this module, you will be able to:

- Describe how to create and invoke methods.
- Define and call methods that can take optional parameters and output parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Defining and Invoking Methods



This lesson introduces methods and explains how to create and call them. This lesson also explains how to create overloaded methods, and methods that can take a variable number of parameters. Finally, this lesson explains how to use the refactoring tools that Microsoft® Visual Studio® provides to create a method from an existing code block, and how to create unit tests to test the functionality of a method.

Objectives

After completing this lesson, you will be able to:

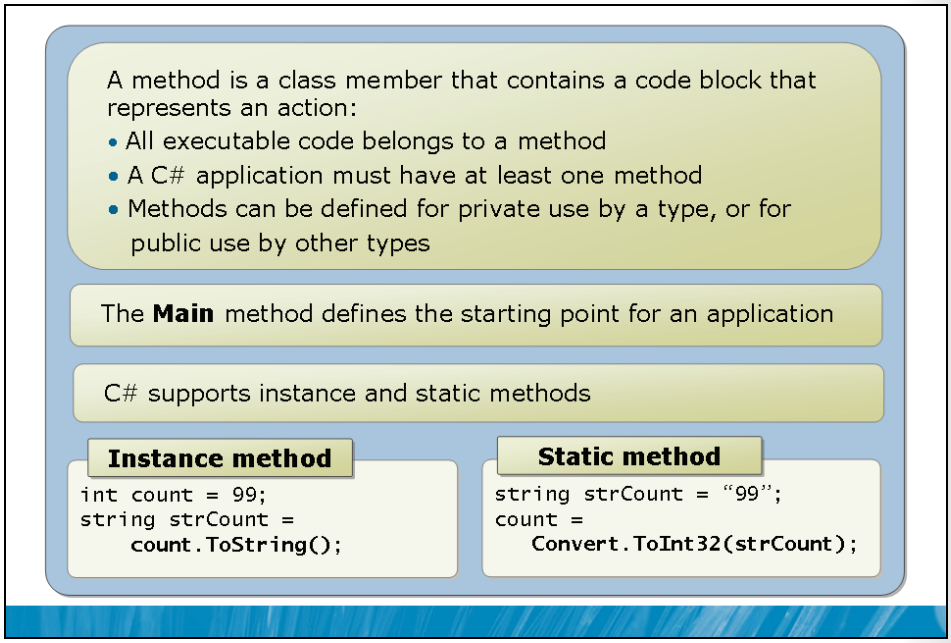
- Describe how to create a method that takes parameters, and returns a value.
- Describe how to call a method and handle a return value.
- Describe how to create and call overloaded methods.
- Describe how to use parameter arrays to pass variable numbers of arguments to methods.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe how to refactor code into a method.
- Describe how to create a unit test for a method.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Method?



Key Points

Methods implement the behavior of a type. A method contains a block of code that defines an action that a type can perform. All code belongs to a method; you cannot write a C# program that does not contain at least one 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 Microsoft Visual C#® will have many methods, each with a specific purpose. Some methods are fundamental to the operation of an application. For example, all C# applications must have a method called **Main** that defines the entry point for the application; when the user runs a C# application, the common language runtime (CLR) executes the **Main** method for that application.

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

MCT USE ONLY. STUDENT USE PROHIBITED

C# supports two classes of methods:

- *Instance methods*. These methods execute in the context of a specific object, and can directly access data that belongs to the object. For example, the **ToString** method that was described in Module 2 is an instance method. You invoke instance methods by specifying the object that they belong to.
- *Static methods*. These methods are associated with a type rather than a specific object. Examples of static methods include those that belong to the **Convert** class that was described in Module 2, such as **Convert.ToInt32**. You invoke these methods by specifying a type rather than an object.

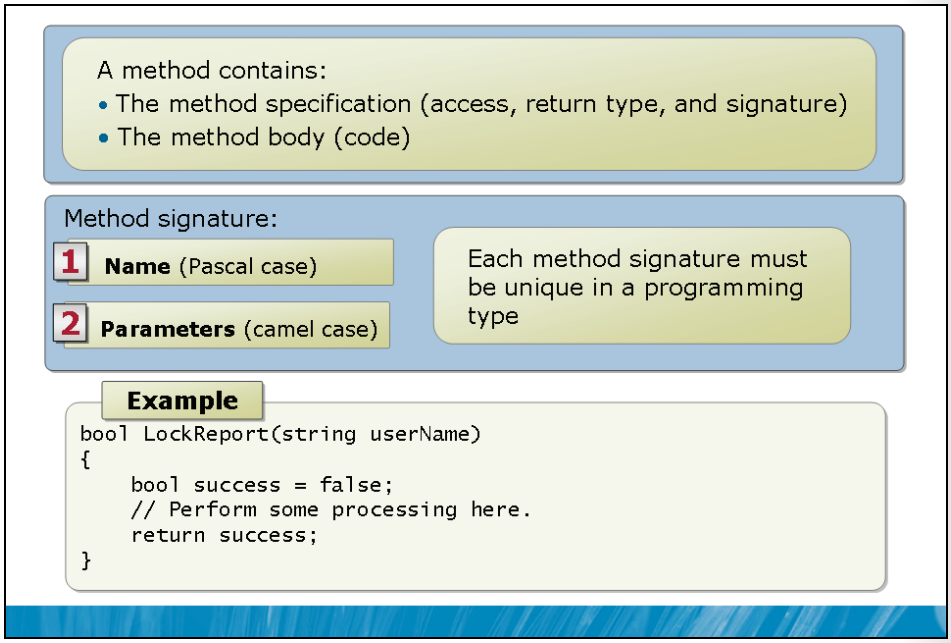


Note: Module 7 describes the differences between instance and static methods in detail.

Question: Why do you need to use methods when developing a .NET Framework application with C#?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating a Method



Key Points

A method contains two elements:

1. The method specification
2. The method 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. Each method in a class must have a unique signature.

Note: Method accessibility will be described in more detail in Module 7.

Naming Methods

A method name has the same syntactic restrictions as a variable name; it must start with a letter or an underscore, and can only contain letters, underscores, and

MCT USE ONLY. STUDENT USE PROHIBITED

numeric characters. Remember that C# is case-sensitive, so a class can contain two methods that have the same name but that differ only in the case of one or more letters, although this is not considered to be good 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 Pascal case.* Do not start method names with an underscore or a lowercase letter.

Implementing a Method Body

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

Inside a method body, you can define variables. These variables only exist while the method is running. When the method finishes, they disappear.

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 camel case.

Note that the names of parameters can be exposed to applications that use your methods through Microsoft IntelliSense® in Visual Studio, so keep the names of parameters meaningful.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

```
string MyMethod ()
{
    return "Hello";
}
```

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.

Method Examples

The following code example shows a method that accepts no parameters and does not return a value.

```
void ClearReport()
{
    // Perform some processing here.
}
```

The following code example shows a method that accepts two string parameters, but does not return a value.

```
void CreateReport(string reportName, string reportDescription)
{
    // Perform some processing here.
}
```

The following code example shows a method that accepts two string parameters and returns a Boolean result by using the **return** statement.

```
bool LockReport(string reportName, string userName)
{
    bool success = false;

    // Perform some processing here.

    return success;
}
```



Note: Any variables that you declare within a method block are only accessible to other statements in that method block.

MCT USE ONLY. STUDENT USE PROHIBITED

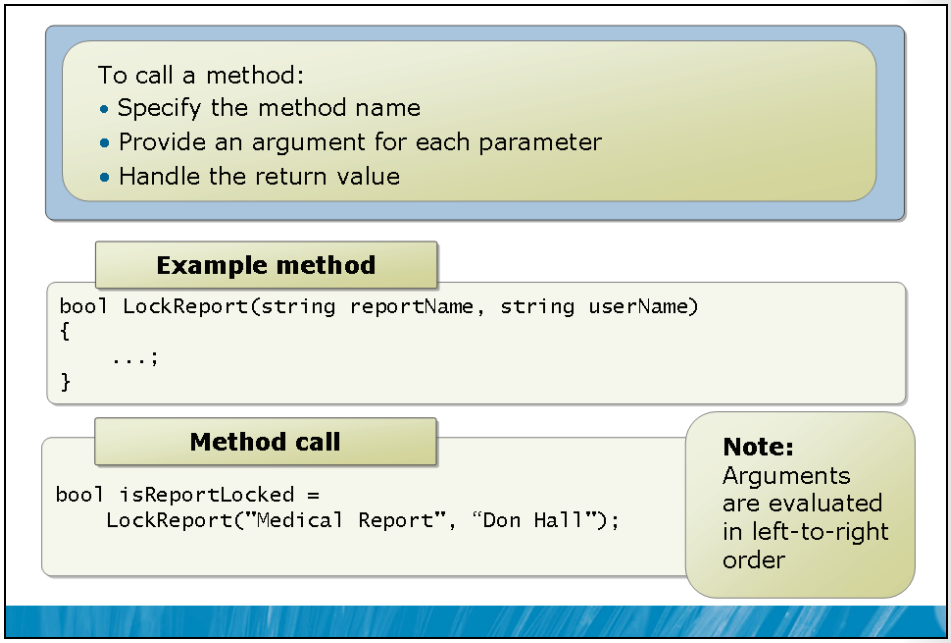
Question: What are the four elements in the method specification?

Additional Reading

For more information about methods, see the Methods (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192905.

MCT USE ONLY. STUDENT USE PROHIBITED

Calling a Method



Key Points

You call a method to run the code in that method. You do not need to understand how the code in a method works; you may not even have access to this 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. If the method returns a value, you specify how to handle this value, typically by assigning it to a variable of the same type.

Example

The method called **LockReport** in the following code example locks a report for a particular user. The method returns a Boolean result to indicate the success of the operation.

```
public bool LockReport(string reportName, string userName)
{
    bool success = false;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Perform some processing here.
    return success;
}
```

The **LockReport** method expects two string parameters. The first parameter represents the name of the report that you want to lock, and the second parameter represents the user who locked the report. The following code example shows how you can call this method. The return value is assigned to a Boolean variable called isReportLocked.

```
bool isReportLocked = LockReport("Medical Report", "Don Hall");
```

The arguments that are passed to a method can be any expression that evaluates to the type that is expected by the corresponding parameters; the parameters are initialized with the values of each of these expressions.

Order of Evaluation

The arguments to a method are evaluated in strict left-to-right order. This is important if evaluating an argument modifies the value of another argument. For example, the method that is defined by the following code example takes two integer parameters and adds them together to return their sum.

```
int Sum(int first, int second)
{
    return first + second;
}
```

If an application invokes this method as shown in the following code example, the value of **i** (1) will be used as the first argument, **i** will then be incremented to 2, and the value 2 + 2 will be used as the second argument. The value that is returned to **result** will therefore be 5.

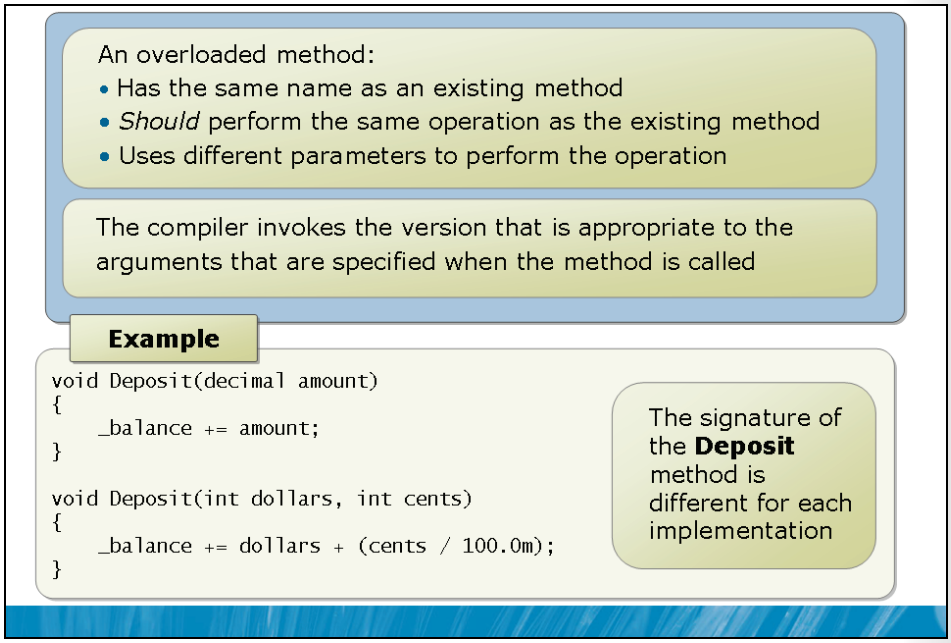
```
int i = 1;
int j = 2;
int result = Sum(i++; i+j);
```

Question: How can you call the method in the following code example?

```
void DeleteReport(string reportName)
```

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Calling Overloaded Methods



Key Points

Sometimes it is useful to define several implementations of a method that takes a different set of parameters. Each version of the method performs the same operation, it just happens to use different data. An example of this in the .NET Framework is the **WriteLine** method of the **Console** class. This method has 19 different versions that enable you to display data specified as a range of types. For example, the following code example displays an integer value and a Boolean value by using two **Console.WriteLine** statements. Notice that the type of the parameter that is specified in each case is different.

```
int intData = 99;
bool booleanData = true;
...
Console.WriteLine(intData);
Console.WriteLine(booleanData);
```

This technique is known as overloading. You can create as many overloaded versions of a method as you need as long as the type and number of parameters is different for each version (each method signature must be unique).

MCT USE ONLY. STUDENT USE PROHIBITED



Note: Only use method overloading to provide different methods that do semantically the same thing.

Defining Overloaded Methods

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 contains 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 how to define three overloaded **Deposit** methods in a class called **BankAccount**:

- The first **Deposit** method takes a parameter that represents the amount to deposit as a fractional number.
- The second **Deposit** method takes a parameter that represents the amount to deposit as a string.
- The third **Deposit** method takes two parameters that represent the amount to deposit as dollars and cents.

```
public class BankAccount
{
    private decimal _balance;

    public void Deposit(decimal amount)
    {
        _balance += amount;
    }

    public void Deposit(string amount)
    {
        _balance += decimal.Parse(amount);
    }

    public void Deposit(int dollars, int cents)
    {
        _balance += dollars + (cents / 100.0m);
    }
}
```

When you call the **Deposit** method, the compiler determines which version to invoke by examining the number and types of the arguments that you specify.

MCT USE ONLY. STUDENT USE PROHIBITED

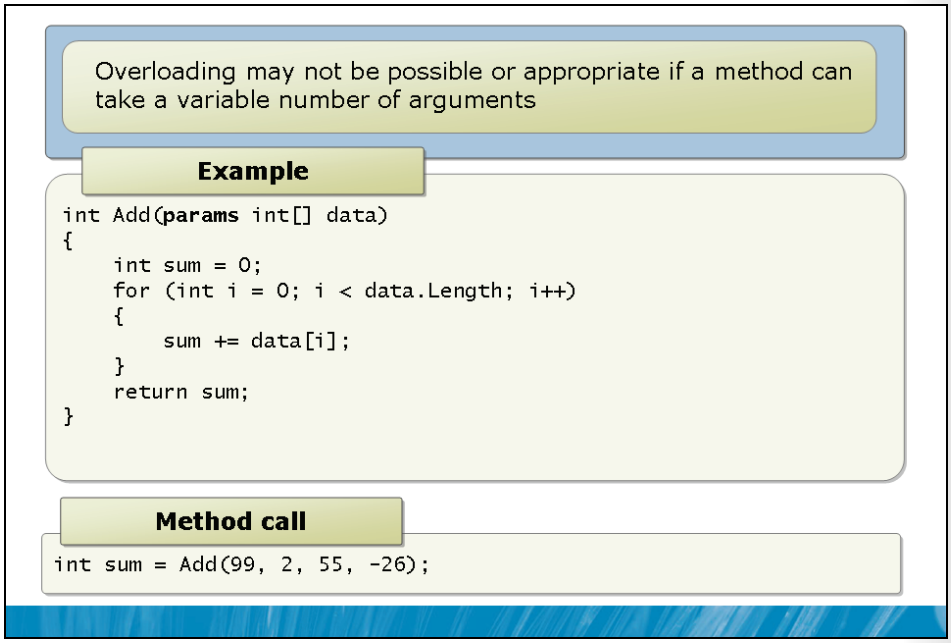
Question: What is meant by overloading a method?

Additional Reading

For more information about method overloading, see the Member Overloading page at http://go.microsoft.com/fwlink/?LinkId=192906.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Parameter Arrays



Key Points

Overloading a method is a useful technique, but it might not always be an appropriate strategy. For example, overloading a method that can take a varying number of parameters may not always be feasible, especially if there is no theoretical limit to the number of parameters. For example, suppose that you wanted to define a method called **Add** that calculated the sum of a set of integer values. You might define overloaded versions of this method as shown in the following code example.

```
int Add(int one, int two)

{
    return one + two;
}

int Add(int one, int two, int three)

{
    return one + two + three;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
int Add(int one, int two, int three, int four)

{
    return one + two + three + four;
}
```

This solution works well if you want to sum two, three, or four integers, but what if you need to sum five, six, seven, or even 100 integers? You could define 99 overloads, but how far should you go?

One way around this is to pass parameters as an array to a method. In theory, there is no limit to the size of an array (in practice, the maximum size of an array is governed by the amount of memory that is available on the computer running your application). Using this approach, you could define a single version of **Add** that looks like the following code example.

```
int Add(int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
```

The downside to this approach is that you would then have to manually declare and populate the array with data, and then pass the array to the method each time you call it, as the following code example shows.

```
int[] myData = new int[...];
myData[0] = 99;
myData[1] = 2;
myData[2] = 55;
myData[3] = -26;
...
int sum = myObject.Add(myData);
```

Using the params Keyword

The **params** keyword provides a useful shorthand approach to implementing this technique. When you define a method with an array parameter prefixed with the **params** keyword, the C# compiler can automatically generate code that creates an array from a set of arguments that is specified when the method is invoked. The following code example shows how to define a method with the **params** keyword, and how you can invoke this method with a variable number of arguments.

MCT USE ONLY. STUDENT USE PROHIBITED

```
int Add(params int[] data)
{
    int sum = 0;
    for (int i = 0; i < data.Length; i++)
    {
        sum += data[i];
    }
    return sum;
}
...
int sum = myObject.Add(99, 2, 55, -26);
```

Notice that the only difference to the **Add** method is the use of the **params** keyword. When the **Add** method is called, the arguments are evaluated. If they have a type that matches the type of the array specified by the **params** keyword, they are collected together into an array and the array is passed as the argument to the **Add** method.



Note: If an overload exists that matches the specified type and number of parameters, it will be called in preference to the version that takes the **params** array.

You can use a **params** array in combination with other parameters, but if you specify a **params** array, it must be the final parameter in the parameter list that the method specified. A method can only take one **params** array as a parameter.

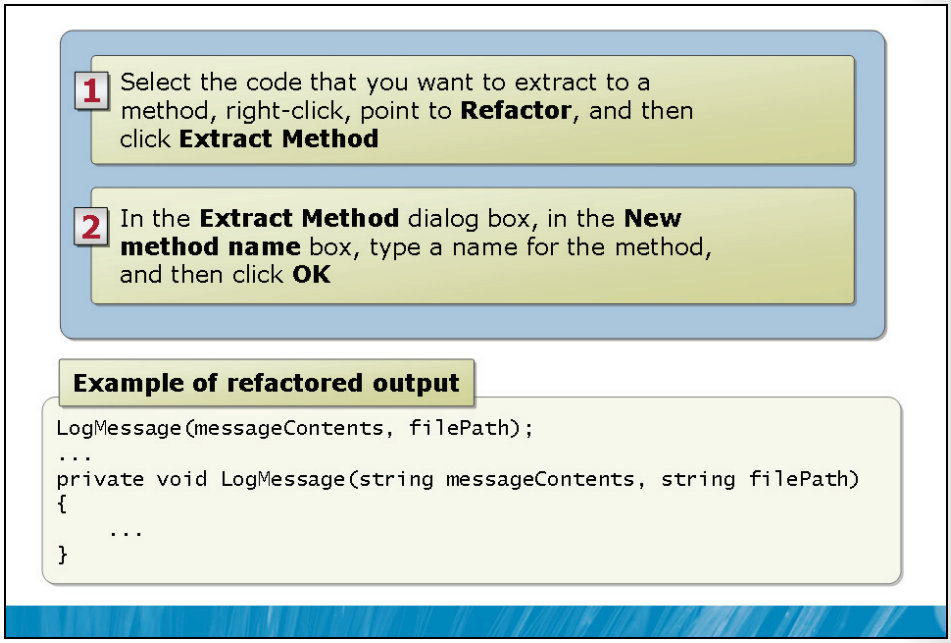
Question: How do you define a method that takes a parameter array?

Additional Reading

For more information about parameter arrays, see the params (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192907.

MCT USE ONLY. STUDENT USE PROHIBITED

Refactoring Code into a Method



Key Points

When writing the code for your applications, you may often find yourself repeatedly writing the same or very similar code. When this happens, you should consider refactoring the code into a method. In this way, if the logic that is implemented by your code changes, you only need to update the code in one place, making your application much easier to maintain.

If you notice code duplication occurring, Visual Studio 2010 provides the Extract Method Wizard, which enables you to create a new method from an existing block of code.

▶ Refactor existing code into a method

1. In Visual Studio 2010, in the Code Editor window, select the code that you want to refactor into a method, right-click, point to **Refactor**, and then click **Extract Method**.
2. In the **Extract Method** dialog box, in the **New method name** box, type a name for the method, and then click **OK**.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code examples show an example of the output from the Visual Studio refactoring process.

Original Code Section

```
string messageContents = "My message text here";
string filePath = @"C:\Users\Student\Desktop";

if (messageContents == null || messageContents == String.Empty)
{
    throw new ArgumentException("Message cannot be empty");
}

if (filePath == null || !System.IO.File.Exists(filePath))
{
    throw new ArgumentException("File path must exist");
}

System.IO.File.AppendAllText(filePath, messageContents);
```

Refactored Code Section

```
string messageContents = "My message text here";
string filePath = @"C:\Users\Student\Desktop";

LogMessage(messageContents, filePath);

...


private void LogMessage(string messageContents, string filePath)
{
    if (messageContents == null || messageContents == String.Empty)
    {
        throw new ArgumentException("Message cannot be empty");
    }

    if (filePath == null || !System.IO.File.Exists(filePath))
    {
        throw new ArgumentException("File path must exist");
    }

    System.IO.File.AppendAllText(filePath, messageContents);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Notice how Visual Studio identified the variables that were used in the code section, and then included them as parameters in the new method signature.



Note: Besides refactoring code into a method, Visual Studio 2010 provides other refactoring operations that can help you to improve the internal structure of your applications. These other operations are described on the Course Companion CD.

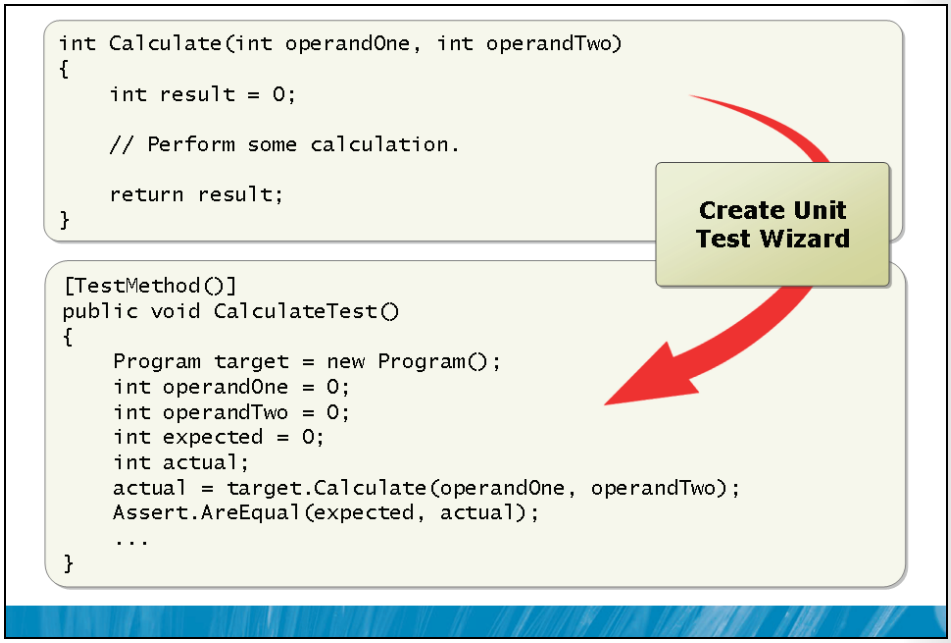
Question: Why would you want to refactor code into a method?

Additional Reading

For more information about refactoring in C#, see the Refactoring (C#) page at http://go.microsoft.com/fwlink/?LinkId=192908.

MCT USE ONLY. STUDENT USE PROHIBITED

Testing a Method



Key Points

When building any application, verifying that the application functions as intended should be part of the development process and not overlooked. In addition, being able to repeat the same test quickly and easily after modifying code is an important software engineering principle.

Unit tests in Visual Studio 2010 can help to simplify the testing process and can help to ensure that your code gets sufficient coverage so that the bug count remains low. Unit tests achieve this by enabling you to create a series of tests that can be run at any time to provide you with feedback that indicates whether your application is still functioning as expected.

Benefits of Unit Tests

Unit tests provide several benefits, which include:

- They provide instant feedback.
- They can help you to document and make it easier for another developer to understand your code.

MCT USE ONLY. STUDENT USE PROHIBITED

- They enable you to constantly run regression test passes on your code, which helps to minimize the introduction of new bugs.
- They can help to reduce the amount of effort required to repeat tests reliably.

▶ Create a unit test

The following steps assume that you have a method that resembles the following code example.

```
public int Calculate(int operandOne, int operandTwo)
{
    int result = 0;

    // Perform some calculation.

    return result;
}
```

1. In Visual Studio 2010, in the Code Editor window that contains your method, right-click, and then click **Create Unit Tests**.
2. In the **Create Unit Tests** dialog box, perform the following, and then click **OK**:
   a. In the **Current selection** list, expand the nodes, and then select the method that you want to create a test for.
   b. In the **Output project** list, ensure that **Create a new Visual C# test project** is selected.



Note: If your solution already contains a Unit Test project, you could select that project in the **Output project** list.

3. In the **New Test Project** dialog box, in the **Enter a name for your new project** box, type a name for the test project, and then click **Create**.

When you click **Create**, Visual Studio 2010 creates a new Unit Test project with the name that you specified, and then adds that project to your solution. The Unit Test project contains a class file that contains several members, the most significant of them being a skeleton test method.

The following code example shows the test method that Visual Studio 2010 created to test the **Calculate** method.

MCT USE ONLY. STUDENT USE PROHIBITED

```
/// <summary>
///A test for Calculate
///</summary>
[TestMethod()]
public void CalculateTest()
{
    Program target = new Program(); // TODO: Initialize to an
    // appropriate value.

    int operandOne = 0; // TODO: Initialize to an appropriate value.
    int operandTwo = 0; // TODO: Initialize to an appropriate value.
    int expected = 0; // TODO: Initialize to an appropriate value.
    int actual;
    actual = target.Calculate(operandOne, operandTwo);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method");
}
```

The **CalculateTest** method contains code to perform the following tasks:

- Initialize the class that contains the **Calculate** method.
- Initialize the two *int* parameters.
- Initialize an *int* parameter for the return value.
- Call the **Calculate** method passing the two *int* parameters.
- Determine whether the result that is returned from the **Calculate** method is as expected by using an **Assert.AreEqual** method call.

The method stub that Visual Studio 2010 generates provides an excellent starting point for you to ensure that methods function as expected.

▶ Run a unit test

After you have created a Unit Test project and defined a test method, you can then run the test in Visual Studio. To do this, perform the following tasks:

1. In Visual Studio 2010, on the **Test** menu, point to **Windows**, and then click **Test View**.

   The Test View window lists all of the test methods in your test project, and provides controls that enable you to run your tests.

2. In the Test View window, select the tests that you want to run, right-click, and then click **Run Selection**.

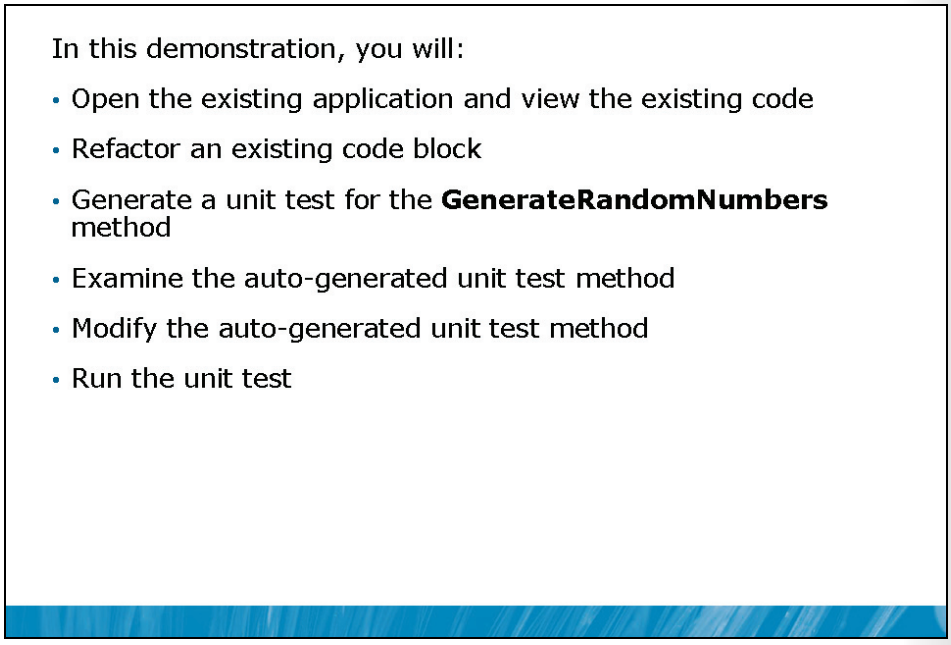
   If you want to debug your code when running the tests, click **Debug Selection**.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: Why would you want to use unit tests when developing your .NET Framework applications?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Refactoring and Testing a Method



Key Points

- Open the existing application and view the existing code.
- Refactor an existing code block.
- Generate a unit test for the **GenerateRandomNumbers** method.
- Examine the auto-generated unit test method.
- Modify the auto-generated unit test method.
- Run the unit test.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open **Microsoft Visual Studio 2010**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

3. In Visual Studio 2010, open the **MethodRefactorAndTestDemo** solution in the **E:\Demofiles\Mod3\Demo1\Starter\MethodRefactorAndTestDemo** folder.
4. Open the **Program.cs** file in the Code Editor window, and examine the code in the **Main** method.
5. Select the contents of the **Main** method, right-click, point to **Refactor**, and then click **Extract Method**.
6. In the **Extract Method** dialog box, in the **New method name** box, type **GenerateRandomNumbers** and then click **OK**.
7. In the Code Editor window, right-click the **GenerateRandomNumbers** method, and then click **Create Unit Tests**.
8. In the **Create Unit Tests** dialog box, click **OK**.
9. In the **New Test Project** dialog box, click **Create**.
10. In the **Add InternalsVisibleTo Attribute** dialog box, click **Yes**.
11. In the Code Editor window, in the **ProgramTests** class, navigate to the **GenerateRandomNumbersTest** method.
12. Make the following changes to the **GenerateRandomNumbersTest** method:
    - Set the max variable to **100**.
    - Set the numberOfRequirednumbers variable to **999**.
    - Remove the **int[] expected = null; // TODO: Initialize to an appropriate value** line.
    - Replace the **Assert.AreEqual(expected, actual);** line with **Assert.AreEqual(1000, actual.Length);**
    - Remove the **Assert.Inconclusive("Verify the correctness of this test method.");** line.

    Your code should resemble the following code example.

```
public void GenerateRandomNumbersTest()
{

    int min = 0; // TODO: Initialize to an appropriate value
    int max = 100; // TODO: Initialize to an appropriate value
    int numberOfRequirednumbers =999;

    // TODO: Initialize to an
    // appropriate value
```

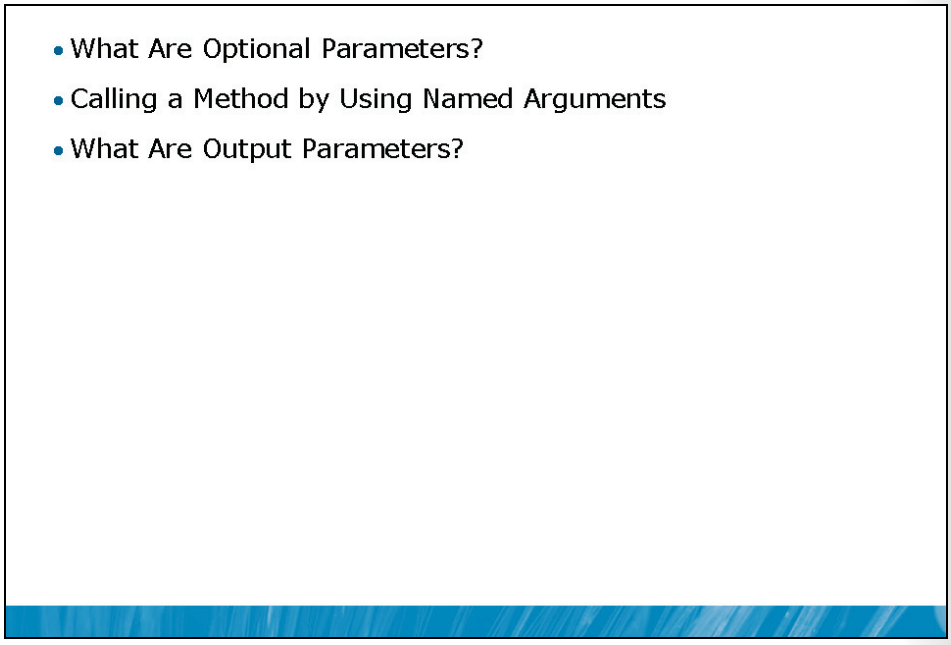
```
    int[] actual;
    actual = Program_Accessor.GenerateRandomNumbers(min, max,
      numberOfRequirednumbers);
    Assert.AreEqual(1000, actual.Length);
}
```

13. Build the solution.
14. Open the Test View window, run the **GenerateRandomNumbersTest** unit test, and examine the results in the Test Results window.
15. In the Code Editor window, navigate to the **GenerateRandomNumbersTest** method, and then set the numberOfRequirednumbers variable to **1000**.
16. Run the **GenerateRandomNumbersTest** unit test, and examine the results in the Test Results window.

Question: Name one way in which you can view and start your unit tests.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Specifying Optional Parameters and Output Parameters



You have seen that you can define a method that takes a variable number of arguments by using a parameter array. However, sometimes 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 achieve this functionality by defining a method that takes optional parameters.

By default, any arguments that you provide when you call a method are passed by value into the parameters that the method specifies. When the method completes, the parameters are destroyed and any changes that you make to the values in these parameters are lost. Output parameters provide a mechanism to enable you to pass data from a method back to the code that calls the method.

This lesson describes how to define and use optional parameters and output parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

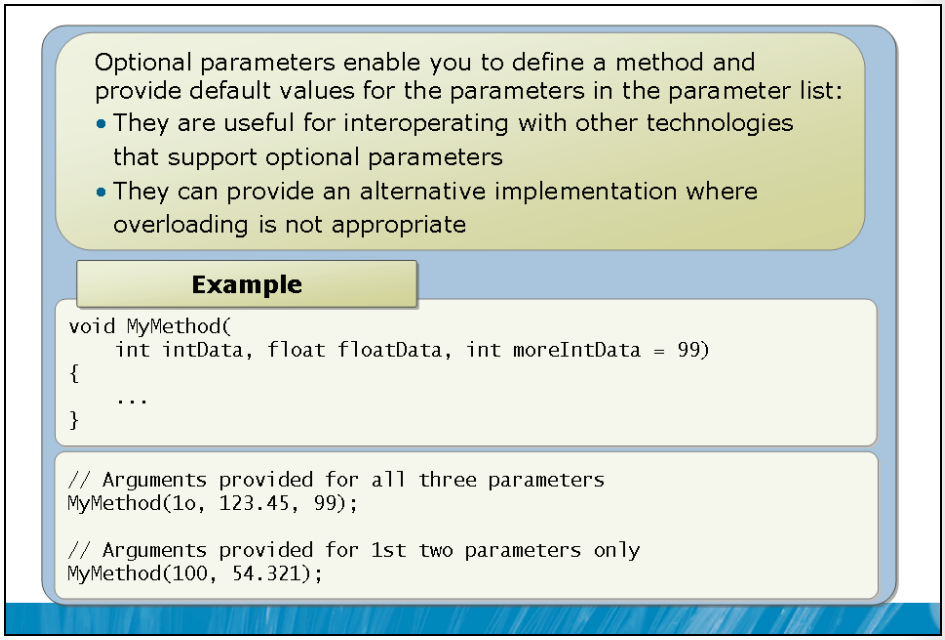
Objectives

After completing this lesson, you will be able to:

- Explain the purpose of optional parameters.
- Describe how to call a method by using named arguments.
- Explain the purpose of output parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Optional Parameters?



Key Points

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 instances of each method it should use to satisfy each method call.

However, there are other languages and technologies that developers can use for building Windows®-based applications and components that do not follow these rules. A key feature of C# and other languages that are designed for the .NET Framework 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 C# solution, 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

MCT USE ONLY. STUDENT USE PROHIBITED

between implementations. For example, consider the method in the following code example.

```
void MyMethod(
    int intData, float floatData, int moreIntData)
{
    ...
}
```

The **MyMethod** method takes three parameters: two *int* parameters and a *float* parameter. If you wanted to provide an implementation of **MyMethod** that took only two parameters, *intData* and *floatData*, you could overload the method, as the following code example shows.

```
void MyMethod (int intData, float floatData)
{
    ...
}
```

If you write a statement that calls the **MyMethod** method, you can provide either two or three parameters of the appropriate types, and the compiler uses the type information to determine which overload to call, as the following code example shows.

```
int arg1 = 99;
float arg2 = 100.0F;
int arg3 = 101;
// Call overload with three parameters
DoWorkWithData(arg1, arg2, arg3);
// Call overload with two parameters
DoWorkWithData(arg1, arg2);
```

However, suppose you want to implement two further versions of **MyMethod** that take only the first parameter and the third parameter. You might try to implement these overloads as shown in the following code example.

```
void MyMethod (int intData)
{
    ...
}
void MyMethod (int moreIntData)
{
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

However, these two overloads have the same signature, so the code will fail to compile and instead generates the error "**Type 'typename' already defines a member called 'MyMethod' with the same parameter types.**".

Using optional parameters can help to solve this problem.

Defining Optional Parameters

Optional parameters enable you to define a method and provide default values for the parameters in the parameter list. You indicate a default value by using the assignment operator.

The following code example shows how to define a method with an optional parameter. All other parameters are mandatory.

```
void MyMethod(int intData, float floatData, int moreIntData = 99)
{
    ...
}
```

When using optional parameters, you must specify all mandatory parameters before any optional parameters. The following code example causes a compiler error.

```
void MyMethod(int intData, float floatData = 101.1F, int moreIntData)
{
    ...
}
```

Calling a Method with Optional Parameters

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. In the following code example, the first call to the **MyMethod** method provides values for all three parameters.

The second call specifies only two arguments, and these values are applied to the first and second parameters. The *moreIntData* parameter receives the default value of 99 when the method runs, as the following code example shows.

```
// Arguments provided for all three parameters
MyMethod(10, 123.45F, 99);
// Arguments provided for 1st two parameters only
MyMethod(100, 54.321F);
```

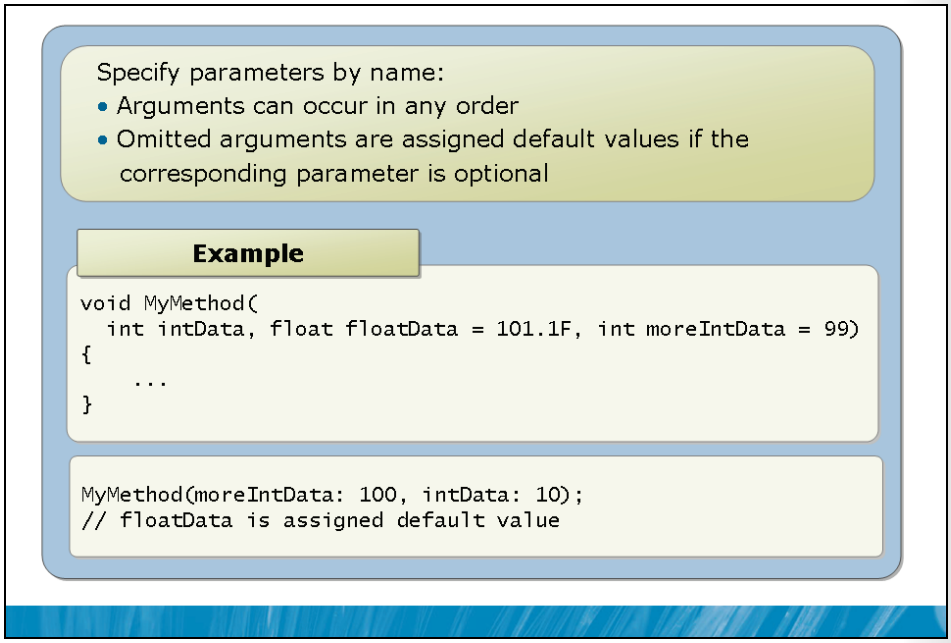
Question: When defining a method with optional parameters, in what order must you specify the parameters?

Additional Reading

For more information about optional parameters, see the Named and Optional Arguments (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192909.

MCT USE ONLY. STUDENT USE PROHIBITED

Calling a Method by Using Named Arguments



Key Points

Traditionally, when calling a method, the order and position of arguments in the method call should correspond to the order of parameters in the method signature. If the arguments were misaligned and the types mismatched, you would get a compile error.

In C#, you can specify parameters by name, and supply arguments in a sequence that differs from that defined by the order of the parameters in the method signature. To use the named arguments feature, you must supply the parameter name and corresponding value separated by a colon. The following code example shows the syntax.

```
// Method declaration.
void MyMethod(int first, double second, string third)
{
}
...
// Method call using named arguments.
MyMethod(third: "Hello", first: 1234, second: 12.12);
```

MCT USE ONLY. STUDENT USE PROHIBITED

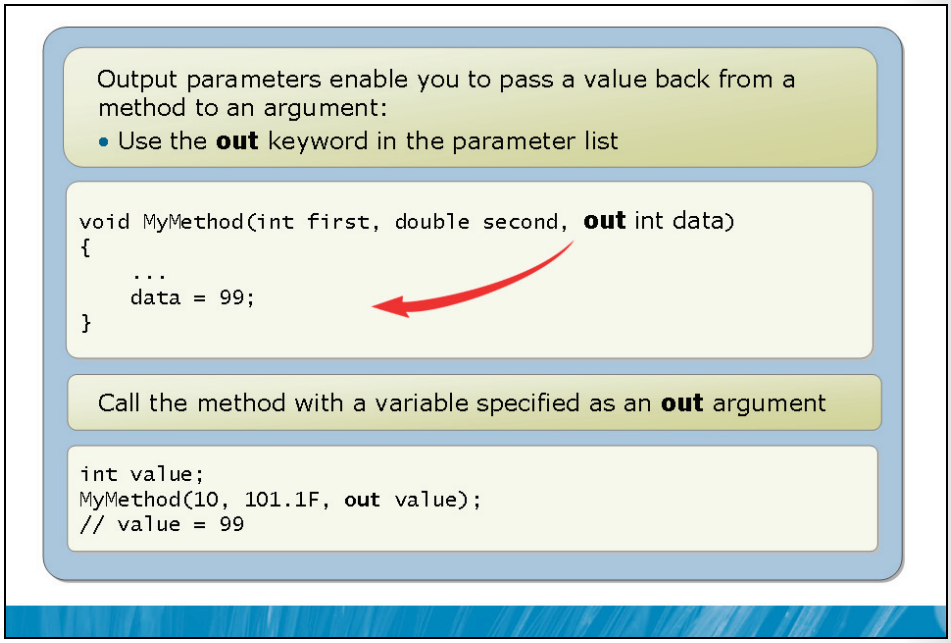
When using named arguments in conjunction with optional parameters, you can easily omit parameters. These parameters will receive their default value. However, if you omit any mandatory parameters, your code will not compile.

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

Question: What is the syntax for using named parameters in method calls?

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Output Parameters?



Key Points

A method can specify a return type, and use a **return** statement to pass a value back to code that calls it. Output parameters enable you to return additional data from a 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 the syntax.

```
void MyMethod(int first, double second, out int data)
{
    ...
    data = 99;
}
```

A method can have as many output parameters as required.

MCT USE ONLY. STUDENT USE PROHIBITED

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 the argument with the **out** keyword. If you attempt to specify an argument that is not a variable, or you omit the **out** keyword, your code will not compile.

The following code example shows how to use an output parameter returned from a method.

```
int value;
MyMethod(10, 101.1F, out value);
// value = 99
```

Question: What happens if you attempt to call **MyMethod** with the code in the following code example?

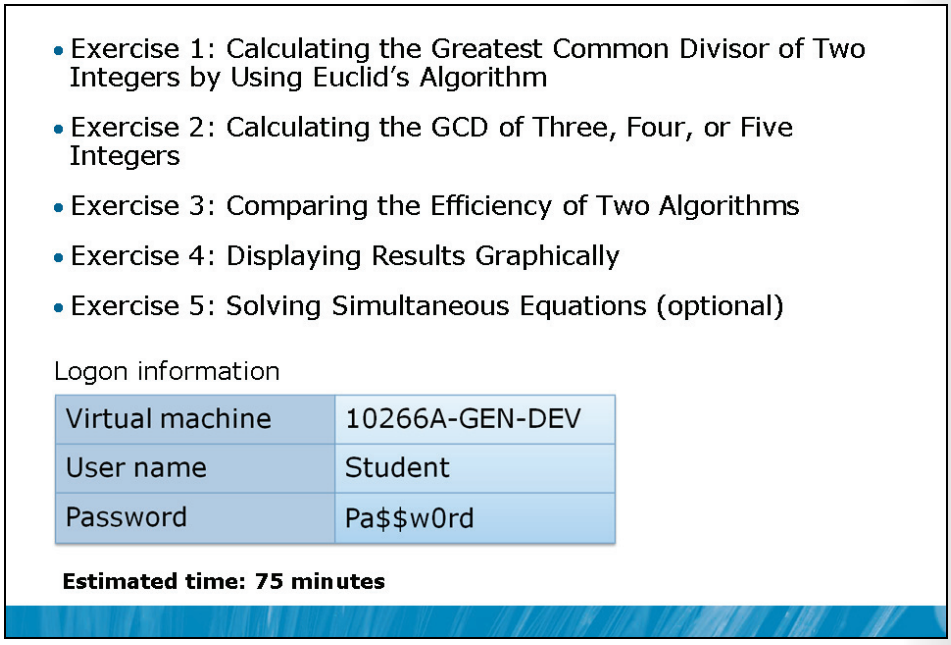
```
MyMethod(10, 101.1F, 20);
```

Additional Reading

For more information about output parameters, see the out parameter modifier (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192910.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Declaring and Calling Methods



Objectives

After completing this lab, you will be able to:

- Create and call methods.
- Define overloaded methods.
- Define methods that take output parameters.
- Define methods that take optional parameters and call them by using named arguments.

Introduction

In this lab, you will create methods to calculate the greatest common divisor (GCD) of a pair of positive integers. You will create an overloaded version of one of these methods that can take up to five integer parameters. You will modify the methods to take an output parameter that returns the time taken to perform the

MCT USE ONLY. STUDENT USE PROHIBITED

calculations. Finally, you will use a method that uses optional parameters to display the relative performance of the methods by displaying a simple graph.

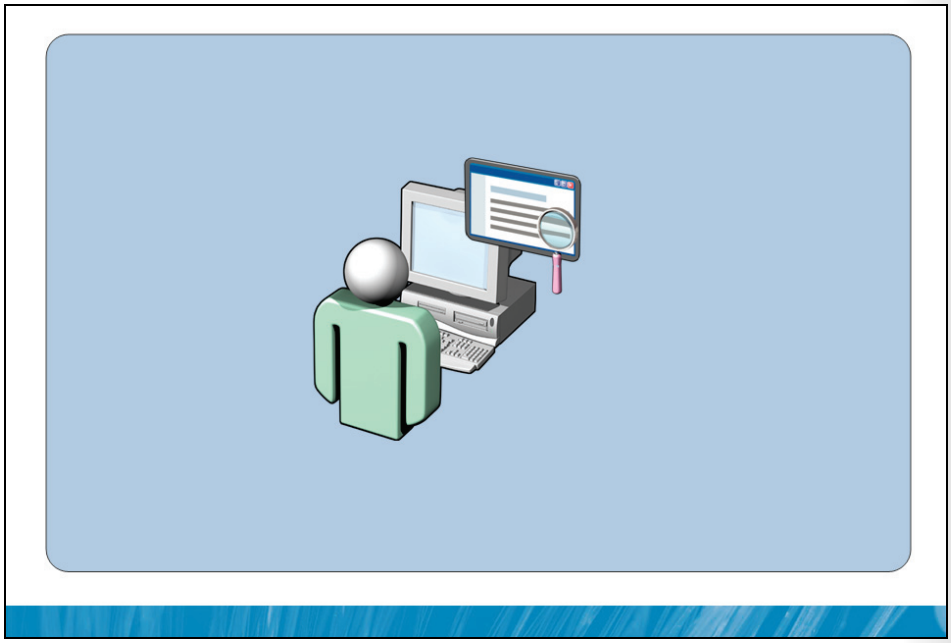
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Some of the calculations that various scientific instruments perform depend on statistical information that is generated by using prime numbers. One of your colleagues has implemented a method for generating prime numbers, but it does not have sufficient performance to meet the requirements of the devices that it will be used with. The software analysts have examined the code and have determined that it can be improved by using a faster algorithm for calculating the GCDs. You have been asked to implement a test application that can calculate the GCD of a set of numbers by using different well-known algorithms, and compare their relative performance.

Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid's Algorithm

In this exercise, you will write a method that implements Euclid's algorithm for calculating the GCD of two integers passed in as parameters. You will test this method by using a Windows® Presentation Foundation (WPF) application that

MCT USE ONLY. STUDENT USE PROHIBITED

prompts the user for the parameter values, and displays the result. You will also generate a unit test project to enable you to automate testing this method.

Scenario

Some of the data that is collected by devices built by Fabrikam, Inc. must be encrypted for security purposes. Encryption algorithms often make use of prime numbers. A part of the algorithm that generates prime numbers needs to calculate the GCD of two numbers.

The GCD of two numbers is the largest number that can exactly divide into the two numbers. For example, the GCD of 15 and 12 is 3. Three is the largest whole number that divides exactly into 15 and 12.

The process for finding the GCD of 2806 and 345 by using Euclid's algorithm is as follows.

1. Keep taking 345 away from 2806 until less than 345 is left and store the remainder.

   In this case, 2806 = (8 × 345) + 46, so the remainder is 46.

2. Keep taking the remainder (46) away from 345 until less than 46 is left, and store the remainder.

   345 = (7 × 46) + 23, so the remainder is 23.

3. Keep taking 23 away from 46 until less than 23 is left, and store the remainder.

   46 = (2 × 23) + 0

4. The remainder is 0, so the GCD of 2806 and 345 was the value of the previously stored remainder, which was 23 in this case.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Implement Euclid's algorithm.
3. Test the **FindGCDEuclid** method.
4. Create a unit test for the **FindGCDEuclid** method.

▶ Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 3\Snippets folder.
4. Open the Euclid solution in the E:\Labfiles\Lab 3\Ex1\Starter folder.

▶ Task 2: Implement Euclid's algorithm

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 1, Task 2** task.

   This task is located in the GCDAlgorithms.cs file.
3. In the **GCDAlgorithms** class, remove the **TODO Exercise 1, Task 2** comment and declare a **public static** method called **FindGCDEuclid**. The method should accept two integer parameters called *a* and *b*, and return an integer value.
4. In the **FindGCDEuclid** method, add code that calculates and returns the GCD of the values specified by the parameters *a* and *b* by using Euclid's algorithm.

   Euclid's algorithm works as follows:

   a. If *a* is zero, the GCD of *a* and *b* is *b*.

   b. Otherwise, repeatedly subtract *b* from *a* (when *a* is greater than *b*) or subtract *a* from *b* (when *b* is greater than *a*) until *b* is zero.

   c. The GCD of the two original parameters is the new value in *a*.

▶ Task 3: Test the FindGCDEuclid method

1. Use the Task List window to navigate to the **TODO Exercise 1, Task 3** task.

   This task is located in the MainWindow.xaml.cs file. This is the code-behind file for a WPF window that you will use to test the **FindGCDEuclid** method and display the results.
2. Remove the **TODO Exercise 1, Task 3** comment, add code to call the static **FindGCDEuclid** method of the **GCDAlgorithms** class, and display the results in the **resultEuclid** label control. In the method call, use the firstNumber and secondNumber variables as arguments (these variables contain values that the user enters in the WPF window). Finally, the result should be formatted as the following code example shows.

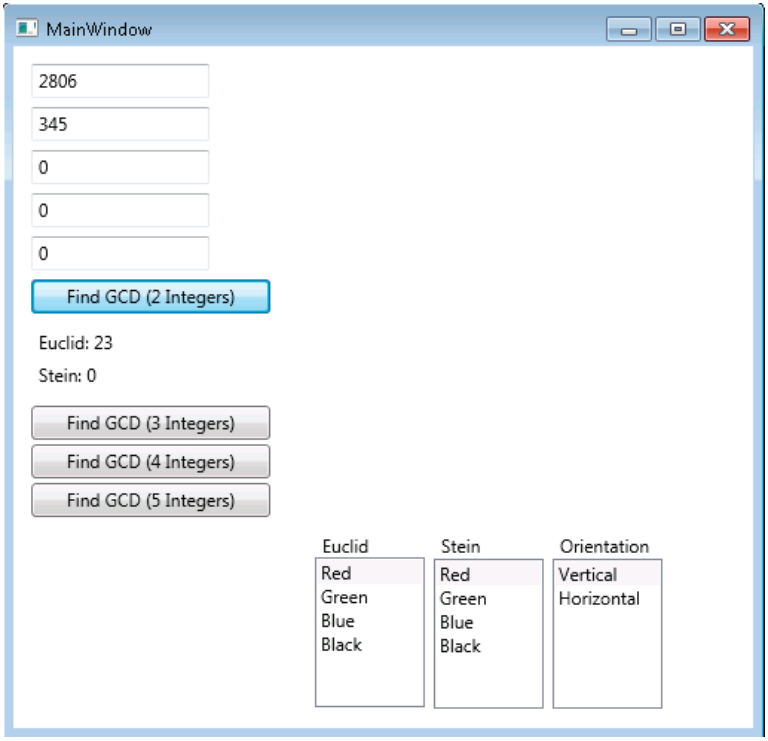
MCT USE ONLY. STUDENT USE PROHIBITED

```
Euclid: result
```



Hint: Set the **Content** property of a label control to display data in a label. Use the **String.Format** method to create a formatted string.

3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first text box, type **2806**
6. In the second text box, type **345** and then click **Find GCD (2 Integers)**. The result of **23** should be displayed, as the following screen shot shows.



7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

MCT USE ONLY. STUDENT USE PROHIBITED

First number	Second number	Result
0	0	0
0	10	10
25	10	5
25	100	25
26	100	2
27	100	1

8. Close the GreatestCommonDivisor application.

▶ Task 4: Create a unit test for the FindGCDEuclid method

1. Open the GCDAlgorithms.cs file.
2. In the **GCDAlgorithms** class, create a unit test for the **FindGCDEuclid** method. Create a new Test Project called GCD Test Project to hold the unit test.
3. In the GCD Test Project project, in the GCDAlgorithmsTest.cs file, locate the **FindGCDEuclidTest** method.
4. In the **FindGCDEuclidTest** method, set the a variable to **2806**, set the b variable to **345**, set the expected variable to **23**, and then remove the **Assert.Inconclusive** method call.
5. Open the Test View window and refresh the display if the unit test is not listed.
6. Run the FindGCDEuclidTest test and verify that the test ran successfully.

Exercise 2: Calculating the GCD of Three, Four, or Five Integers

In this exercise, you will create overloaded versions of this method that can take three, four, or five integer parameters and calculate the GCD of all of these parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Scenario

Some of the encryption algorithms used by devices that Fabrikam, Inc. builds require calculating the GCD of sets of numbers, not just pairs. You have been asked to provide implementations of the Euclid algorithm that can calculate the GCD of three, four, or five integers.

The process for finding the GCD of three numbers *x*, *y*, and *z* is straightforward:

1. Calculate the GCD of *x* and *y* by using the algorithm for two numbers, and store the result in a variable r.
2. Calculate the GCD of r and *z*. The result is the GCD of *x*, *y*, and *z*.

You can apply the same technique to calculate the GCD of four or five integers:

- GCD(*w*, *x*, *y*, *z*) = GCD(*w*, GCD(*x*, *y*, *z*))
- GCD(*v*, *w*, *x*, *y*, *z*) = GCD(*v*, GCD(*w*, *x*, *y*, *z*))

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add overloaded methods to the **GCDAlgorithms** class.
3. Test the overloaded methods.
4. Create unit tests for the overloaded methods.

▶ Task 1: Open the starter project

- Open the Euclid solution in the E:\Labfiles\Lab 3\Ex2\Starter folder.

  This solution contains a completed copy of the code from Exercise 1.

▶ Task 2: Add overloaded methods to the GCDAlgorithms class

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 2** task.
3. In the **GCDAlgorithms** class, remove the **TODO Exercise 2, Task 2** comment, and then declare an overloaded version of the **FindGCDEuclid** method. The method should accept three integer parameters called a, b, and c, and return an integer value.

4. In the new method, add code that uses the original **FindGCDEuclid** method, to find the GCD for the parameters *a* and *b*. Store the result in a new variable called d.
5. Add a second call to the original **FindGCDEuclid** method to find the GCD for variable d and parameter *c*. Store the result in a new variable called e.
6. Add code to return the parameter *e* from the **FindGCDEuclid** method.
7. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept four integer parameters called *a*, *b*, *c*, and *d*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.
8. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept five integer parameters called *a*, *b*, *c*, *d*, and *e*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.

▶ Task 3: Test the overloaded methods

1. Use the Task List window to navigate to the **TODO Exercise 2, Task 3** task.

   This task is located in the code for the WPF window that you can use to test your code.
2. Remove the **TODO Exercise 2, Task 3** comment, locate the **else if (sender == findGCD3)** block, and modify the statement that sets the **Content** property of the **resultEuclid** label to "N/A" as follows:
   a. Call the **FindGCDEuclid** overload that accepts three parameters and pass the variables firstNumber, secondNumber, and thirdNumber as arguments.
   b. Display the results in the **resultEuclid** label control. The result should be formatted as the following code example shows.

```
Euclid: result
```

3. Locate the **else if (sender == findGCD3)** block, the **else if (sender == findGCD4)** block, and the **else if (sender == findGCD5)** block, and modify the statements that set the **Content** property of the **resultEuclid** label to "N/A". Call the appropriate **FindGCDEuclid** overload by using the firstNumber, secondNumber, thirdNumber, fourthNumber, and fifthNumber variables as arguments. Display the results in the **resultEuclid** label control.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Build the solution and correct any errors.
5. Run the GreatestCommonDivisor application.
6. In the GreatestCommonDivisor application, in the MainWindow window, type the values **7396 1978 1204 430 258** and then click **Find GCD (5 Integers)**.

   Verify that the result **86** is displayed.
7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

First number	Second number	Third number	Fourth number	Fifth number	Result
2806	345	0	0	0	23
0	0	0	0	0	0
0	0	0	0	1	1
12	24	36	48	60	12
13	24	36	48	60	1
14	24	36	48	60	2
15	24	36	48	60	3
16	24	36	48	60	4
0	24	36	48	60	12

8. Close the GreatestCommonDivisor application.

▶ Task 4: Create unit tests for the overloaded methods

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 4** task.
3. Remove the **TODO Exercise 2, Task 4** comment and add a test method called **FindGCDEuclidTest1**.
4. In the **FindGCDEuclidTest1** method, declare four variables called a, b, c, and expected, and assign them values **7396**, **1978**, **1204**, and **86** respectively.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Declare a variable called actual, and assign it the result of a call to the **FindGCDEuclid** method call. Use the variables a, b, and c as arguments.
6. Call the **AreEqual static** method of the **Assert** class, and pass the expected and actual variables as arguments.
7. Repeat steps 4–6 to create two more test methods to test the other **FindGCDEuclid** method overloads. Create test methods called **FindGCDEuclidTest2** and **FindGCDEuclidTest3**. Use the values **7396**, **1978**, **1204**, and **430** for the **FindGCDEuclidTest2** method, and the values **7396**, **1978**, **1204**, **430**, and **258** for the **FindGCDEuclidTest3** method. The result should be **86** in both cases.
8. Open the Test View window and refresh the display if the unit test is not listed.
9. Run the FindGCDEuclidTest, FindGCDEuclidTest1, FindGCDEuclidTest2, and FindGCDEuclidTest3 tests and verify that the tests ran successfully.

Exercise 3: Comparing the Efficiency of Two Algorithms

In this exercise, you will write another method that implements Stein's algorithm for calculating the GCD of two integer parameters. The method will take an output parameter that contains the time taken to perform the calculation. You will also modify the method that implements Euclid's algorithm for calculating the GCD of two parameters to take an output parameter, also containing the time taken to perform the calculation. You will then modify the WPF application to test the relative performance of the methods and display the times taken.

Scenario

Stein's algorithm is an alternative algorithm for finding the GCD of two numbers. You have been told that it is more efficient than Euclid's algorithm. A colleague has previously implemented Stein's algorithm, but you decide to test this hypothesis by comparing the time taken to calculate the GCD of pairs of numbers with that taken by using Euclid's algorithm.

The following steps describe the process of calculating the GCD of two numbers, *u* and *v*, by following Stein's algorithm:

1. gcd(0, *v*) = *v* because everything divides by zero, and *v* is the largest number that divides *v*.

   Similarly, gcd(*u*, 0) = *u*. gcd(0, 0) is not typically defined, but it is convenient to set gcd(0, 0) = 0.

MCT USE ONLY. STUDENT USE PROHIBITED

2. If u and v are both even, $\gcd(u, v) = 2 \cdot \gcd(u/2, v/2)$ because 2 is a common divisor.
3. If u is even and v is odd, $\gcd(u, v) = \gcd(u/2, v)$ because 2 is not a common divisor.

   Similarly, if u is odd and v is even, $\gcd(u, v) = \gcd(u, v/2)$.
4. If u and v are both odd, and $u \geq v$, $\gcd(u, v) = \gcd((u - v)/2, v)$.

   If both are odd and $u < v$, $\gcd(u, v) = \gcd((v - u)/2, u)$.

   These are combinations of one step of the simple Euclidean algorithm, which uses subtraction at each step, and an application of step 4 above. The division by 2 results in an integer because the difference of two odd numbers is even.
5. Repeat steps 3–5 until $u = v$, or (one more step) until $u = 0$.

   In either case, the result is $2^k v$, where k is the number of common factors of 2 found in step 2.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Implement Stein's algorithm.
3. Test the **FindGCDStein** method.
4. Add code to test the performance of the algorithms.

▶ Task 1: Open the starter project

- Open the Stein solution in the E:\Labfiles\Lab 3\Ex3\Starter folder.

  This solution contains a completed copy of the code from Exercise 2.

▶ Task 2: Implement Stein's algorithm

1. Open the GCDAlgorithms.cs file.
2. At the end of the **GCDAlgorithms** class, remove the TODO comment and declare a **public static** method called **FindGCDStein**. The method should accept two integer parameters called u and v, and return an integer value.
3. In the **FindGCDStein** method, add the code in the following code example, which calculates and returns the GCD of the values that are specified by the

MCT USE ONLY. STUDENT USE PROHIBITED

parameters *u* and *v* by using Stein's algorithm. You can either type this code manually, or use the Mod03Stein code snippet.



Note: For the purposes of this exercise, it is not necessary for you to understand this code. However, if you have time, you may like to compare this method to the algorithm that is described in the exercise scenario. Note that this code uses the left-shift (<<) and right-shift (>>) operators to perform fast multiplication and division by 2. If you left-shift an integer value by one place, the result is the same as multiplying the integer value by 2. Similarly, if you right-shift an integer value by one place, the result is the same as dividing the integer value by 2. In addition, the **|** operator performs a bitwise **OR** operation between two integer values. Consequently, if either *u* or *v* are zero, the expression **u | v** is a fast way of returning the value of whichever variable is non-zero, or zero if both are zero. Similarly, the **&** operator performs a bitwise **AND** operation, so the expression **u & 1** is a fast way to determine whether the value of *u* is odd or even.

```
static public int FindGCDStein(int u, int v)
{
    int k;

    // Step 1.
    // gcd(0, v) = v, because everything divides zero,
    // and v is the largest number that divides v.
    // Similarly, gcd(u, 0) = u. gcd(0, 0) is not typically
    // defined, but it is convenient to set gcd(0, 0) = 0.
    if (u == 0 || v == 0)
        return u | v;

    // Step 2.
    // If u and v are both even, then gcd(u, v) = 2·gcd(u/2, v/2),
    // because 2 is a common divisor.
    for (k = 0; ((u | v) & 1) == 0; ++k)
    {
        u >>= 1;
        v >>= 1;
    }

    // Step 3.
    // If u is even and v is odd, then gcd(u, v) = gcd(u/2, v),
    // because 2 is not a common divisor.
    // Similarly, if u is odd and v is even,
    // then gcd(u, v) = gcd(u, v/2).

    while ((u & 1) == 0)
        u >>= 1;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Step 4.
    // If u and v are both odd, and u ≥ v,
    // then gcd(u, v) = gcd((u – v)/2, v).
    // If both are odd and u < v, then gcd(u, v) = gcd((v – u)/2, u).
    // These are combinations of one step of the simple
    // Euclidean algorithm,
    // which uses subtraction at each step, and an application
    // of step 3 above.
    // The division by 2 results in an integer because the
    // difference of two odd numbers is even.
    do
    {
        while ((v & 1) == 0)  // Loop x
            v >>= 1;

        // Now u and v are both odd, so diff(u, v) is even.
        //   Let u = min(u, v), v = diff(u, v)/2.
        if (u < v)
        {
            v -= u;
        }
        else
        {
            int diff = u - v;
            u = v;
            v = diff;
        }
        v >>= 1;
        // Step 5.
        // Repeat steps 3–4 until u = v, or (one more step)
        // until u = 0.
        // In either case, the result is (2^k) * v, where k is
        // the number of common factors of 2 found in step 2.
    } while (v != 0);

    u <<= k;

    return u;
}
```

▶ Task 3: Test the FindGCDStein method

1. Open the MainWindow.xaml.cs file.
2. In the **MainWindow** class, in the **FindGCD_Click** method, locate the **TODO Exercise 3, Task 2** comment. Remove this comment and replace the statement that sets the **Content** property of the **resultStein** label with code that calls the

MCT USE ONLY. STUDENT USE PROHIBITED

FindGCDStein method by using the variables firstNumber and secondNumber as arguments. Display the results in the **resultStein** label control. The result should be formatted as the following code example shows.

```
Stein: result
```

3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**.

   Verify that the value **4** is displayed in both labels.
6. Close the GreatestCommonDivisor application.
7. Open the GCDAlgorithmsTest.cs file.
8. At the end of the **GCDAlgorithmsTest** class, locate the **TODO Exercise 3, Task 2** comment, remove the comment, and then add a test method called **FindGCDSteinTest**.
9. In the **FindGCDSteinTest** method, declare three variables called u, v, and expected, and assign them values **298467352**, **569484**, and **4** respectively.
10. Declare a variable called actual, and assign it the result of a call to the **FindGCDStein** method call. Use the variables u and v as arguments.
11. Call the **static AreEqual** method of the **Assert** class, and pass the expected and actual variables as arguments.
12. Open the Test View window and refresh the display if the unit test is not listed.
13. Run the FindGCDSteinTest test, and verify that the test ran successfully.

▶ Task 4: Add code to test the performance of the algorithms

1. Open the GCDAlgorithms.cs file.
2. In the **GCDAlgorithms** class, locate the **FindGCDEuclid** method that accepts two parameters, and modify the method signature to take an *out* parameter called *time* of type **long**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. At the start of the **FindGCDEuclid** method, add code to initialize the time parameter to zero, create a new **Stopwatch** object called **sw**, and start the stop watch.

   The **Stopwatch** class is useful for timing code. The **Start** method starts an internal timer running. You can subsequently use the **Stop** method to halt the timer, and establish how long the interval was between starting and stopping the timer by querying the **ElapsedMilliseconds** or **ElapsedTicks** properties.

4. At the end of the **FindGCDEuclid** method, before the **return** statement, add code to stop the **Stopwatch** object, and set the *time* parameter to the number of elapsed ticks of the **Stopwatch** object.

5. Comment out the other **FindGCDEuclid** method overloads.

6. Modify the **FindGCDStein** method to include the time output parameter, and add code to record the time each method takes to run. Note that the **FindGCDStein** method contains two **return** statements, and you should record the time before each one.

7. Open the MainWindow.xaml.cs file.

8. In the **FindGCD_**Click method, modify each of the calls to the **FindGCDEuclid** method and the **FindGCDStein** method to use the updated method signatures, as follows:

   a. For calling the Euclid algorithm, create a long variable called timeEuclid.

   b. For calling the Stein algorithm, create a long variable called timeStein.

   c. Format the results displayed in the labels as the following code example shows.

```
[Euclid]
Euclid: result, Time (ticks): result

[Stein]
Stein: result, Time (ticks): result
```

9. Comment out the code that calls the overloaded versions of the **FindGCDEuclid** method.

10. Open the GCDAlgorithmsTest.cs file.

11. Modify the **FindGCDEuclidTest** and **FindGCDSteinTest** methods to use the new method signatures. Comment out the methods **FindGCDEuclidTest1**, **FindGCDEuclidTest2**, and **FindGCDEuclidTest3**.

MCT USE ONLY. STUDENT USE PROHIBITED

12. Build the solution and correct any errors.
13. Run the GreatestCommonDivisor application.
14. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The result of **4** should be displayed. The time reported for Euclid's algorithm should be approximately three times more than that for Stein's algorithm.



Note: The bigger the difference between the two values, the more efficient Stein's algorithm becomes compared to Euclid's. If you have time, try experimenting with different values.

15. Close the GreatestCommonDivisor application.
16. Open the Test View window and refresh the display if the unit test is not listed.
17. Run the **FindGCDEuclidTest** and **FindGCDSteinTest** methods and verify that the tests ran successfully.

Exercise 4: Displaying Results Graphically

In this exercise, you will add a method to the application that displays the results graphically by using a bar graph. The parameters to the method are the two times taken, the orientation of the graph, and the colors to use to display the bars. The graph orientation and color parameters will be optional parameters. The default values will generate a vertical bar graph with a red bar for the first value and a blue bar for the second.

Scenario

You want to display the results of the timing comparisons graphically by using a simple, customizable bar graph.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Display the algorithm timings graphically.
3. Modify the **DrawGraph** method.
4. Modify the code that calls the **DrawGraph** method.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the starter project

- Open the Charting solution in the E:\Labfiles\Lab 3\Ex4\Starter folder.

  This solution contains a completed copy of the code from Exercise 3.

▶ Task 2: Display the algorithm timings graphically

1. Open the MainWindow.xaml.cs file.
2. In the **FindGCD_Click** method, locate the Call **DrawGraph** comment, and add a call to the **DrawGraph** method, using the timeEuclid and timeStein variables as parameters.
3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The result of **4** should be displayed. The time reported for both algorithms should be represented by a simple bar graph in the window.
6. Close the GreatestCommonDivisor application.

▶ Task 3: Modify the DrawGraph method

1. In the **MainWindow** class, locate the **DrawGraph** method and add the following three optional parameters:
   a. A parameter called orientation of type **Orientation** with a default value of **Orientation.Horizontal**.
   b. A parameter called *colorEuclid* of type **string** with a default value of **"Red"**.
   c. A parameter called *colorStein* of type **string** with a default value of **"Blue"**.
2. In the **DrawGraph** method, locate the **Use optional orientation parameter** comment, and remove the existing declaration of the orientation variable.
3. Locate the **Use optional color parameters** comment, and modify the assignment of the bEuclid and bStein variables to use the optional parameters in the method signature. To do this, you will need to use the **BrushConverter** class and the **ConvertFromString** instance method as shown in the following code example.

```
...
private void DrawGraph(long euclidTime, long steinTime,
    Orientation orientation = Orientation.Horizontal,
    string colorEuclid = "Red",
    string colorStein = "Blue")
{
    ...
    BrushConverter bc = new BrushConverter();
    Brush bEuclid = (Brush)bc.ConvertFromString(colorEuclid);
    Brush bStein = (Brush)bc.ConvertFromString(colorStein);
    ...
}
...
```

4. Build the solution and correct any errors.
5. Run the GreatestCommonDivisor application.
6. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The graph should be displayed as before, except the **DrawGraph** method call is now using the default parameter values, and the graph is displayed as a pair of red and blue vertical bars.
7. Close the GreatestCommonDivisor application.

▶ Task 4: Modify the code that calls the DrawGraph method

1. Open the MainWindow.xaml.cs file.
2. In the **FindGCD_Click** method, locate the **Modify the call to Drawgraph to use the optional parameters** comment, and modify the **DrawGraph** method call to use the *orientation*, *colorEuclid*, and *colorStein* optional parameters as follows:
   a. *orientation*—set to the selected value of the **chartOrientation** list box.
   b. *colorEuclid*—set to the selected item of the **euclidColor** list box.
   c. *colorStein*—set to the selected item of the **steinColor** list box.

   These list boxes are already included in the user interface; they appear in the lower part of the window. The user can select the values in these list boxes to change the appearance of the graph that is displayed.
3. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484**
6. In the **Euclid** list box, select **Green**, in the **Stein** list box, select **Black**, in the **Orientation** box, select **Horizontal**, and then click **Find GCD (2 Integers)**. The graph should be displayed with the specified colors and direction.
7. Close the GreatestCommonDivisor application.

Exercise 5: Solving Simultaneous Equations (optional)

In this exercise, you will write a method that solves simultaneous linear equations with four variables (w, x, y, and z). You will use a WPF application to obtain input from the user (the coefficients of w, x, y, and z and the result for four equations) to simulate the data captured by a device, and call the method. The method will use Gaussian Elimination (a well-known algorithm for solving simultaneous linear equations) to generate solutions for w, x, y, and z, which will be returned as an array. The WPF application will then display these values.

Scenario

A key requirement of one of the engineering applications produced by Fabrikam, Inc. is the ability to solve simultaneous linear equations based on some of the data captured by various measuring devices.

Suppose you need to find the values of x, y, and z given the equations in the following code example.

```
2x + y – z = 8              (equation E1)
-3x – y + 2z = -11          (equation E2)
-2x + y + 2z = -3           (equation E3)
```

The method to solve these equations, known as Gaussian Elimination, proceeds as follows:

1. Eliminate x from equations E2 and E3:
   - To eliminate x from E2, calculate (3 ÷ 2) × E1 + E2

     The coefficient of x in E2 is −(3 ÷ 2) times that of the coefficient of x in E1, so multiplying E1 by (3 ÷ 2) and adding E2 removes x from E2.
   - To remove x from E3, calculate E1 + E3

MCT USE ONLY. STUDENT USE PROHIBITED

The coefficient of x in E3 is −1 times that of the coefficient of x in E1, so adding E1 to E3 removes x from E3.

The result is shown in the following code example.

```
2x + y – z = 8          (E1)
(1/2)y + (1/2)z = 1     (E2)
2y + z = 5              (E3)
```

2. Next, eliminate y from E3:

   - To eliminate y from E3, calculate −4 × E2 + E3

     The coefficient of y in E3 is four times that of the coefficient of y in E2, so multiplying E2 by −4 and adding E3 removes y from E3.

   The result is shown in the following code example.

```
2x + y – z = 8          (E1)
(1/2)y + (1/2)z = 1     (E2)
-z = 1                  (E3)
```

   The equations are now in triangular form—three unknowns in the first equation, two in the second equation, and one in the third equation.

3. Solve E3 and calculate the value for z, as the following code example shows.

```
z = -1          (E3)
```

4. Substitute the value of z into E2 to calculate the value of y, as the following code example shows.

```
(1/2)y – 1/2 = 1  =>
(1/2)y = 3/2      =>
y = 3            (E2)
```

5. Substitute the values of z and y into E1 to calculate the value of x, as the following code example shows.

```
2x + 3 + 1 = 8   =>
2x = 4           =>
x = 2           (E1)
```

   This process is known as back substitution.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create methods to copy arrays.
3. Convert the equations to triangular form.
4. Perform back substitution.
5. Test the solution.

▶ Task 1: Open the starter project

1. Open the SimultaneousEquations solution in the E:\Labfiles\Lab 3\Ex5 \Starter folder.
2. Open the MainWindow.xaml file.

   This is a different application from the one that the previous exercises have used. It is a WPF application that enables a user to enter the coefficients for four simultaneous equations that contain four variables (w, x, y, and z), and then uses Gaussian Elimination to find a solution for these equations. The results are displayed in the lower part of the screen.

▶ Task 2: Create methods to copy arrays

1. Open the Gauss.cs file.

   This file contains a class called **Gauss** that provides a method called **SolveGaussian**. This method takes two arrays as parameters:

   - A two-dimensional array of **double** values containing the coefficients for the variables w, x, y, and z specified by the user for each equation.
   - An array of **double** values containing the result of each equation specified by the user (the value to the right of the equal sign).

   The method returns an array of **double** values that will be populated with the values of w, x, y, and z that provide the solutions to these equations.

   You will implement the body of this method in this exercise.

2. In the **Gauss** class, locate the **TODO Exercise 5, Task 2** comment. Remove this comment and declare a **private static** method called **DeepCopy1D**. The method should accept and return a **double** array.

The **SolveGaussian** method will make a copy of the arrays passed in as parameters to avoid changing the original data that the user provided.

3. In the **DeepCopy1D** method, add code to create a deep copy of the one-dimensional array that was passed into the method. Your code should perform the following tasks:
   a. Create and initialize an array with the same number of columns as the array that was passed in.
   b. Copy the values in the array that was passed as a parameter into the new array.
   c. Return the new array.
4. In the **Gauss** class, declare another **private static** method called **DeepCopy2D**. The method should accept and return a two-dimensional **double** array.
5. In the **DeepCopy2D** method, add code to create a deep copy of the two-dimensional array that was passed into the method. Your code should do the following:
   a. Create and initialize an array with the same number of columns and rows as the array that was passed in.
   b. Copy the values in the array that was passed in as the parameter into the new array.
   c. Return the new array.

▶ Task 3: Convert the equations to triangular form

1. In the **SolveGaussian** method, use the **DeepCopy1D** and **DeepCopy2D** methods to create deep copies of the **rhs** and **coefficients** arrays.
2. Locate the **Convert the equation to triangular form** comment, and add code to convert the equations represented by the copies of the **coefficients** and **rhs** arrays into triangular form.



Note: The **Gauss** class defines a constant integer called **numberOfEquations** that specifies the number of coefficients that the application can resolve.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Perform back substitution

- In the **Gauss** class, in the **SolveGaussian** method, locate the **Perform the back substitution and return the result** comment, and then add code to perform back substitution. To do this, you will need to work back from the equation with one unknown and substituting the values calculated at each stage to solve the remaining equations.

▶ Task 5: Test the solution

1. Open the MainWindow.xaml.cs file.
2. In the **MainWindow** class, locate the **TODO Exercise 5, Step 5** comment, and add code to call the **SolveGaussion** method. Use the coefficients and rhs variables as parameters and set the **answers** array to the result.
3. Run the GaussianElimination application.
4. In the GaussianElimination application, in the MainWindow window, enter the following equations, and then click **Solve**.



Note: Enter a value of zero in the corresponding text if no value is specified for w, x, y, or z in the equations below.

$2w + x - y + z = 8$

$-3w - x + 2y + z = -11$

$-2w + x - 2y = -3$

$3w - x + 2y - 2z = -5$

Verify that the following results are displayed:

$w = 4$

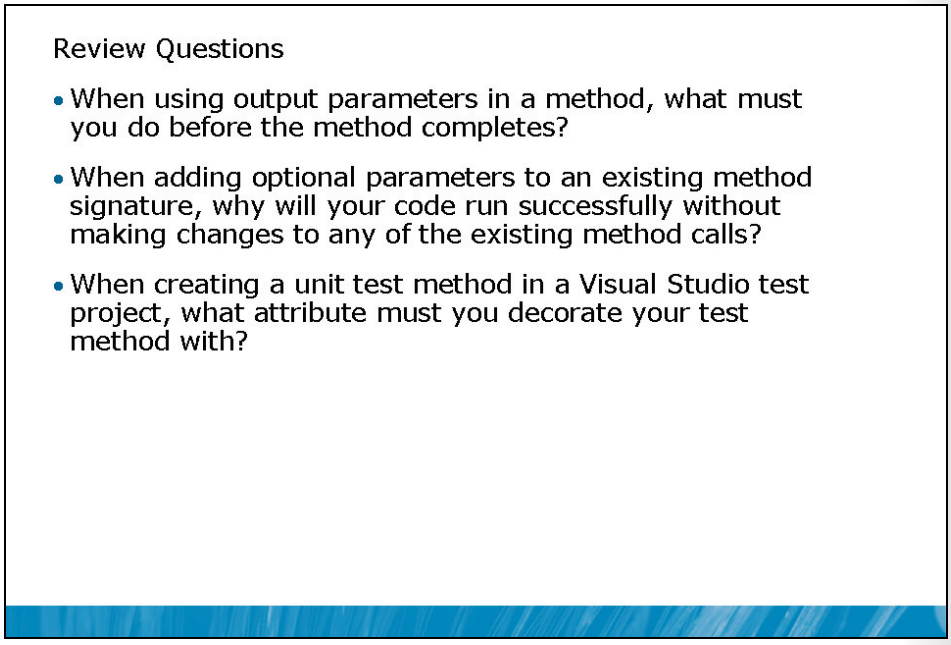
$x = -17$

$y = -11$

$z = 6$

5. Experiment with other equations. Note that not all systems of equations have a solution. How does your code handle this situation?
6. Close the MainWindow window.
7. Close Visual Studio.

Lab Review

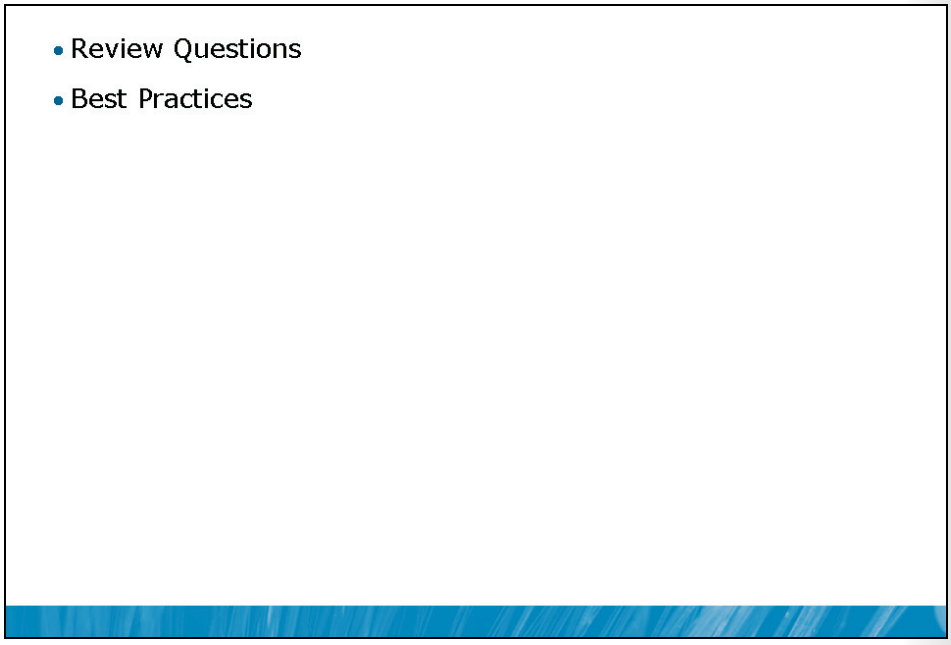


Review Questions

1. When using output parameters in a method, what must you do before the method completes?
2. When adding optional parameters to an existing method signature, why will your code run successfully without making changes to any of the existing method calls?
3. When creating a unit test method in a Visual Studio test project, what attribute must you decorate your test method with?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What return type do you specify for a method that does not return any data?
2. What term is given to the process of defining multiple methods with the same name, but different parameter lists?
3. What is the difference between an optional parameter and a named argument?
4. What is the purpose of output parameters?

Best Practices Related to Using Methods

Supplement or modify the following best practices for your own work situations:

- Keep methods as small and lightweight functional units. If methods start to become large, consider refactoring code into smaller logical methods.
- Create unit tests for all **public** methods. You can assume that any **private** methods that you create will be tested when the **public** methods are called.

MCT USE ONLY. STUDENT USE PROHIBITED

- Use output parameters only when it is absolutely necessary. If you find yourself using output parameters too often, reconsider the purpose of the method.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 4

Handling Exceptions

Contents:

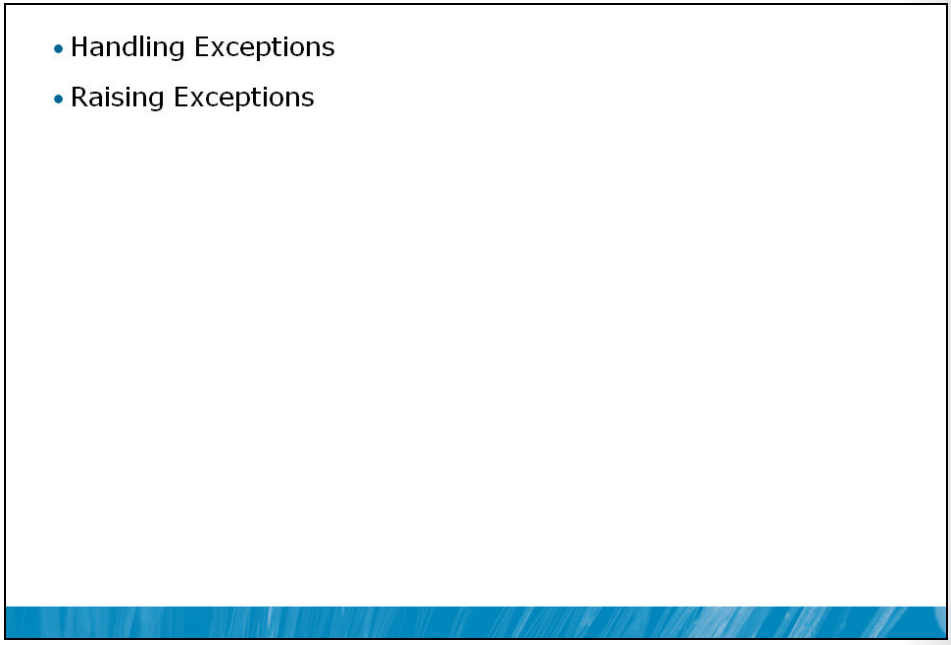
Lesson 1: Handling Exceptions 4-3

Lesson 2: Raising Exceptions 4-23

Lab: Handling Exceptions 4-34

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



In the previous modules, you have been introduced to some important concepts that will enable you to develop Microsoft® .NET Framework applications. Until this point, if your application caused an exception, it would crash in an ungainly manner. This approach to handling exceptions is clearly not acceptable in a professional application.

Exception handling is an important concept and your applications should be designed with exception handling in mind. This module explains how you can 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.

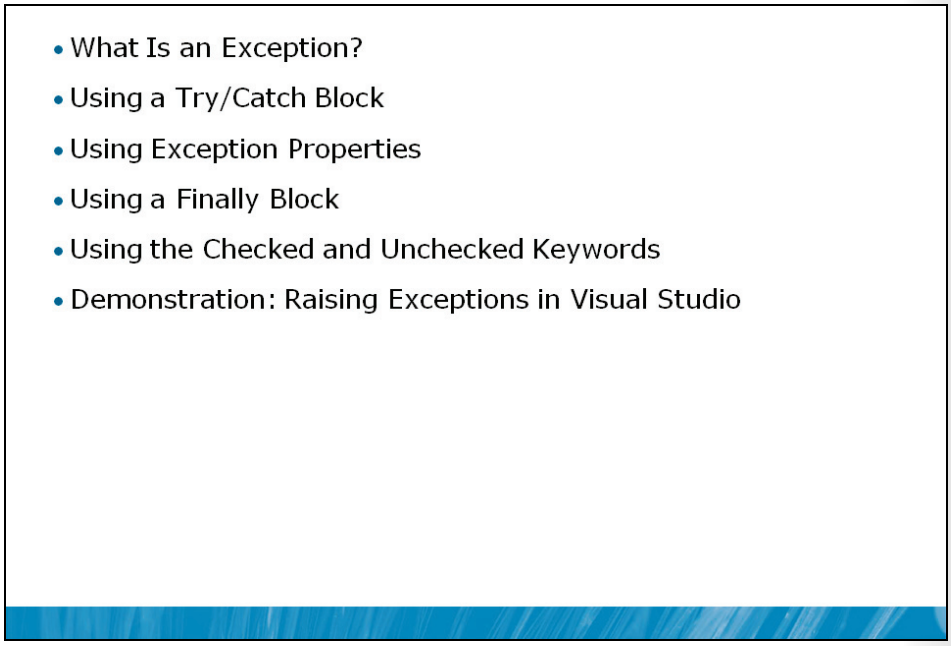
Objectives

After completing this module, you will be able to:

- Describe how to catch and handle exceptions.
- Describe how to create and raise exceptions.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Handling Exceptions



Applications may function as expected during development, with limited use and controlled input. However, when an application is deployed to its live environment and subject to constraints such as dynamically changing data at greater volumes, errors are likely to emerge. To manage the user experience and ensure that your application remains useable when exceptions occur, you need to handle these exceptions.

This lesson introduces concepts such as the **try/catch/finally** block, which will enable you to implement structured exception handling (SEH) in your applications.

Objectives

After completing this lesson, you will be able to:

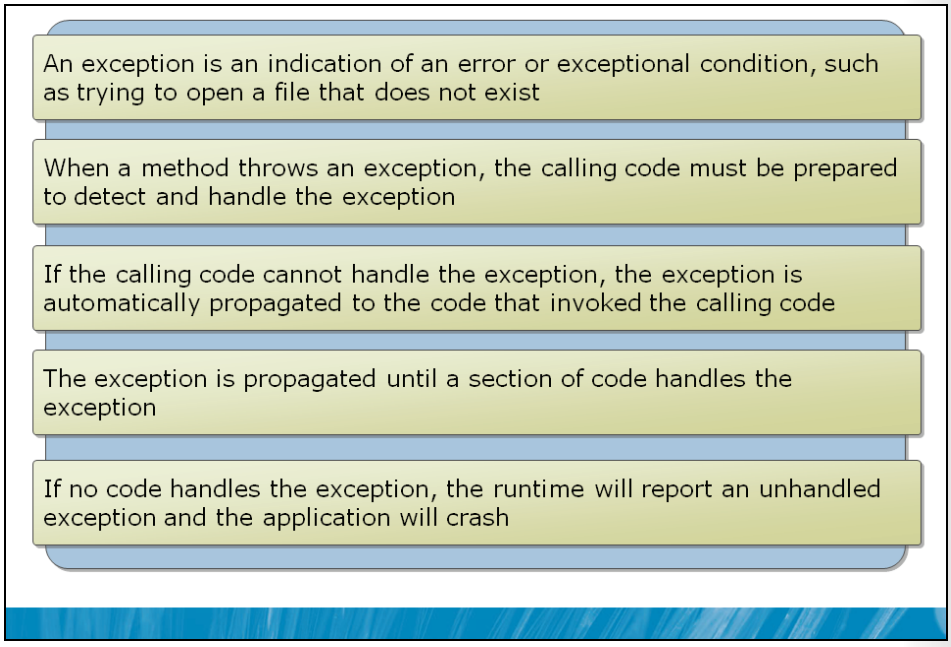
- Describe the purpose of exceptions.
- Describe how to use a **try/catch** block.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe how to use some of the properties that the **Exception** class exposes.
- Explain how to use a **finally** block.
- Explain how to use the **checked** and **unchecked** keywords to control numeric overflow checking.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Exception?



Key Points

Many things can go wrong as an application runs, not just because of faults in the logic, but because applications typically depend on many variables outside the scope of the application, such as the existence of files on the file system and connections to databases. During the design of your application, you must consider how to ensure that your application can recover gracefully when such problems arise. It is common practice to check the return values from methods to ensure that they have executed correctly. However, there are issues with this approach:

- Not all methods return a value.
- You need to know why the method call has failed, not just that it has failed.
- This approach does not cover how to handle unexpected errors such as running out of memory.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Many older systems 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. Needless to say, this approach was not robust because it was too easy for a programmer to forget to handle errors.

How Exceptions Propagate

This is where exceptions in the .NET Framework prove useful. 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, it tries to open a file, but the file does not exist.

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, it 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.

As an example, suppose the **A** method calls the **B** method. As part of its processing, the **B** method calls the **C** method. While it is running, the **C** method throws an exception. This exception may cause the **C** method to abort, and the exception is passed back to the **B** method. If the **B** method is not prepared to handle the exception, it also aborts and the same exception is passed back to the **A** method. If the **A** method handles the exception, execution continues in the **A** method after the exception-handling logic.

If the **A** method is not prepared to handle the exception, the exception will be propagated back to the method that called the **A** method. If this is the **Main** method, and **Main** is also not prepared to handle the exception, the application reports the unhandled exception to the user and then terminates.

A method can catch and handle its own exceptions to provide a degree of robustness that the calling code may not even be aware of. For example, a method that updates a database may catch an exception that occurs if the connection to the database fails. It may try connecting again, possibly with an alternative set of credentials. This process can be hidden from the code that called the method.

The Exception Type

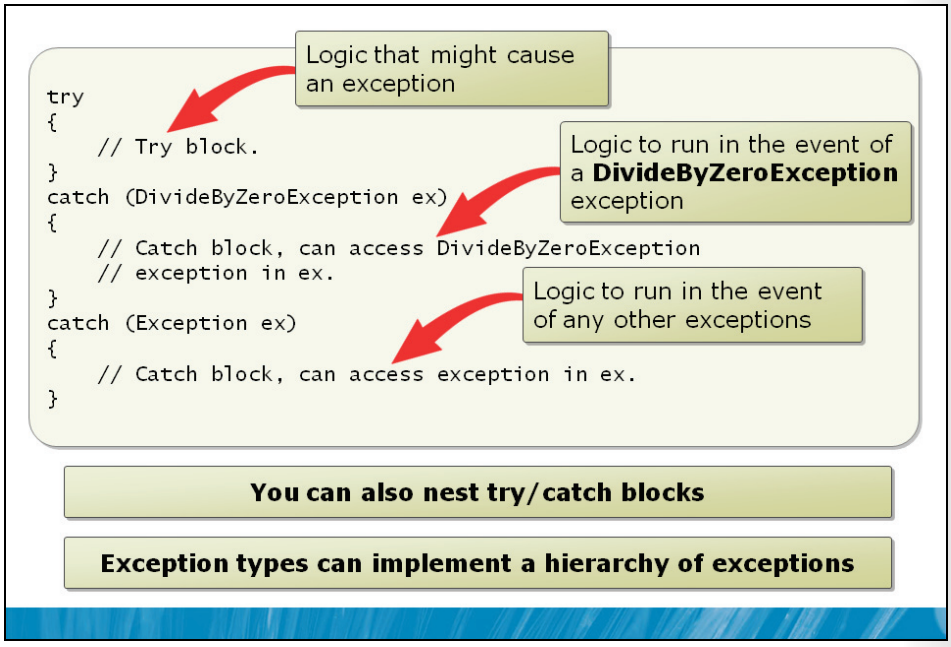
In the previous scenario, the **A** method is not aware that the **B** method called the **C** method. Consequently, when the **B** method aborted because it could not handle the exception, it was the **B** method that caused the exception as far as the **A**

method was concerned. When an exception occurs, it is therefore 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. This object is passed to the code that handles the exception, which can use it to determine the best way to handle the exception.

Question: Discuss your experiences of applications that have crashed with other students and the instructor.

MCT USE ONLY. STUDENT USE PROHIBITED

Using a Try/Catch Block



Key Points

The **try**/**catch** block is the key programming construct for SEH. 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.

Try/Catch Block Syntax

The syntax for using a **try**/**catch** block is shown in the following code example.

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

MCT USE ONLY. STUDENT USE PROHIBITED

The statements that are enclosed in the braces in the **try** block can be any 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.



Note: When the code in the **catch** block has completed, execution will continue at the first statement after the **try/catch** block.

The **catch** specification for each block determines what exceptions will be caught and what variable is used to store the exception, if any. You can specify **catch** blocks for different types of exceptions. The .NET Framework defines many different exception types for many of the common exceptions that can occur. For example, some methods in the **System.IO** namespace that handle file I/O throw the **FileNotFoundException** exception if an application attempts to access a nonexistent file. In addition, the common language runtime (CLR) itself throws a **DivideByZeroException** exception if you attempt to perform numeric division by zero.

When an exception occurs, you do not have to include a **catch** block for every type of exception, and exceptions that are not matched will be propagated as described earlier.

The most general form of **catch** block is one that has no catch specification, so it catches any type of exception. This is illustrated in the following code example.

```
try
{
    // Try block.
}

catch
{
    // Catch block.
}
```

In this code example, any exception that is thrown in the **try** block will transfer control to the **catch** block. However, you will not be able to determine the cause of the exception. To access this information, you must provide a variable to use in the **catch** specification, as the following code example shows.

```
try
{
    // Try block.
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
catch (Exception ex)
{
    // Catch block, can access exception in ex.
}
```

The exception information that is generated by the code that threw the exception is passed in this variable. Note that this code will also catch any type of exception.

You frequently use the **Exception** type to catch all exceptions that have not been otherwise handled. In the following code example, if the code in the **try** block causes a **DivideByZeroException** 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.

```
try
{
    // Try block.
}
catch (DivideByZeroException ex)
{
    // Catch block, can access DivideByZeroException exception in ex.
}
catch (Exception ex)
{
    // Catch block, can access exception in ex.
}
```

Sequencing Catch Blocks

You must put your **catch** blocks 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.



Note: Exception types can implement a hierarchy of exceptions. For example, the .NET Framework provides an exception type called **ArithmeticException**, which you can use to indicate an error when evaluating an arithmetic expression. The **DivideByZeroException** type is a specific classification of **ArithmeticException** (the .NET Framework also defines two other types of **ArithmeticException** called **OverflowException** and **NotFiniteNumberException**). If you catch the **DivideByZeroException** exception, only that exception type is caught. However, if you catch the **ArithmeticException** exception, this **catch** block will trap **DivideByZeroException**, **OverflowException**, and **NotFiniteNumberException**. Therefore, if you have multiple **catch** blocks, you must ensure that you place the blocks

MCT USE ONLY. STUDENT USE PROHIBITED

for more specific exceptions before those for less specific ones. The **Exception** type is the least specific of all, and should be the final **catch** block if you use it.

Nesting Try/Catch Blocks

A **try**/**catch** block is a programming construct like any other statement in C#. You can nest **try**/**catch** blocks, so a **try** block can contain a **try**/**catch** block, as the following code example shows.

```
try
{
    // Outer try block.
    ...
    try
    {
        // Nested try block
    }
    catch (FileNotFoundException ex)
    {
        // Catch block for nested try block
    }
    ...
    // Outer try block continued
}
catch (DivideByZeroException ex)
{
    // Catch block, can access DivideByZeroException exception in ex.
}
catch (Exception ex)
{
    // Catch block, can access exception in ex.
}
```

If a **FileNotFoundException** exception occurs in the nested **try** block, the nested **catch** block runs. Execution continues in the nested **try** block, at the first statement after the nested **catch** block.

If any other type of exception occurs in the nested **try** block, the exception is propagated to the outer **try** block, where it is caught by the **catch** block for the **Exception** type. Execution then continues at the first statement after the outer **try**/**catch** block.

Nesting provides a convenient mechanism for handling and recovering certain types of exception locally within a method.

MCT USE ONLY. STUDENT USE PROHIBITED

Try/Catch Example

The following code example shows an example of a **try/catch** block used for file access.

```
StreamReader reader = null;
try
{
    string fileName = GetFileName();
    reader = new StreamReader(fileName);
    string savedData = reader.ReadToEnd();
}
catch (IOException ioex)
{
    // Handle the IO exception.
}
catch (Exception ex)
{
    // Handle all other types of exceptions.
}
```

In this code example, the **try** block contains code that attempts to read data from a file. If an exception of type **IOException** is thrown, the **catch** block is executed. If any other type of exception is thrown, the generic **catch** block is executed.

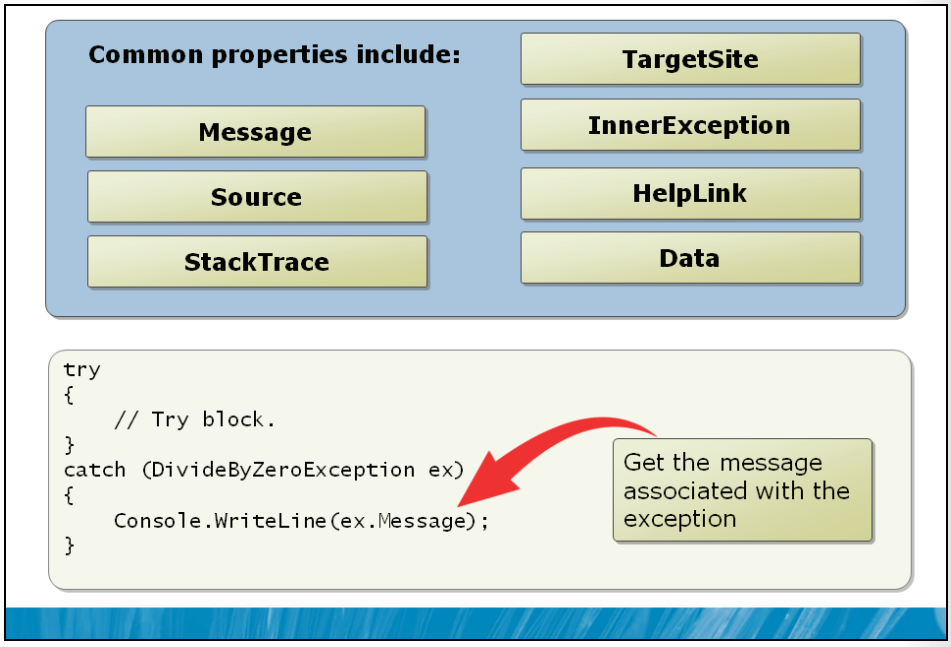
Question: How would you use the **try/catch** block to catch all exceptions regardless of type, and then execute some generic additional logic?

Additional Reading

For more information about **try/catch** blocks, see the try-catch (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192911.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Exception Properties



Key Points

All exception classes provide the same basic information that is common to all exceptions, but they may also provide additional information that is specific to the type of the exception.

The properties that are common to all exceptions are shown in the following table.

Property	Description
Message	This property is the most commonly used and contains a string that describes the error that has occurred.
Source	This property contains a string that indicates the object or application that caused the error.
StackTrace	This property is a string that contains the call stack at the point where the exception was thrown.
TargetSite	This property is a string that contains the name of the method that generated the exception.

MCT USE ONLY. STUDENT USE PROHIBITED

Property	Description
InnerException	This property is a member of type **Exception** that can be used to contain an additional exception. You can use this property to drill down into the cause of a problem in some circumstances. This property is often used in the **catch** block of nested SEH code to take an exception that has been thrown and wrap it in a new exception that is then thrown and caught by code further up the call stack.
HelpLink	This string property can be used to store a link to additional information on the error that occurred.
Data	This property is an object that you can use to store additional information about an error.

The following code example shows how to display the message that is provided when a **DivideByZeroException** exception occurs.

```
try
{
    // Try block.
}
catch (DivideByZeroException ex)
{
    Console.WriteLine(ex.Message);
}
```

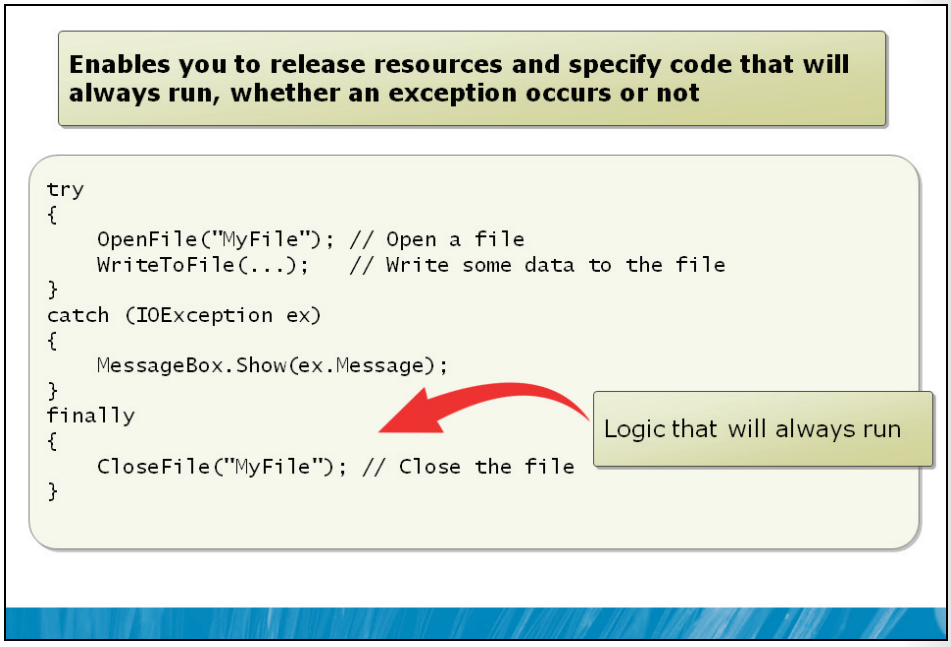
Question: You have a **catch** block that contains some logic to write details of any exceptions to a log file. The **catch** block will catch all types of exceptions. What members of the exception class would you use to get a description and the source of the error?

Additional Reading

For more information about the members in the **System.Exception** class, see the Exception Members page at http://go.microsoft.com/fwlink/?LinkId=192912.

MCT USE ONLY. STUDENT USE PROHIBITED

Using a Finally Block



Key Points

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, have occurred. (If an exception is caught and handled, the exception handler in the **catch** block will run first, 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.

Finally Block Syntax

The syntax for using a **finally** block is shown in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

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

Flow of Control for Try/Catch/Finally

When you use **finally** blocks, the flow of control is more complicated than in **try**/**catch** blocks. The flow of control is as follows:

1. The **try** block runs.
2. If an exception is thrown:
   - If there is a matching **catch** block for the exception:
     i. The **catch** block that matches the exception is executed.
     ii. The **finally** block executes.
   - If there is a matching **catch** block for the exception, and this **catch** block itself causes an exception:
     i. The **catch** block that matches the original exception is executed.
     ii. The **finally** block executes.
     iii. The exception caused by the **catch** handler is propagated to any enclosing **try**/**catch** block, or to the calling method if there is no enclosing **try**/**catch** block.
   - If there is no matching **catch** block for the exception:
     i. The **finally** block executes.

MCT USE ONLY. STUDENT USE PROHIBITED

ii. The exception is propagated to any enclosing **try/catch** block, or to the calling method if there is no enclosing **try/catch** block.

3. If no exception is thrown, the **finally** block executes.

The important thing here is that the **finally** block is always executed. This enables the code in a **finally** block to tidy up after an exception before any other code deals with the exception.

Try/Catch/Finally Example

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

```
try
{
    OpenFile("MyFile"); // Open a file
    WriteToFile(...);   // Write some data to the file
}
catch (IOException ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    CloseFile("MyFile"); // Close the file
}
```

The code in the **try** block calls methods that open a file and write some data to that file. If an **IOException** exception occurs, the **catch** block displays the details of the exception. The **finally** block calls the **CloseFile** method to close the file. This code will always run and the file will always be closed, no matter what exceptions occur.

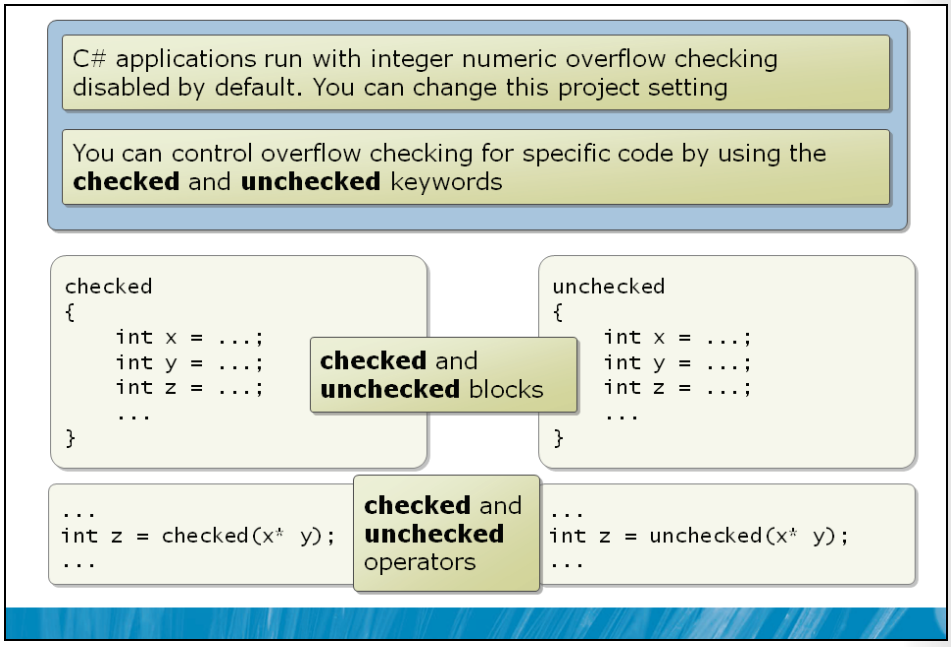
Question: Describe the differences between a **catch** block and a **finally** block.

Additional Reading

For more information about the **finally** block, see the try-finally (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192913.

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Checked and Unchecked Keywords



Key Points

Integer arithmetic is a large part of most applications. It is very common to see idioms such as the following code example.

```
for (int i = 0; i < 10; i++)
{
    ...
}
```

All numeric variables, including integers, have a maximum value. If you increment an integer that has this maximum value, the result is a numeric overflow. However, integer arithmetic is so common that checking for numeric overflow after every integer numeric operation might seriously impact the performance of applications. Consequently, Microsoft Visual C#® applications run with integer numeric overflow checking disabled by default. In these applications, there is a risk that numeric overflow may lead to incorrect results; if you increment an integer variable that has the largest integer value possible, the result is a negative value.

MCT USE ONLY. STUDENT USE PROHIBITED

If you have a section of code that may cause numeric overflow, you can reinstate overflow checking by using the **checked** keyword.

You can also enable overflow checking for an entire application and disable it locally by using the **unchecked** keyword.



Note: You can activate and disable overflow checking in Microsoft Visual Studio® 2010 by setting the project properties. In Solution Explorer, click ***YourProject*** (where ***YourProject*** is the name of your project). On the **Project** menu, click ***YourProject*** **Properties**. In the **Project Properties** dialog box, click the **Build** tab. Click the **Advanced** button in the lower-right corner of the page. In the **Advanced Build Settings** dialog box, select or clear the **Check for arithmetic overflow/underflow** check box.

Using a Checked Block

You use the **checked** keyword to define a block of code, enclosed in braces, that includes numeric overflow checking. If numeric overflow occurs, the statement that caused the overflow will cause an **OverflowException** exception to be thrown. The following code example shows how to enable overflow checking and catch the **OverflowException** exception.

```
checked

{
    int x = ...;
    int y = ...;
    int z = ...;
    ...

    try
    {
        z = x * y;  // May cause numeric overflow
    }

    catch (OverflowException ex)
    {
        ... // Handle the overflow exception
    }
    ...
}
```

You can also apply the **checked** operator to an individual expression, as shown in the following code example. The scope of overflow checking is limited to the expression.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
public int Multiply(short operandX, short operandY)
{
    return checked((short)(operandX * operandY));
}
...
```

Using an Unchecked Block

If you have enabled overflow checking for an application, you can use the **unchecked** keyword to suppress overflow checking in a block or an individual expression. The syntax is the same as for a **checked** block or statement.

Question: In what scenario would you want to use the **checked** keyword?

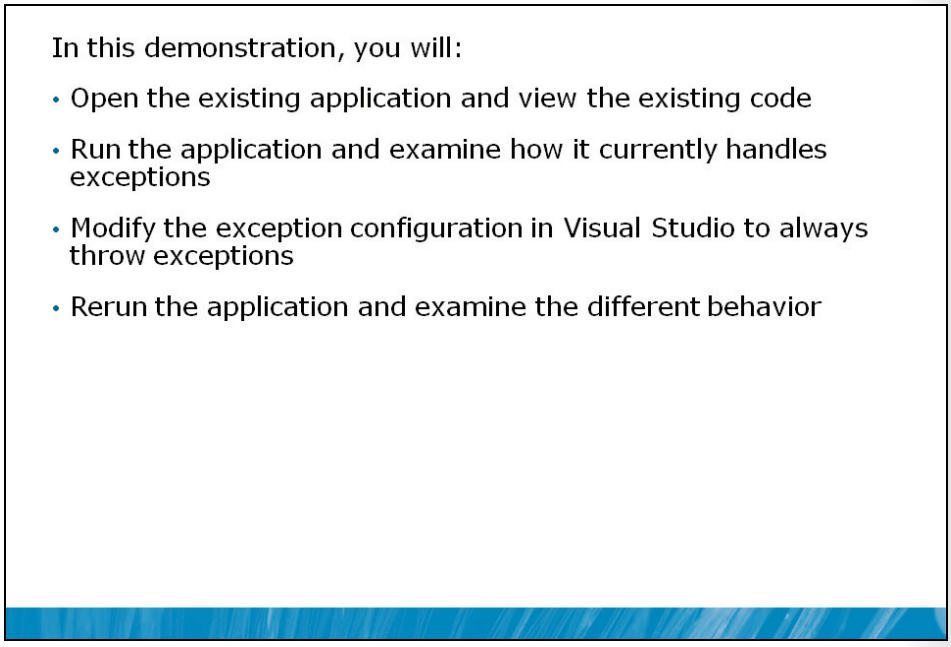
Additional Reading

For more information about using the **checked** and **unchecked** keywords, see the Checked and Unchecked (*C# Reference*) page at

http://go.microsoft.com/fwlink/?LinkId=192914.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Raising Exceptions in Visual Studio



Key Points

- Open the existing application and view the existing code.
- Run the application and examine how it currently handles exceptions.
- Modify the exception configuration in Visual Studio to always throw exceptions.
- Rerun the application and examine the different behavior.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Microsoft Visual Studio 2010.
3. In Visual Studio 2010, open the **FabrikamUserManagement** solution in the **E:\Demofiles\Mod4\Demo1\Starter\FabrikamUserManagement** folder.
4. In the Code Editor window, examine the following code in the **Main** method:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

a. The **Main** method contains a call to the **Users.GetUserById** method, which returns a **user** object for the provided user ID.

b. If you specify a user ID that does not exist, the method returns **null**.

c. When the method returns, the application displays the **userName** field returned.

d. The method call is in a **try**/**catch** block.

e. The **catch** block contains code to display details of any exceptions to the Command Prompt window.

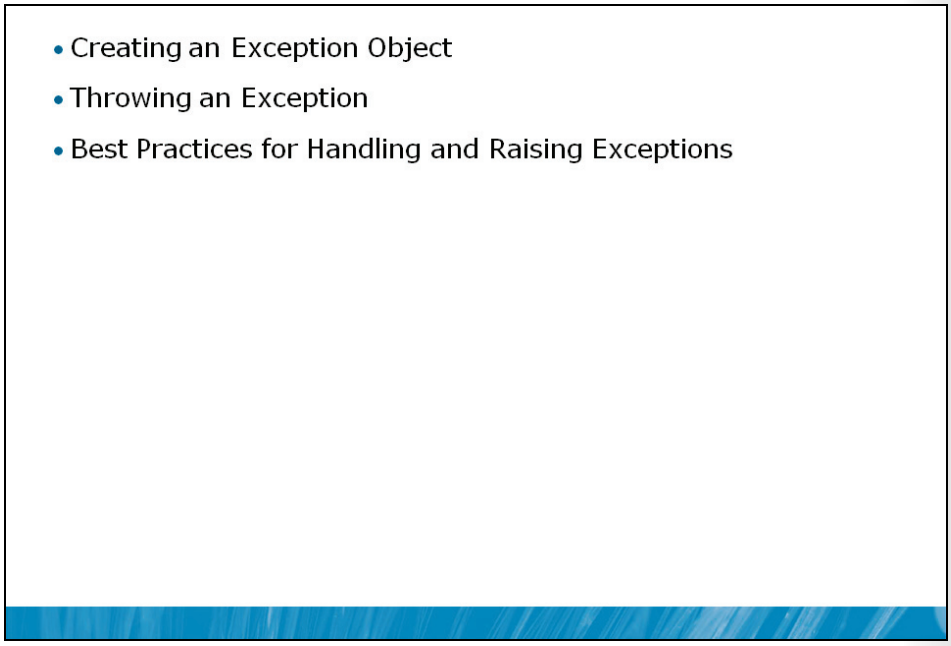
5. Run the application with debugging.
6. Switch to the Command Prompt window, and examine the **Object reference not set to an instance of an object** exception message.

   The application generated this message because a user could not be found with the ID of 5, so the **GetUserById** method returned **null**. Subsequently, any code that tried to use that user object would generate a **null** reference exception. Because the code is enclosed in a **try**/**catch** block, the exception was caught and error logic was executed.
7. Stop debugging.
8. On the **Debug** menu, click **Exception**.
9. In the **Exceptions** dialog box, in the **Break when an exception is** list, expand **Common Language Runtime Exceptions**, and then expand **System**.
10. In the **Break when an exception is** list, under **System**, locate the **System.NullReferenceException** row.
11. For the **System.NullReferenceException** row, clear the **User-unhandled** check box, and then select the **Thrown** check box.
12. In the **Exceptions** dialog box, click **OK**.
13. Run the application with debugging.

    Now when the application tries to use the user object and generates a **null** reference exception, Visual Studio stops the application and notifies you.

Question: How can you guarantee that Visual Studio will always notify you if an exception occurs instead of automatically propagating the exception to a **catch** block?

Lesson 2
Raising Exceptions



Using a **try**/**catch** block enables an application to catch and handle exceptions. These exceptions may be thrown by the CLR if an application attempts to perform an illegal operation, such as attempting to divide by zero, or access a file for which the user running the application does not have permission. However, an application may also detect its own fault conditions, such as an invalid combination of arguments passed as parameters into a method. In this case, it is useful for the application itself to throw an exception that indicates the reason for the fault.

This lesson explains the key concepts that enable you to create and raise exceptions.

Objectives

After completing this lesson, you will be able to:

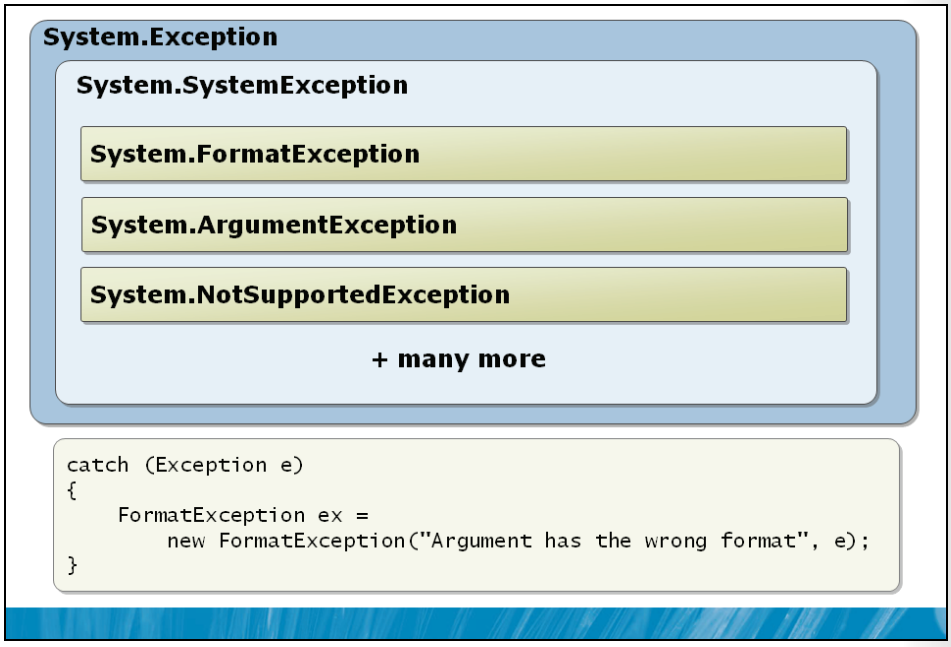
- Describe how to create a new exception object by using some of the predefined exception types that are provided with the .NET Framework.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain how to throw an exception by using the **throw** keyword.
- Describe some of the best practices for raising and handling exceptions.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating an Exception Object



Key Points

You can add your code to your own methods that detect fault conditions, and then throw a corresponding exception to indicate the nature of the fault to the caller. The calling code can catch and handle the exception, as described in the previous lesson.

The .NET Framework provides a wide range of built-in exception types, which all inherit from the **Exception** class. (You will learn more about inheritance in C# in Module 8, "Inheriting from Classes and Implementing Interfaces.") Each exception type is intended to indicate a specific classification of exception. For example, the **FileNotFoundException** exception type indicates that an attempt was made to open a file that does not exist, and **DivideByZeroException** is used to indicate an attempt to divide by zero in a mathematical expression. There is nothing to stop you throwing any type of exception in a method, but it is considered good practice to throw an exception of a type that is appropriate to the fault condition that is detected. The following table lists some of the more commonly used exception types.

MCT USE ONLY. STUDENT USE PROHIBITED

Exception type	Description
ArgumentException	You can throw this exception if the caller specifies an argument to a method that does not conform to the requirements of the method. You can use this exception type to indicate generalized errors with arguments, or you can use the **ArgumentOutOfRangeException** and **ArgumentNullException** types to indicate more specific errors (for example, if you pass the value 100 to a method, and the method expects a value between 1 and 99, or if you pass the value **null** as an argument).
FormatException	You can throw this exception if the caller specified an argument that contains data that does not have the required format. For example, if the caller passes a string argument that does not contain information in the format that the method expects, the method should throw a **FormatException** exception.
NotImplementedException	You can throw this exception to indicate that you have not yet implemented the code in a method. This exception is primarily useful while you are developing code when you have defined the method, but have not written the code for the body of the method.
NotSupportedException	You can throw this exception if a caller attempts to perform an unsupported operation by using your method, such as specifying arguments that indicate that the caller wants to write to a read-only file.
FileNotFoundException **DirectoryNotFoundException** **DriveNotFoundException**	You can throw these exceptions in methods that attempt to open files on behalf of a caller. If the name of the file that is indicated by arguments that the caller specifies reference a file that does not exist, or the file is in a folder or drive that does not exist.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: You can also create your own custom exception types by inheriting from the **System.Exception** class. Inheritance in C# is described in Module 8, "Inheriting from Classes and Implementing Interfaces."

Syntax for Creating an Exception Object

You use the **new** keyword to create an exception object. Specify the type of the exception, and provide information that indicates the cause of the exception. You typically provide this information as a string that contains an error message, although you can also include another exception object if your exception was the result of another exception. The text of the error message is made available to the **catch** block that handles the exception in the **Message** property of the exception. If you include another exception object in your exception, the details are available to the **catch** block that handles the exception in the **InnerException** property.

The following code example shows two examples of how to create a **FormatException** object.

```
// Example 1
// Create a FormatException containing an error message.
FormatException ex =
    new FormatException("Argument has the wrong format");

...

// Example 2
try
{
    ... // Statements that might cause an exception if data
    ... // is in the wrong format
}
catch (Exception e)
{
    // Create a FormatException containing an error message
    // and a reference to the original exception.
    FormatException ex =
        new FormatException("Argument has the wrong format", e);
    ...
}
```

Different exception classes can provide constructors that take additional parameters. The following code example shows the **ArgumentOutOfRangeException** exception. This exception type has a constructor that can take two string parameters. The first parameter is the name of a parameter that is out of range, and the second parameter is the text of the error message.

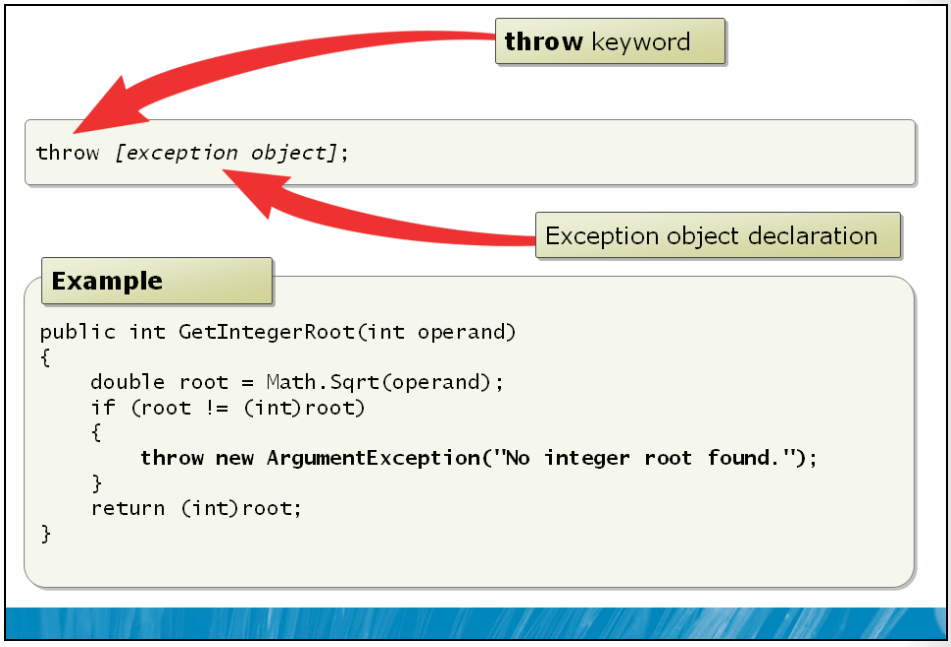
MCT USE ONLY. STUDENT USE PROHIBITED

```
ArgumentException argEx = new ArgumentOutOfRangeException("param1",
    "Parameter param1 too large.");
```

Question: You are in the process of adding several new methods to your application. So far you have added the method signatures. What else should you do to indicate that the method is not complete and functional?

MCT USE ONLY. STUDENT USE PROHIBITED

Throwing an Exception



Key Points

After you have created an exception object, you can throw it 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, as described in Lesson 1 of this module.

Note: Throwing an exception is an expensive operation in terms of CPU cycles, so you should use it with care.

Syntax for Throwing an Exception

To throw an exception, you use the **throw** keyword and specify the exception object to throw. The following code example shows the syntax.

```
throw [exception object];
```

MCT USE ONLY. STUDENT USE PROHIBITED

For example, you create and throw a **FormatException** exception as shown in the following code example.

```
FormatException ex =
    new FormatException("Argument has the wrong format");
throw ex;
```

Rethrowing an Exception

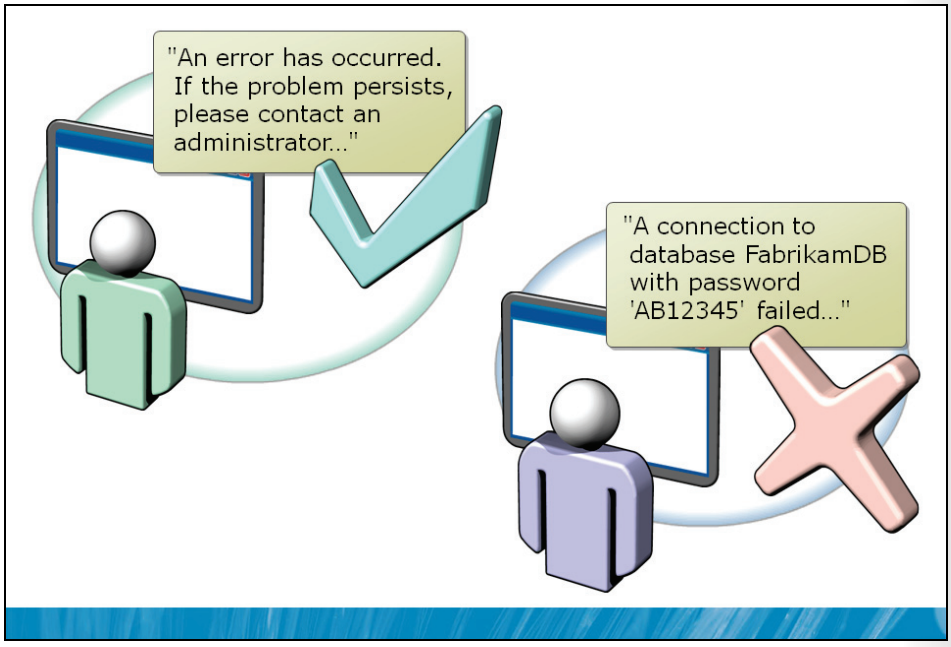
A common strategy is for a method or block of code to catch any exceptions and attempt to handle them. If the **catch** block for an exception cannot resolve the error, it can rethrow the exception and propagate it to the caller. To do this, specify the **throw** keyword, as the following code example shows.

```
try
{
    ... // Statements that might cause an exception
}
catch (Exception e)
{
    // Attempt to handle the exception
    ...
    // If this catch handler cannot resolve the exception,
    // throw it to the calling code
    throw;
}
```

Question: Where does execution continue after you perform a **throw** statement?

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices for Handling and Raising Exceptions



Key Points

The constructs for implementing exception handling in your applications are straightforward to use, but as with any programming constructs, it is important to follow a good design. The following list explains some of the best practices for handling exceptions:

- Throw an exception that is appropriate to the error condition that is detected.
- The logic in your application should not rely on **try** and **catch** blocks to function under nonexceptional conditions. You should design your methods so that, under normal circumstances, they will not throw exceptions. Only catch and throw exceptions for conditions that are outside the expected logical flow of an application.
- When you define multiple **catch** blocks, order them from the most specific to the least specific. If you catch the **Exception** type, it must be the final handler in a set of **catch** blocks.
- Catch and log detailed exception messages for diagnostic purposes, and then display user-friendly messages to the user. Remember that any text that is

MCT USE ONLY. STUDENT USE PROHIBITED

displayed to the user should be localizable, and the text should be retrieved from resource files.

The following code example shows how you can write a message to the Windows® event log.

```
using System.Diagnostics;

...

// The event source name.
string source = "My C# application";

// The event log to write to.
string log = "Application";

// The message you want to write.
string message = "An error with code ex1032 has occurred...";

// Check to see if the event source exists, and if not, create it.
if (!EventLog.SourceExists(source))
{
    EventLog.CreateEventSource(source, log);
}

// Write the message to the event log.
EventLog.WriteEntry(source, message, EventLogEntryType.Error);
```

- Don't display detailed exception messages to the user because a malicious user could use detailed information to cause your application to malfunction, or even gain access to protected information. A common mistake that is made in data access layers is to provide detailed error information resulting from an incorrect database query. This can enable a malicious user to understand the underlying logic in your application and use knowledge of this to attack your system.
- Effective exception handling should enable your application to recover from exceptions, and enable the user to continue using your application. In the event of an exception, the user should not lose data, and your application should not crash.

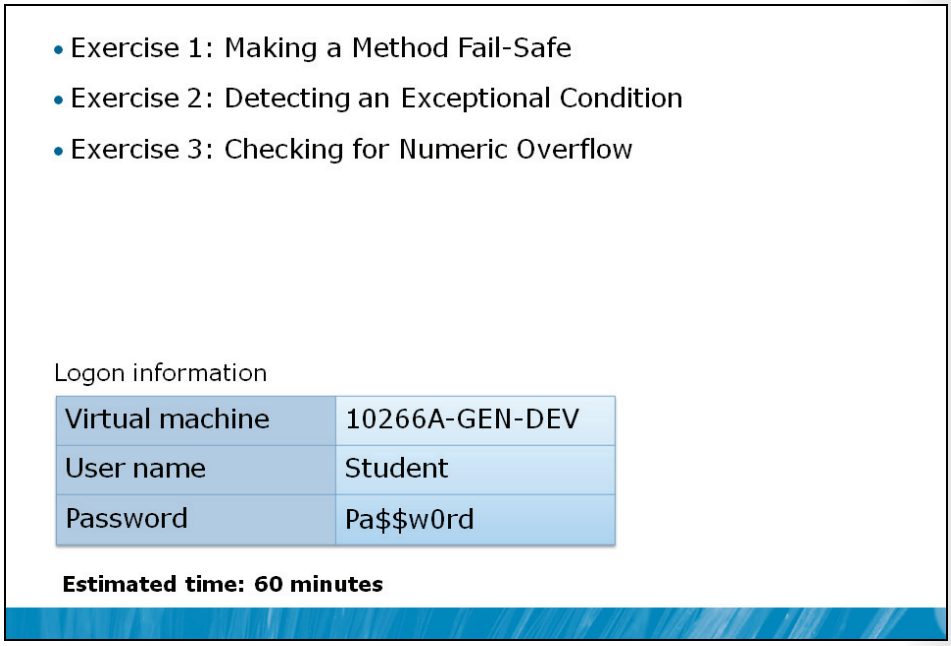
Question: In your application, you have a method that returns a user object. When you have the user object, you are going to use it as a parameter in another method call. There is a possibility that some of the data in the user object is

MCT USE ONLY. STUDENT USE PROHIBITED

incorrectly formatted and that, if you try to use this data, it would cause an exception. What would you do in this situation?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Handling Exceptions



Objectives

After completing this lab, you will be able to:

- Add code to make a method fail-safe.
- Add code to a method to detect a condition and throw an exception if that condition is met.
- Add code to use the **checked** keyword to test for numeric overflow.

Introduction

In this lab, you will catch and handle the possible exceptions that can occur in a method. You will also use the **finally** construct to implement code that runs even if an exception occurs. You will also add code that throws an exception if an error condition is detected in a method. Finally, you will enable integer overflow checking in an application.

MCT USE ONLY. STUDENT USE PROHIBITED

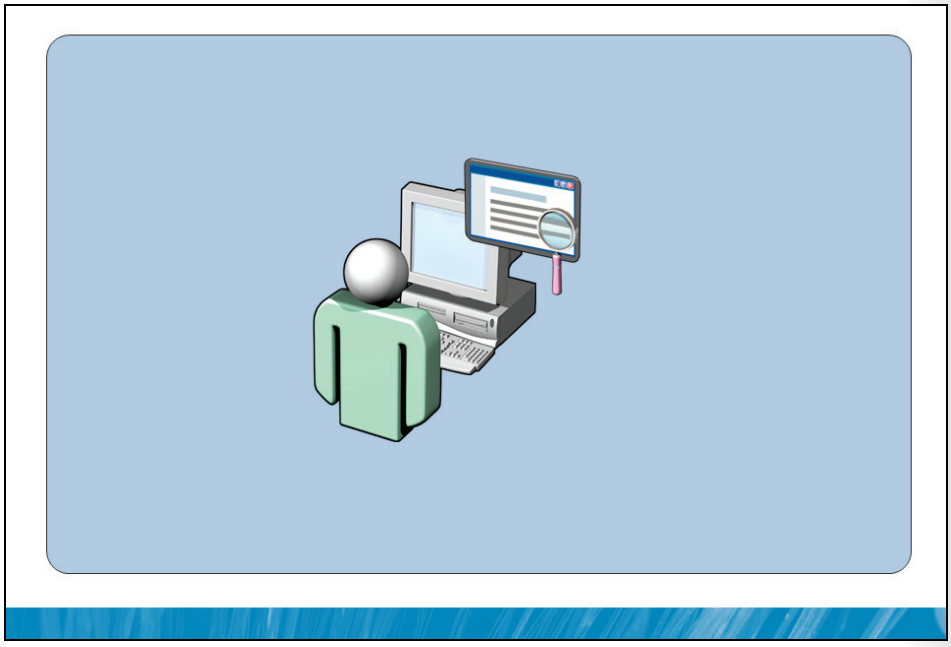
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Exception handling and resource management are a critical part of all of the applications that Fabrikam, Inc. develops. Failure to handle exceptions correctly in software that drives a large piece of machinery could result in life-threatening situations. Even in smaller, less critical scientific devices, an unhandled exception could result in lost data and the need to repeat experiments.

Exercise 1: Making a Method Fail-Safe

In this exercise, you will add fail-safe functionality to an application to ensure that it continues to function even if one or more exceptions occur. The code itself is located in a Windows Presentation Foundation (WPF) application that acts as a test harness.

Scenario

Fabrikam, Inc. provides intelligent switching devices that can monitor the environment for a critical condition (such as the temperature exceeding a specified

MCT USE ONLY. STUDENT USE PROHIBITED

value), and trigger a shutdown operation. These switching devices are used in applications in the energy industry to initiate the shutdown of nuclear reactors. Needless to say, the correct operation of these devices is essential. Fabrikam, Inc. is developing a new model of switching device, and requires you to write part of the software that controls its operation. You have been provided with the code that performs the shutdown operation. This code contains a number of steps, and they must all be run. If any step fails, the code must report the failure, but continue with the next step.

The main tasks for this exercise are as follows:

1. Open the Failsafe solution and run the application.
2. Examine the **Switch** class.
3. Handle the exceptions that the **Switch** class throws.
4. Test the application.

▶ Task 1: Open the Failsafe solution and run the application

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Visual Studio 2010.
3. Open the Failsafe solution in the E:\Labfiles\Lab 4\Ex1\Starter folder.
4. Run the Failsafe project and repeatedly click **Shutdown** until an exception occurs.



Note: The **Switch** class is designed to randomly throw an exception, so you may not encounter an exception the first time that you click the button. Repeatedly click the **Shutdown** button until an exception occurs.

▶ Task 2: Examine the Switch class

1. If it is not already open, open the Switch.cs file in Visual Studio.
2. Examine the **Switch** class.

   Note that the class contains several methods, each of which is capable of throwing at least one exception, dependent on the outcome of a random number generation. Toward the bottom of the file, note the definitions of each

MCT USE ONLY. STUDENT USE PROHIBITED

of the custom exceptions that the **Switch** class can throw. These are very basic exception classes that simply encapsulate an error message.

▶ Task 3: Handle the exceptions that the Switch class throws

The SwitchTestHarness project contains a reference to the **SwitchDevice** class, and invokes each method in the **Switch** class to simulate polling multiple sensors and diagnostic devices. Currently, the project contains no exception handling, so when an exception occurs, the application will fail. You must add exception-handling code to the SwitchTestHarness project, to protect the application from exceptions that the **Switch** class throws.

1. Open the MainWindow.xaml.cs file in Visual Studio.
2. In the **MainWindow** class, locate the **Button1_Click** method. This method runs when the user clicks the **Shutdown** button.
3. Remove the comment **TODO - Add exception handling**, and then locate the **Step 1 - disconnect from the Power Generator** and **Step 2 - Verify the status of the Primary Coolant System** comments. Enclose the code between these comments in a **try/catch** block that catches the **SwitchDevices.PowerGeneratorCommsException** exception. This is the exception that the **DisconnectPowerGenerator** method can throw.
4. In the **catch** block, add code to append a new line of text to the **textBlock1** control with the message "*** Exception in step 1:" and then the contents of the **Message** property of the exception. The **Message** property contains the error message that the **Switch** object specified when it threw the exception.



Hint: To append a line of text to a **TextBlock** control, use the **+=** operator on the **Text** property of the control.

5. Enclose the code between the **Step 2 - Verify the status of the Primary Coolant System** and **Step 3 - Verify the status of the Backup Coolant System** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, following the same pattern as step 3, print a message on a new line in the **textBlock1** control (note that this is step 2, not step 1 of the shutdown process).
6. Enclose the code between the **Step 3 - Verify the status of the Backup Coolant System** and **Step 4 - Record the core temperature prior to shutting**

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

down the reactor comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 3).

7. Enclose the code between the **Step 4 - Record the core temperature prior to shutting down the reactor** and **Step 5 - Insert the control rods into the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 4).

8. Enclose the code between the **Step 5 - Insert the control rods into the reactor** and **Step 6 - Record the core temperature after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.RodClusterReleaseException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 5).

9. Enclose the code between the **Step 6 - Record the core temperature after shutting down the reactor** and **Step 7 - Record the core radiation levels after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 6).

10. Enclose the code between the **Step 7 - Record the core radiation levels after shutting down the reactor** and **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.CoreRadiationLevelReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 7).

11. Enclose the two statements after **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.SignallingException** exception. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 8).

12. Build the solution and correct any errors.

▶ Task 4: Test the application

- Run the application, and then click the **Shutdown** button. Examine the messages displayed in the MainWindow window, and verify that exceptions are now caught and reported.



Note: The **Switch** class randomly generates exceptions as before, so you may not see any exception messages the first time that you click the button. Repeat the process of clicking the button and examining the output until you see exception messages appear.

Exercise 2: Detecting an Exceptional Condition

In this exercise, you will modify a method so that it throws an **ArgumentException** exception if it is invoked with arguments that contain erroneous or invalid data.

Scenario

One of the engineering devices that Fabrikam, Inc. produces performs several calculations that involve matrices. These matrices represent the coordinates of sets of points within the bounds of a multidimensional mesh. The device itself collects the data for these points and constructs the matrices. Then, it uses a C# method to multiply them together to generate a new set of data points. Under normal operations, none of the data items in any of the matrices should be negative. However, sometimes the data that the device captures contains an error—if the device detects a value that is out of range, it generates the value −1 for a data point. Unfortunately, the code that multiplies matrices together fails to detect this condition, and calculates a result that is erroneous. You have been provided with a copy of this code as a method that is embedded in a WPF application.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication solution.
2. Add code to throw exceptions in the **MatrixMultiply** method.
3. Handle the exceptions that the **MatrixMultiply** method throws.
4. Implement test cases and test the application.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the MatrixMultiplication solution

1. In Visual Studio, open the MatrixMultiplication solution in the

   E:\Labfiles\Lab 4\Ex2\Starter folder.

2. Open the Matrix.cs file, and then locate the **MatrixMultiply** method.

   The **MatrixMultiply** method performs the arithmetic to multiply together the two matrices passed as parameters and return the result.

   Currently, the method accepts matrices of any size, and performs no validation of data in the matrices before calculating the results. You will add checks to ensure that the two matrices are compatible (the number of columns in the first matrix is equal to the number of rows in the second matrix), and that no value in either matrix is a negative number.

   If the matrices are not compatible, or either of them contain a negative value, the method must throw an exception.

▶ Task 2: Add code to throw exceptions in the MatrixMultiply method

1. In the **MatrixMultiply** method, locate and remove the comment **TODO – Evaluate input matrices for compatibility**. Below the comment block, add code to perform the following actions:

   a. Compare the number of columns in **matrix1** to the number of rows in **matrix2**.

   b. Throw an **ArgumentException** exception if the values are not equal. The exception message should specify that the number of columns and rows should match.



Hint: You can obtain the number of columns in a matrix by examining the length of the first dimension. You can obtain the number of rows in a matrix by examining the length of the second dimension.

2. Locate and remove the comment **TODO – Evaluate matrix data points for invalid data**. At this point, the method iterates through the data points in each matrix, multiplying the value in each cell in **matrix1** against the value in the corresponding cell in **matrix2**. Add code below the comment block to perform the following actions:

a. Check that the value in the current column and row of **matrix1** is greater than zero. The cell and row variables contain the column and row that you should examine.

b. Throw an **ArgumentException** exception if the value is not greater than zero. The exception should contain the message "Matrix1 contains an invalid entry in cell[*x*, *y*]." where *x* and *y* are the column and row values of the cell.



Hint: Use the **String.Format** method to construct the exception message.

3. Add another block of code to check that the value in the current column and row of **matrix2** is greater than zero. If it is not, throw an **ArgumentException** exception with the message "Matrix2 contains an invalid entry in cell[*x*, *y*].". The column and cell variables contain the column and row that you should examine.

▶ Task 3: Handle the exceptions that the MatrixMultiply method throws

1. Open the MainWindow WPF window in the Design View window and examine the window.

   This window provides the user interface that enables the user to enter the data for the two matrices to be multiplied. The user clicks the **Calculate** button to calculate and display the result.

2. Open the code file for the MainWindow WPF window.
3. In the **MainWindow** class, locate the **ButtonCalculate_Click** method. This method runs when the user clicks the **Calculate** button.
4. In the **ButtonCalculate_Click** method, locate the line of code that invokes the **Matrix**.**MatrixMultiply** method, and enclose this line of code in a **try**/**catch** block that catches an **ArgumentException** exception named **ex**.
5. In the **catch** block, add a statement that displays a message box that contains the contents of the **Message** property of the exception object.



Hint: You can use the **MessageBox.Show** method to display a message box. Specify the message to display as a string passed in as a parameter to this method.

6. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

7. Start the application without debugging.
8. In the MainWindow window, in the first drop-down list box, select **Matrix 1: 2 Columns**, in the second drop-down list box, select **Matrix 1: 2 Rows**, and then in the third drop-down list box, select **Matrix 2: 2 Columns**.

   This creates a pair of 2 × 2 matrices initialized with zeroes.
9. Enter some non-negative values in the cells in both matrices, and then click **Calculate**.

   Verify that the result is calculated and displayed, and that no exceptions occur.
10. Enter one or more negative values in the cells in either matrix, and then click **Calculate** again.

    Verify that the appropriate exception message is displayed, and that it identifies the matrix and cell that is in error.
11. Close the MainWindow window and return to Visual Studio.

The application throws and catches exceptions, so you need to test that the application functions as expected. Although you can test for negative data points by using the application interface, the user interface does not let you create arrays of different dimensions. Therefore, you have been provided with unit test cases that will invoke the **MatrixMultiply** method with data that will cause exceptions. These tests have already been created; you will just run them to verify that your code works as expected.

▶ Task 4: Implement test cases and test the application

1. In the Matrix Unit Test Project, open the **MatrixTest** class, and then examine the **MatrixMultiplyTest1** method.

   The **MatrixMultiplyTest1** method creates four matrices: **matrix1**, **matrix2**, **expected**, and **actual**. The **matrix1** and **matrix2** matrices are the input matrices that are passed to the **MatrixMultiply** method during the test. The **expected** matrix contains the expected result of the matrix multiplication, and the **actual** matrix stores the result of the **MatrixMultiply** method call. The method invokes the **MatrixMultiply** method before using a series of **Assert** statements to verify that the **expected** and **actual** matrices are identical.

   This test method is complete and requires no further work.
2. Examine the **MatrixMultiplyTest2** method.

   This method creates two compatible matrices, but **matrix2** contains a negative value. This should cause the **MatrixMultiply** method to throw an exception.

MCT USE ONLY. STUDENT USE PROHIBITED

The **MatrixMultiplyTest2** method is prefixed with the **ExpectedException** attribute, indicating that the test method expects to cause an **ArgumentException** exception. If the test does not cause this exception, it will fail.

3. Examine the **MatrixMultiplyTest3** method.

   This method creates two incompatible matrices and passes them to the **MatrixMultiply** method, which should throw an **ArgumentException** exception as a result. Again, the method is prefixed with the **ExpectedException** attribute, indicating that the test will fail if this exception is not thrown.

4. Run all tests in the solution, and verify that all tests execute correctly.

Exercise 3: Checking for Numeric Overflow

In this exercise, you will examine what happens by default if an integer calculation causes numeric overflow. You will then modify the application to check for numeric overflow exceptions and repeat the calculation.

Scenario

Part of the software for a measuring device performs integer multiplication, but the integer values used can be very large. You want to ensure that the software does not generate errors that are caused by numeric overflow.

The main tasks for this exercise are as follows:

1. Open the IntegerOverflow solution.
2. Add a **checked** block.
3. Test the application.

▶ Task 1: Open the IntegerOverflow solution

1. Open the IntegerOverflow solution in the E:\Labfiles\Lab 4\Ex3\Starter folder.
2. Run the application, and then click **Multiply**. Observe the result that is displayed and note that it is incorrect.

   The application multiplies 2147483647 by 2, and displays the result **−2**. This is because the multiplication causes an integer numeric overflow. By default, overflow errors of this nature do not cause an exception. However, in many

MCT USE ONLY. STUDENT USE PROHIBITED

situations, it is better to catch the overflow error than to let an application proceed with incorrect data.

3. In Visual Studio, on the **Debug** menu, click **Stop Debugging.**

▶ Task 2: Add a checked block

1. In Solution Explorer, open the MainWindow.xaml.cs file.
2. Locate the **DoMultiply_Click** method

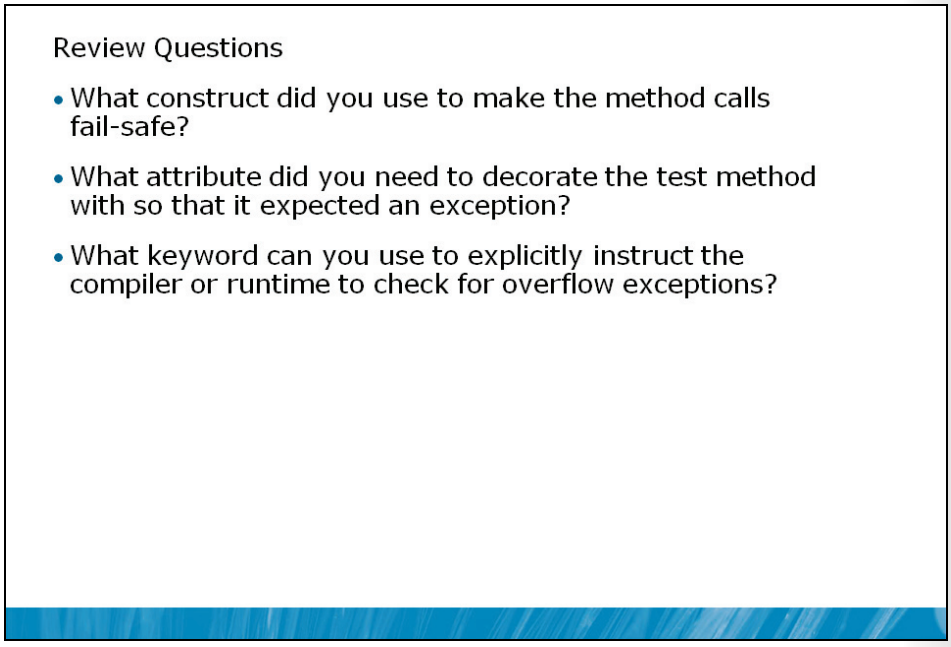
   This method runs when the user clicks the **Multiply** button.
3. Remove the **TODO - Place the multiplication in a checked block** comment. Add a **try/catch** block around the line of code that performs the multiplication operation, and then catch the **OverflowException** exception.
4. Inside the **try** block, add a **checked** block around the line of code that performs the multiplication arithmetic.
5. Build the solution and correct any errors.

▶ Task 3: Test the application

1. Start the application.
2. Click **Multiply**. Verify that the application now displays a message informing you that the arithmetic operation resulted in an overflow.
3. Click **OK**, close the MainWindow window, and then return to Visual Studio.
4. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

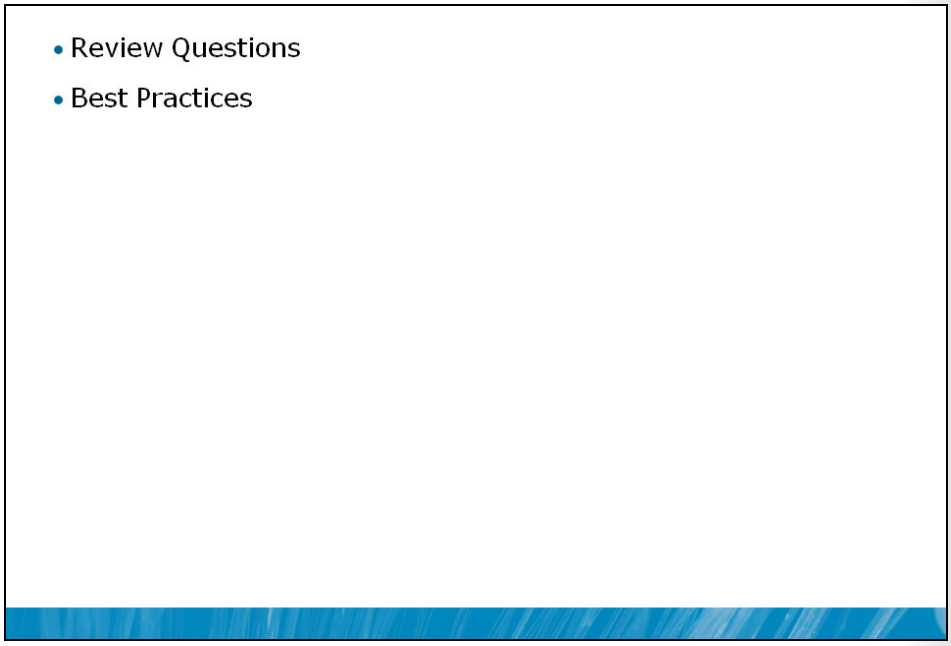


Review Questions

1. What construct did you use to make the method calls fail-safe?
2. What attribute did you need to decorate the test method with so that it expected an exception?
3. What keyword can you use to explicitly instruct the compiler or runtime to check for overflow exceptions?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. In your application, you have a method call that depends on many variables that are out of the control of your application. It is very likely that this method call will throw an exception. You have implemented a centralized exception-handling system so that all exceptions are caught and handled in a single place. When you make the method call, if an exception is thrown, you just want to ensure that you manage and close any resources. Which construct would you use?
2. In your application, you have defined several custom exception classes. You have several **catch** blocks that catch this type of exception. In your **catch** blocks, you want to wrap this type of exception in a more generic exception type. What constructor parameter can you set to ensure that the more specific exception is included in the chain?
3. What should you do with detailed exception messages?

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices Related to Implementing Exception Handling

Supplement or modify the following best practices for your own work situations:

- Always design your applications with errors in mind. Users will always find ways to break your application.
- Design your exception handling in such a way that all exceptions are handled in a centralized location.
- Do not design your application to rely on exceptions to function normally.
- Do not display detailed exception messages to the user because a malicious user could use detailed technical information to make your application malfunction.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 5

Reading and Writing Files

Contents:

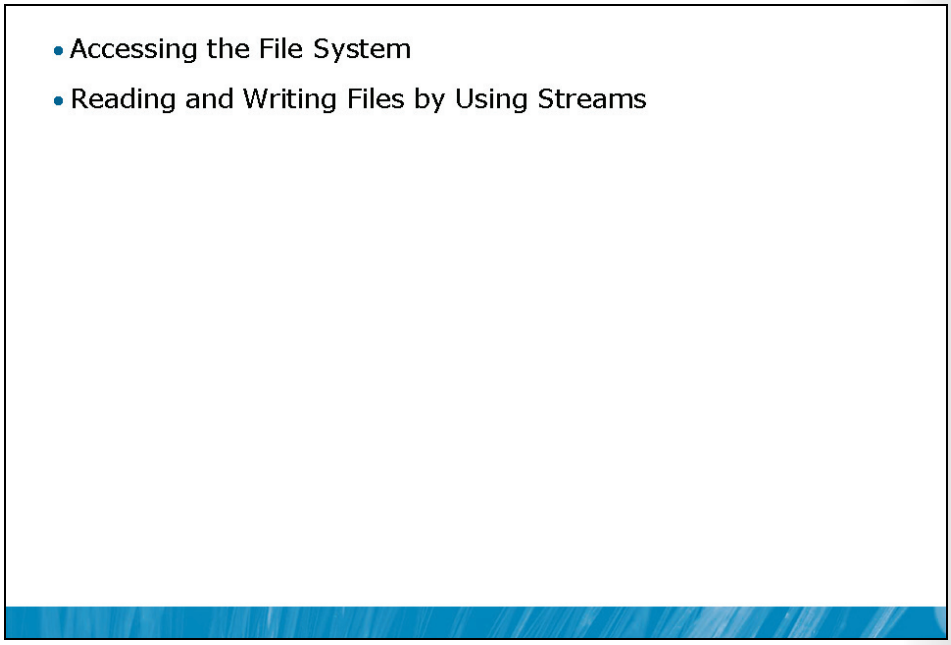
Lesson 1: Accessing the File System 5-3

Lesson 2: Reading and Writing Files by Using Streams 5-27

Lab: Reading and Writing Files 5-45

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



The ability to access and manipulate the files on the file system is a common requirement for many applications. Files provide a means of storage, whether for ordinary text files or for binary data files.

This module shows how to read and write to files by using the classes in the Microsoft® .NET Framework. This module also describes the different approaches that you can take, and how to read and write different formats of data.

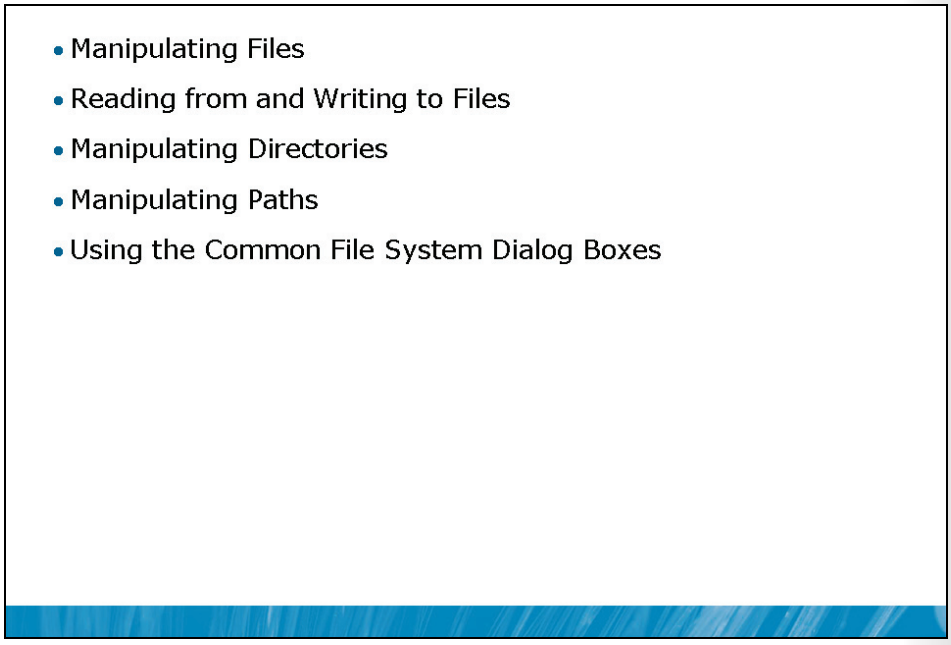
Objectives

After completing this module, you will be able to:

- Describe how to access the file system by using the classes that the .NET Framework provides.
- Describe how to read and write files by using streams.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Accessing the File System



This lesson introduces several classes that provide functionality that an application can use to interact with files and directories.

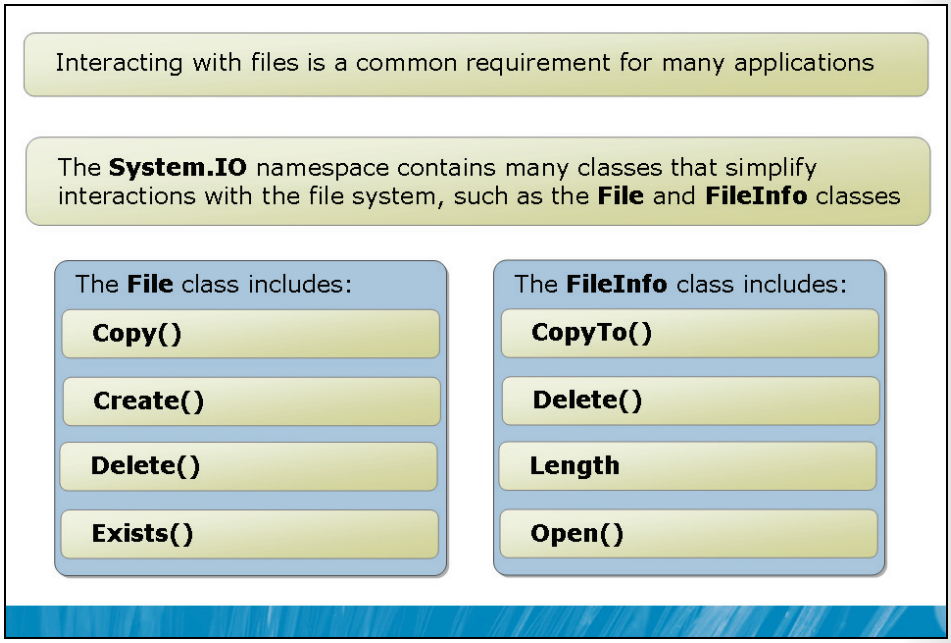
Objectives

After completing this lesson, you will be able to:

- Describe how to control files by using the **File** and **FileInfo** classes.
- Describe how to read from and write to a file by using the **File** class.
- Describe how to manipulate directories by using the **Directory** and **DirectoryInfo** classes.
- Describe how to specify file paths by using the **Path** class.
- Describe how to use the common file system dialog boxes.

MCT USE ONLY. STUDENT USE PROHIBITED

Manipulating Files



Key Points

A common requirement for many applications is the ability to interact with files that are stored on the file system. This can involve creating a new file, copying or deleting a file, or moving a file from one directory to the next. To help simplify these interactions, the .NET Framework provides several classes in the **System.IO** namespace. These include the **File** and **FileInfo** classes.

The File Class

The **File** class is a utility class that wraps various file-related functions. These functions are exposed through static methods. The following table describes some of the key methods that the **File** class provides and shows some code examples.

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example

AppendAllText	Enables you to open an existing file, append text to that file, and then close the file, all in a single operation.	`string filePath = "...";`
		`string fileContents = "...";`
		`File.AppendAllText(filePath, fileContents);`

Copy	Enables you to copy an existing file to a new location.	`string sourceFile = "...";`
		`string destFile = "...";`
		`bool overwrite = false;`
		`File.Copy(sourceFile, destFile, overwrite);`

Create	Enables you to create a new file on the Windows® file system. The **Create** method returns a **FileStream** object that enables you to interact with the file by using the streaming model. Streams are covered in the next lesson.	`string filePath = "...";`
		`int bufferSize = 128;`
		`FileStream file = File.Create(filePath, bufferSize, FileOptions.None);`

Delete	Enables you to delete a file from the Windows file system.	`string filePath = "...";` `File.Delete(filePath);`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example

Exists	Enables you to determine whether a file exists.	`string filePath = "...";`
		`bool exists =`
		`  File.Exists(`
		`  filePath);`

GetCreationTime	Enables you to get the creation time of a file.	`string filePath =`
		`  "...";`
		`DateTime time =`
		`  File.`
		`  GetCreationTime(`
		`  filePath);`

GetLastAccessTime	Enables you to get the last access time of a file.	`string filePath =`
		`"...";`
		`DateTime time =`
		`  File.`
		`  GetLastAccessTime(`
		`  filePath);`

Move	Enables you to move a file to a new location. You can also use this method to rename a file.	`string sourceFile =`
		`  "...";`
		`string destFile = "...";`
		`File.Move(sourceFile,`
		`  destFile);`

ReadAllText	Enables you to read all of the text from a file into a string variable.	`string filePath = "...";`
		`string fileContents =`
		`  File.ReadAllText(`
		`  filePath);`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example
SetCreationTime	Enables you to set the creation time of a file.	`string filePath = "...";` `File.SetCreationTime(filePath, DateTime.Now);`
SetLastAccessTime	Enables you to set the last access time of a file.	`string filePath = "...";` `File.SetLastAccessTime(filePath, DateTime.Now);`

WriteAllText	Enables you to create a new file, write text to that file, and then close the file, all in a single operation.	`string filePath = "...";`
		`string fileContents = "...";`
		`File.WriteAllText(filePath, fileContents);`

The FileInfo Class

The **FileInfo** class provides several properties and instance methods that enable you to create, copy, and move files and process the contents of files.

When you create an instance of the **FileInfo** class, you specify the path to a file on the file system. The following code example shows how to create a new **FileInfo** object for controlling the myFile.txt file in the C:\Temp folder.

```
string filePath = @"C:\Temp\myFile.txt";
FileInfo file = new FileInfo(filePath);
```

MCT USE ONLY. STUDENT USE PROHIBITED

You can then use the **FileInfo** object as a wrapper for the file, which exposes various data and functions through properties and methods. You can also use the **FileInfo** class to create new files. The following table describes some of the key properties and methods and provides some code examples.

Member	Description	Code example

CreationTime (property)	Enables you to get or set the creation time for a particular file.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`file.CreationTime =`
		`  DateTime.Now;`
		
		`...`
		`DateTime time =`
		`  file.CreationTime;`

CopyTo (method)	Enables you to copy the file to a new location on the file system.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`string destPath = "...";`
		`file.CopyTo(destPath);`

Delete (method)	Enables you to delete a file.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`file.Delete();`

DirectoryName (property)	Enables you to get the directory path to the file.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`string dirPath =`
		`  file.DirectoryName`

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Description	Code example

Exists (property)	Enables you to determine whether the file exists.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`bool exists =`
		`file.Exists;`

Extension (property)	Enables you to get the extension of the file.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`string ext =`
		`file.Extension;`

Length (property)	Enables you to get the length of the file in bytes.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`long length =`
		`  file.Length;`

Name (property)	Enables you to get the name of the file.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`string name = file.Name;`

Open (method)	Enables you to open a file on the Windows file system. The **Open** method returns a **FileStream** object that enables you to interact with the file by using the streaming model. Streams are covered in the next lesson.	`string filePath = "...";`
		`FileInfo file = new`
		`  FileInfo(filePath);`
		`FileStream stream =`
		`  file.Open(`
		`  FileMode.OpenOrCreate);`

MCT USE ONLY. STUDENT USE PROHIBITED

Question: In your application, you use files as a temporary storage mechanism while the application is running. When the application stops running, you want to make sure that the file exists, and then delete the file. What is the easiest way to achieve this?

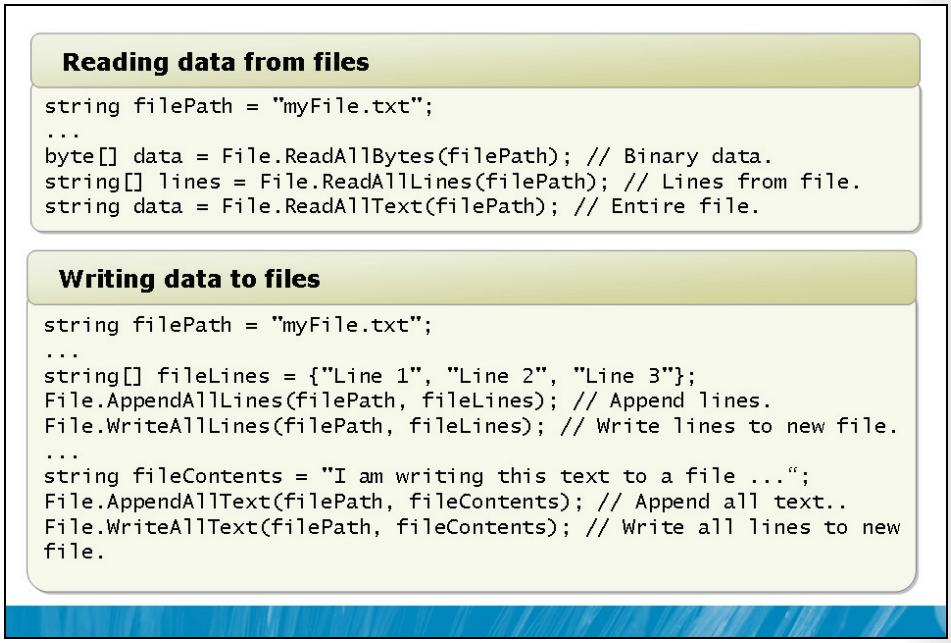
Additional Reading

For more information about the **File** class, see the File Class page at http://go.microsoft.com/fwlink/?LinkId=192915.

For more information about the **FileInfo** class, see the FileInfo Class page at http://go.microsoft.com/fwlink/?LinkId=192916.

MCT USE ONLY. STUDENT USE PROHIBITED

Reading from and Writing to Files



Key Points

The **File** and **FileInfo** classes provide several methods that you can use to read from and write to a file.

The **File** class contains static methods that you can use to perform atomic operations for direct reading from and writing to files. These methods are atomic because they wrap several underlying functions into a single method call. For example, the **AppendAllLines** method wraps operations to acquire the file handle, open a stream to the file, write data to the file, and then release the file handle.

The **FileInfo** class contains instance methods that, when reading from and writing to files, rely on the **FileStream** and **StreamReader** classes. The use of streams is covered in Lesson 2: Reading and Writing Files by Using Streams. This topic focuses on the static methods provided by the **File** class that do not use streams, but provide single atomic operations.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Reading from Files

When you use the **File** class to read data from a file, there are many alternative methods that you can use, each offering different behavior. The following list describes some of these methods:

- 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 myFile.txt file into a byte array called **data**.

```
string filePath = "myFile.txt";

byte[] data = File.ReadAllBytes(filePath);
```

- The **ReadAllLines** method enables you to read a text file from start to finish, line by line, and store each line in a string array. The following code example shows how to read the contents of the myFile.txt file and store each line in the string array called **lines**.

```
string filePath = "myFile.txt";

string[] lines = File.ReadAllLines(filePath);
```

- The **ReadAllText** method enables you to read a file from start to finish, and store the data from the file in a string variable. The following code example shows how to read the contents of the myFile.txt file and the data in a string called **data**.

```
string filePath = "myFile.txt";

string data = File.ReadAllText(filePath);
```

Writing to Files

When you write data to a file by using the **File** class, several options are available, depending on the type of data that you want to write. With each option, you can either append the data to an existing file, or create a new file and then perform the write operation.

The following list describes some of these methods:

- The **AppendAllLines** method enables you to write the contents of a string array to a text file. If the path that you specify does not exist, the operation will create a new file. The following code example shows how to write the contents of a string array called **fileLines** to the myFile.txt file.

```
string filePath = "myFile.txt";

string[] fileLines = {"Line 1", "Line 2", "Line 3"};

File.AppendAllLines(filePath, fileLines);
```

- The **AppendAllText** method enables you to write the contents of a string variable to a text file. Similar to the **AppendAllLines** method, if the file does not exist, the operation will create the file, and then perform the write operation. The following code example shows how to write the contents of a string variable called fileContents to the myFile.txt file.

```
string filePath = "myFile.txt";

string fileContents = "I am writing this text to a file called
    myFile.txt";

File.AppendAllText(filePath, fileContents);
```

- The **WriteAllBytes** method enables you to write the contents of a byte array to a binary file. If the file already exists, this operation will overwrite the file. The following code example shows how to write the contents of a byte array called **fileBytes** to a new file called myFile.txt.

```
string filePath = "myFile.txt";

byte[] fileBytes = {12, 134, 12, 8, 32};

File.WriteAllBytes(filePath, fileBytes);
```

- The **WriteAllLines** method behaves in a similar way to the **AppendAllLines** method in that you can write the contents of a string array to a text file. The main difference is that, if the file exists, the file will be overwritten. If the file does not exist, a new file will be created. The following code example shows how to write the contents of a string array called **fileLines** to a new file called myFile.txt.

```
string filePath = "myFile.txt";

string[] fileLines = { "Line 1", "Line 2", "Line 3" };

File.WriteAllLines(filePath, fileLines);
```

- The WriteAllText method behaves in a similar way to the AppendAllText method in that you can write the contents of a string variable to a text file. The

MCT USE ONLY. STUDENT USE PROHIBITED

main difference is that, if the file exists, the file will be overwritten. If the file does not exist, a new file will be created. The following code example shows how to write the contents of a string variable called fileContents to a new file called myFile.txt.

```
string filePath = "myFile.txt";

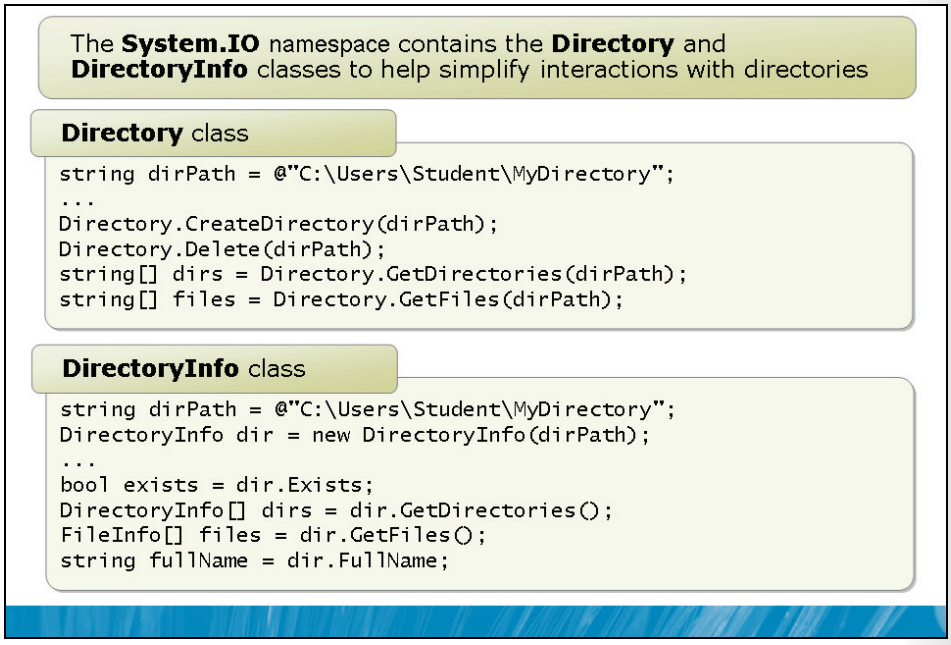
string fileContents = "I am writing this text to a file called
    myFile.txt";

File.WriteAllText(filePath, fileContents);
```

Question: In your application, you have just added some logic to handle exceptions. You now want to extend this logic further to store details of these exceptions to a log file on the file system so that you can diagnose any problems. You will be writing a string variable and you should want to never overwrite any existing log records in a file. Which method would you use?

MCT USE ONLY. STUDENT USE PROHIBITED

Manipulating Directories



Key Points

Files are stored in directories and folders. The .NET Framework provides a pair of classes that are similar to the **File** and **FileInfo** classes that enable you to query and manage directories. Whether you want to create a new directory, delete an existing directory, or enumerate the contents of a directory, you can achieve this by using the **Directory** and **DirectoryInfo** classes in the **System.IO** namespace.

The Directory Class

Similar to the **File** class, the **Directory** class is a utility class that provides various operations that enable you to manage folders and directories. The **Directory** class exposes its functionality through static methods. The following table describes some of the methods and provides some code examples.

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example

CreateDirectory	Enables you to create all of the directories that are specified in the path that don't already exist.	`string dirPath =`
		`@"C:\NewFolder\SubFolder";`
		`Directory.CreateDirectory(`
		`  dirPath);`

DeleteDirectory	Enables you to delete one or more directories from the file system.	`string dirPath =`
		`  @"C:\Users\Student\"`
		`  + "MyDirectory";`
		
		`bool deleteSubFolders = true;`
		
		`Directory.Delete(`
		`  dirPath,`
		`  deleteSubFolders);`

GetDirectories	Enables you to get all of the subdirectories in the specified path.	`string dirPath = "...";`
		`string[] dirs =`
		`  Directory.GetDirectories(`
		`  dirPath);`

GetFiles	Enables you to get all of the files in the specified path.	`string dirPath = "...";`
		`string[] files =`
		`  Directory.GetFiles(`
		`  dirPath);`

Exists	Enables you to determine whether a directory exists at the specified path.	`string dirPath = "...";`
		`bool dirExists =`
		`  Directory.Exists(`
		`  dirPath);`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example

Move	Enables you to move a directory. You cannot use the **Move** method to move directories to different drives.	`string sourcePath = "...";`
		`string destPath = "...";`
		`Directory.Move(`
		`  sourcePath, destPath);`

The DirectoryInfo Class

The **DirectoryInfo** class provides several properties and instance methods that enable you to work with directories. Similar to the **FileInfo** class, when you create an instance of the **DirectoryInfo** class, you typically specify the path to a directory on the file system. The following code example shows how to create an instance of the **DirectoryInfo** class.

```
string dirPath =
    @"C:\Users\Student\Music\";

DirectoryInfo dir = new DirectoryInfo(dirPath);
```

You can then use the **DirectoryInfo** object as a wrapper for the directory that exposes various data and functions through properties and methods. You can also use the **DirectoryInfo** class to create a new directory. For example, the following code example shows how you could determine whether the directory exists, and if it does not exist, how you could create the directory.

```
string dirPath =
    @"C:\Users\Student\Music\";

DirectoryInfo dir = new DirectoryInfo(dirPath);

if (!dir.Exists)
{
    dir.Create();
}
```

The following table describes some of the key properties and methods and provides some code examples.

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Description	Code example

Create (method)	Enables you to create the directories in the path specified. If the directory already exists, it is ignored.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`dir.Create();`

Delete (method)	Enables you to delete several directories. If the directory cannot be found, a **DirectoryNotFoundException** exception is thrown.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`dir.Delete();`

Exists (property)	Enables you to determine whether a directory exists at the specified path.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`bool exists = dir.Exists;`

FullName (property)	Enables you to get the full path of the directory.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		
		`string fullName = dir.FullName;`

GetDirectories (method)	Enables you to get all of the subdirectories in the specified path. This method returns a **DirectoryInfo** array, which enables you to use each of the **DirectoryInfo** members on all subdirectories.	`string dirPath = "...";`
		
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		
		`DirectoryInfo[] dirs =`
		`  dir.GetDirectories();`

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Description	Code example

GetFiles (method)	Enables you to get all of the files in the specified path. This method returns a **FileInfo** array, which enables you to use each of the **FileInfo** members on all of the files in the directory.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`FileInfo[] files =`
		`  dir.GetFiles();`

MoveTo (method)	Enables you to move a directory. You cannot use the **MoveTo** method to move directories to different drives.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`string destPath = "...";`
		`dir.MoveTo(destPath);`

Name (property)	Enables you to get the name of the directory.	`string dirPath = "...";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`string dirName =`
		`  dir.Name;`

Parent (property)	Enables you to get the parent directory.	`string dirPath =`
		`  @"C:\Users\Student\Music\";`
		`DirectoryInfo dir = new`
		`  DirectoryInfo(dirPath);`
		`DirectoryInfo parentDir =`
		`  dir.Parent;`

MCT USE ONLY. STUDENT USE PROHIBITED

Enumerating Directory Contents

The following code example shows how you can enumerate a directory and display details of all subdirectories and the files that they contain.

```
string dirPath =
     @"C:\Users\Student\Documents";

// Get all sub directories in the Documents directory.
string[] subDirs = Directory.GetDirectories(dirPath);

foreach (string dir in subDirs)
{
    // Display the directory name.
    Console.WriteLine("{0} contains the following files:", dir);

    // Get all the files in each directory.
    string[] files = Directory.GetFiles(dir);

    foreach (string file in files)
    {
        // Display the file name.
        Console.WriteLine(file);
    }
}
```

Question: What class would you use to retrieve an instance of a directory in the file system, which you can then interact with?

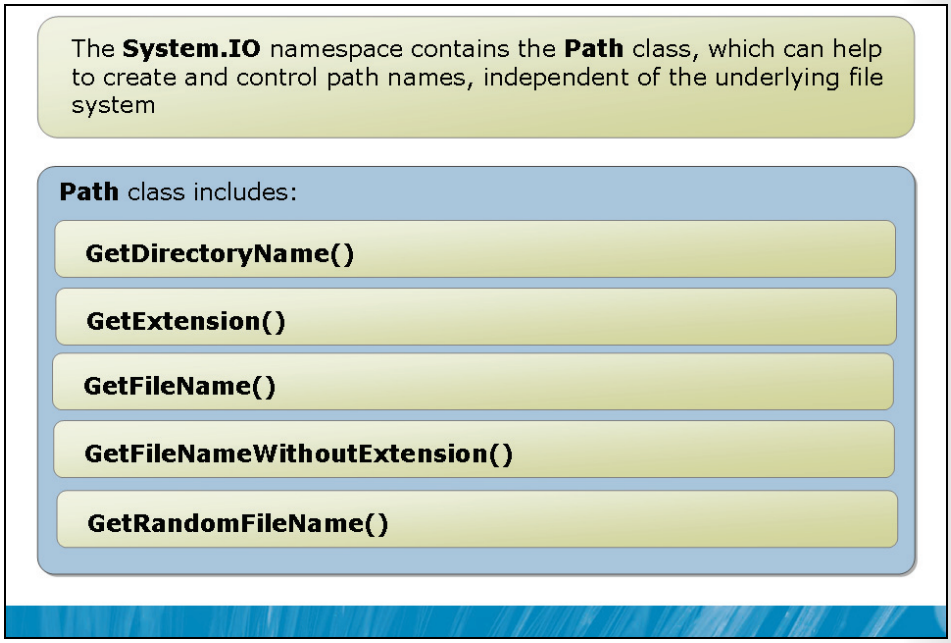
Additional Reading

For more information about the **Directory** class, see the Directory Class page at http://go.microsoft.com/fwlink/?LinkId=192917.

For more information about the **DirectoryInfo** class, see the DirectoryInfo Class page at http://go.microsoft.com/fwlink/?LinkId=192918.

MCT USE ONLY. STUDENT USE PROHIBITED

Manipulating Paths



Key Points

Files are held in folders. All files and folders have a name. The combination of the name of a file and the folder where it is located constitute the path to that file. Different file systems can have different conventions and rules for what constitutes a legal file and path name. The **Path** class provides methods that you can use to parse and construct legal file and folder names for a specified file system.

The Path Class

The **Path** class exposes its functionality through various static methods. The following table describes some of the methods and provides some code examples.

Method	Description	Code example

GetDirectory Name	Enables you to get all of the directories in the path.	`string path = @"C:\Temp\SubFolder\MyFile.txt";`
		`string dirs =`
		`  Path.GetDirectoryName(path);`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Code example

GetExtension	Enables you to get the extension of the specified file.	`string path = @"C:\Temp\SubFolder\MyFile.txt";`
		`string ext =`
		`Path.GetExtension(path);`

GetFileName	Enables you to get the file name including the extension from the specified path.	`string path = @"C:\Temp\SubFolder\MyFile.txt";`
		`string fileName =`
		`Path.GetFileName(path);`

GetFileName Without Extension	Enables you to get the file name without the extension from the specified path.	`string path = @"C:\Temp\SubFolder\MyFile.txt";`
		`string fileName =`
		`Path.`
		`GetFileNameWithoutExtension(`
		`path);`

GetRandom FileName	Enables you to generate a random folder or file name.	`string fileName =` `Path.GetRandomFileName();`
GetTempFile Name	Enables you to create a new temp file in your local Windows temp folder. This method then returns the absolute path to that file.	`string tempFilePath =` `Path.GetTempFileName();`
GetTempPath	Enables you to get the path to the local Windows temp folder.	`string tempPath =` `Path.GetTempPath();`

MCT USE ONLY. STUDENT USE PROHIBITED

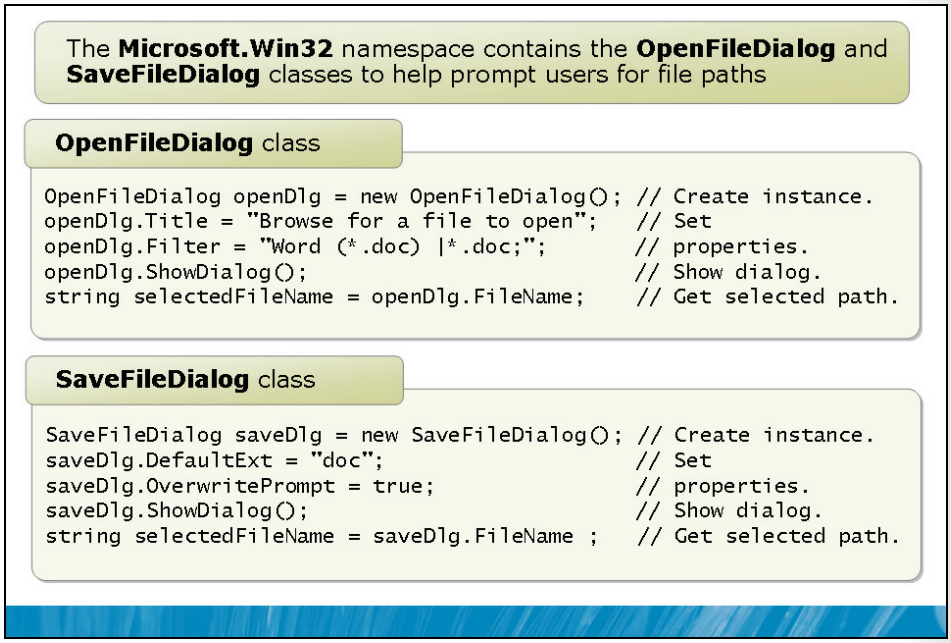
Question: You are creating a filter that enables users to browse files by extension. To start with, you need to get the extensions of each file and then run some logic depending on the result. You also want to display the file name including the extension in a list. Which methods would you use to query the files?

Additional Reading

For more information about the **Path** class, see the Path Class page at http://go.microsoft.com/fwlink/?LinkId=192919.

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Common File System Dialog Boxes



Key Points

When you are building an application with a graphical user interface, it is unreasonable to expect users to type long, unwieldy path and file names. Users expect the ability to browse to files and directories through dialog boxes.

Creating a dialog box, such as an open or save file dialog box found in any Microsoft application, would take a considerable amount of development and test effort. Fortunately, the .NET Framework provides the **OpenFileDialog** and **SaveFileDialog** classes in the **Microsoft.Win32** namespace.



Note: You can also find the **OpenFileDialog** and **SaveFileDialog** classes in the **System.Windows.Forms** namespace. Before the introduction of Windows Presentation Foundation (WPF), Windows Forms used to be the primary technology for implementing Windows-based client applications in the .NET Framework, hence the inclusion in the **System.Windows.Forms** namespace.

Both the **OpenFileDialog** and **SaveFileDialog** classes provide the functionality to enable the user to browse for a file or specify a file name and create any folders that

MCT USE ONLY. STUDENT USE PROHIBITED

are required. The functionality is made accessible through various properties and methods, which you can use to customize the behavior of the dialog boxes to your requirements. However, note that neither dialog box actually opens or saves the specified file; all they do is construct a path and file name that your application can use to open or save the file.

The following table describes some of the key properties that are common to both the **OpenFileDialog** and **SaveFileDialog** classes.

Property	Description
CheckFileExists	Enables you to instruct the dialog box to display a warning if the user specifies a file that does not exist.
FileName	Enables you to get or set the path to the file that is selected in the dialog box.
Filter	Enables you to restrict the type of files that the user can select from the dialog box.
InitialDirectory	Enables you to get or set the default directory that is displayed when the dialog box is first shown.
Title	Enables you to specify a title for the dialog box.

Using the OpenFileDialog and SaveFileDialog Classes

You can use the **OpenFileDialog** and **SaveFileDialog** classes in the same way that you would use any other .NET Framework class. The first step is to create an instance of the class, as the following code example shows.

```
OpenFileDialog openDlg = new OpenFileDialog();
...
SaveFileDialog saveDlg = new SaveFileDialog();
```

After you have created an instance of either dialog class, you can use their properties to customize their behavior. Most properties that are exposed through both classes are the same, but there are some exceptions such as the **Multiselect** property in the **OpenFileDialog** class, and the **OverwritePrompt** property in the **SaveFileDialog** class, as the following code example shows.

```
...
openDlg.Title = "Browse for a file to open";
openDlg.Multiselect = false;
openDlg.InitialDirectory = @"C:\Users\Student\Documents";
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
openDlg.Filter = "Word (*.doc) |*.doc;";
...
saveDlg.Title = "Browse for a save location";
saveDlg.DefaultExt = "doc";
saveDlg.AddExtension = true;
saveDlg.InitialDirectory = @"C:\Users\Student\Documents";
saveDlg.OverwritePrompt = true;
```

For the dialog boxes to appear when your application is running, you need to call the **ShowDialog** method, as the following code example shows.

```
...
openDlg.ShowDialog();
...
saveDlg.ShowDialog();
```

Finally, to get the paths that the user selected, query the **FileName** property, as the following code example shows.

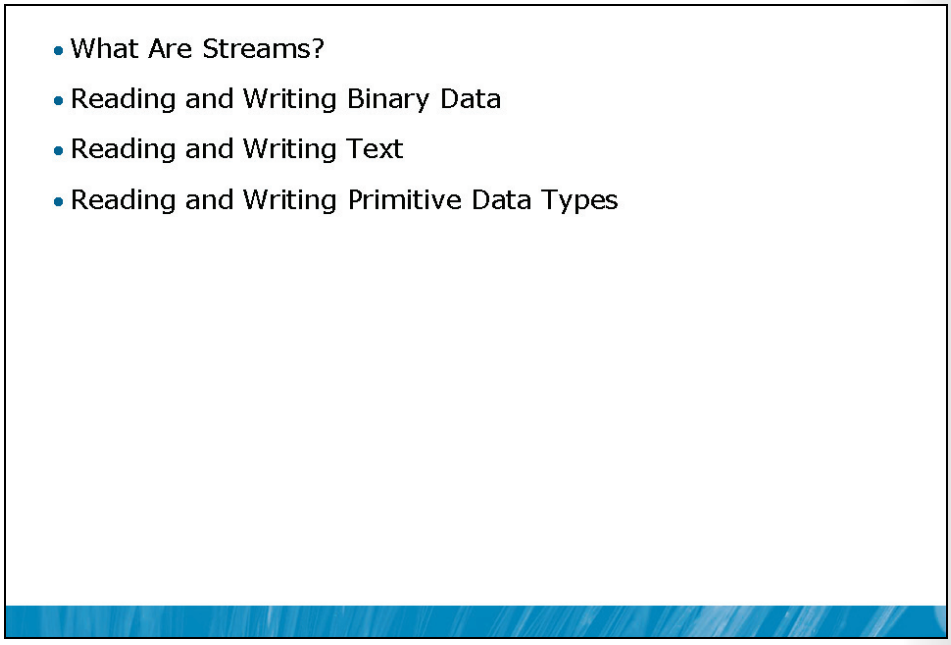
```
...
string selectedFileName = openDlg.FileName;
...
string selectedFileName = saveDlg.FileName;
```

Depending on whether the user selected a file, or just closed the dialog box, the value that is returned from the **FileName** property may be a valid absolute path, or an empty string. Therefore, you should perform some validation at this point before using the result.

Question: You have almost completed your implementation of a text editor, and the final step is to get users to browse to a save location, and prompt them for a file name. What class would you use and how would you use it?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Reading and Writing Files by Using Streams



Reading and writing data in single atomic operations as described in the previous lesson is acceptable with small amounts of data. However, when you are working with large data volumes, such operations are inefficient and can consume too many resources.

An alternative approach is to use streams. This lesson introduces the .NET Framework streaming model, and the classes that you can use to implement streaming in your applications.

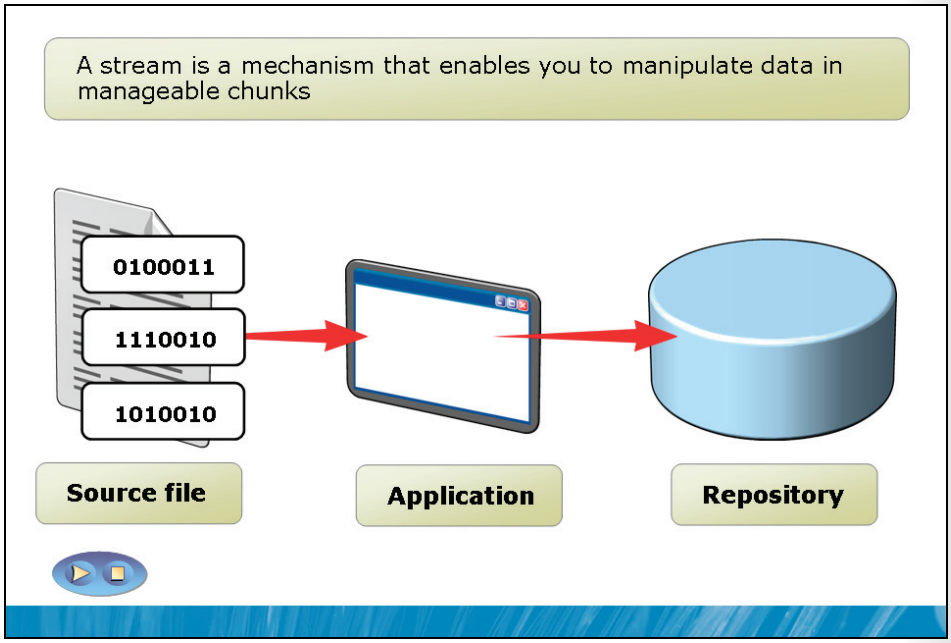
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of streams.
- Describe how to read and write binary data.
- Describe how to read and write text.
- Describe how to read and write primitive data types.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Streams?



Key Points

When you work with data, whether the data is stored in a file on the file system or on a Web server that is accessible over an HTTPS connection, the data sometimes becomes too large to load into memory and transmit in a single atomic operation. For example, imagine trying to load a 100-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.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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; it 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 to 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 source. 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.



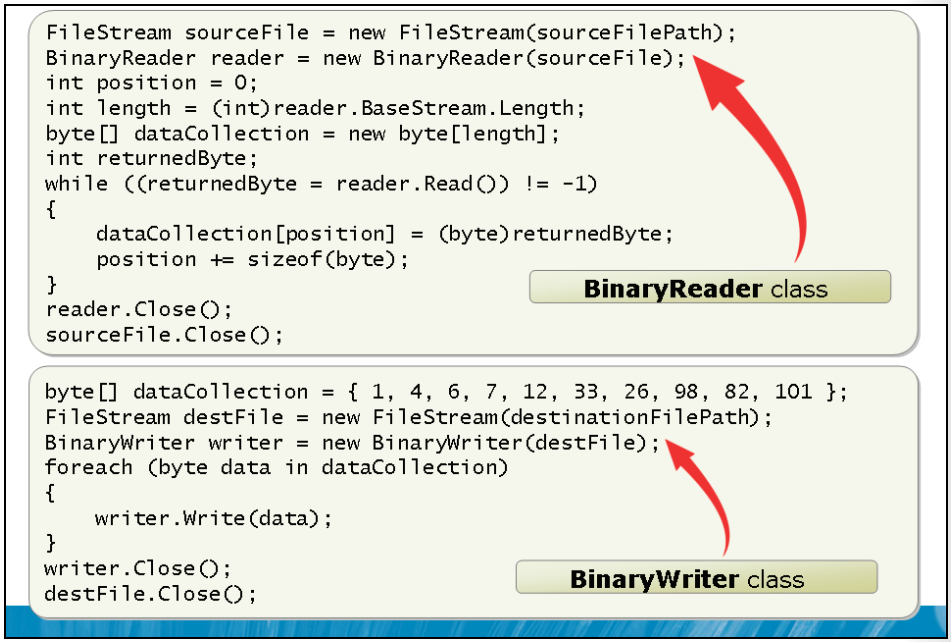
Note: The remaining topics in this lesson focus on reading data from and writing data to files on the file system, so will use the **FileStream** class. However, these topics involve reading and writing a variety of data, so will be using classes such as **BinaryReader**, **BinaryWriter**, **StreamReader**, and **StreamWriter**. For more information about the **FileStream** class, see the content for this topic on the Course Companion CD.

Question: What do you think are the benefits of streaming data?

Additional Reading

For more information about the **FileStream** class, see the FileStream Class page at http://go.microsoft.com/fwlink/?LinkId=192920.

Reading and Writing Binary Data



Key Points

A stream that is established by using a **FileStream** object is just a raw sequence of bytes. If a file contains structured data, you must convert the byte sequence into the appropriate types. This can be a time-consuming, error-prone task. However, the .NET Framework class library contains other classes that you can use to read and write textual data and primitive types in a stream that you have opened by using a **FileStream** object. These classes include **StreamReader**, **StreamWriter**, **BinaryReader**, and **BinaryWriter**.

The BinaryReader and BinaryWriter Classes

Many applications store data in raw binary form because writing binary is fast, it takes up less space on disk, and because it is not human readable. You can take advantage of using the binary format in your .NET Framework applications by using the **BinaryReader** and **BinaryWriter** classes.

You construct a **BinaryReader** or **BinaryWriter** object by providing a stream that that is connected to the source of the data that you want to read or write. The following code example shows how to initialize the **BinaryReader** and **BinaryWriter** classes, passing a **FileStream** object.

MCT USE ONLY. STUDENT USE PROHIBITED

```
string filePath = "...";
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 table describes some of the key members.



Important: When you have finished using a **StreamReader** or **StreamWriter** object, you must call the **Close** method to flush the stream and release any resources that are associated with the stream. You must also close the **FileStream** object that is providing the data for the **StreamReader** and **StreamWriter** objects.

Member	Description
BaseStream (property)	Enables you to access the underlying stream that the **BinaryReader** object uses.
Close (method)	Enables you to close the **BinaryReader** object and the underlying stream.
Read (method)	Enables you to read the number of remaining bytes in the stream from a particular index.
ReadByte (method)	Enables you to read the next byte from the stream, and advance the stream to the next byte.
ReadBytes (method)	Enables you to read a specified number of bytes into a byte array.



Note: The **BinaryReader** class contains a further 16 methods that can read a binary stream and convert the data into the various primitive data types that are available with C#. These methods will be discussed in more detail later in this lesson.

Similarly, the **BinaryWriter** object exposes various members to enable you to write data to an underlying stream. The following table describes some of the key members.

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Description
BaseStream (property)	Enables you to access the underlying stream that the **BinaryWriter** object uses.
Close (method)	Enables you to close the **BinaryWriter** object and the underlying stream. Any data in the buffer will be flushed to the underlying stream.
Flush (method)	Enables you to explicitly flush any data in the current buffer to the underlying stream.
Seek (method)	Enables you to set your position in the current stream, thus writing to a specific byte.
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.

```
// Source file path.
string sourceFilePath =
    @"C:\Users\Student\Documents\BinaryDataFile.bin";

// 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 BinaryWriter 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;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
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();
```



Note: If a file read or file write operation throws an exception, you need to ensure that streams and file handles are released. You can use the **try finally** block to ensure that resources are released. Typically, you should place the logic that performs the read or write in the **try** block, and place any logic that closes streams and releases file handles in the **finally** block.

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.

```
string destinationFilePath =
@"C:\Users\Student\Documents\BinaryDataFile.bin";


// 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);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
// 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();
```

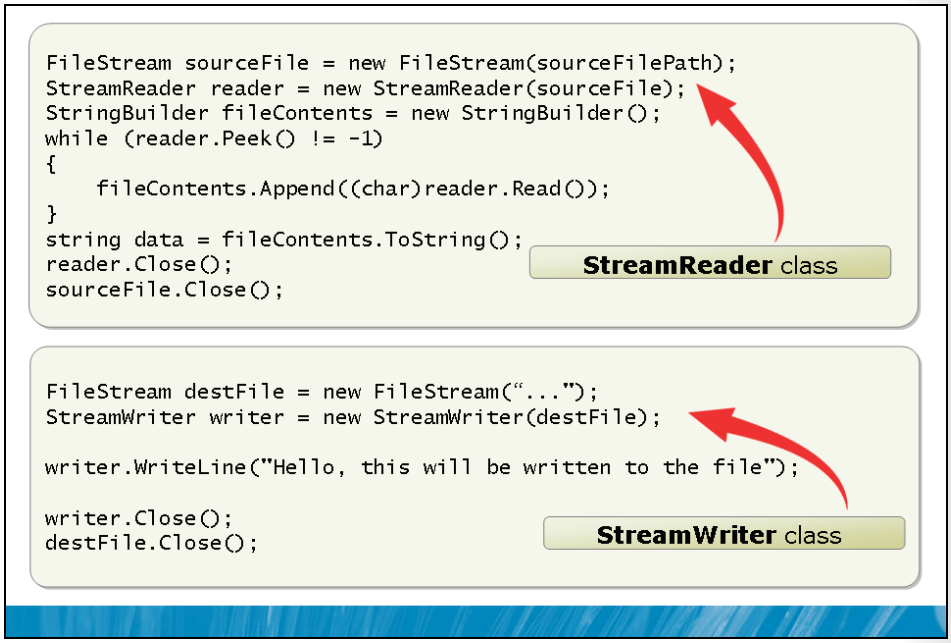
The above code produces a file with the following contents.

```
00000000  01 04 06 07 0C 21 1A 62  52 65
```

Question: Why is it important to close streams when you have finished using them?

MCT USE ONLY. STUDENT USE PROHIBITED

Reading and Writing Text



Key Points

In addition to storing data in raw binary form, you can also store data as plain text. The process for reading and writing textual data to a file is very similar to reading and writing binary data, except that you use the **StreamReader** and **StreamWriter** classes.

Note: The **Console** class that you can use for reading from and writing to the console contains a **StreamReader** property called **In** and a **StreamWriter** property called **Out**. The **Console.ReadLine** method reads text data from the stream that the **In** property identifies, and the **Console.WriteLine** method writes text data to the stream that the **Out** property identifies.

The StreamReader and StreamWriter Classes

Similar to using the **BinaryReader** and **BinaryWriter** classes, when you initialize the **StreamReader** or **StreamWriter** classes, you must provide a stream object to handle the interaction with the data source, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
string destinationFilePath = "...";
FileStream file = new FileStream(destinationFilePath);
...
StreamReader reader = new StreamReader(file);
...
StreamWriter writer = new StreamWriter(file);
```

The following table describes some of the key members that the **StreamReader** class provides to enable you to read text from an underlying stream.

Member	Description
Close (method)	Enables you to close the **StreamReader** object and the underlying stream.
EndOfStream (property)	Enables you to determine whether you have reached the end of the stream.
Peek (method)	Enables you to get the next available character in the stream, but does not consume it.
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.
ReadBlock (method)	Enables you to read an entire block of characters from a specific index from the stream.
ReadLine (method)	Enables you to read an entire line of characters from the stream.
ReadToEnd (method)	Enables you to read all characters from the current position in the stream.

The following table shows some of the key members that the **StreamWriter** class provides to enable you to write text to a stream.

Member	Description
AutoFlush (property)	Enables you to instruct the **StreamWriter** object to flush data to the underlying stream after every write call.
Close (method)	Enables you to close the **StreamWriter** object and the underlying stream.

MCT USE ONLY. STUDENT USE PROHIBITED

Member	Description
Flush (method)	Enables you to explicitly flush any data in the current buffer to the underlying stream.
NewLine (property)	Enables you to get or set the characters that are used for new line breaks.
Write (method)	Enables you to write your data to the stream, and advance the stream.
WriteLine (method)	Enables you to write your data to the stream followed by a new line break, and then advance the stream.



Note: The **Write** and **WriteLine** methods each provide several overloads that enable you to write various types of data, other than text.

Reading Text

The following code example shows how to use the **StreamReader** and **FileStream** classes to read a text file. This example uses the **Peek** and **Read** methods to manually get each character in the file.

```
string sourceFilePath =
    @"C:\Users\Student\Documents\TextDataFile.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());
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
// 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();
```

The following code example provides an alternative approach to manually retrieving each character from the stream, by using the **ReadToEnd** method.

```
string sourceFilePath =
    @"C:\Users\Student\Documents\TextDataFile.txt";

string data;

// 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);

// Read the entire file into a single string variable.

data = reader.ReadToEnd();

// Always close the underlying streams release any file handles.

reader.Close();
sourceFile.Close();
```

Writing 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.

```
string destinationFilePath =
    @"C:\Users\Student\Documents\TextDataFile.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(
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    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();
```

Question: You want to write a series of strings to a text file, and add a line break after each string. What is the easiest way to achieve this?

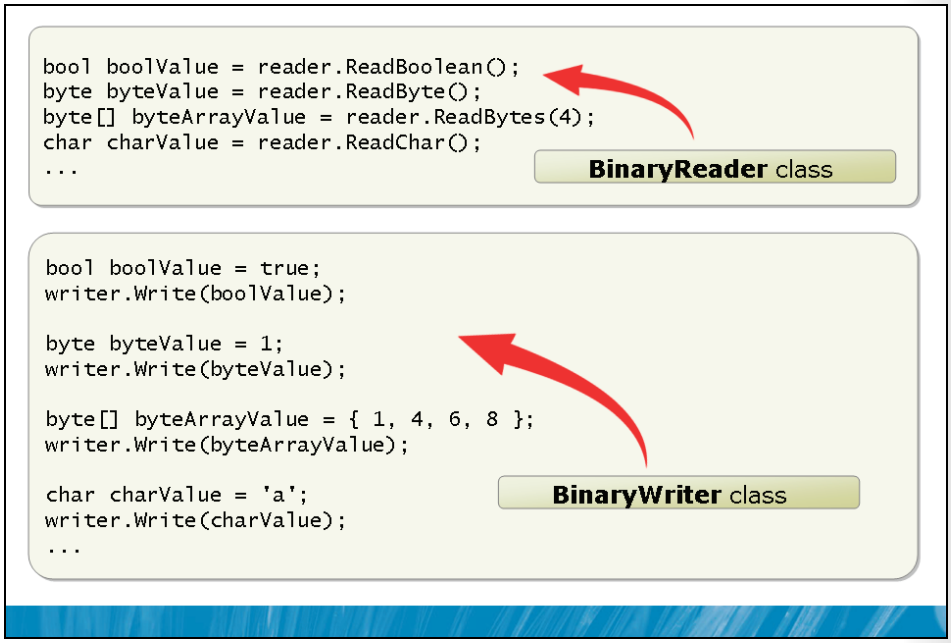
Additional Reading

For more information about the **StreamWriter** class, see the StreamWriter Class page at http://go.microsoft.com/fwlink/?LinkId=192921.

For more information about the **StreamReader** class, see the StreamReader Class page at http://go.microsoft.com/fwlink/?LinkId=192922.

MCT USE ONLY. STUDENT USE PROHIBITED

Reading and Writing Primitive Data Types



Key Points

When you use the **BinaryReader** and **BinaryWriter** classes, you are not restricted to using unstructured byte arrays. These classes also provide methods that enable you to read and write any data into any primitive data type, which includes integers, doubles, Booleans, and strings.

Note: The streaming model that the .NET Framework implements also supports streaming of nonprimitive types such as classes and structures that you define. These types must be serializable, and you use a formatter such as a **BinaryFormatter** object with a **FileStream** object to specify how to read and write the data. Serialization and formatting objects is outside the scope of this course.

Reading Primitive Data Types

The **BinaryReader** class enables you to read any primitive data type by using 16 specific read methods. The following table describes some of the read methods that the **BinaryReader** class provides.

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description
ReadBoolean	Enables you to read a **true/false** value from a stream.
ReadChar	Enables you to read a single character from a stream.
ReadChars	Enables you to read a collection of characters from a stream. When you use this method, you must specify the number of characters that you want the method to return.
ReadDouble	Enables you to read a double value from a stream.
ReadInt	Enables you to read an **int** value from a stream.
ReadLong	Enables you to read a **long** value from a stream.
ReadString	Enables you to read a **string** value from a stream.

Each of the read methods is designed to work with a specific data type. The method reads the required number of bytes for that type, and then advances the stream to the next block of bytes.

The following code example shows how to read a file that contains a variety of primitive types.



Note: When you read an array, you must specify the number of items in the array that you want to read.

```
// Source file path.
string sourceFilePath =
    @"C:\Users\Student\Documents\PrimitiveDataTypeFile.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 BinaryWriter object passing in the FileStream object.
BinaryReader reader = new BinaryReader(sourceFile);

bool boolValue = reader.ReadBoolean();
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
byte byteValue = reader.ReadByte();

byte[] byteArrayValue = reader.ReadBytes(4);

char charValue = reader.ReadChar();

char[] charArrayValue = reader.ReadChars(4);

decimal decimalValue = reader.ReadDecimal();

double doubleValue = reader.ReadDouble();

float floatValue = reader.ReadSingle();

int intValue = reader.ReadInt32();

long longValue = reader.ReadInt64();

sbyte sbyteValue = reader.ReadSByte();

short shortValue = reader.ReadInt16();

string stringValue = reader.ReadString();

uint unintValue = reader.ReadUInt32();

ulong ulongValue = reader.ReadUInt64();

ushort ushortValue = reader.ReadUInt16();


// Close the streams to release any file handles.
reader.Close();
sourceFile.Close();
```

Writing Primitive Data Types

The **BinaryWriter** class enables you to write any primitive data type with the write method, which provides several overloads. The following code example shows how you can use the **BinaryWriter** class to write a variety of primitive data types to a file.

MCT USE ONLY. STUDENT USE PROHIBITED

```
string destinationFilePath =
    @"C:\Users\Student\Documents\PrimitiveDataTypeFile.txt";

// 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);

bool boolValue = true;
writer.Write(boolValue);

byte byteValue = 1;
writer.Write(byteValue);

byte[] byteArrayValue = { 1, 4, 6, 8 };
writer.Write(byteArrayValue);

char charValue = 'a';
writer.Write(charValue);

char[] charArrayValue = {'a', 'b', 'c', 'd'};
writer.Write(charArrayValue);

decimal decimalValue = 1.00m;
writer.Write(decimalValue);

double doubleValue = 2.5;
writer.Write(doubleValue);

float floatValue = 4.5f;
writer.Write(floatValue);

int intValue = 999999999;
writer.Write(intValue);

long longValue = 999999999999999999;
writer.Write(longValue);

sbyte sbyteValue = 99;
writer.Write(sbyteValue);

short shortValue = 9999;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
writer.Write(shortValue);

string stringValue = "MyString";
writer.Write(stringValue);

uint unintValue = 999999999;
writer.Write(unintValue);
ulong ulongValue = 999999999999999999;
writer.Write(ulongValue);

ushort ushortValue = 9999;
writer.Write(ushortValue);
// Close both streams to flush the data to the file.
writer.Close();
destFile.Close();
```

The above code example produces a file with the following contents.

```
00000000  01 01 01 04 06 08 61 61  62 63 64 64 00 00 00 00   ......aabcdd....
00000010  00 00 00 00 00 00 00 00  00 02 00 00 00 00 00 00   ................
00000020  00 04 40 00 00 90 40 FF  C9 9A 3B FF FF 63 A7 B3   ..@...@...;..c..
00000030  B6 E0 0D 63 0F 27 08 4D  79 53 74 72 69 6E 67 FF   ...c.'.MyString.
00000040  C9 9A 3B FF FF 63 A7 B3  B6 E0 0D 0F 27            ..;..c......'
```

Question: What method would you use to read a 64-bit signed integer from a binary stream?

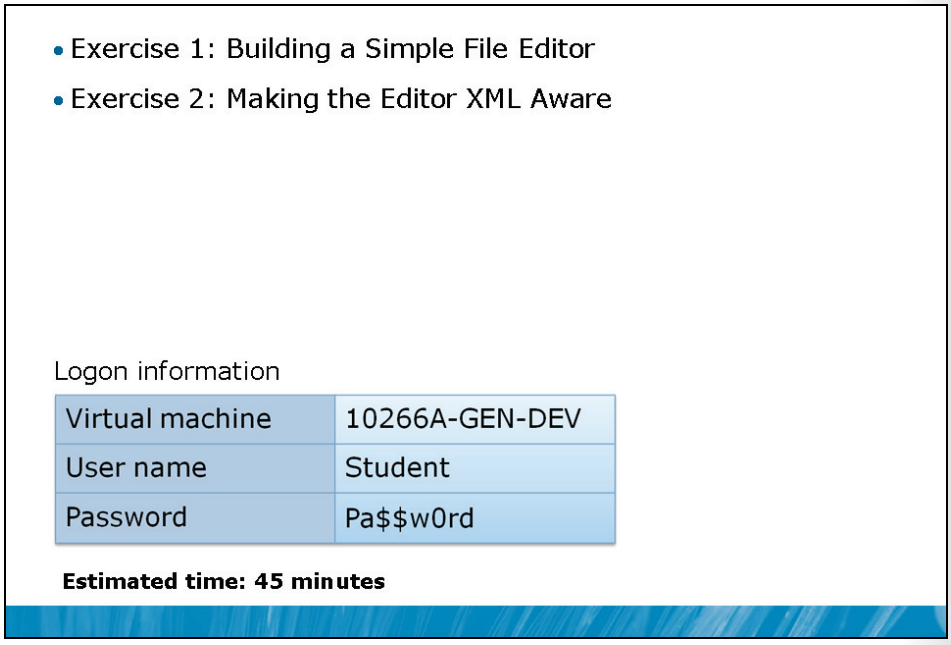
Additional Reading

For more information about the **BinaryWriter** class, see the BinaryWriter Class page at http://go.microsoft.com/fwlink/?LinkId=192923.

For more information about the **BinaryReader** class, see the BinaryReader Class page at http://go.microsoft.com/fwlink/?LinkId=192924.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Reading and Writing Files



Objectives

After completing this lab, you will be able to:

- Read and write data by using the **File** class.
- Read and write data by using a **FileStream** class.

Introduction

In this lab, you will use the **File** class in the **System.IO** namespace to read and write data to a file on the file system. You will then use a stream class to process this file.

Lab Setup

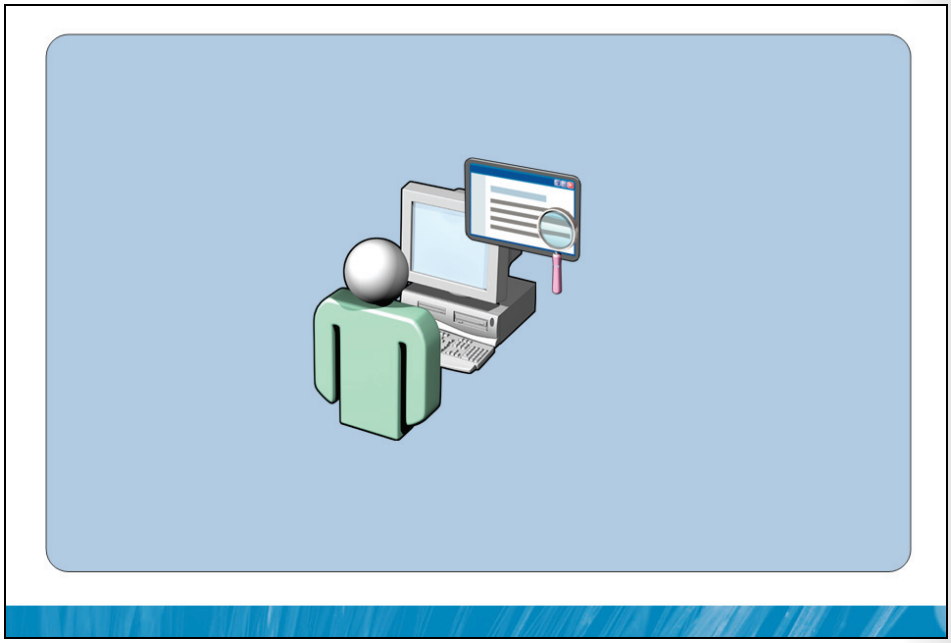
For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

MCT USE ONLY. STUDENT USE PROHIBITED

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Many of the robotic devices that Fabrikam, Inc. builds are controlled by using instructions that are held in a text file that is stored on the device. You have been asked to write a simple application that a user can use to open, display, and edit one of these text files (the device will not have Notepad installed). The application will run on the device, and make use of a small screen and keypad that is built into the device. The application must be easy to use, and include full exception handling.

Exercise 1: Building a Simple File Editor

In this exercise, you will add functionality to a simple WPF application that can be used to edit text files. The WPF application expects the user to enter the name and path of a text file by using the **Open File** common dialog box. The application will then open this file and display its contents in a text box on the WPF form. The user can edit this text, and then save the amended text back to the file.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The user interface for this application has already been completed, but you will implement the logic to enable the user to specify the file to edit, and to load and save the file.

The main tasks for this exercise are as follows:

1. Open the SimpleEditor project.
2. Display a dialog box to accept a file name from the user.
3. Implement a new class to read and write text to a file.
4. Update the MainWindow event handlers to consume the **TextFileOperations** class.
5. Implement test cases.

▶ Task 1: Open the SimpleEditor project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio® 2010.
3. Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex1\Starter folder.

▶ Task 2: Display a dialog box to accept a file name from the user

1. Display the MainWindow.xaml window.

   The MainWindow window implements a very simple text editor. The main part of the window contains a text box that a user can use to display and edit text. The **Open** button enables the user to open a file, and the **Save** button enables the user to save the changes to the text back to a file. You will add the code that implements the logic for these two buttons.

2. Review the task list.
3. Locate the task **TODO - Implement a method to get the file name**. Double-click this task.

   This task is located in the MainWindow.xaml.cs class file.

4. Delete the comment, and then define a new private method named **GetFileName** that accepts no parameters and returns a string value that holds the file name that the user specified.

5. In the method body, declare a new string member named **fname**, and then initialize it with the **String.Empty** value.
6. At the end of the collection of **using** statements at the top of the file, add a statement to bring the **Microsoft.Win32** namespace into scope.
7. In the **GetFileName** method, after the statement that declares the fname variable, add code to the method to perform the following actions:
   a. Create a new instance of the **OpenFileDialog** dialog box, named **openFileDlg**.
   b. Set the **InitialDirectory** property of **openFileDlg** to point to the E:\Labfiles\Lab 5\Ex1\Starter folder.



Note: When including file paths in code, you should prefix the string with the @ symbol. This symbol instructs the C# compiler to treat any '\' characters as literals rather than escape characters.

   c. Set the **DefaultExt** property of **openFileDlg** to ".txt";
   d. Set the **Filter** property of **openFileDlg** to "**Text Documents (.txt)|*.txt**".
8. Add code to perform the following tasks:
   a. Call the **ShowDialog** method of **openFileDlg**, and then save the result.



Note: The value that **ShowDialog** returns is a nullable Boolean value, so save the result in a nullable Boolean variable.

   b. If the result is **true**, assign the value of the **FileName** property of **openFileDlg** to the fname variable.
9. At the end of the method, return the value in the fname variable.

▶ Task 3: Implement a new class to read and write text to a file

1. Add a new class named **TextFileOperations** to the FileEditor project.

   You will use this class to wrap some common file operations. This scheme enables you to change the way in which files are read from or written to without affecting the rest of the application.

MCT USE ONLY. STUDENT USE PROHIBITED

2. At the top of the class file, add a statement to bring the **System.IO** namespace into scope.
3. In the **TextFileOperations** class, add a public static method named **ReadTextFileContents**. The method should accept a string parameter named *fileName*, and return a string object.
4. In the **ReadTextFileContents** method, add code to return the entire contents of the text file whose path is specified in the *fileName* parameter.



Hint: Use the static **ReadAllText** method of the **File** class.

5. Below the **ReadTextFileContents** method, add a public static method named **WriteTextFileContents**. The method should not return a value type, and should accept the following parameters:
   a. A string parameter named *fileName*.
   b. A string parameter named *text*.
6. In the **WriteTextFileContents** method, add code to write the text that is contained in the *text* parameter to the file that is specified in the *fileName* parameter.



Hint: Use the static **WriteAllText** method of the **File** class.

7. Build the solution and correct any errors.

▶ Task 4: Update the MainWindow event handlers to consume the TextFileOperations class

1. In the task list, locate the task **TODO - Update the OpenButton_Click method**. Double-click this task.

   This task is located in the **OpenButton_Click** method of the **MainWindow** class.
2. Remove the comment, and then add code to perform the following tasks:
   a. Invoke the **GetFileName** method. Store the result of the method in the **fileName** member.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

b. If **fileName** is not an empty string, call the static **ReadTextFileContents** method of the **TextFileOperations** class, and then pass *fileName* as the parameter. Store the result in the **Text** property of the editor **TextBox** control in the WPF window.

3. In the task list, locate the task **TODO - Update the SaveButton_Click method**. Double-click this task.

   This task is located in the **SaveButton_Click** method of the **MainWindow** class.

4. In the **SaveButton_Click** method, remove the comment, and then add code to perform the following tasks:

   a. Check that the **fileName** member is not an empty string.

   b. If **fileName** is not an empty string, call the static **WriteTextFileContents** method of the **TextFileOperations** class. Pass **fileName** and the **Text** property of the editor **TextBox** control as the parameters.

5. Build the solution and correct any errors.

6. Start the application without debugging.

7. In the MainWindow window, click **Open**.

8. In the **Open** dialog box, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder, click **Commands.txt**, and then click **Open**.

9. In the MainWindow window, verify that the text in the following code example is displayed in the editor **TextBox** control.

```
Move x, 10
Move y, 20
If x < y Add x, y
If x > y & x < 20 Sub x, y
Store 30
```

   This is the text from the Commands.txt file.

10. Change the **Store 30** line to **Save 50**, and then click **Save**.

11. Close the MainWindow window.

12. Using Windows Explorer, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder.

13. Open the Commands.txt file by using Notepad.

14. In Notepad, verify that the last line of the file contains the text **Save 50**.

15. Close Notepad and return to Visual Studio.

▶ Task 5: Implement test cases

1. In the task list, locate the task **TODO - Complete Unit Tests**. Double-click this task.

   This task is located in the **TextFileOperationsTest** class.

2. Remove the comment.

3. Examine the **ReadTextFileContentsTest1** method, and then uncomment the commented line.

   This method creates three strings:

   a. The **fileName** string contains the path of a prewritten file that contains specific content.

   b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.

   c. The **actual** string is initialized by calling the **ReadTextFileContents** method that you just implemented.

   The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

4. Examine the **WriteTextFileContentsTest1** method, and then uncomment the commented line.

   This method creates two strings:

   a. The **fileName** string contains the path of a nonexistent file, which the method will create when run.

   b. The **text** string contains some text that the method will write to the file.

   The method calls the **WriteTextFileContents** method, passing the **fileName** and **text** strings as parameters. This creates the file at the specified location, and writes to the file. The method then creates a further string, **expected**, by calling the **File**.**ReadAllText** method and reading the text from the written file. The method then checks that the text string and the expected string are the same, before deleting the file that was created during the test.

5. Run all tests in the solution, and verify that all tests execute correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Making the Editor XML Aware

The applications that control a robotic device read the instructions from the file and then encode them as an XML document before passing them to the instruction execution module on the device. For example, imagine that a text file contains the instructions in the following code example.

```
Move x, 10
Move y, 20
If x < y Add x, y
If x > y Sub x, y
Store 30
```

The control applications will wrap them in a pair of XML tags, as the following code example shows.

```
<ControlApplication>
    <Instructions Code = "
       Move x, 10
       Move y, 20
       If x < y Add x, y
       If x > y Sub x, y
       Store 30"
     />
</ControlApplication>
```

However, some of the data in these instructions can contain characters such as ">" and "<" that might be misinterpreted as XML tags rather than data.

In this exercise, you will modify the WPF application to look for data that contains XML tags in the text file as it is read in and encode this data as XML escape sequences before displaying it. For example, the "<" character will be replaced with ">", the ">" symbol will be replaced with "<", and so on. The WPF application will use a file stream to read the data.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add a new method to filter XML characters to the **TextFileOperations** class.
3. Update the user interface to invoke the new method.
4. Implement test cases.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the starter project

- Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex2\Starter folder.

  This project is a completed version of the SimpleEditor project from Exercise 1.

▶ Task 2: Add a new method to filter XML characters to the TextFileOperations class

1. Review the task list.
2. In the task list, locate the **TODO - Implement a new method in the TextFileOperations class** task. Double-click this task.

   This task is located in the **TextFileOperations** class.
3. Remove the comment, and then add a new public static method named **ReadAndFilterTextFileContents**. The method should accept a string parameter named *fileName*, and return a string.
4. In the **ReadAndFilterTextFileContents** method, add the following local variables:
   a. A **StringBuilder** object named **fileContents**, initialized to a new instance of the **StringBuilder** class.
   b. An integer variable called charCode.
5. Add a statement that instantiates a **StreamReader** object, named **fileReader**, by using the *fileName* parameter.
6. Add a **while** statement that reads each character in the **StreamReader** object until the end of the file is reached.



Hint: Use the **Read** method of the **StreamReader** class to read the next character from a stream. This method returns –1 if there is no more data.

7. In the **while** block, add a **switch** statement that evaluates the charCode variable.

   In the **switch** statement, add **case** statements for each of the characters in the following table. In each statement, append the **fileContent StringBuilder** object with the alternative representation shown in the table.

charCode	Standard representation	Alternative representation
34	" (straight quotation mark)	"
38	& (ampersand)	&
39	' (apostrophe)	'
60	< (less than)	<
62	> (greater than)	>

8. Add a default **case** statement that appends the actual character read from the stream to the **fileContent StringBuilder** object.



Note: The **Read** method returns the value read from the file as an integer and stores it in the charCode variable. You must cast this variable to a character before you append it to the end of the **StringBuilder** object.

9. At the end of the method, return the contents of the **fileContent StringBuilder** object as a string.
10. Build the solution and correct any errors.

▶ Task 3: Update the user interface to invoke the new method

1. In the task list, locate the **TODO - Update the UI to use the new method** task. Double-click this task.

   This task is located in the **OpenButton_Click** method of the **MainWindow.xaml.cs** class.

2. Delete the comment, and then modify the line of code that calls the **TextFileOperations.ReadTextFileContents** method to call the **TextFileOperations.ReadAndFilterTextFileContents** method instead. Pass the **fileName** field as the parameter, and then save the result in the **Text** property of the editor **TextBox** control.
3. Build the solution and correct any errors.
4. Start the application without debugging.
5. In the MainWindow window, click **Open**.

MCT USE ONLY. STUDENT USE PROHIBITED

6. In the **Open** dialog box, move to the **E:\Labfiles\Lab 5\Ex2\Starter** folder, click **Commands.txt**, and then click **Open**.
7. In the MainWindow window, verify that the text in the following code example is displayed in the editor **TextBox** control.

```
Move x, 10
Move y, 20
If x &lt; y Add x, y
If x &gt; y &amp; x &lt; 20 Sub x, y
Store 30
```

   This is the text from the Commands.txt file. Notice that the **<**, **>**, and **&** characters have been replaced with the text **<**, **>**, and **&**.

8. Close the MainWindow window and return to Visual Studio.

▶ Task 4: Implement test cases

1. In the task list, locate the **TODO - Complete Unit Tests** task. Double-click this task.

   This task is located in the **TextFileOperationsTest** class.

2. Examine the **ReadAndFilterTextFileContentsTest** method, and then uncomment the commented line.

   This method creates three strings:

   a. The **filename** string contains the path of a prewritten file that contains specific content.

   b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.

   c. The **actual** string is initialized by calling the **ReadAndFilterTextFileContents** method that you just implemented.

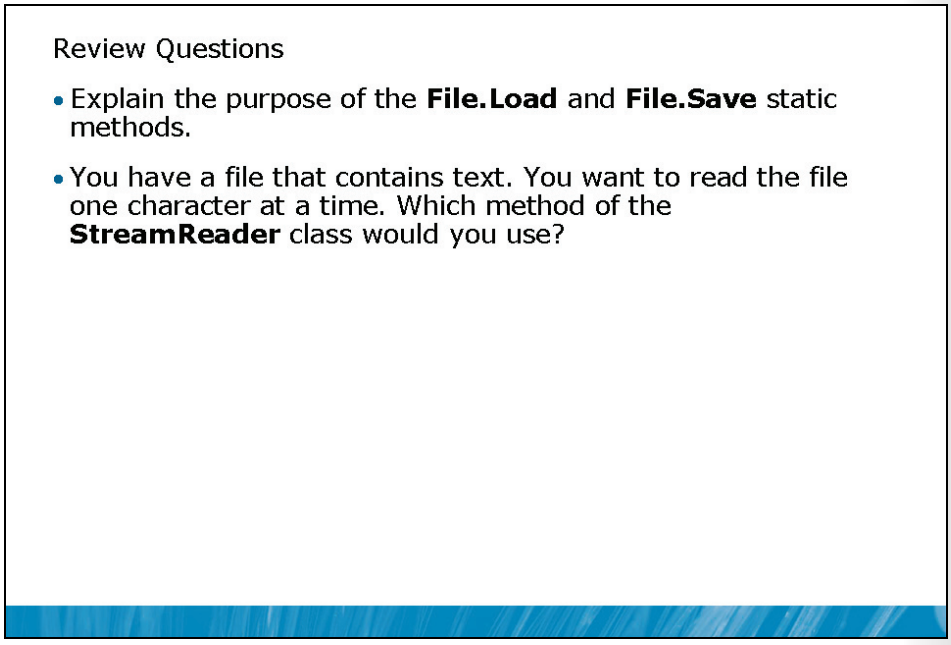
   The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

   This method is complete, and requires no further work.

3. Run all tests in the solution, and verify that all tests execute correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

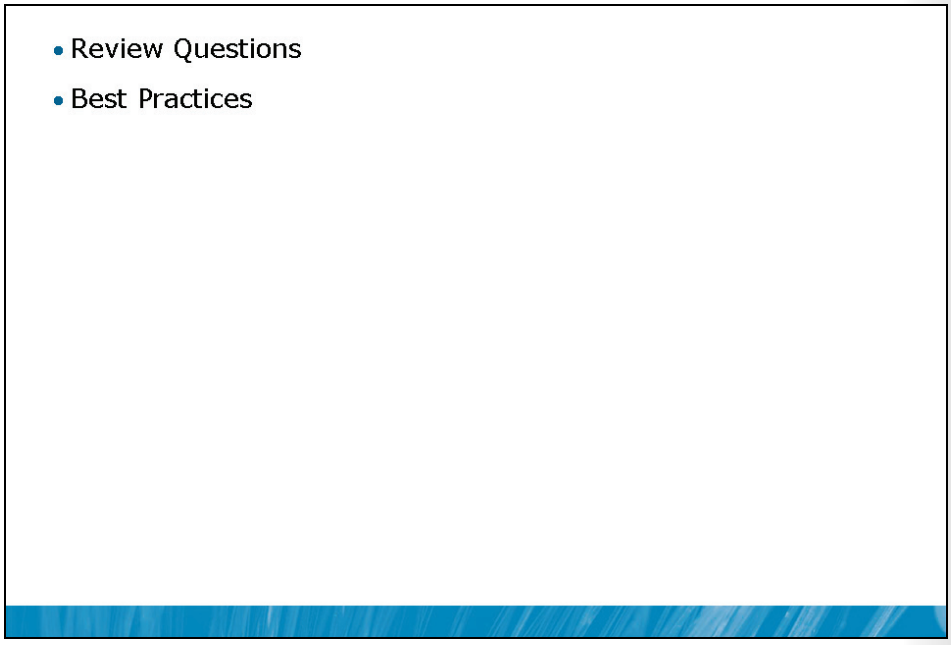


Review Questions

1. Explain the purpose of the **File.Load** and **File.Save** static methods.
2. You have a file that contains text. You want to read the file one character at a time. Which method of the **StreamReader** class would you use?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. When you write data to a stream, name two methods that you could use to ensure that any buffered data is written to the underlying data source.
2. Which two classes does the .NET Framework provide that display a graphical control that enables you to capture a save file and open file path from a user?
3. Which stream class would you use to write textual data?

Best Practices Related to Reading and Writing Data on the File System

Supplement or modify the following best practices for your own work situations:

- Always check to make sure that the file exists before you try to read from it or write to it.
- Do not assume that the contents in the file are going to be correct. Remember that files are stored on the file system, which users have access to. Users are more than capable of editing a file that they should not edit. Always parse a file

MCT USE ONLY. STUDENT USE PROHIBITED

to ensure that it is valid, or be prepared to catch and handle an appropriate exception.

- When you use streams, always ensure that you close the stream after use to ensure that you release any handles on the underlying data source.
- It is easy to assume that you will have permissions to write and read files anywhere in the live environment. Typically, this is not the case. Make sure that your development environment mirrors the live environment.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 6

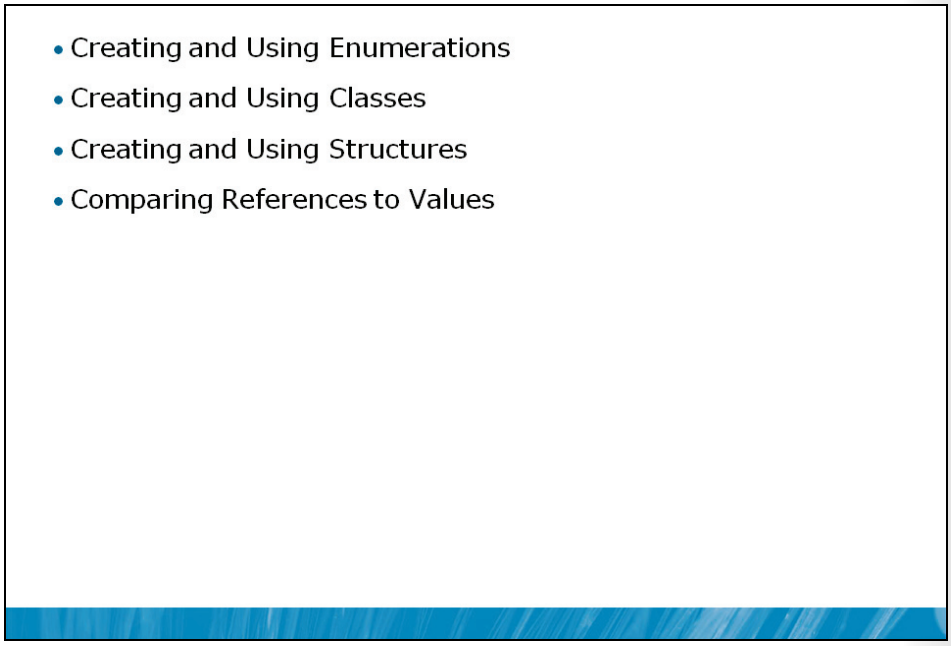
Creating New Types

Contents:

Lesson 1: Creating and Using Enumerations 6-3
Lesson 2: Creating and Using Classes 6-12
Lesson 3: Creating and Using Structures 6-33
Lesson 4: Comparing References to Values 6-41
Lab: Creating New Types 6-55

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



The Microsoft® .NET Framework base class library consists of many types that you can use in your applications. However, in all applications, you must also build your own types that implement the logic for your solution.

This module explains how to create your own types and describes the differences between reference types and value types.

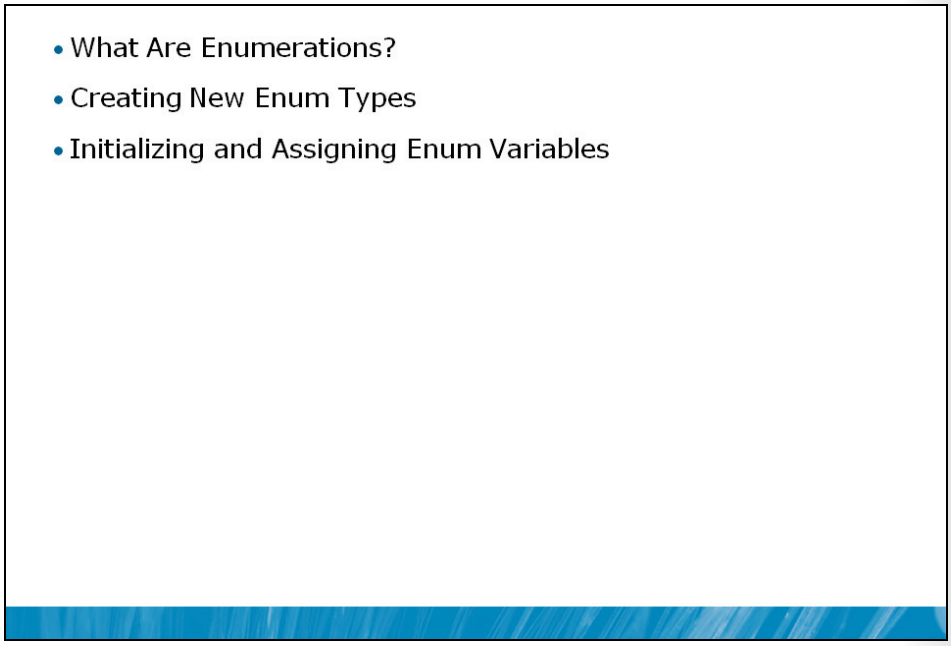
Objectives

After completing this module, you will be able to:

- Describe how to create and use enumerations.
- Describe how to create and use classes.
- Describe how to create and use structures.
- Explain the differences between reference and value types.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Creating and Using Enumerations



An enumeration is a set of related constant values that have a predefined order. They are very useful when you work with data that has a specific range of values. For example, if you model the days of the week, you can use the numbers 0 through 6 to indicate Sunday through Saturday, but this strategy does not lead to readable or easily maintainable code; if your application contains the statement that is shown in the following code example, it is easy to see that the statement assigns the value 5 to variable d, but the purpose of this is not apparent.

```
d = 5;
```

However, the statement in the following code example is immediately more intuitive, and it becomes obvious that d must refer to a day of the week.

```
d = DaysOfWeek.Friday;
```

This lesson describes the purpose of enumerations. It also explains how to create new enumeration types and instantiate and assign existing enumeration types.

MCT USE ONLY. STUDENT USE PROHIBITED

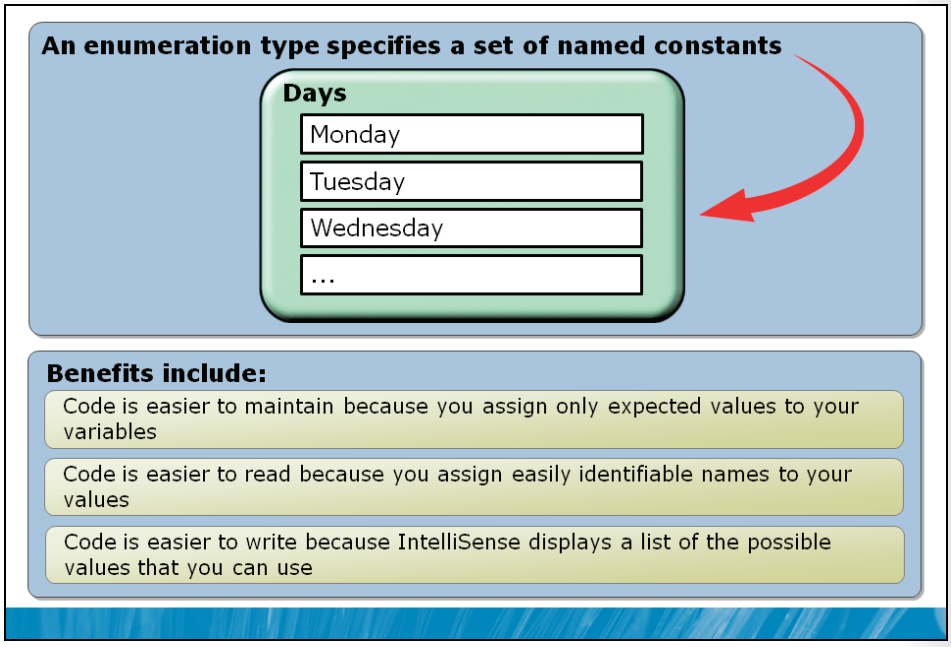
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of enumerations.
- Describe how to create new enumeration types.
- Describe how to initialize and assign existing enumeration types.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Enumerations?



Key Points

An enumeration type specifies a set of related, named constants. An enumeration type is a scalar type that has a user-defined range of values. You can create an enumeration type, declare variables of that type, and assign values to those variables in much the same way that you can use the built-in scalar types of C#, such as **int** or **float**. You can use an enumeration type to represent a set of values in a specific domain. Enumerations also help to make your code easier to read and maintain.

The .NET Framework base class library contains various enumerations that you can use in your applications. Many of the .NET Framework classes use these enumerations as method return values and method parameters.

You may have used enumerations in other languages, such as Java and C++; however, there are a few subtle differences. The main difference is that enumerations in Microsoft Visual C#® are based on the integral data types (such as **int** and **long**), whereas enumerations in Java derive from objects, which means that each enumeration can contain its own modifiable fields. The implementation of enumerations in Visual C# and C++ is very similar.

MCT USE ONLY. STUDENT USE PROHIBITED

Benefits

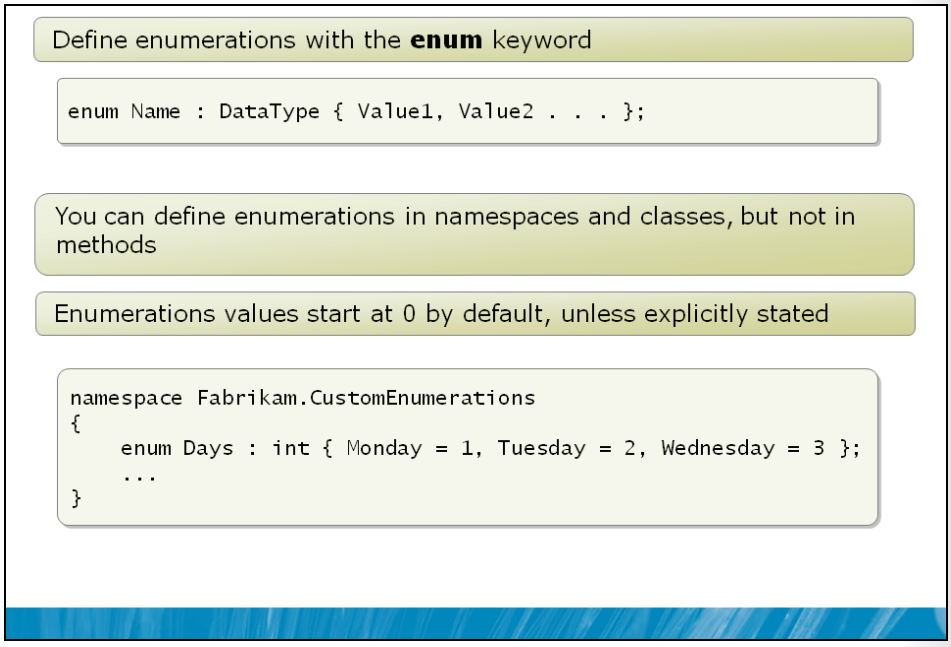
Enumerations provide all of the advantages that constants provide and the following additional benefits:

- Code is easier to maintain because you assign only anticipated values to your variables.
- Code is easier to read because you assign easily identifiable names to your values.
- Code is easier to type because Microsoft IntelliSense® displays a list of the possible values that you can use.
- Code is well formed because you can specify a set of constant values and define a type that will accept values from only that set.

Question: Discuss with other students and the instructor where and how you have used enumerations before.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating New Enum Types



Key Points

You can create your own enumeration types by using the **enum** keyword. You must assign a name to the enumeration and then list the values that your enumeration accepts. Enumerations are types, so you can declare enumerations in a class or a namespace, but not in a method.

The following code example shows the syntax to create an enumeration.

```
enum Name { Value1, Value2 . . . };
```

The following code example declares an enumeration for the seasons of the year.

```
enum Seasons { Spring, Summer, Fall, Winter};
```

Internally, an enumeration type associates an integer value with each element of the enumeration. By default, the numbering starts at 0 for the first element and increments in steps of 1. If you prefer, you can associate a specific integer constant (such as 1) with an enumeration literal (such as **Spring**), as in the following code

MCT USE ONLY. STUDENT USE PROHIBITED

example. In this case, the enumeration literals **Summer**, **Fall**, and **Winter** automatically have the values 2, 3, and 4.

```
enum Season { Spring = 1, Summer, Fall, Winter }
```

The numeric value associated with each enumeration literal becomes significant if you write code that iterates through the possible values that an enumeration variable can have. You can also use the ++ and -- operators on an enumeration variable to advance or retract the value that the variable has.

You can give more than one enumeration literal the same underlying value. For example, in the United Kingdom, fall is referred to as autumn. You can cater to both cultures, as the following code example shows.

```
enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }
```

When you declare an enumeration, the enumeration literals are given values of type **int**. You can also base an enumeration on a different underlying integer type. The following code example declares that the underlying type of the **Season** enumeration is a **short** rather than an **int**.

```
enum Season : short { Spring, Summer, Fall, Winter }
```

The main reason to do this is to save memory; an **int** occupies more memory than a **short**. If you do not require the entire range of values that are available to an **int**, it can make sense to use a smaller data type.

You can base an enumeration on any of the eight integer types: **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, or **ulong**. The values of all of the enumeration literals must fit inside the range of the chosen base type.

Question: Does the following code example show a legal enumeration?

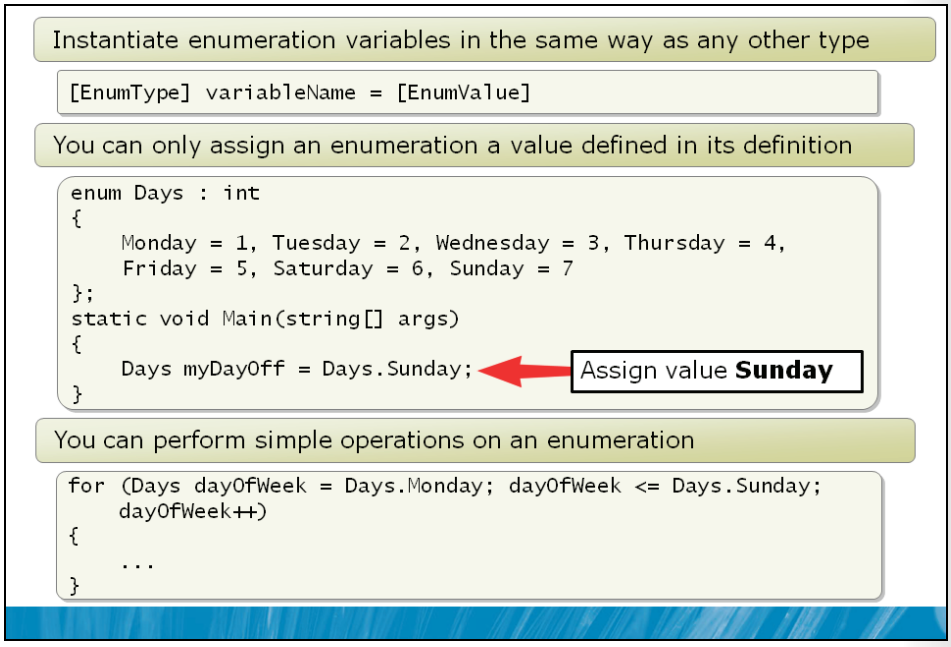
```
enum Season : sbyte {Spring = -3, Summer, Fall, Winter};
```

Additional Reading

For more information about enumerations, see the Enumeration Types (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192925.

MCT USE ONLY. STUDENT USE PROHIBITED

Initializing and Assigning Enum Variables



Key Points

The way in which you define and assign a variable that is based on an enumeration type is very similar to the way in which you use other types in C#. The type of the variable is the name of the enumeration, and the values that you can assign are the literals that the enumeration defines.

The following code example uses an enumeration called **Days**, which contains enumeration values for each day of the week.

```
enum Days
{
    Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4,
    Friday = 5, Saturday = 6, Sunday = 7
};

static void Main(string[] args)
{
    Days myDayOff = Days.Sunday;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

The variable myDayOff is declared by using the **Days** type. Notice that when you assign a value to the myDayOff variable, you explicitly specify the enumeration to which the literal value belongs (**Days.Sunday** in the example). When you create an instance of the **Days** enumeration, you can only assign it one of the literal values that the **Days** enumeration defines: **Monday**, **Tuesday**, **Wednesday**, **Thursday**, **Friday**, **Saturday**, or **Sunday**.

Using an Enum Variable

You can perform simple operations on an enumeration variable in much the same way that you can use an integer variable. The following code example uses an enumeration variable to iterate through the days of the week and display each one in turn.

```
enum Days
{
    Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4,
    Friday = 5, Saturday = 6, Sunday = 7
};
...

for (Days dayOfWeek = Days.Monday; dayOfWeek <= Days.Sunday;
     dayOfWeek++)
{
    Console.WriteLine(dayOfWeek);
}

/* Output is:

Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
*/
```

Notice that you can perform comparisons by using the literal values that the enumeration defines. The comparisons are performed by using the underlying integer values for each literal. Additionally, you can perform integer operations such as `++` and `--` on an enumeration variable. The effect of `++` is to advance the enumeration variable to its next value, and `--` will retreat to the previous value. Finally, when you display the value of an enumeration variable, the value that is displayed is the corresponding literal from the enumeration type. However, if you increment an enumeration variable outside the range of integer values that the

MCT USE ONLY. STUDENT USE PROHIBITED

enumeration type uses, the value that is displayed is the underlying integer value instead (this is usually the result of a programming error, but the C# compiler does not check whether the integer value that is assigned to an enumeration variable in this way is outside the range of values that are used for the literals that the enumeration defines).

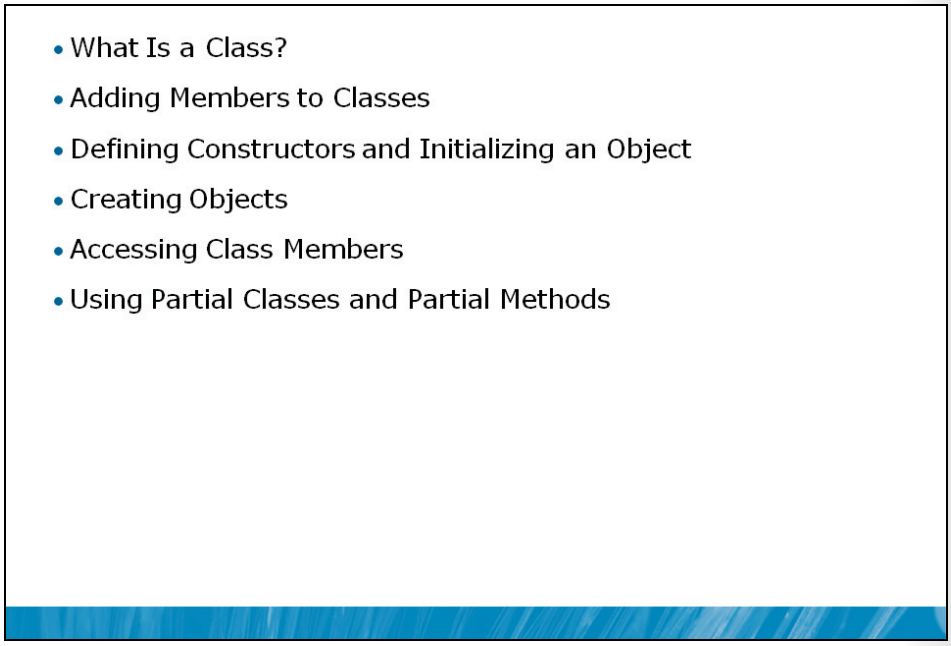


Note: Other than **++** and **—**, you cannot perform any other arithmetic operations on an enumeration variable unless you cast the variable to the underlying integer type first. This is not really a restriction; in most cases, the semantics of arithmetic operations are meaningless for enumeration types (for example, what would the expression **Days.Monday + Days.Wednesday** mean?).

Question: Describe how to initialize an enumeration variable.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Creating and Using Classes



Visual C# is an object-oriented programming language. All of the logic for a C# application is contained in classes and structs. This lesson explains how to create your own classes and use them in your own .NET Framework applications. It also introduces concepts such as partial classes and partial methods.

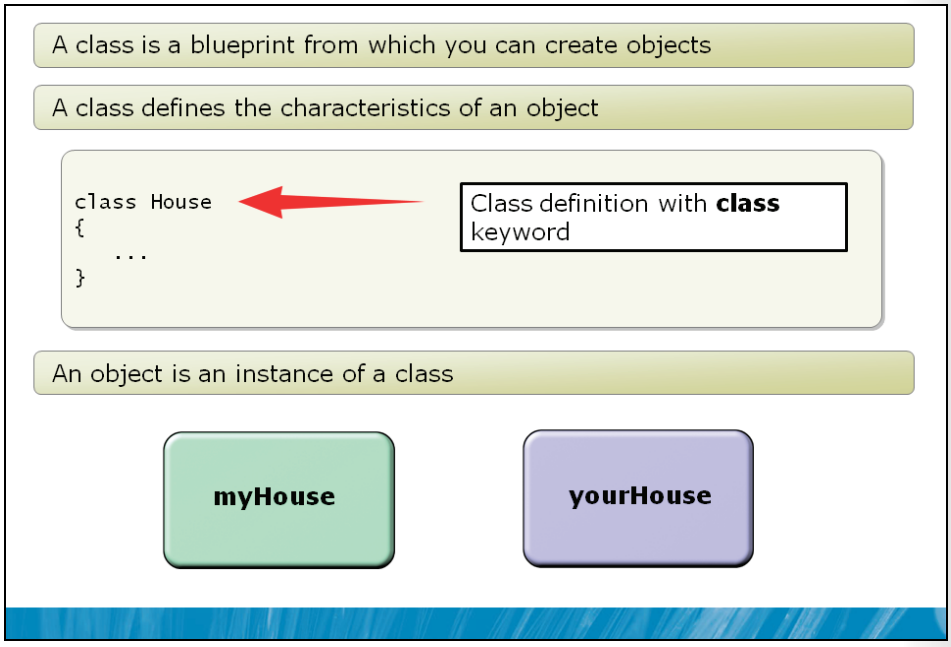
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of classes.
- Describe how to add fields and methods to a class.
- Describe how to define a constructor.
- Explain how to create an instance of a class.
- Describe how to access fields and methods in a class.
- Describe how to create and define partial classes and partial methods.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Class?



Key Points

When you create a C# application, you use classes that represent the principal data types in your application. The .NET Framework provides a large number of reusable utility classes, but you can also define your own classes that encapsulate data and logic that is specific to your own applications.

What Is a Class?

You can think of a class as a blueprint from which you can create objects. A class defines the characteristics of an object, such as the data that the object can contain and the operations that the object can perform. The characteristics of a class are also known as members; members are covered in the next topic.

What Is an Object?

An object is an instance of a class. If a class is like a blueprint, an object is an item that you create by using that blueprint. The class is the definition of an item; the object is the item itself.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: The term *instance* is often used as an alternative to *object*.

In the real world, the plans for a house are like a class, and a house that is built by following these plans is like an object. You can build many instances of houses by following the same set of plans. All of the houses will have the same layout and structure (the same rooms), but they are still different houses. In object-oriented programming, you can define a **House** class that specifies a particular room layout and dimensions. You can then create one or more **House** objects by using this class. Each **House** object will have the room layout and dimensions that the class defines, but some other aspects of each **House** object may be different, such as the location of the **House** object or the color of the front door.

Defining a New Class

You can use Microsoft Visual Studio® to add a new class to a project. Typically, you place each class in a separate source file and give the source file the same name as the class. Visual Studio 2010 generates template code in the source file for the new class. The template code typically includes **using** statements to bring namespaces into scope, the definition of the namespace that the class belongs to, and the class definition itself.

▶ Add a new class to a project

1. In Solution Explorer, right-click the project, point to **Add**, and then click **Class**.
2. In the **Add New Item** dialog box, enter a name for the source file that will contain the new class, and then click **Add**.

The following code example shows a new class definition called **House**.

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

namespace HouseSystem
{
    class House
    {
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

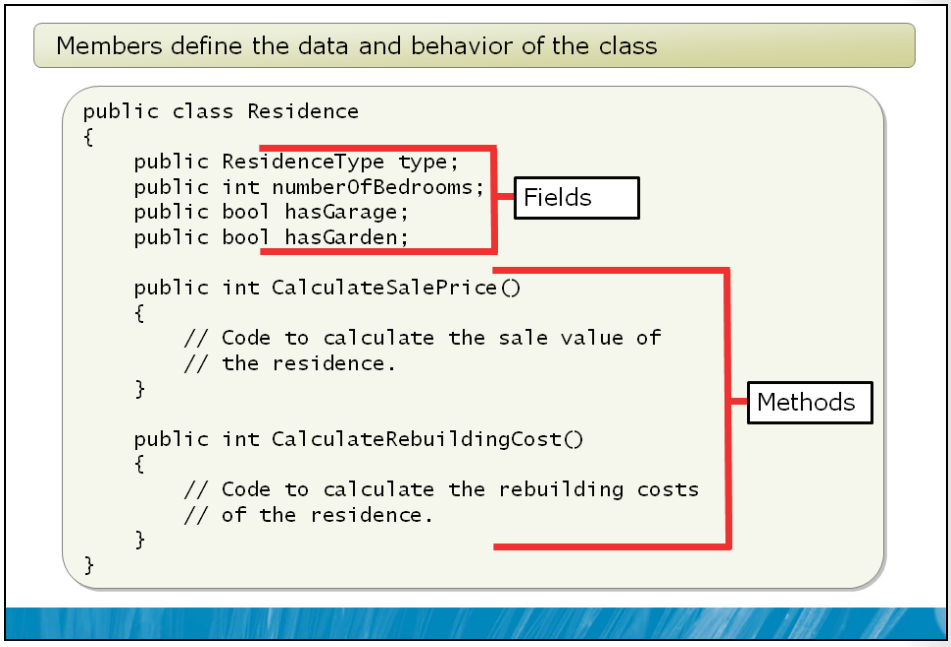


Note: The default namespace generated for the class is determined by the application that you add the class to. You can change this namespace by overwriting it in the source file for the class.

Question: Explain the difference between a class and an object.

MCT USE ONLY. STUDENT USE PROHIBITED

Adding Members to Classes



Key Points

You can add fields and methods to a class that define the data and behavior of that class. You can define any number of fields and methods in a class, depending on the purpose and intended functionality of the class.

Note: The fields and methods that are described in this topic are *instance fields* and *instance methods*. An instance field is a per-instance piece of data, and an instance method is a per-instance operation. Two objects that are based on the same class have their own copy of the instance fields. However, you can share fields between instances by creating **static** members. The **static** keyword is described in detail in a later module.

Defining Fields

You can think of a field as a variable that is scoped to the class. All methods that are defined in the class can access the field. Like a variable, each field has a name, a data type, and an access modifier. If you do not explicitly specify an access modifier for a field, the default access level is private, which means that it can be accessed only by methods that are defined in the class. If you want to make the

MCT USE ONLY. STUDENT USE PROHIBITED

field available to methods that are defined in other classes, you can mark the field as public.



Note: Access modifiers are described in more detail in a later module.

You can place field definitions anywhere in a class. Some programmers prefer to place their field definitions near the start of the class to make the code easy to read for other programmers.

When you define a field, you can also assign a default value to that field, although you can use a constructor to change the value that is assigned to a field when an object is created.



Note: The next topic describes how you can use constructors to initialize class members.

Defining Methods

A method is a procedure or function inside a class. You use methods to implement the behavior of a class. Each method has a name, a parameter list, a return type, and an access modifier.

A method has complete and unrestricted access to all of the other members in the class. This is an important aspect of object-oriented programming; methods encapsulate operations on the fields in the class. When you refer to a field in the class, you can prefix the field with the **this** keyword, as shown in bold in the following code example. This approach helps to disambiguate any references (for example, a parameter to a method can have the same name as a field in a class, although this is not recommended practice) and helps to make your code easier to maintain.

```
public bool hasGarage;

public void OpenGarageDoor(int doorId)
{
    if (this.hasGarage)
    {
        // Code to run if a residence has a garage.
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Example

The following code example shows a **Residence** class that is used as part of a real-estate application. The class has four fields that represent the type of residence, the number of bedrooms, whether the residence has a garage, and whether the residence has a garden. The class has methods that calculate the value of the residence for sale purposes and the cost of rebuilding the residence for insurance purposes.

```
public enum ResidenceType { House, Flat, Bungalow, Apartment };

public class Residence
{
    public ResidenceType type;
    public int numberOfBedrooms;
    public bool hasGarage;
    public bool hasGarden;

    public int CalculateSalePrice()
    {
        // Code to calculate the sale value of the residence.
    }

    public int CalculateRebuildingCost()
    {
        // Code to calculate the rebuilding costs of the residence.
    }
}
```

Using the Class Designer

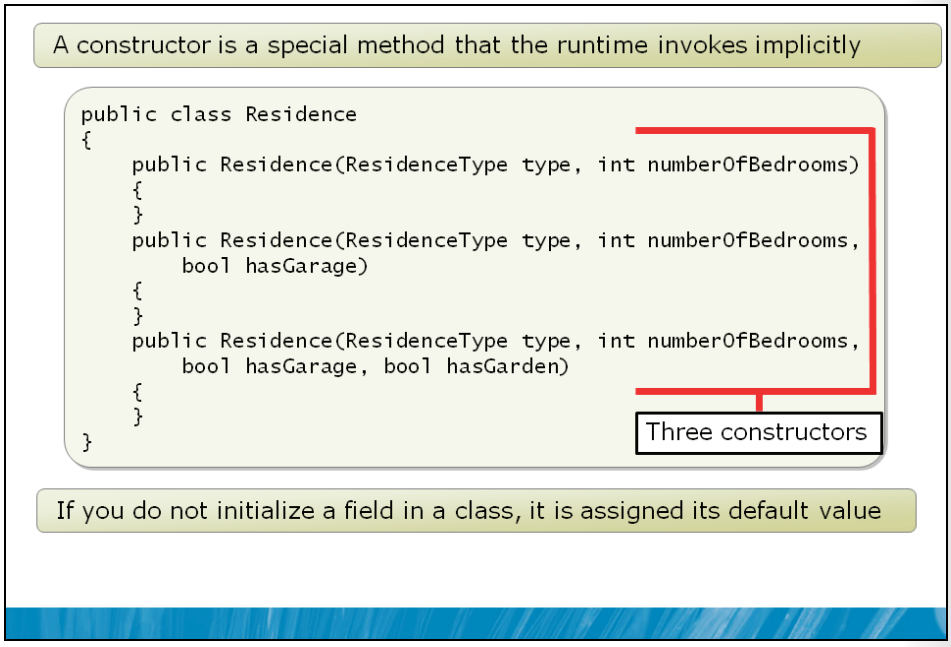
You can design a class manually by writing code in the Code Editor window. However, you can also use the Class Designer window to design a class and add fields and methods graphically.

To use the Class Designer window, you add a class diagram to your project. To add a class diagram to your Visual Studio solution, in Solution Explorer, right-click your project, and then click **View Class Diagram**. The class diagram automatically includes all enumerations, classes, and structures that you have defined in your project. It also provides a toolbox that you can use to add new items to the diagram and add fields and methods to them.

Question: What is the difference between a field and a method?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Constructors and Initializing an Object



Key Points

When you create an object, it is important to ensure that the object is fully initialized and that all of its fields are set to meaningful values. To achieve this, define one or more constructors in the class. The common language runtime (CLR) automatically invokes a constructor when an object is created.

Note: If you do not initialize a field in a class, it is assigned its default value. If the field is a numeric value, it is initialized to zero. If the field is a Boolean value, it is initialized to **false**. If the field is a string, it is initialized to null. If the property is a class, it is also initialized to null.

Defining Constructors

A constructor is a special method that the CLR invokes automatically when you create an object.

The following rules and guidelines apply when you define a constructor:

- Constructors have the same name as the class in which they are defined.

MCT USE ONLY. STUDENT USE PROHIBITED

- Constructors must not specify a return value, not even **void**, but they can take parameters. You can define any number of constructors in a class, provided each constructor has a unique parameter list. A constructor that takes no parameters is known as a *default constructor*.
- Constructors are typically declared with public accessibility to enable any part of the application to create and initialize objects. If you want to limit the parts of the application that can create and initialize objects, you can define a more restrictive access level for the constructors.
- Constructors typically initialize some or all of the fields in the object and can also perform any additional initialization tasks that the class requires.



Important: If you do not define any constructors for a class, the C# compiler automatically generates a default constructor (a constructor that take no parameters) for you. This constructor does nothing, but it enables you to create an instance of the class. However, if you define one or more constructors yourself, the C# compiler will not generate a default constructor.

Example

The following code example shows how to define three constructors for the **Residence** class. The following list describes the constructors:

- The first constructor takes two parameters and sets the type of residence and the number of bedrooms that the residence has.
- The second constructor takes three parameters and sets the type of residence, the number of bedrooms that the residence has, and whether the residence has a garage.
- The third constructor takes four parameters and sets the type of residence, the number of bedrooms that the residence has, whether the residence has a garage, and whether the residence has a garden.



Note: Notice how the code example uses the **this** keyword to distinguish between fields and parameters with the same name.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public enum ResidenceType { House, Flat, Bungalow, Apartment };


public class Residence
{
    public ResidenceType type;
    public int numberOfBedrooms;
    public bool hasGarage;
    public bool hasGarden;

    public Residence(ResidenceType type, int numberOfBedrooms)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
    }

    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
    }

    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }
    ...
}
```

It is also possible to call one constructor from another by using the **this** keyword as part of the constructor declaration. The constructor with the matching signature will be run. Using this feature, you can implement a default constructor that calls a parameterized constructor with a set of default values for each parameter, as shown in the following code example.

```
public class Residence
{
    ...

    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }

    // Default constructor creates a 3-bedroom residence
    // with a garage and a garden
    public Residence() : this(ResidenceType.House, 3, true, true)
    {
    }
    ...
}
```

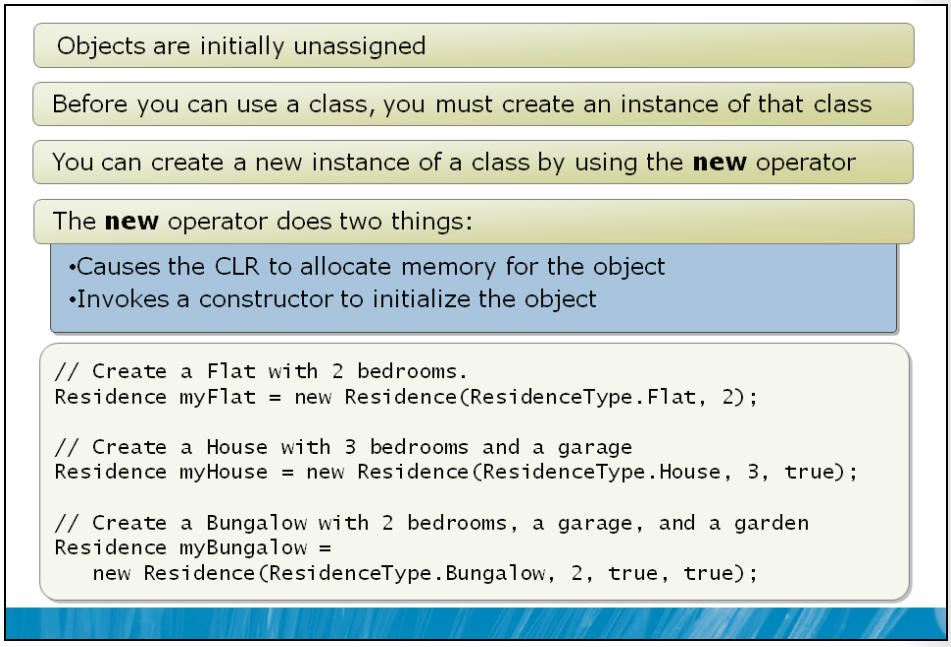
Question: What happens if you do not define a default constructor for a class?

Additional Reading

For more information about constructors, see the Instance Constructors (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192926.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating Objects



Key Points

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

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 and use instances of the **Residence** class by using the constructors that were defined in the previous topic.

```
// Create a flat with two bedrooms.
Residence myFlat = new Residence(ResidenceType.Flat, 2);

// Create a house with three bedrooms and a garage.
Residence myHouse = new Residence(ResidenceType.House, 3, true);
// Create a bungalow with two bedrooms, a garage, and a garden.
Residence myBungalow = new Residence(ResidenceType.Bungalow, 2, true,
true);
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

If you call **new** and do not specify any parameters, the default constructor runs. Remember that if you define one or more constructors for a class, the C# compiler does not create a default constructor for you automatically.

Using an Object Initializer

You instantiate an object by calling a constructor. A constructor may take parameters that specify the values to initialize the fields in the object. However, an object may have any number of fields, and it may not always be possible or feasible to provide constructors that can initialize all possible combinations of these fields. For example, suppose that the **Residence** class currently provides the three constructors that are shown in the following code example.

```
public enum ResidenceType { House, Flat, Bungalow, Apartment };

public class Residence
{
    public ResidenceType type;
    public int numberOfBedrooms;
    public bool hasGarage;
    public bool hasGarden;
    public Residence(ResidenceType type, int numberOfBedrooms)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
    }
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage, bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
        this.hasGarden = hasGarden;
    }
    ...
}
```

Using these constructors, you can create a **Residence** object and initialize various fields, but there is one combination missing. You can only specify that the residence has a garden if you explicitly state whether the residence has a garage; there is no constructor that enables you to initialize the **hasGarden** field without

setting the **hasGarage** property. You may be tempted to define an additional constructor, as shown in the following code example.

```
public class Residence
{
    ...
    public bool hasGarden;

    ...
    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarage)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarage = hasGarage;
    }

    // Constructor to initialize the hasGarden field without setting
    // hasGarage.

    public Residence(ResidenceType type, int numberOfBedrooms,
        bool hasGarden)
    {
        this.type = type;
        this.numberOfBedrooms = numberOfBedrooms;
        this.hasGarden = hasGarden;
    }

    ...
}
```

The problem is that constructors follow the same overloading rules as methods, and you cannot define two or more constructors that have the same signature. In this example, the **Residence** class will not compile because the two constructors have the same signature.

You can solve this problem by using an object initializer.

An object initializer creates an object by using a constructor and also initializes any other fields that are mentioned in the same statement. You specify the fields to initialize and the values to set them to in braces after the call to the constructor, as the following code example shows.

```
// Create a house with three bedrooms and a garden.
Residence myHouse = new Residence(ResidenceType.House, 3) {hasGarden =
true};
```

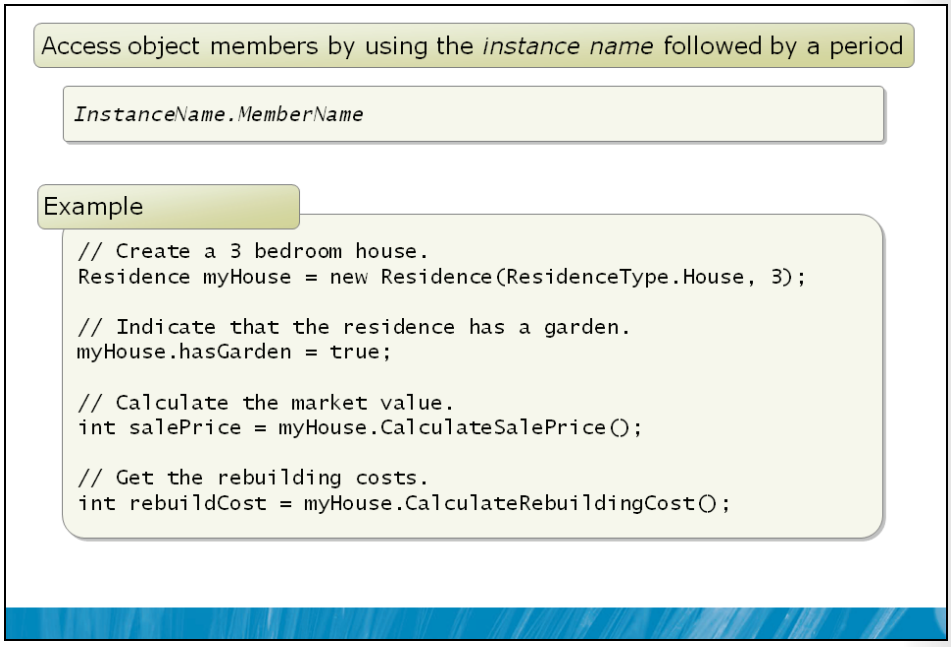
MCT USE ONLY. STUDENT USE PROHIBITED

When you create an object by using an object initializer, the appropriate constructor runs first, and then the property values are assigned. The property assignment may override the initialization that the constructor performs.

Question: Which operator must you use when you initialize a class to create an instance of that class?

MCT USE ONLY. STUDENT USE PROHIBITED

Accessing Class Members



Key Points

To access a member on an instance, use the name of the instance, followed by a period, followed by the name of the member. The following rules and guidelines apply when you access a member on 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 field, use the field name. You can then get the value of that field or set the value of that field.

The following code example performs the following tasks:

- Creates a **Residence** instance by using the constructor that specifies the residence type and the number of bedrooms.

MCT USE ONLY. STUDENT USE PROHIBITED

- Sets the **hasGarden** property to **true** to indicate that the residence has a garden. (You could also do this by using object initialize when the object was created.)
- Calls the **CalculateSalePrice** method to determine the current market value of the residence.
- Calls the **CalculateRebuildingCost** method to determine the cost of rebuilding the residence for insurance purposes.

```
// Create a three-bedroom house.
Residence myHouse = new Residence(ResidenceType.House, 3);

// Indicate that the residence has a garden.
myHouse.hasGarden = true;

// Calculate the market value.
int salePrice = myHouse.CalculateSalePrice();

// Get the rebuilding costs.
int rebuildCost = myHouse.CalculateRebuildingCost();
```

Question: Highlight the syntax errors in the following code example.

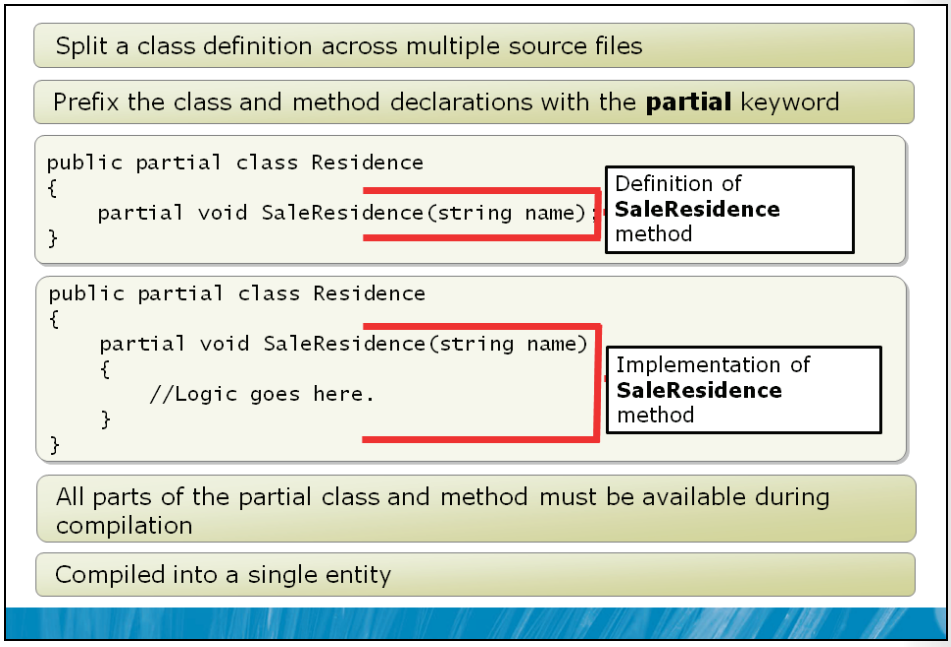
```
Car myCar = new Car("Ford", "Black");

// Set a field to indicate the car's transmission.
myCar,isManual() = true;

// Call a method to calculate the car's value.
double value = myCar,CalculateValue;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Using Partial Classes and Partial Methods



Key Points

There may be situations where you want to split a class definition across multiple source files. For example, you may have multiple developers who want to work concurrently on a class, or you may have parts of a class that should never be modified. The .NET Framework provides the concept of a partial class for this purpose.

Some classes in the .NET Framework and Visual Studio projects use the partial concept. For example, Windows® Presentation Foundation (WPF) applications use partial classes to separate out the code that Visual Studio generates to initialize a window from the programmatic logic that you add to process the user input and display the results.

Defining Partial Classes

Defining a class as partial enables you to split a class over multiple files. To define a class as partial, you must use the **partial** keyword, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
// File1.cs
namespace HouseSystem
{
    public partial class Residence
    {
        //...
    }
}
```

```
// File2.cs
namespace HouseSystem
{
    public partial class Residence
    {
        //...
    }
}
```

The following rules and guidelines apply when you define a partial class:

- Each part of the class must be available when your application is compiled, because the compiler compiles the class into a single entity.
- Each part of the class must be prefixed with the **partial** keyword.
- The partial type cannot be split over multiple assemblies. Each part of the partial type must exist in the same assembly.
- The **partial** keyword must prefix the **class** keyword.

Defining Partial Methods

When you define a partial class, you can define one or more methods in that class as partial methods.

A partial method specifies the method signature in one file that holds the partial class, and it optionally specifies the code that implements the method in another file that holds the partial class. If the partial method is not implemented, it is effectively removed from the class, and any statements that call that method are also ignored when the class is compiled. Partial methods are typically used by frameworks; they provide a mechanism for the classes in the framework to invoke methods when developers outside the framework implement the code for these methods.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code examples demonstrate splitting a method declaration and implementation across a partial file. In these examples, the first code file shows the partial class that is provided as part of a framework of classes. The **FrameworkClass** class defines a partial method called **DoWork**. The **FrameworkMethod** method in this class calls the **DoWork** method. The **DoWork** method is implemented by another developer in a separate file. Note that if this second file does not implement the **DoWork** method, the call to this method in the **FrameworkMethod** method will be ignored by the compiler.

```
// Code provided by the Framework.

public partial class FrameworkClass
{
    partial void DoWork(int data); // The definition of the partial
    // method.

    public void FrameworkMethod()
    {
        ...
        DoWork(99);                // Call the partial method.
        ...
    }
}
```

```
// Code provided by a developer to link into the Framework.

public partial class FrameworkClass
{
    partial void DoWork(int data)
    {
            // Code that implements the DoWork method.
    }
}
```

The following rules and guidelines apply when you define partial methods:

- All partial methods must be void and cannot return a value.
- All partial methods are implicitly private. You cannot access a partial method from outside the class that it is defined in.
- All partial method declarations must be prefixed with the **partial** keyword.
- Partial methods can have ref parameters but not out parameters. Ref parameters are covered later in this module.

MCT USE ONLY. STUDENT USE PROHIBITED

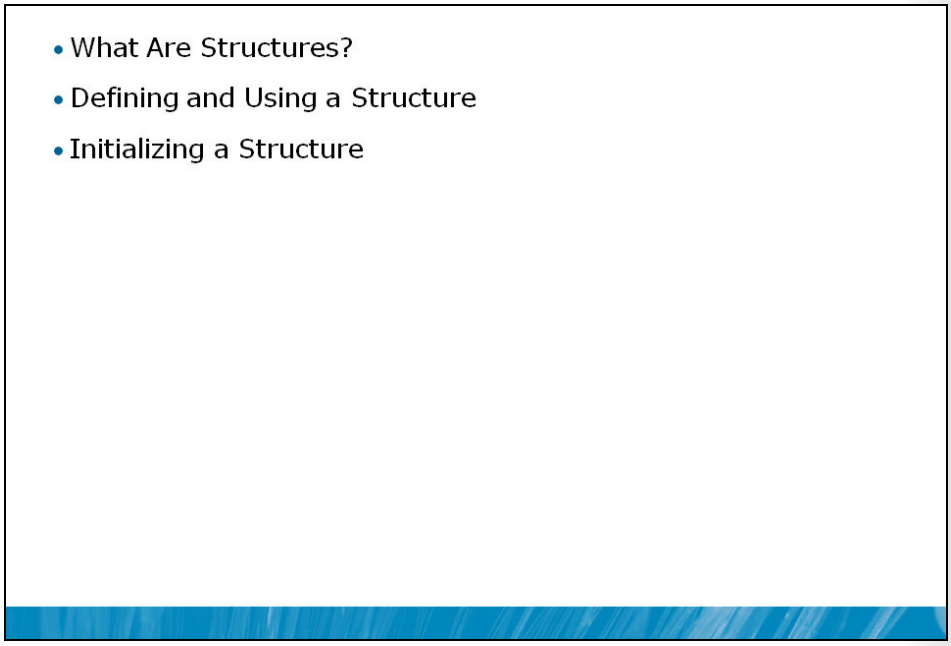
Question: What happens if you define a partial method, but do not provide an implementation of this method?

Additional Reading

For more information about partial classes and methods, see the Partial Classes and Methods (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192927.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Creating and Using Structures



Classes are very useful when you want to model real-world entities in an application and encapsulate their associated business logic and data. However, when you create instances of objects, you will incur an overhead, and sometimes you require a more lightweight solution. Structures have many of the characteristics of classes but without some of the overhead, although they have some limitations.

This lesson describes how to define structures and explains some of the differences between classes and structures.

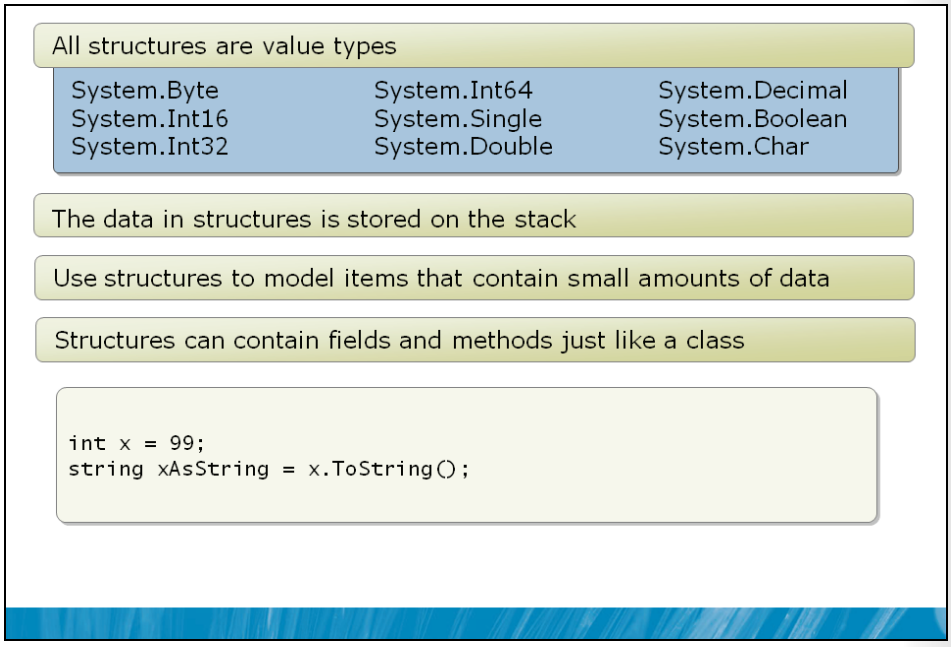
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of structures.
- Describe how to add members to structures.
- Describe how to initialize and access members in a structure.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Structures?



Key Points

A structure is very similar to a class in many respects, except that it has a reduced overhead because of the way in which the CLR creates and manages instances of structures (you will see more about this later in this module). However, structures also have some limitations, which will be discussed later in this course.

You typically use structures to model items that contain relatively small amounts of data. You have used structures throughout the course, although you may not have been aware of this. Many of the primitive types in the C# language are just aliases for some of the structures that the .NET Framework defines, and you can use these aliases or the corresponding structures interchangeably. The following table describes some of these structures.

MCT USE ONLY. STUDENT USE PROHIBITED

Structure type	C# keyword
System.Byte	**byte**
System.Int16	**short**
System.Int32	**int**
System.Int64	**long**
System.Single	**float**
System.Double	**double**
System.Decimal	**decimal**
System.Boolean	**bool**
System.Char	**char**

Like a class, a structure can contain fields and implement methods. For example, the **System.Int32** structure defines the **ToString** method, which returns a string representation of the integer value that is held. This means that you can perform operations on an **int**, as the following code example shows.

```
int x = 99;
string xAsString = x.ToString();
```

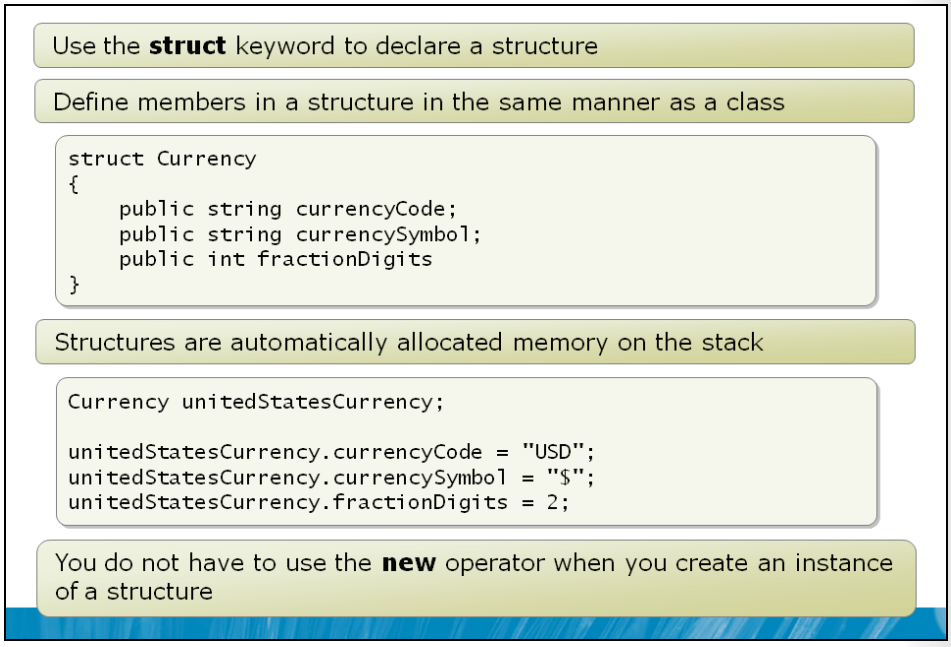
Note that by default, you cannot use many of the common operators such as **==** and **!=** on structure types unless you provide definitions of these operators. The syntax that you use for this is described in a later module. The types that are listed in the previous table provide their own implementations of these operators.

Question: Is the following code legal?

```
int x = 99;
System.Int32 y = x + 1;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Defining and Using a Structure



Key Points

The syntax that you use to declare a structure is similar to the syntax that you use to declare a class, except that you use the **struct** keyword rather than the **class** keyword. The syntax that you use to define members in structures is also very similar to the way in which you define members in classes. The main difference is that when you define instance fields in a structure, you cannot assign a value in the declaration.

The following code example shows a structure type named **Currency**, which can be used to represent a monetary value.

```
using System;
using System.Collections.Generic;
using System.Text;
struct Currency
{
    public string currencyCode;   // The ISO 4217 currency code
    public string currencySymbol; // The currency symbol ($,£,...)
    public int fractionDigits;    // The number of decimal places
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Using a Structure

The CLR manages structures in a different way from classes. When you declare a structure variable, the memory for that variable is allocated automatically. Consequently, you do not have to use the **new** operator to create an instance of a structure type; you simply declare a variable of that type. You can then assign the individual values of the fields by using the same dot notation that you use to reference members of a class. You can read the values of fields in the same way. The following code example shows how to create and use an instance of the **Currency** type.

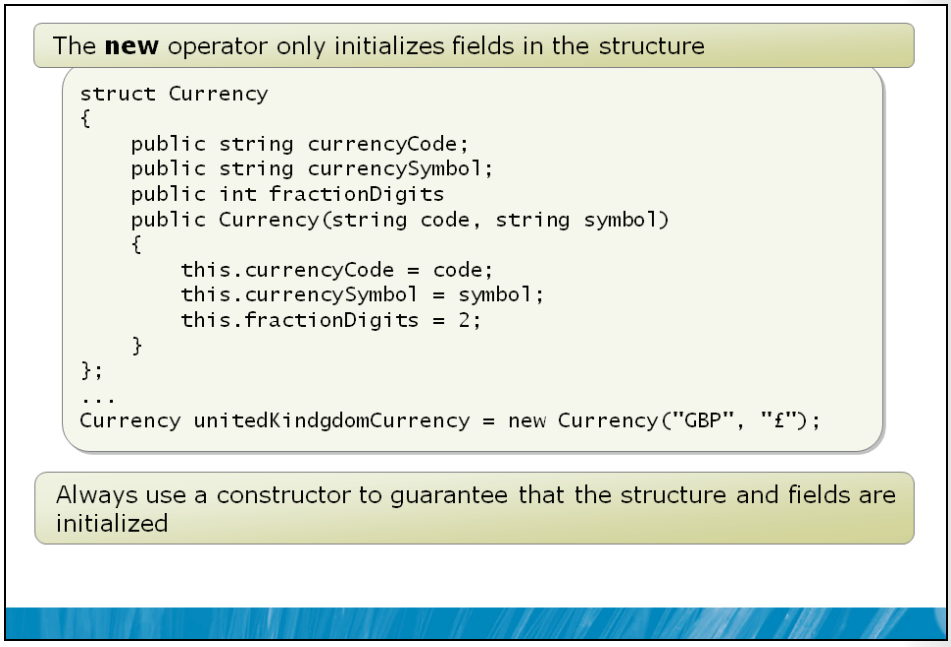
```
Currency unitedStatesCurrency;

unitedStatesCurrency.currencyCode = "USD";
unitedStatesCurrency.currencySymbol = "$";
unitedStatesCurrency.fractionDigits = 2;
```

Question: What keyword do you use to declare a structure?

MCT USE ONLY. STUDENT USE PROHIBITED

Initializing a Structure



Key Points

When you create an object by using a class, you use the **new** operator to allocate memory for the corresponding object and invoke a constructor. You do not need to use the **new** operator to create an instance of a structure, because the memory is allocated automatically when you declare a struct variable. However, if you want to initialize the fields in a structure when you create the instance, you can define one or more constructors.

Constructors for structs are syntactically very similar to constructors for classes, but there are some semantic differences. The biggest differences are as follows:

- You cannot define a default (parameterless) constructor for a struct. This is because, unlike a class, the compiler *always* generates its own default constructor for a struct, regardless of whether you define any other constructors.
- All constructors *must* explicitly initialize every field in the struct. In addition, a constructor cannot call other methods in a struct before all of the fields have been assigned a value.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows the **Currency** struct again, but this time it has a constructor defined that takes two parameters that specify the currency code and symbol to use. The variable unitedKingdomCurrency is initialized by using this constructor.

```
struct Currency
{
    public string currencyCode;   // The ISO 4217 currency code.
    public string currencySymbol; // The currency symbol ($,£,...).
    public int fractionDigits;    // The number of decimal places.

    public Currency(string code, string symbol)
    {
        this.currencyCode = code;
        this.currencySymbol = symbol;
        this.fractionDigits = 2;
    }
};

...
Currency unitedKindgdomCurrency = new Currency("GBP", "£");
```



Important: If you create an instance of a struct, but do not use a constructor, the struct is considered to be uninitialized. Although you can read and write individual fields in an uninitialized struct, you cannot use it as an argument to a method or copy it to another variable until you have explicitly assigned a value to every field in that struct. The simplest way to guarantee that a struct is fully initialized is to always use a constructor. Remember that the compiler generates a default constructor for you automatically, so you do not have to write your own if you simply want a struct to be populated with default values.

Question: You decide to add further constructors to the **Currency** type, and you attempt to factor out common initialization code into a method in the type, as shown in the following code example. Why does this approach not work?

```
struct Currency
{
    public string currencyCode;   // The ISO 4217 currency code
    public string currencySymbol; // The currency symbol ($,£,...)
    public int fractionDigits;    // The number of decimal places

    public Currency(string code)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

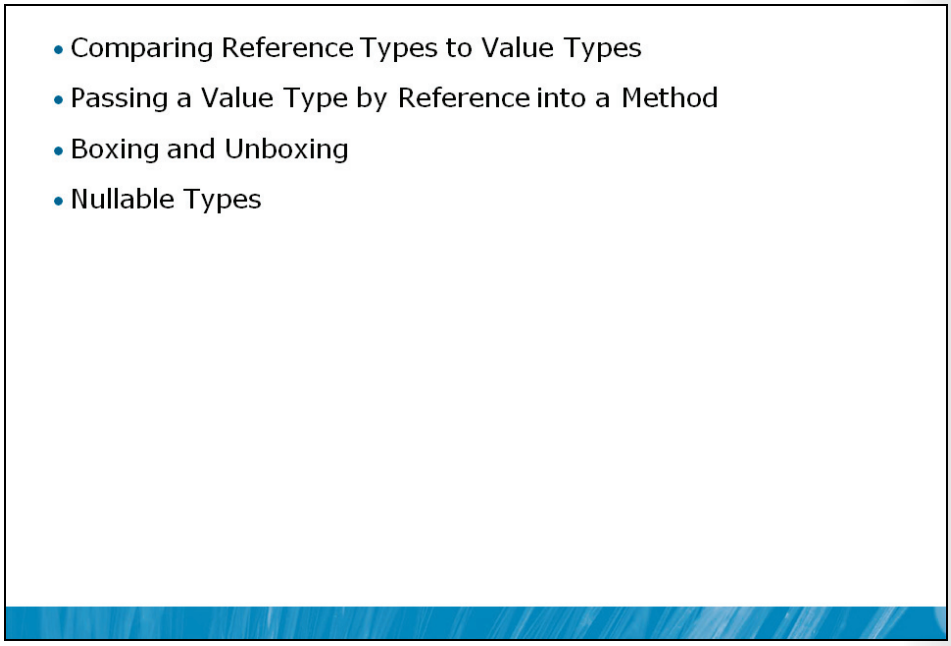
```
        this.initialize();
        this.currencyCode = code;
    }

    public Currency(string code, string symbol)
    {
        this.initialize();
        this.currencyCode = code;
        this.currencySymbol = symbol;
    }

    // Specify default values
    // Constructors may override these values
    public void initialize()
    {
        this.currencyCode = "USD";
        this.currencySymbol = "$";
        this.fractionDigits = 2;
    }
};
```

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Comparing References to Values



Structure types, such as user-defined structs and the primitive types that C# uses, are also called *value* types. When you declare a variable as a structure type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value. For example, declaring an int variable causes the compiler to allocate 4 bytes of memory (32 bits). A statement that assigns a value (such as 42) to the int variable causes the data for this value to be copied into this block of memory.

Class types, such as the **Residence** class that was defined in Lesson 1, are called *reference* types. The CLR handles these types differently. When you declare a Residence variable, the compiler does not generate code that allocates a block of memory big enough to hold a **Residence** object All the compiler does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory that contains a **Residence** object. Finally, the compiler initializes this reference to the **null** value to indicate that the object has not yet been initialized. The memory for the **Residence** object is allocated only when you use the **new** keyword to call a constructor and create the object.

MCT USE ONLY. STUDENT USE PROHIBITED

This lesson describes the differences between reference and value types and explains how their behavior differs when you use them as parameters in methods. This lesson also describes how to convert a value into a reference and back again by using boxing and unboxing and how to create value types that can hold null references.

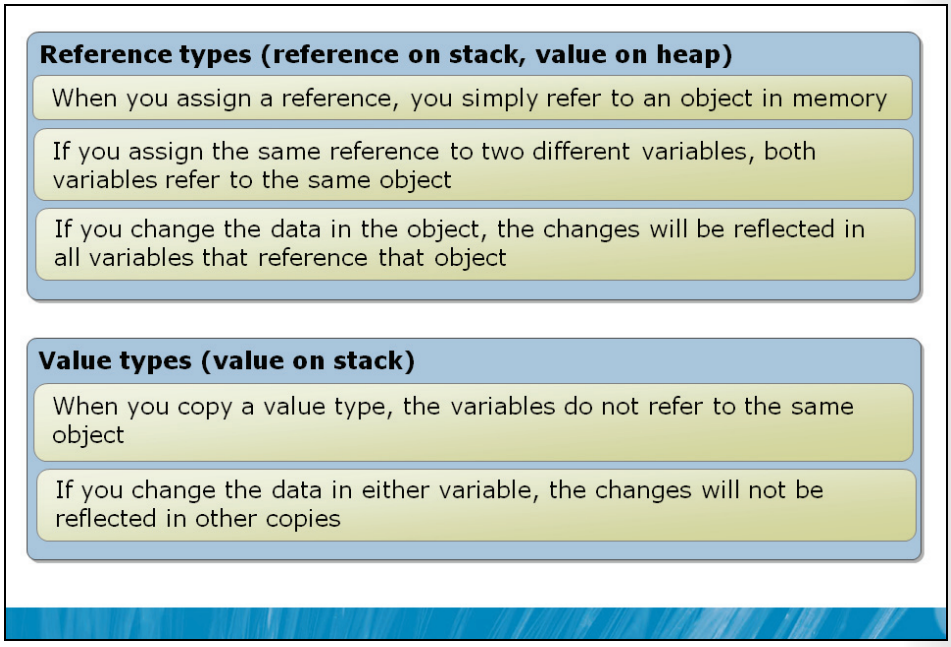
Objectives

After completing this lesson, you will be able to:

- Describe the differences between reference and value types.
- Describe how to pass a value type by reference into a method.
- Describe how perform boxing and unboxing.
- Describe how to create and use a null value type.

MCT USE ONLY. STUDENT USE PROHIBITED

Comparing Reference Types to Value Types



Key Points

The CLR divides its memory into two main areas: the stack and the heap. Most of the time, value types are created on the stack, and reference types are created on the heap. These two areas use memory in different ways. The details of how memory is managed are described in a later module.

The main difference between value types and reference types is what happens when you copy them. In the following code examples, the **Residence** type is a class (a reference type), and the **Currency** type is a struct (a value type).

When you assign a reference, you simply refer to an object in memory. If you assign the same reference to two different variables, both variables refer to the same object. In the following code example, the myHouse variable refers to a new **House** object. The variable refToMyHouse refers to the same object.

```
// Create a two-bedroom House object.
Residence myHouse = new Residence(ResidenceType.House, 2);
Residence refToMyHouse = myHouse;
```

MCT USE ONLY. STUDENT USE PROHIBITED

If you change the data that the myHouse variable refers to, you are changing the same object that the refToMyHouse variable refers to. The following code example updates the number of bedrooms in the **House** object to three by using the myHouse reference. The **Console.WriteLine** statement that prints out the number of bedrooms displays the value 3 despite the fact that this statement uses the refToMyHouse reference, because both references refer to the same object.

```
myHouse.numberOfBedrooms = 3;
Console.WriteLine(refToMyHouse.numberOfBedrooms);
```

In the next example, myCurrency and mySecondCurrency are both Currency variables. The Currency variable is a value type. When you assign the myCurrency variable to the mySecondCurrency variable, the CLR creates a copy of the data and assigns it to the mySecondCurrency variable. The two variables do not refer to the same data in memory, so you can change the values in the myCurrency variable and the information in the mySecondCurrency variable will not change.

```
// Create a Currency object.
Currency myCurrency = new Currency("USD", "$");

// Create a second Currency object that is a copy of the first.
Currency mySecondCurrency = myCurrency;

myCurrency.currencyCode = "GBP";
Console.WriteLine(mySecondCurrency.currencyCode); // Displays "USD"
```



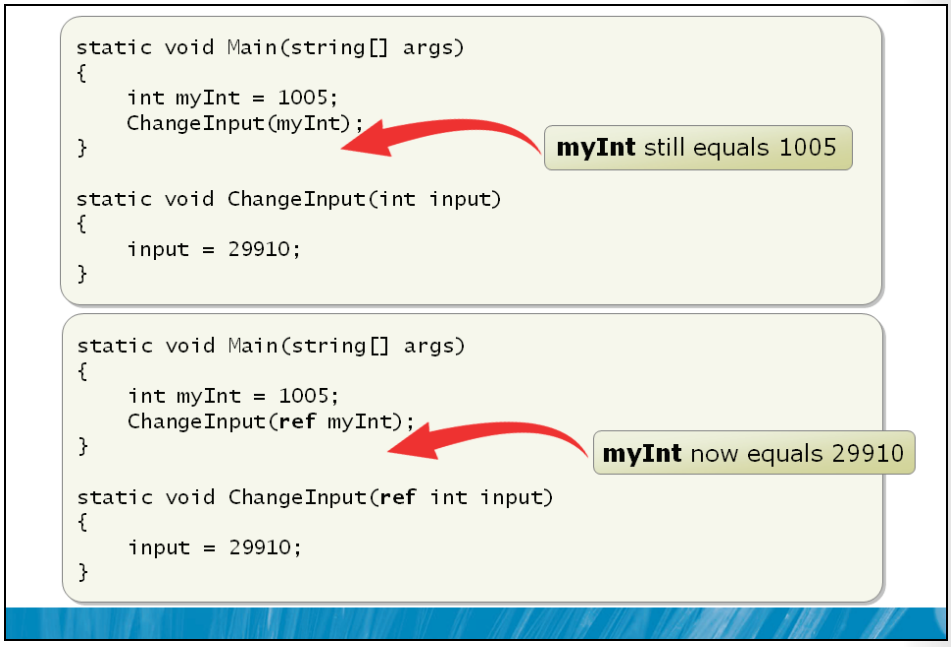
Note: Enumerations are also value types and follow the same copy behavior as structs.

Question: If **Residence** is a class (a reference type), what message does the following code example display?

```
Residence myHouse = new Residence(ResidenceType.House, 2);
Residence anotherHouse = new Residence(ResidenceType.House, 2);
if (myHouse == anotherHouse)
{
    Console.WriteLine("They are the same house");
}
else
{
    Console.WriteLine("They are different houses");
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Passing a Value Type by Reference into a Method



Key Points

The fundamental difference in behavior between value and reference types has a significant impact on what happens if you pass parameters of these types into a method.

For example, the following code example shows a method called **UpdateCurrency**. This method takes a *Currency* parameter (a value type) and changes the **currencyCode** field in this parameter.

```
public void UpdateCurrency(Currency currencyParam)
{
    currencyParam.currencyCode = "EUR";
}

...
Currency myCurrency = new Currency(...);
myCurrency.currencyCode = "USD";
UpdateCurrency(myCurrency);
Console.WriteLine(myCurrency.currencyCode);
```

MCT USE ONLY. STUDENT USE PROHIBITED

The code creates a Currency variable called myCurrency and assigns the **currencyCode** field of this variable to the value **"USD"** before calling the **UpdateCurrency** method. When the method is called, the expression **myCurrency** is evaluated, and the value of this expression is passed as the parameter to the **UpdateCurrency** method. Note that this value is a copy of the data in the myCurrency variable. Consequently, the **UpdateCurrency** method only changes the data in this copy. When the method completes, this copy is lost. The value in the myCurrency variable is unchanged, so the **Console.WriteLine** statement displays the string **"USD"**.

You can contrast this to what happens in the following code example when you pass a *Residence* parameter into a method (the **Residence** parameter is a reference type).

```
public void UpdateResidence(Residence residenceParam)
{
    residenceParam.numberOfBedrooms = 3;
}

...
// Create a two-bedroom house.
Residence myResidence = new Residence(ResidenceType.House, 2);
UpdateResidence(myResidence);
Console.WriteLine(myResidence.numberOfBedrooms);
```

In this case, when you call the **UpdateResidence** method, the expression **myResidence** is evaluated, and this is a reference to a **Residence** object. This reference is passed as the parameter to the **UpdateResidence** method. The parameter *residenceParam* and the variable myResidence both refer to the same **Residence** object in memory. Consequently, when the code in the **UpdateResidence** method modifies the **numberOfBedrooms** field in the parameter, it updates the same object that the myResidence variable references. When the method finishes, the **Console.WriteLine** statement displays the value 3.

Using the ref Keyword

If you want to pass a value parameter by reference into a method, you can use the **ref** keyword. The **ref** keyword causes the method to pass a reference to data into a method rather than passing a copy. This means any changes to that parameter in the method are made to the referenced object and will remain when the method has completed.

To use the **ref** keyword, you must do the following:

- Prefix the parameter in the method signature with the **ref** keyword.

MCT USE ONLY. STUDENT USE PROHIBITED

- Prefix the object in the method call with the **ref** keyword.

The following code example shows how to use the **ref** keyword with the **Currency** variable value type.

```
public void UpdateCurrency(ref Currency currencyParam)
{
    currencyParam.currencyCode = "EUR";
}

...
Currency myCurrency = new Currency(...);
myCurrency.currencyCode = "USD";
UpdateCurrency(ref myCurrency);
Console.WriteLine(myCurrency.currencyCode);
```

This time, the **UpdateCurrency** method takes a reference to a Currency variable. The argument that is passed in is a reference to the myCurrency variable. In the method, the **currencyParam** parameter refers to the myCurrency variable (it is not a copy), and any changes made through this reference modify the data in the myCurrency variable. When the method finishes, the **Console.WriteLine** statement displays the value "**EUR**".

Question: In the following code example, what is the value of the myString variable after the **ChangeInput** method completes?

```
class Program
{
    static void Main(string[] args)
    {
        string myString = "Original value";

        ChangeInput(myString);
    }

    static void ChangeInput(string input)
    {
        input = "Changed value";
    }
}
```

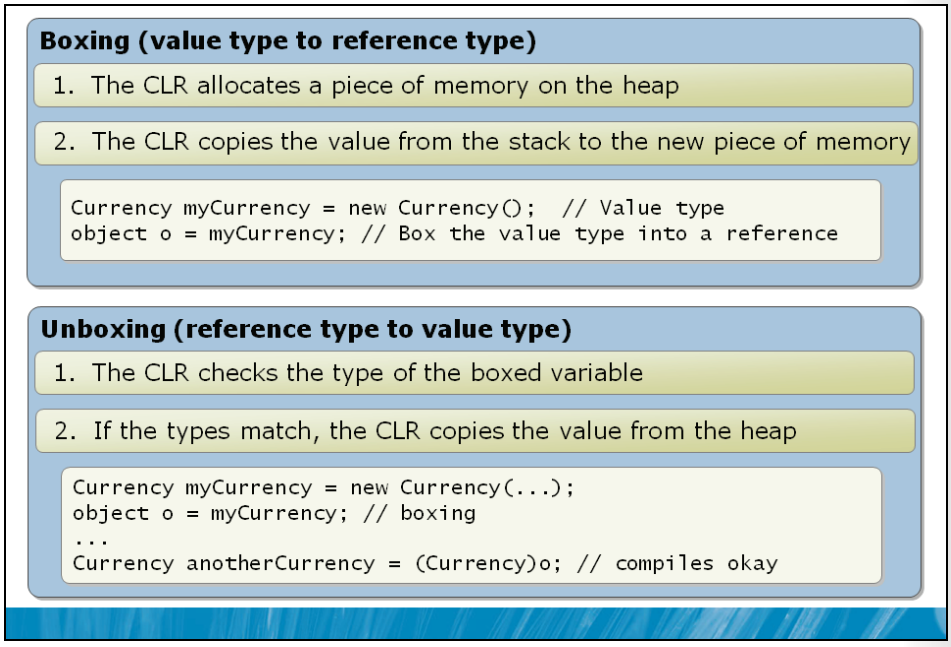
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about using the **ref** keyword, see the ref (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192928.

MCT USE ONLY. STUDENT USE PROHIBITED

Boxing and Unboxing



Key Points

Reference types refer to *objects* and value types hold *values*. The C# language defines a special type called **object** that you can use to refer to any type, as the following code example shows.

```
Residence myHouse = new Residence(...);
object  obj = myHouse;
```

The **object** type is useful if you want to define methods that can take parameters of different types, and you do not know in advance what those types are. For example, the collection classes in the .NET Framework class library enable you to build collections of objects of almost any type, and the methods that these classes define use the **object** type. You will see more about the collection classes in a later module.

Note: The **object** type is an alias for the **System.Object** class. This class underpins the entire type system that the .NET Framework implements; all data types are really just

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

specialized versions of the **object** type. The purpose of the **System.Object** class and how it relates to other types is described in more detail in a later module.

In some cases, you may need to convert a value type to a reference type, such as **object**. You can achieve this quite simply, as the following code example shows.

```
Currency myCurrency = new Currency();  // Value type.
object o = myCurrency;         // Box the value type into a reference.
```

The second statement requires a little explanation. Remember that the myCurrency variable is a value type that is created on the stack. If the reference inside the o variable referred directly to the myCurrency variable, the reference would refer to the stack. However, all references must refer to objects on the heap; creating references to items on the stack can seriously compromise the robustness of the CLR and create a potential security risk, so it is not allowed. Therefore, the CLR allocates a piece of memory from the heap, copies the value of the Currency variable myCurrency to this piece of memory, and then refers the object **o** to this copy. This automatic copying of an item from the stack to the heap is called *boxing*.

Because a variable of type **object** can refer to a boxed copy of a value, it is only reasonable to allow you to access that boxed value through the variable. You may expect to be able to access the boxed Currency variable value that a variable o refers to by using a simple assignment statement, as the following code example shows.

```
Currency anotherCurrency = o;
```

However, if you try this syntax, you will get a compile-time error. This is because the o variable could be referencing anything and not just a Currency variable value, as the following code example shows.

```
Residence myHouse = new Residence(...);
Currency myCurrency;
object o;
o = myHouse;    // o refers to a Residence
myCurrency = o; // what is stored in myCurrency?
```

To obtain the value of the boxed copy, you must use a cast. The cast causes the compiler to generate code that checks whether it is safe to convert the object variable into the specified type. The following code example shows how to use a cast in this scenario.

MCT USE ONLY. STUDENT USE PROHIBITED

```
Currency myCurrency = new Currency(...);
object o = myCurrency; // boxing
...
Currency anotherCurrency = (Currency)o; // compiles okay
```

If the compiler-generated code that checks the type of the object successfully determines that the o variable refers to a Currency variable value, this statement extracts the value from the boxed **Currency** object on the heap and copies it to the **anotherCurrency** object, which is held on the stack (it is a value type). This process is called unboxing. However, if o does not refer to a boxed **Currency** object, there is a type mismatch, which causes the cast to fail, and the compiler-generated code throws an **InvalidCastException** exception at run time.



Important: Boxing and unboxing only occur when you convert from a value type to a reference type (such as an object) and back again. If you convert from one reference type to another, no copies are made, and all that happens is that a new reference is created to the existing object on the heap.

Question: Is the following code an example of boxing or unboxing?

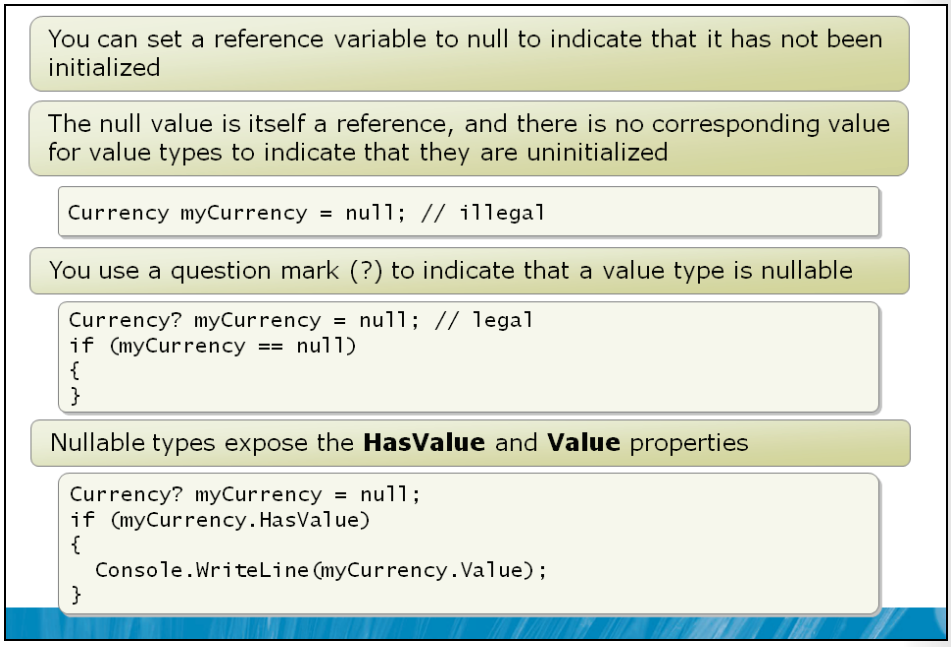
```
object amount = "1234";
int convertedAmount = (int)amount;
```

Additional Reading

For more information about boxing and unboxing, see the Boxing and Unboxing (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192929.

MCT USE ONLY. STUDENT USE PROHIBITED

Nullable Types



Key Points

When you create a reference variable, it is initially unassigned. You cannot use a reference variable until you have assigned it a value, but at the point in your code at which you declare the variable, you may not know what to initialize it to. In this case, you can set the reference variable to null to indicate that it has not been initialized. The null value is useful because you can explicitly check for it later in your code, and if a reference variable is null, you can initialize it by using the **new** operator, as the following code example shows.

```
Residence myHouse = null;
...
if (myHouse == null)
{
    myHouse = new Residence(...);
}
```

The null value is itself a reference. There is no corresponding value for value types. This can cause a problem in your code. For example, it may not be easy to determine whether a value type has been initialized (remember that if you try to

MCT USE ONLY. STUDENT USE PROHIBITED

pass an uninitialized value type into a method, your code will not compile). Because the null value is a reference, the statement that is shown in the following code example is illegal in C#.

```
Currency myCurrency = null; // Illegal
```

However, C# defines a modifier that you can use to declare that a variable is a nullable value type. A nullable value type behaves in a similar manner to the original value type, but you can assign the null value to it. You use a question mark (?) to indicate that a value type is nullable. Later in your application, you can ascertain whether a nullable variable contains null by testing it in the same way as a reference type, as the following code example shows.

```
Currency? myCurrency = null; // Legal
...
if (myCurrency == null)
{
    myCurrency = new myCurrency(...);
}
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following code examples are all legal (remember that **int** is a value type in C#).

```
int? i = null;
int j = 99;
i = 100; // Copy a value-type constant to a nullable type.
i = j;   // Copy a value-type variable to a nullable type.
```

You should note that the converse is not true. You cannot assign a nullable value to an ordinary value-type variable. So, given the definitions of variables i and j from the previous example, the statement that is shown in the following code example is not allowed.

```
j = i; // Illegal
```

This is because the variable i may contain null and j variable is a value type that cannot contain null. This also means that you cannot use a nullable variable as a parameter to a method that expects an ordinary value type.

Properties of Nullable Types

Nullable types expose a pair of properties that you can use to determine whether a nullable variable has a null value and what its value is:

MCT USE ONLY. STUDENT USE PROHIBITED

- **HasValue** .This is a Boolean property that indicates whether a nullable type contains a value or is null. If this property is **true**, the nullable variable has a value; if it is **false**, the nullable variable is null.
- **Value**. This is the value of a variable. You should only attempt to read this value if the **HasValue** property is **true**, otherwise your code will throw an exception.

The following code example shows how to use these properties with a nullable Currency variable.

```
Currency? myCurrency = null;
...
if (myCurrency.HasValue)
{
  Console.WriteLine(myCurrency.Value);
}
```



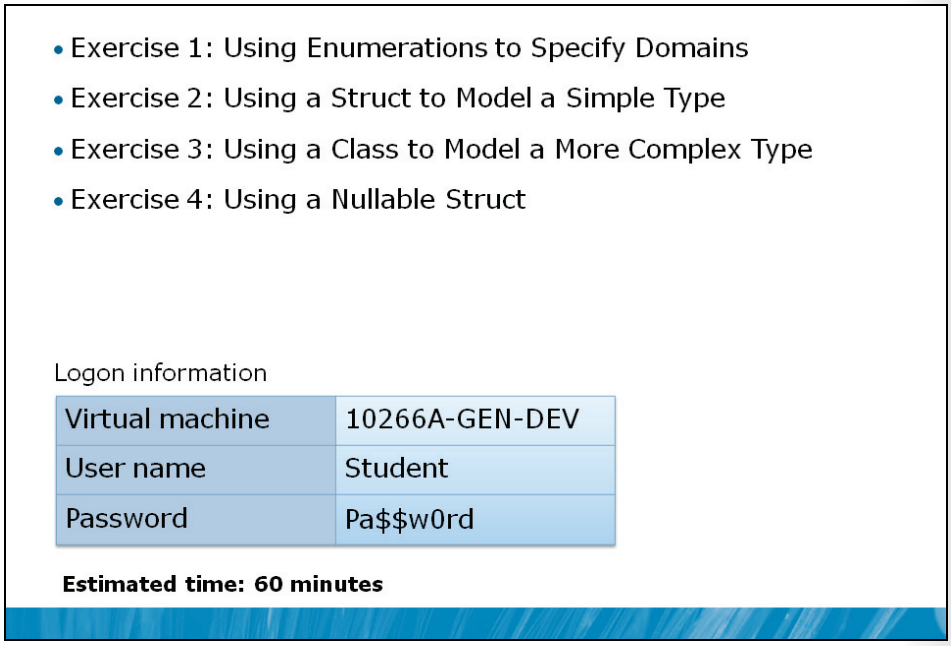
Note: The **Value** property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

Question: What is wrong with the following code?

```
int amount = null;
if (amount != null)
{
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Creating New Types



Objectives

After completing this lab, you will be able to:

- Use enumerations to specify domains.
- Use a struct to model a simple type.
- Use a class to model a more complex type.
- Use a nullable struct.

Introduction

In this lab, you will define an enumeration and then use this type to create variables. You will also define a struct. Finally, you will define a class and use the struct as the type of a data member in the class.

MCT USE ONLY. STUDENT USE PROHIBITED

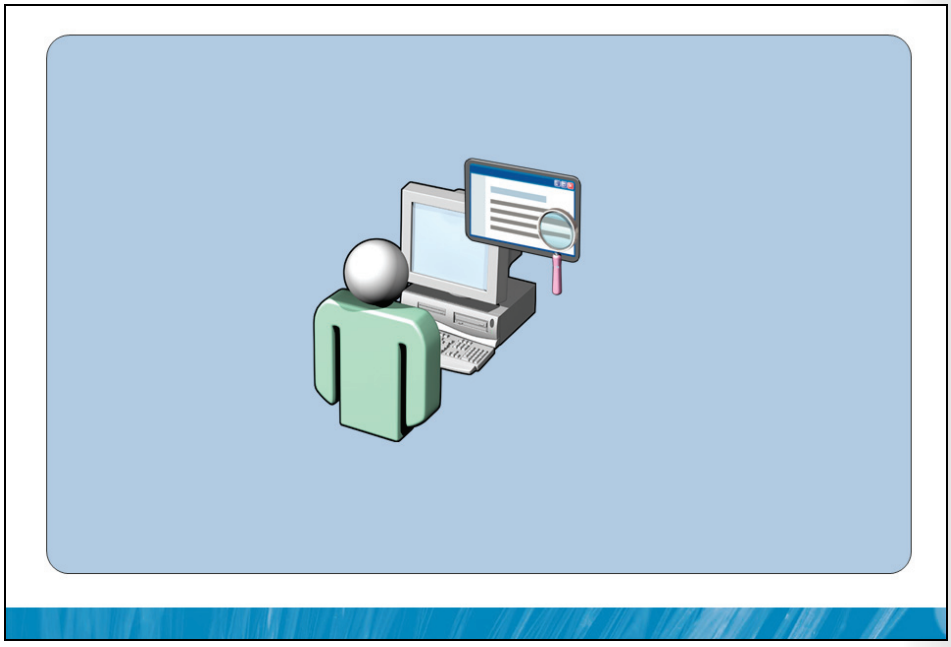
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

You are building an application that supports a machine that stress-tests girders for constructing high-rise buildings, bridges, and other critical structures.

Exercise 1: Using Enumerations to Specify Domains

In this exercise, you will define enumerations that represent different materials under stress (stainless steel, aluminum, reinforced concrete, and titanium) and the cross-section of the girders (I-Beam, Box, Z-Shaped, and C-Shaped). You will also define another enumeration called **TestResult** that represents the results of a stress test.

The main tasks for this exercise are as follows:

1. Open the Enumeration solution.
2. Add enumerations to the **StressTest** namespace.
3. Retrieve the enumeration values.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Display the selection results.
5. Test the solution.

▶ Task 1: Open the Enumerations solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Visual Studio 2010.
3. Open the Enumerations solution in the E:\Labfiles\Lab 6\Ex1\Starter folder.

▶ Task 2: Add enumerations to the StressTest namespace

1. Review the task list.
2. Locate the **TODO - Implement Material, CrossSection, and TestResult enumerations** task, and then double-click this task. This task is located in the StressTestType.cs file.
3. In the **StressTest** namespace, define a new enumeration named **Material**. The enumeration should have the following values:
   a. **StainlessSteel**
   b. **Aluminum**
   c. **ReinforcedConcrete**
   d. **Composite**
   e. **Titanium**
4. Below the **Material** enumeration, define a new enumeration named **CrossSection**. The enumeration should have the following values:
   a. **IBeam**
   b. **Box**
   c. **ZShaped**
   d. **CShaped**
5. Below the **CrossSection** enumeration, define a new enumeration named **TestResult**. The enumeration should have the following values:
   a. **Pass**

MCT USE ONLY. STUDENT USE PROHIBITED

b. **Fail**

6. Build the solution and correct any errors.

▶ Task 3: Retrieve the enumeration values

1. In the TestHarness project, display the MainWindow.xaml window.

   The purpose of the TestHarness project is to enable you to display the values from each of the enumerations. When the application runs, the three lists are populated with the values that are defined for each of the enumerations. The user can select an item from each list, and the application will construct a string from the corresponding enumerations.

2. In the task list, locate the **TODO - Retrieve user selections from the UI** task, and then double-click this task. This task is located in the **MainWindow.xaml.cs** class.

3. Remove the comment, and add code to the **selectionChanged** method to perform the following tasks:

   a. Create a **Material** object called **selectedMaterial** and initialize it to the value of the **SelectedItem** property in the **materials** list box.

   b. Create a **CrossSection** object called **selectedCrossSection** and initialize it to the value of the **SelectedItem** property in the **crosssections** list box.

   c. Create a **TestResult** object called **selectedTestResult** and initialize it to the value of the **SelectedItem** property in the **testresults** list box.



Hint: The **SelectedItem** property of a **ListBox** control has the **object** type. You must cast this property to the appropriate type when you assign it to an enumeration variable.

▶ Task 4: Display the selection results

1. In the **selectionChanged** method, after the code that you added in the previous task, add a statement to create a new **StringBuilder** object named **selectionStringBuilder**.

2. Add a **switch** statement to evaluate the selectedMaterial variable. In the **switch** statement, add **case** statements for each potential value of the **Material** enumeration. In each **case** statement, add code to append the text "Material: <*selectedMaterial*>, " to the **selectionStringBuilder** object. Substitute the text

MCT USE ONLY. STUDENT USE PROHIBITED

"<*selectedMaterial*>" in this string with the corresponding value for the selectedMaterial variable that is shown in the following table.

Material enumeration value	<***selectedMaterial***> string
Material.StainlessSteel	Stainless Steel
Material.Aluminum	Aluminum
Material.ReinforcedConcrete	Reinforced Concrete
Material.Composite	Composite
Material.Titanium	Titanium

3. Add another **switch** statement to evaluate the selectedCrossSection variable. In this **switch** statement, add **case** statements for each potential value of the **CrossSection** enumeration. In each **case** statement, add code to append the text "Cross-section: <*selectedCrossSection*>," to the **selectionStringBuilder** object. Substitute the text "<*selectedCrossSection*>" in this string with the corresponding value for the selectedCrossSection variable that is shown in the following table.

Material enumeration value	<***selectedCrossSection***> string
CrossSection.IBeam	I-Beam
CrossSection.Box	Box
CrossSection.ZShaped	Z-Shaped
CrossSection.CShaped	C-Shaped

4. Add a final **switch** statement to evaluate the **selectedTestResult** member. In the **switch** statement, add **case** statements for each potential value of the **TestResult** enumeration. In each **case** statement, add code to append the text "Result: <*selectedTestResult*>." to the **selectionStringBuilder** object. Substitute the text "<*selectedTestResult*>" in this string with the corresponding value for the selectedTestResult variable that is shown in the following table.

Material enumeration value	<***selectedTestResult***> string
TestResult.Pass	Pass

MCT USE ONLY. STUDENT USE PROHIBITED

Material enumeration value	*<selectedTestResult>* string
TestResult.Fail	Fail

5. At the end of the **selectionChanged** method, add code to display the string that is constructed by using the **selectionStringBuilder** object in the **Content** property of the **testDetails** label.

▶ Task 5: Test the solution

1. Build the application and correct any errors.
2. Run the application.
3. In the MainWindow window, in the **Material** list, click **Titanium**, in the **CrossSection** list, click **Box**, and then in the **Result** list, click **Fail**.

   At the bottom of the window, verify that the label updates with your selections.
4. Experiment by selecting further values from all three lists, and verify that with each change, the label updates to reflect the changes.
5. Close the application, and then return to Visual Studio.

Exercise 2: Using a Struct to Model a Simple Type

In this exercise, you will define a type called **TestCaseResult** that holds the result of a stress test. It will have the following public fields:

- **Result** : **TestResult**
- **ReasonForFailure**: **string**

This type is small, so it is best implemented as a struct. You will provide a constructor that initializes these fields.

The main tasks for this exercise are as follows:

1. Open the Structures solution.
2. Add the **TestCaseResult** structure.
3. Add an array of **TestCaseResult** objects to the user interface project.
4. Fill the **results** array with data.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Display the array contents.
6. Test the solution.

▶ Task 1: Open the Structures solution

- Open the Structures solution in the E:\Labfiles\Lab 6\Ex2\Starter folder.

▶ Task 2: Add the TestCaseResult structure

1. Review the task list:
2. In the task list, locate the **TODO - Declare a Structure** task, and then double-click this task. This task is located in the StressTestTypes.cs file.
3. Delete the comment, and then declare a new structure named **TestCaseResult**. In the **TestCaseResult** structure, add the following members:
   a. A **TestResult** object named **Result**.
   b. A **string** object named **ReasonForFailure**.

▶ Task 3: Add an array of TestCaseResult objects to the user interface project

1. In the TestHarness project, display the MainWindow.xaml window.

   This project simulates running stress tests and displays the results. It tracks the number of successful and failed tests, and for each failed test, it displays the reason for the failure.
2. In the task list, locate the **TODO - Declare a TestCaseResult array** task, and then double-click this task.
3. Remove the comment, and then declare a new array of **TestCaseResult** objects named **results**.

▶ Task 4: Fill the results array with data

1. In the **RunTests_Click** method, after the statement that clears the **reasonsList** list, add code to initialize the **results** array. Set the array length to **10**.
2. Below the statement that creates the array, add code that iterates through the items in the array and populates each one with the value that the static

MCT USE ONLY. STUDENT USE PROHIBITED

GenerateResult method of the **TestManager** class returns. The **GenerateResult** method simulates running a stress test and returns a **TestCaseResult** object that contains the result of the test and the reason for any failure.

▶ Task 5: Display the array contents

- Locate the comment **TODO - Display the TestCaseResult data**. Delete the comment, and then add code that iterates through the **results** array. For each value in the array, perform the following tasks:

  a. Evaluate the **result** value. If the **result** value is **TestResult**.**Pass**, increment the **passCount** value.

  b. If the **result** value is **TestResult.Fail**, increment the **failCount** value, and add the **ReasonForFailure** string to the **reasonsList** list box that is displayed in the window.



Note: To add an item to a list box, you use the **ListBox**.**Items**.**Add** method and pass the item to add to the list as a parameter to the method.

▶ Task 6: Test the solution

1. Build the application and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Tests**.

   Verify that the **Successes** and **Failures** messages are displayed. Also verify that a message appears in the **Failures** list if failures occur.
4. Click **Run Tests** again to simulate running another batch of tests and display the results of these tests.
5. Close the application, and then return to Visual Studio.

Exercise 3: Using a Class to Model a More Complex Type

In this exercise, you will define another type called **StressTestCase** that represents a stress test case for a girder. This type will be more complex than the

MCT USE ONLY. STUDENT USE PROHIBITED

TestCaseResult struct and is best implemented as a class. The **StressTestCase** class will have the following public data members:

- **girderMaterial**: **MaterialType**
- **crossSection**: **CrossSection**
- **lengthInMm**: **int**
- **heightInMm**: **int**
- **widthInMm**: **int**
- **testCaseResult**: **TestCaseResult**

You will also define two constructors: a default constructor that initializes these fields (apart from **testCaseResult**) to default values and an overloaded constructor that enables a programmer to specify nondefault values. You will then add the following public methods to the class:

- **PerformStressTest**. This method will simulate performing a stress test and set the result to indicate whether the test passed or failed, together with a reason for failure.
- **GetStressTestResult**. This method will return the value of the **testCaseResult** field.
- **ToString**. This method will return a representation of the object as a string for display purposes.

The main tasks for this exercise are as follows:

1. Open the Classes solution.
2. Define the **StressTestCase** class.
3. Add a parameterized constructor and a default constructor to the class.
4. Add the **PerformStressTest** and **GetStressTestResult** methods to the class.
5. Override the **ToString** method to return a custom string representation
6. Create an array of **StressTestCase** objects.
7. Display the **StressTestCases** collection.
8. Test the solution.
9. Examine and run unit tests.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the Classes solution

- Open the Classes solution in the E:\Labfiles\Lab 6\Ex3\Starter folder.

▶ Task 2: Define the StressTestCase class

1. In the TestHarness project, display the MainWindow.xaml window.

   This project is an extended version of the test harness from the previous two exercises. In addition to simulating stress-test results, it displays the details of the girder under test.

2. Review the task list.
3. In the task list, locate the **TODO - Add the StressTestCase class** task, and then double-click this task.
4. Remove the comment, and then add code to declare a public class named **StressTestCase** with the following public members:
   a. A **Material** object named **GirderMaterial**.
   b. A **CrossSection** object named **CrossSection**.
   c. An integer named **LengthInMm**.
   d. An integer named **HeightInMm**.
   e. An integer named **WidthInMm**.
   f. A **TestCaseResult** object named **TestCaseResult**.

▶ Task 3: Add a parameterized constructor and a default constructor to the class

1. Below the member declarations, add a constructor for the **StressTestCase** class that accepts the following parameters:
   a. A **Material** object named **girderMaterial**.
   b. A **CrossSection** object named **crossSection**.
   c. An integer named **lengthInMm**.
   d. An integer named **heightInMm**.
   e. An integer named **widthInMm**.

In the constructor, add code to store the value for each parameter in the corresponding member.



Hint: In the constructor, to make it clear which items are member variables and which items are parameters, use the **this** keyword (which represents the current object) with all member variables.

2. Above the constructor, add a default constructor.



Hint: A default constructor is a constructor that accepts no parameters and implements functionality to create a default instance of a class.

In the default constructor, initialize the members of the **StressTestCase** object with default values by using the parameterized constructor and the data that are shown in the following table.

Parameter name	Value
girderMaterial	**Material.StainlessSteel**
crossSection	**CrossSection.IBeam**
lengthInMm	**4000**
heightInMm	**20**
widthInMm	**15**



Hint: Remember that you can invoke one constructor directly from another by using the syntax in the following code example.

```
public MyDefaultConstructor() : this(parameter1, parameter2, ...)
{
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Add the PerformStressTest and GetStressTestResult methods to the class

1. Below the class constructors, add code to declare a new method named **PerformStressTest**. The **PerformStressTest** method should take no parameters and should not return a value.

   This method will simulate performing a stress test and then populate a **StressTestCase** object with the details of the test.

2. In the **PerformStressTest** method, create an array of strings called **failureReasons** that contains the following values:

   a. **"Fracture detected"**

   b. **"Beam snapped"**

   c. **"Beam dimensions wrong"**

   d. **"Beam warped"**

   e. **"Other"**

3. Add a statement that invokes the **Next** method of the static **Rand** method of the **Utility** class. Pass the value **10** as a parameter.



Note: The **Utility.Rand.Next** method accepts an integer parameter and then returns a random integer value between zero and the value of the integer parameter. In this case, the method will return an integer between 0 and 9.

   If the value that the **Rand** method returns is 9, add code to perform the following tasks:

   a. Set the **TestCaseResult**.**Result** member value to **TestResult.Fail**.

   b. Invoke the **Utility**.**Rand**.**Next** method with a parameter value of 5. Store the result in a new integer member named **failureCode**.

   c. Set the **TestCaseResult.ReasonForFailure** value to the value in the **failureReasons** array that the **failureCode** value indicates.



Note: This code simulates a 10 percent chance of a test case failing. The **failureReasons** array contains five possible causes of failure, and this code selects one of these causes at random.

MCT USE ONLY. STUDENT USE PROHIBITED

4. If the **Rand** method returns a value other than 9, add code to set the **TestCaseResult.Result** member value to **TestResult.Pass**.
5. Below the **PerformStressTest** method, add a public method named **GetStressTestResult**, which accepts no parameters and returns a **TestCaseResult** object.
6. In the **GetStressTestResult** method, add code to return a reference to the **TestCaseResult** member.

▶ Task 5: Override the ToString method to return a custom string representation

1. Below the **GetStressTestResult** method, add the following public method named **ToString**.



Note: This overrides the **ToString** method that is inherited from the **object** type. You will see more about inheritance in a later module.

```
...
public class StressTestCase
{
    ...
    public override string ToString()
    {

    }
}
...
```

2. In the **ToString** method, add code to return a string with the format shown in the following code example, where each value in angle brackets is replaced with the corresponding member in the class.

```
Material: <girderMaterial>, CrossSection: <crossSection>, Length:
<lengthInMm>mm, Height: <heightInMm>mm, Width:<widthInMm>mm.
```



Hint: Use the **String.Format** method to build the string.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 6: Create an array of StressTestCase objects

1. In the task list, locate the **TODO - Create an array of sample StressTestCase objects** task, and then double-click this task. This task is located in the **MainWindow.xaml.cs** class.
2. Remove the comment, and add a private method named **CreateTestCases**. The **CreateTestCases** method should accept no parameters and return an array of **StressTestCase** objects.
3. In the **CreateTestCases** method, add code to create an array of **StressTestCase** objects named **stressTestCases**. The array should be able to hold 10 objects.
4. Add code to generate 10 **StressTestCase** objects, and store each of them in the **stressTestCases** array. Use the following table to determine the parameters to pass to the constructor for each instance.

Array position	Material	CrossSection	Length	Height	Width
0	Use default constructor				
1	**Material.Composite**	**CrossSection.CShaped**	3500	100	20
2	Use default constructor				
3	**Material.Aluminium**	**CrossSection.Box**	3500	100	20
4	Use default constructor				
5	**Material.Titanium**	**CrossSection.CShaped**	3600	150	20
6	**Material.Titanium**	**CrossSection.ZShaped**	4000	80	20
7	**Material.Titanium**	**CrossSection.Box**	5000	90	20
8	Use default constructor				
9	**Material.StainlessSteel**	**CrossSection.Box**	3500	100	20

5. At the end of the method, return the **stressTestCases** array.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 7: Display the StressTestCases collection

1. In the task list, locate the **TODO - Iterate through the StressTestCase samples displaying the results** task, and then double-click this task. This task is located in the **doTests_Click** method that runs when the user clicks **Run Stress Tests**.
2. Remove the comment, and then add code to invoke the **CreateTestCases** method. Store the result of the method call in a new array of **StressTestCase** objects named **stressTestCases**.
3. Add code to create a **StressTestCase** object named **currentTestCase** and a **TestCaseResult** object named **currentTestResult**. You will add code to instantiate these objects shortly.
4. Add code that iterates through the **StressTestCase** objects in the **stressTestCases** array. For each **StressTestCase** object, add code to perform the following tasks:
   a. Set the **currentTestCase** object to refer to the **StressTestCase** object.
   b. Invoke the **currentTestCase**.**PerformStressTest** method on the **currentTestCase** object.
   c. Add the **currentTestCase** object to the **testList** list box that is displayed in the window.
   d. Invoke the **currentTestCase**.**GetStressTestResult** method, and store the result in the **currentTestResult** object.
   e. Add a string to the **resultList** list box that is displayed in the window. This string should consist of the **currentTestResult**.**Result** value and the **currentTestResult**.**ReasonForFailure** message.

▶ Task 8: Test the solution

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**.

   Verify that the **Girder Tested** list contains a list of different girder compositions and the **Results** list contains a series of test results.
4. Click **Run Stress Tests** again. You should see a different set of results.
5. Close the application, and then return to Visual Studio

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 9: Examine and run unit tests

1. In the task list, locate the **TODO - Examine and Run Unit Tests** task, and then double-click this task. This task is located in the **StressTestCaseTest** class.
2. Examine the **StressTestCaseConstructorTest** method.

   This method uses the parameterized constructor to create a new **StressTestCase** object the uses defined values. The method then uses a series of **Assert** statements to ensure that the properties of the created object match the values that are passed to the constructor.
3. Examine the **StressTestCaseConstructorTest1** method.

   This method uses the default constructor to create a new **StressTestCase** object, passing no parameters. The method then uses a series of **Assert** statements to ensure that the properties of the created object match the intended default values.
4. Examine the **GetStressTestResultTest** method.

   This method creates a new **StressTestCase** object and then retrieves a **TestCaseResult** object by calling the **StressTestCase**.**GetStressTestResult** method. The test method then uses **Assert** statements to ensure that the **TestCaseResult.Result** and **TestCaseResult.ReasonForFailure** properties contain the expected values.
5. Examine the **PerformStressTestTest** method.

   This method creates a **StressTestCase** object, calls the **PerformStressTest** method, and then retrieves the **TestCaseResult** object. The method then checks that, if the test failed, the **TestCaseResult.ReasonForFailure** member contains some text. If the test passed, the method uses **Assert** statements to verify that the **ReasonForFailure** member contains no data. The method iterates 30 times.
6. Examine the **ToStringTest** method.

   This method creates a default **StressTestCase** object, and then verifies that the object's **ToString** method returns a string that contains the correct details.
7. Run all of the tests in the solution, and verify that all of the tests execute successfully.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Using a Nullable Struct

In this exercise, you will modify the constructor for the **StressTestCase** class to initialize the **testCaseResult** field to null (to indicate no result yet). However, you cannot set a value-type field or variable to a reference value such as null. Therefore, you will convert the **testCaseResult** field to a nullable field to support null values. You will then modify the methods in the **StressTestCase** class, and the test harness that invokes the **GetStressTestResult** method that displays the result of a test case, to dereference the value of this struct through the **Value** property.

The main tasks for this exercise are as follows:

1. Open the NullableStructs solution.
2. Modify the **TestCaseResult** field to make it nullable.
3. Modify the parameterized constructor to initialize the **TestCaseResult** member.
4. Modify the **PerformStressTest** method.
5. Modify the **GetStressTestResult** method.
6. Modify the **GetStressTestResult** method call.
7. Test the solution.
8. Update the unit tests.

▶ Task 1: Open the NullableStructs solution

- Open the NullableStructs solution in the E:\Labfiles\Lab 6\Ex4\Starter folder.

▶ Task 2: Modify the TestCaseResult field to make it nullable

1. Review the task list.
2. In the task list, locate the **TODO - Make TestCaseResult nullable** task, and then double-click this task. This task is located in the **StressTestTypes** class.
3. Remove the comment, and then modify the **TestCaseResult** member definition to allow it to store a null value.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Modify the parameterized constructor to initialize the TestCaseResult member

- In the **StressTestCase** parameterized constructor, remove the comment **TODO – Initialize TestCaseResult to null**, and then add code to initialize the **TestCaseResult** member to null.

▶ Task 4: Modify the PerformStressTest method

1. In the **PerformStressTest** method, remove the comment **TODO – Update the PerformStressTest method and work with the nullable type**, and then add code to declare a new TestCaseResult variable named currentTestCase.
2. Modify the **if** statement to perform the following tasks:
   a. In all instances, modify the **currentTestCase** object rather than the **TestCaseResult** member.
   b. At the end of the **if** block, assign the **currentTestCase** object to the **TestCaseResult** member.
3. Modify the **else** block to perform the following tasks:
   a. Modify the **currentTestCase** object rather than the **TestCaseResult** member.
   b. At the end of the **if** block, store the **currentTestCase** object in the **TestCaseResult** member.

▶ Task 5: Modify the GetStressTestResult method

- In the **GetStressTestResult** method, modify the method definition to return a nullable **TestCaseResult** value.

▶ Task 6: Modify the GetStressTestResult method call

1. In the task list, locate the **TODO - Modify call to GetStressTestResult method to handle nulls** task, and then double-click this task.
2. Remove the comment, and then modify the code to create a nullable **TestCaseResult** object named **currentTestResult**.
3. In the **for** block, after retrieving the value of the **currentTestResult** object from the **currentStressTest**.**GetStressTestResult** method, add code to check whether the **currentTestResult** object contains a value. If a value exists, add a

string that contains the **StressTestResult Result** and **ReasonForFailure** properties to the **resultList** list box.

▶ Task 7: Test the solution

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**.

   Verify that the application functions in the same way as before.
4. Close the application, and then return to Visual Studio.

▶ Task 8: Update the unit tests

1. In the task list, locate the **TODO - Examine and run unit tests updated to deal with nullable type** task, and then double-click this task. This task is located in the **StressTestCaseTest** class.



Note: Most of the test cases are identical to those in Exercise 3. The only changes are in the **GetStressTestResult** and **PerformStressTestTest** methods.

2. Examine the **GetStressTestResult** method.

   This method creates a new **StressTestCase** object. It then evaluates the **HasValue** property on the result of the **GetStressTestResult** method call to verify that property contains no value. The test then calls the **PerformStressTest** method, which generates a **TestCaseResult** value in the **StressTestCase** object. The test method again evaluates the **HasValue** property to verify that a value now exists.
3. Examine the changes to the **PerformStressTestTest** method.

   This method creates a **StressTestCase** object and then calls the **PerformStressTest** method on that object. The method calls the **GetStressTestResult** method on the **StressTestCase** object and stores the result in a local nullable **TestCaseResult** object. The method then uses an **Assert** statement to evaluate the **HasValue** property of the **TestCaseResult** object to verify that the result is not null. The method then evaluates the **Value** property of the **TestCaseResult** object to determine whether the result indicates that the stress test failed or passed. If the stress test failed, an **Assert**

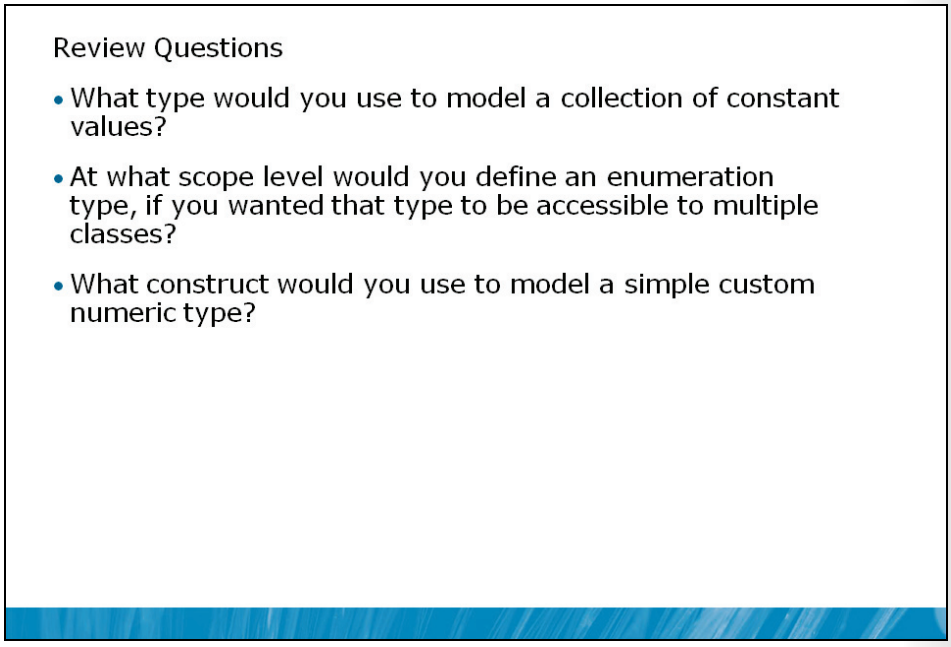
MCT USE ONLY. STUDENT USE PROHIBITED

statement is used to verify that the **ReasonForFailure** string contains a value. If the stress test passed, an **Assert** statement is used to verify that the **ReasonForFailure** string is null. The method iterates 30 times.

4. Run all of the tests in the solution, and verify that all of the tests execute successfully.
5. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

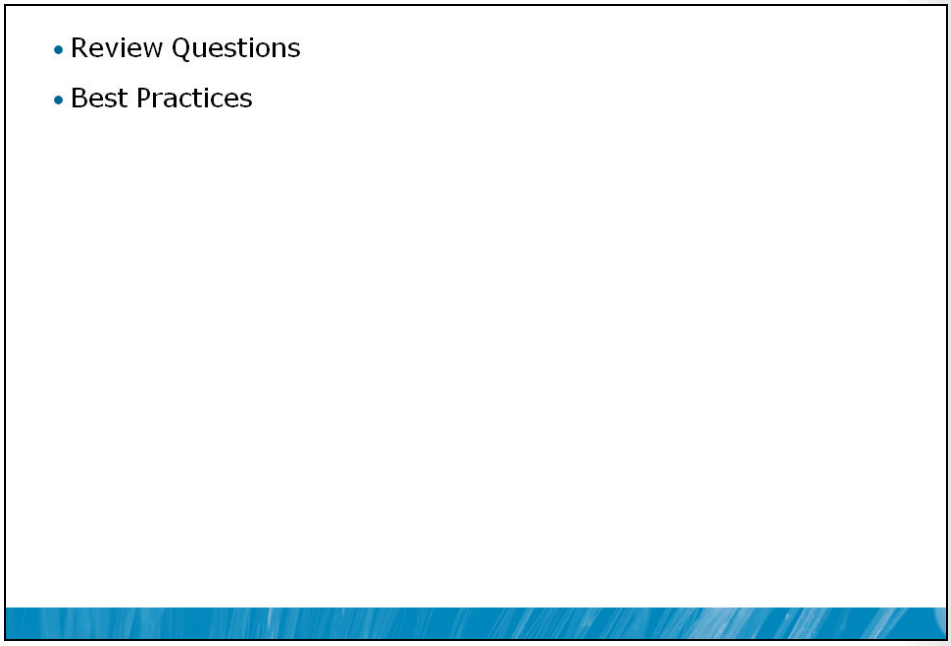


Review Questions

1. What type would you use to model a collection of constant values?
2. At what scope level would you define an enumeration type, if you wanted that type to be accessible to multiple classes?
3. What construct would you use to model a simple custom numeric type?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. When you define the first value in an enumeration, the value defaults to the index of zero. How can you change the default index?
2. What is a class?
3. What keyword can you use to split a class definition over multiple files?
4. Is a Boolean variable a value type or a reference type?
5. How can you pass a value type by reference into a method?
6. What is the process called when you explicitly convert a value type to a reference type?

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices Related to Creating and Using Types

Supplement or modify the following best practices for your own work situations:

- When you use a series of related constants, create an enumeration to encapsulate those constants into an object.
- Use structures to implement simple concepts whose main feature is their value. Also use structures for small data items where it is just as—or nearly as—efficient to copy the value as it would be to copy an address.
- Use classes for more complex data that is too big to copy efficiently.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 7

Encapsulating Data and Methods

Contents:

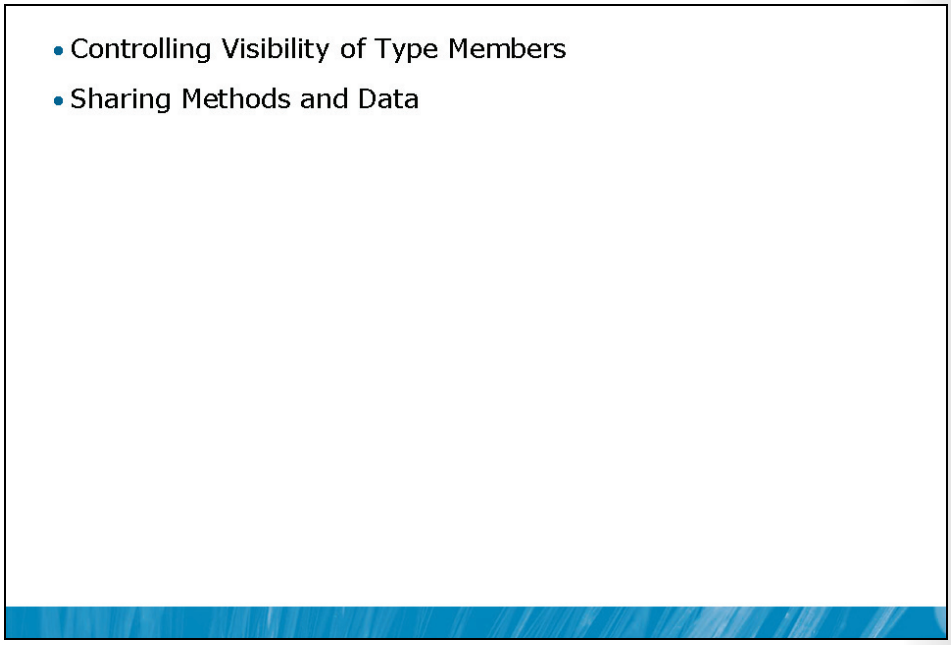
Lesson 1: Controlling Visibility of Type Members 7-4

Lesson 2: Sharing Methods and Data 7-15

Lab: Encapsulating Data and Methods 7-29

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



Previous modules have shown you how to add functionality to Microsoft® .NET Framework applications by using existing types and creating your own types. When you create your own types, you will rarely want to expose all of the members in that type, because some of these members may represent helper functions and internal state data that are not relevant to other classes. Typically, your types should expose only the fields and methods that other types can use.

In addition, all types to this point have been instance types, in that they model data and behavior for a specific instance of a type. Sometimes it is useful to define shared data and behavior that spans all of the instances of a type.

This module describes how to use some of the access modifiers that C# provides to enable you to implement encapsulation. This module also introduces the **static** modifier, which enables you to define members that can be shared over multiple instances of the same type.

MCT USE ONLY. STUDENT USE PROHIBITED

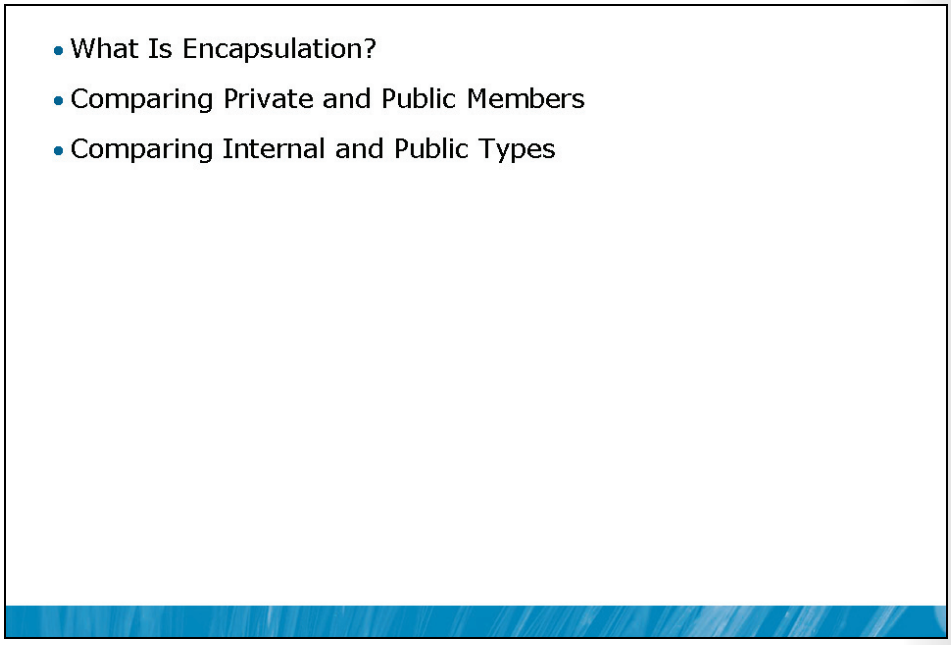
Objectives

After completing this module, you will be able to:

- Describe how to control the visibility of type members.
- Describe how to share methods and data.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Controlling Visibility of Type Members



Encapsulation is a fundamental object-oriented principle. Encapsulation is the ability to hide the private data and inner workings of a type so that it cannot be accessed by code that is defined in other types. A type can expose public members that define its behavior to the outside world, but it should keep the implementation of this behavior private. This feature isolates the way in which a type operates from the applications that use it and can reduce the scope for any inadvertent dependencies that may otherwise occur.

This lesson explains how to use access modifiers to control the visibility of types and members in types.

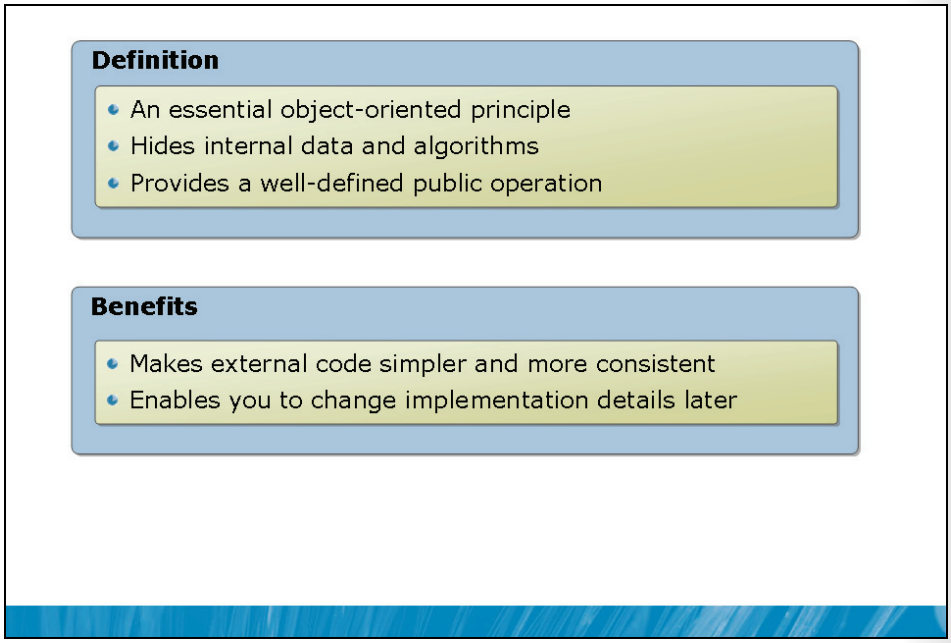
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of encapsulation.
- Describe the difference between creating public and private members.
- Describe the difference between creating internal and public types.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Encapsulation?



Key Points

All applications manipulate data. Many older legacy applications built by using programming languages that are not object-oriented, such as C and COBOL, typically separate the code that processes data from the code that manages and stores this data. Data management code was frequently provided in the form of code libraries, and the data-processing elements of an application were frequently built into the business logic of the application. Several applications that implement different parts of a business system might all use the same data that they access through a code library, but perform their own operations on that data.

However, in this scenario, if the format of the data changes, the code library that manages the data may also need to change. All of the applications that use the original code library may need to be refreshed to use the new version, and the logic in these applications may also need to change to handle the new structure of the data. Such changes can be difficult to perform (or even locate), and any applications that are not updated correctly may exhibit bugs and generate errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Encapsulation?

Encapsulation is the ability of a type to hide its internal data and implementation details, making only specific parts of the type accessible to applications.

Encapsulation is an essential object-oriented principle. For example, when you define a class, you should always define the fields as private members so that external code cannot access the fields directly. The only way for external code to interact with an object or class is through a well-defined set of public methods and properties.



Note: Properties provide a structured way to expose the ability to retrieve and set the values of private fields. Properties are covered in a later module in this course.

Benefits of Encapsulation

Encapsulation enables you to hide information. When you hide information, such as the internal state and implementation details of a type, the external code focuses on only the useful characteristics of the object. For example, the internal mechanism of a telephone is hidden from users; the wires, switches, and other internal parts of a telephone are encapsulated by its cover and are inaccessible to users. You may not have any idea about how a telephone works internally, but you can still use it.

In addition, when you hide the internal state of a type, client applications cannot modify or corrupt this state; this prevents changes that can cause the type to malfunction and produce unexpected results.

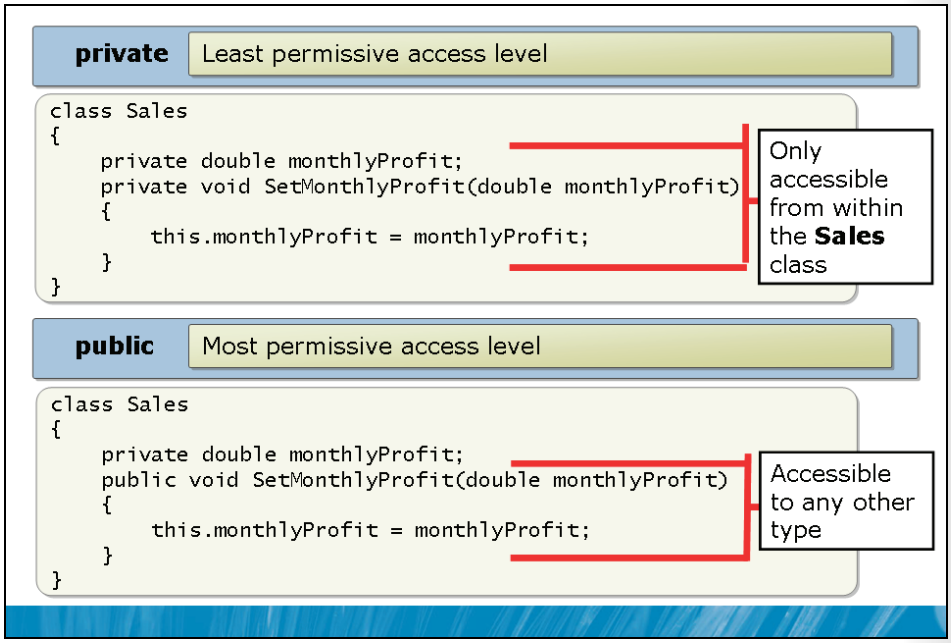
By using encapsulation, you can also easily change the implementation details of your type; applications that use your type do not need to be rewritten. As long as the public methods and properties that a class exposes do not change, any existing client applications should still work correctly.

Question: Discuss your experiences of encapsulation with other students.

Additional Reading

For more information about encapsulation, see the Classes and Structs (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192935.

Comparing Private and Public Members



Key Points

C# provides keywords known as *access modifiers*, which enable you to specify the access level for types and their members. You use these access modifiers to hide data and methods from applications that consume a type to implement encapsulation.

C# provides several different access modifiers that provide varying degrees of protection. Some of these modifiers apply to types, and others apply to members of a type. This topic focuses on how to use the **private** and **public** access modifiers to hide and expose members.

Using the Private and Public Modifiers with Type Members

If you do not specify an access modifier for a member in a type, the default access level for the member is **private**. The member can only be accessed by code in the type and is not visible to other types.

The **Sales** class definition in the following code example contains a private field called **monthlyProfit**, a private method called **SetMonthlyProfit**, and a private method called **GetAnnualProfitForecast**.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: To use an access modifier, prefix the member declaration with the access modifier that you want to use.

```
class Sales

{
    private double monthlyProfit;

    private void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }

    private double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }

}
```

All members are declared as private; therefore, they are only accessible to other members in the **Sales** class. The **SetMonthlyProfit** and **GetAnnualProfitForecast** methods have access to the **monthlyProfit** field because the field is declared at class level and therefore is in scope. If you try to use a member that you do not have access to, you will get a compile error.

By using the **private** access modifier, you can protect the implementation and state of a type from consuming types and thus encapsulate data. However, types are not very useful if they do not expose any members for other types to consume. For example, the **Sales** class definition in the preceding code example is unusable because there is no entry point into the class; no class can access its data or invoke any of its functionality. To expose members to other types, you can use the **public** access modifier.

In contrast to the **private** access modifier, the **public** access modifier is the most permissive access level and does not impose any restrictions. If you declare a member as **public**, any other type can access that member.

Using the example of the **Sales** class, if you declare the **SetMonthlyProfit** and **GetAnnualProfitForecast** methods as **public**, other types can then invoke these methods. The following code example shows that the **Program** class can now invoke the **SetMonthlyProfit** and **GetAnnualProfitForecast** methods.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Sales

{

    private double monthlyProfit;

    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }

    public double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }

}

class Program

{

    static void Main()
    {
        Sales companySales = new Sales();
        companySales.SetMonthlyProfit(3400);
        Console.WriteLine(companySales.GetAnnualProfitForecast());
    }

}
```

When you name private and public members, it is important to adopt a consistent naming convention. This course uses camel case for private fields and Pascal case for public methods. All methods use PascalCase, whether they are public or private. Your organization may have its own naming conventions that differ from this.

Question: You have created a class called **Product** to encapsulate information about the products that your organization sells. The following code example shows the definition of this class. You want to use this class in an application that creates **Product** objects and displays their details. What is the main problem with the **Product** class that may mean that a client application cannot use the **Product** type in this way?

MCT USE ONLY. STUDENT USE PROHIBITED

```
public class Product
{
    // Make these fields private so that an application cannot change
    // them after the Product object has been created.

    private int productID;
    private string productName;

    // Public methods that a client application can use to
    // get the product ID and name.
    public int GetProductID()
    {
        return this.productID;
    }
    public string GetProductName()
    {
        return this.productName;
    }

    // Provide a constructor to enable a client application
    // to create a Product object.
    Product(int ID, string name)
    {
        this.productID = ID;
        this.productName = name;
    }
}
```

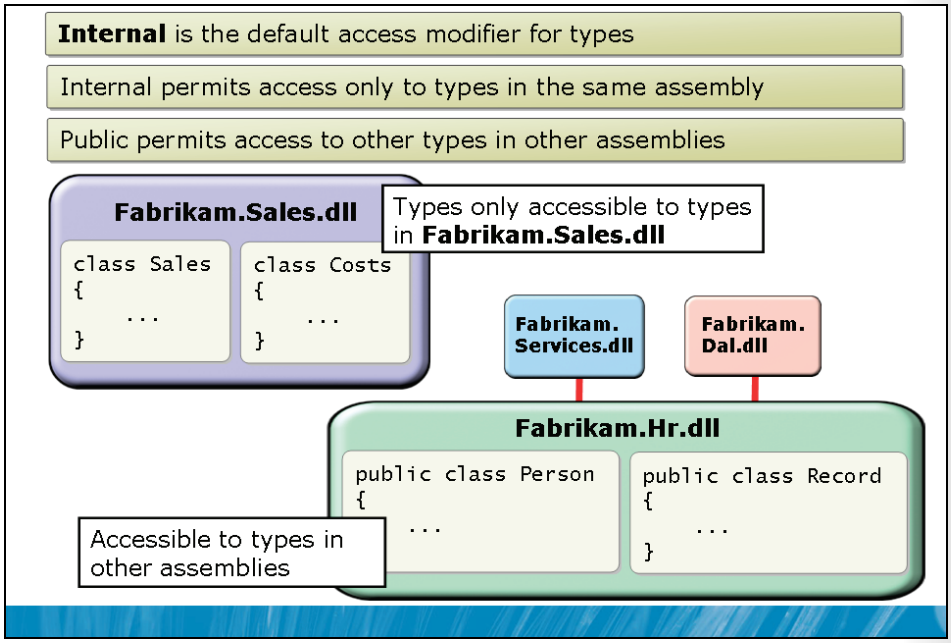
Additional Reading

For more information about the private access modifier, see the private (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192930.

For more information about the public access modifier, see the public (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192931.

MCT USE ONLY. STUDENT USE PROHIBITED

Comparing Internal and Public Types



Key Points

You have already seen how to use access modifiers to show and hide members in types; however, you can also use access modifiers to show and hide types themselves.

Note: This topic focuses primarily on access modifiers for class definitions, although the same principles can be applied to any type, whether it is a class, a structure, or an enumeration.

Internal Types

If you do not specify an access modifier for a type, the default access level applied for the type is **internal**. The **internal** access modifier restricts visibility to only code in types that are defined in the same assembly. The **public** access modifier makes the type available to code in all types.

The **Sales** class definition in the following code example does not explicitly define an access modifier, so the **internal** access modifier is implicitly assigned.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Sales
{
    private double monthlyProfit;

    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
    public double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }
}
```

Other types in the same assembly can now access the **Sales** class and use any exposed members.

You can also use internal members in a type. Internal methods are available to other types that are part of the same assembly, but are inaccessible to types that are defined in other assemblies. The same rules apply to internal fields and properties.

Public Types

The .NET Framework organizes types into assemblies. An assembly may represent the code for an application, or it may contain a library of types and data that applications can use. The .NET Framework class library contains several assemblies with a large number of reusable types. For example, the **System.IO** assembly in the .NET Framework class library provides the necessary functionality to interact with the file system.

These assemblies would not be much use if they only exposed internal types, because no other assembly would be able to access their functionality. Therefore, when developers build a reusable application programming interface (API), it is common to define **public** types.

The following code example shows how the **Sales** class definition has been explicitly assigned the **public** access modifier. Other types in other assemblies can now access the **Sales** class.

```
public class Sales
{
    private double monthlyProfit;

    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    public double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }
}
```

Private Types

You can also declare types as **private**. You can only define a type as **private** if it is nested within another type. It is most frequently used to define private enumerations, although you can define classes and structs that you want to use in your own code, but do not want to expose to other types.

The following code example shows the public **Sales** class, which contains a private **Revenue** structure. The **Revenue** structure is encapsulated by the **Sales** class and is not accessible directly.

```
public class Sales
{
    private Revenue salesRevenue;

    public void SetRevenue(string currency, double amount)
    {
        this.salesRevenue = new Revenue(currency, amount);
    }

    private struct Revenue
    {
        string currency;
        double amount;

        public Revenue(string currency, double amount)
        {
            this.currency = currency;
            this.amount = amount;
        }
    }
}
```

Question: In the **Revenue** structure that is shown in the preceding code example, the constructor is defined as **public** although the type is defined as **private**. Does this mean that a type other than the **Sales** type can invoke this constructor?

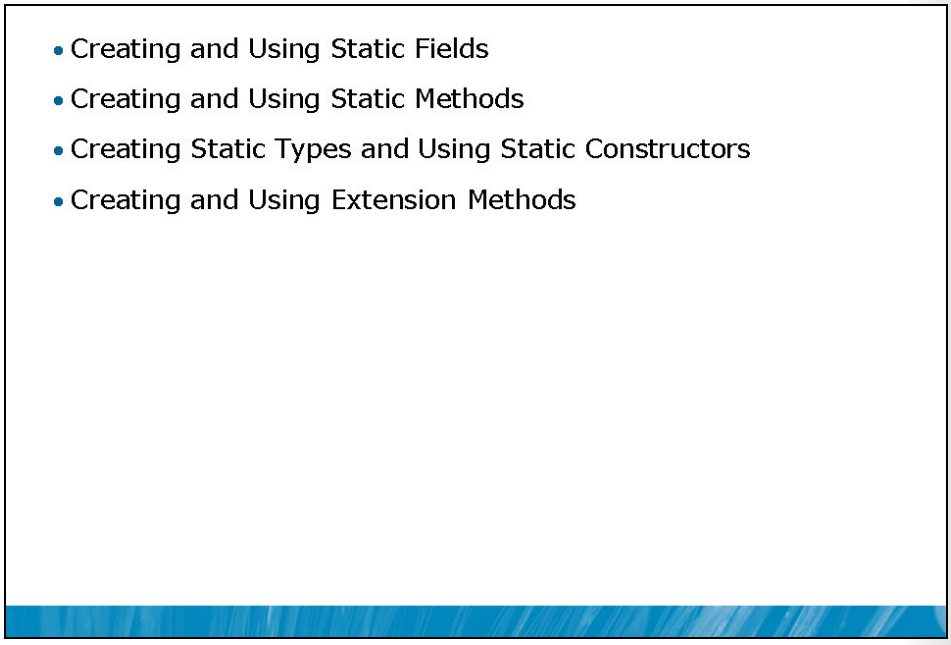
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about the internal access modifier, see the internal (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192932.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Sharing Methods and Data



This lesson introduces the concept of static types and members, which you can use to implement singleton types. This lesson also explains how to extend existing types by using extension methods.

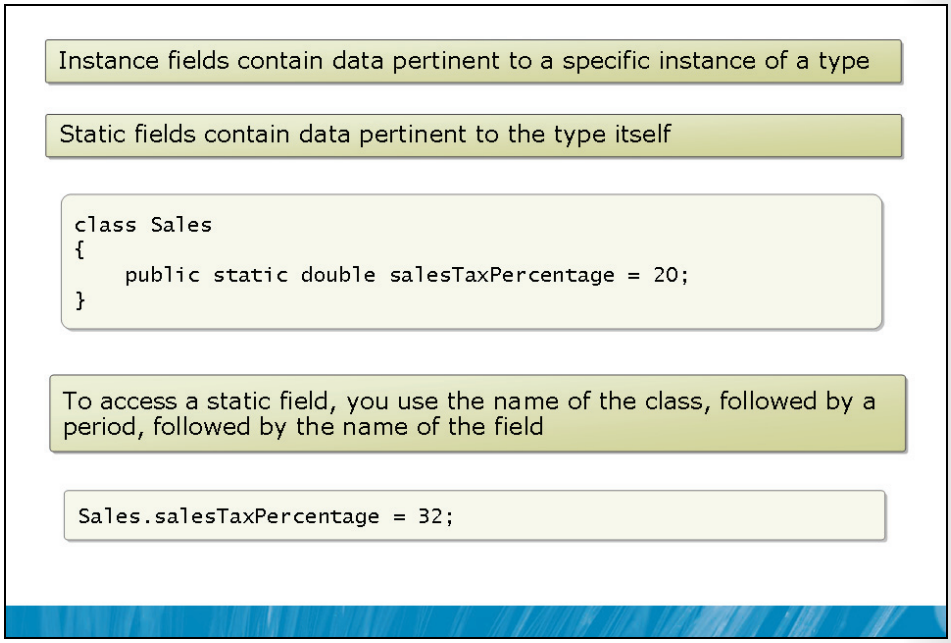
Objectives

After completing this lesson, you will be able to:

- Describe how to create and use static fields.
- Describe how to create and use static methods.
- Describe how to create and use static types and constructors.
- Describe how to create and use extension methods.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Using Static Fields



Key Points

Until this point, the primary focus has been on how to create and use instance members in types. Instance members typically contain data and implement functionality that is pertinent to a specific instance of a type. When you create an instance of a type, memory is allocated for each field, enabling the instance to hold unique data. For example, in the following code example, the **sales2010** object is an instance of the **Sales** class, and any data that is stored in the **sales2010** object is only accessible through the **sales2010** instance. If you create a second instance of the **Sales** class called **sales2011**, it cannot access any of the instance data in the **sales2010** object, and the **sales2010** object cannot access its instance data.

The following code example shows how data is not shared across instances of the **Sales** class.

```
Sales sales2010 = new Sales();
sales2010.SetMonthlyProfit(34672);
Console.WriteLine(sales2010.GetAnnualProfitForecast());
Sales sales2011 = new Sales();
sales2011.SetMonthlyProfit(98675);
Console.WriteLine(sales2011.GetAnnualProfitForecast());
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
class Sales
{
    private double monthlyProfit;

    public void SetMonthlyProfit(double monthlyProfit)
    {
        this.monthlyProfit = monthlyProfit;
    }

    public double GetAnnualProfitForecast()
    {
        return (this.monthlyProfit * 12);
    }
}
```

Alternatively, static fields do not belong to an instance of a type; they belong to the type itself. Memory is allocated independently of any instance, and all references to a static field refer to the same piece of memory. You do not need to create an instance of a type to access a static member of that type. The member is created and the memory is allocated the first time that you reference it.

Using Static Fields

To create a static field, you must use the **static** modifier in the declaration for that field. The following code example shows how to declare a static field called **salesTaxPercentage** and assign the field the value **20**.

```
class Sales
{
    public static double salesTaxPercentage = 20;
}
```

You can initialize a static field in the declaration, as the preceding code example shows, or if the field is public, you can access the field from another type. You do not need to create an instance of the **Sales** type to access the **salesTaxPercentage** field because the field is static. Instead, you access the field directly on the **Sales** type by specifying the type name, followed by a period, followed by the field name, as the following code example shows.

```
Sales.salesTaxPercentage = 32;
```



Note: You can also initialize static fields from a constructor, which is covered in a later topic in this module.

MCT USE ONLY. STUDENT USE PROHIBITED

You cannot access static fields from an instance of the type, because static fields belong to the type itself. If you try, you will get a compile error that explains that the member cannot be accessed with an instance reference. For example, the following code example will generate a compile error.

```
Sales sales = new Sales();
sales.salesTaxPercentage = 23; // Compile error.

...

class Sales
{
    public static double salesTaxPercentage;
}
```

However, you can access static fields from instance methods and constructors, which enables you to share data with multiple instances of the same type. For example, the **User** class in the following code example represents a user who accesses the Fabrikam, Inc. Web site. Every time a user accesses the site, a new **User** object is created, and the static **usersOnline** field is incremented. Other instances of the class can access the data and increment it because the field is static; therefore, the **usersOnline** field will represent the total number of users online at that point in time.

```
class User
{

    internal static int usersOnline;

    internal User()
    {
        usersOnline++;
    }

}

User a = new User();
User b = new User();
User c = new User();
User d = new User();

int totalUsersOnline = User.usersOnline; // Returns the value 4.
```

If the **usersOnline** field was not static, data would not be shared, and the **usersOnline** field would only ever reach the value of 1, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class User
{
    internal int usersOnline;

    internal User()
    {
        usersOnline++;
    }
}

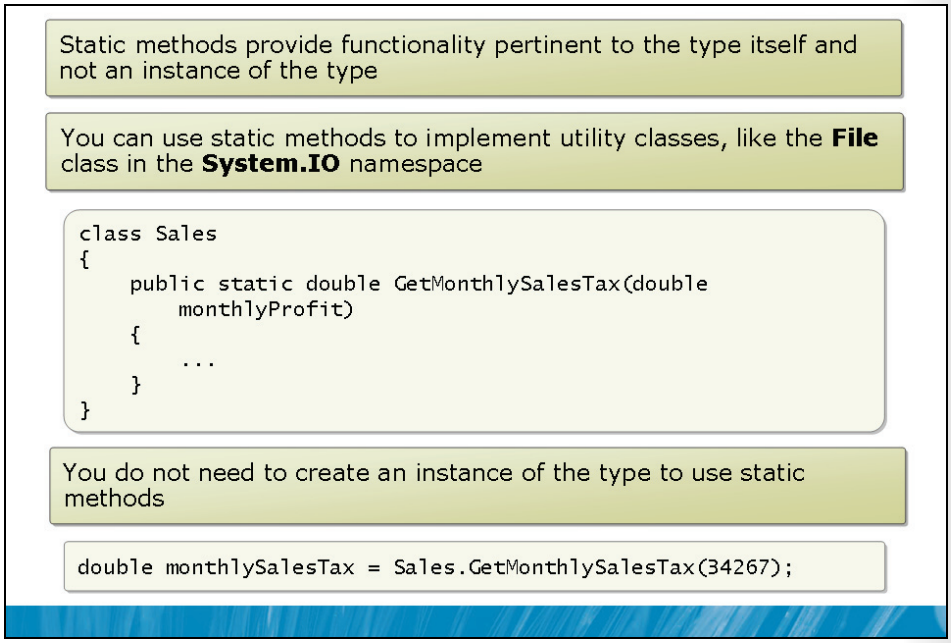
User a = new User();
User b = new User();
User c = new User();
User d = new User();

int totalUsersOnline = d.usersOnline; // Returns the value 1.
```

Question: What happens if you try to access a public static field through an instance of the type?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Using Static Methods



Key Points

You can also use the **static** modifier to create static methods. Static methods are very useful and are typically used in utility classes to perform atomic operations that do not rely on instance data. For example, the **File** class in the **System.IO** namespace contains several static methods, such as **Exists**, to perform atomic file operations.

Using Static Methods

To define a static method, you must prefix the method declaration with the **static** modifier, as the following code example shows.

```
class Sales
{
    public static double GetMonthlySalesTax(double monthlyProfit)
    {
        ...
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED



Note: It is common practice to place the static modifier after the access modifier, as the preceding code example shows, although you can place it before the access modifier. Your organization may have a preference.

If you define a static method, that method cannot reference any instance members that are declared in that class. Static methods can only reference other static members, but instance methods can access both instance and static members. Typically, instance methods use data that is stored in instance fields, which has been collected during the object's initialization and other method calls. Static methods can only use data that is stored in static fields and the data that is passed as parameters in the method's signature.

For example, the **Sales** class in the following code example exposes a **GetMonthlySalesTax** method that uses data that is passed by the *monthlyProfit* parameter and the private static **salesTaxPercentage** field.

```
class Sales
{
    private static double salesTaxPercentage = 20;
    public static double GetMonthlySalesTax(double monthlyProfit)
    {
        return (salesTaxPercentage * monthlyProfit) / 100;
    }
}
```

The syntax that you use to access a **static** method is different from the syntax that you use to access an instance method—you do not need to create an instance of the type.

To access a **static** method on a class, you use the name of the class, followed by a period, followed by the name of the method. For example, the following code example shows how to access the **GetMonthlySalesTax** method in the **Sales** class.

```
double monthlySalesTax = Sales.GetMonthlySalesTax(34267);
...
class Sales
{
    public static double salesTaxPercentage = 20;
    public static double GetMonthlySalesTax(double monthlyProfit)
    {
        return (salesTaxPercentage * monthlyProfit) / 100;
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Question: The **Person** class definition in the following code example contains functionality to calculate the number of years that a person must work before he or she reaches a set retirement age. The class does not compile; can you identify the problem?

```
class Person
{
    public int retirementAge;

    public Person(int retirementAge)
    {
        this.retirementAge = retirementAge;
    }

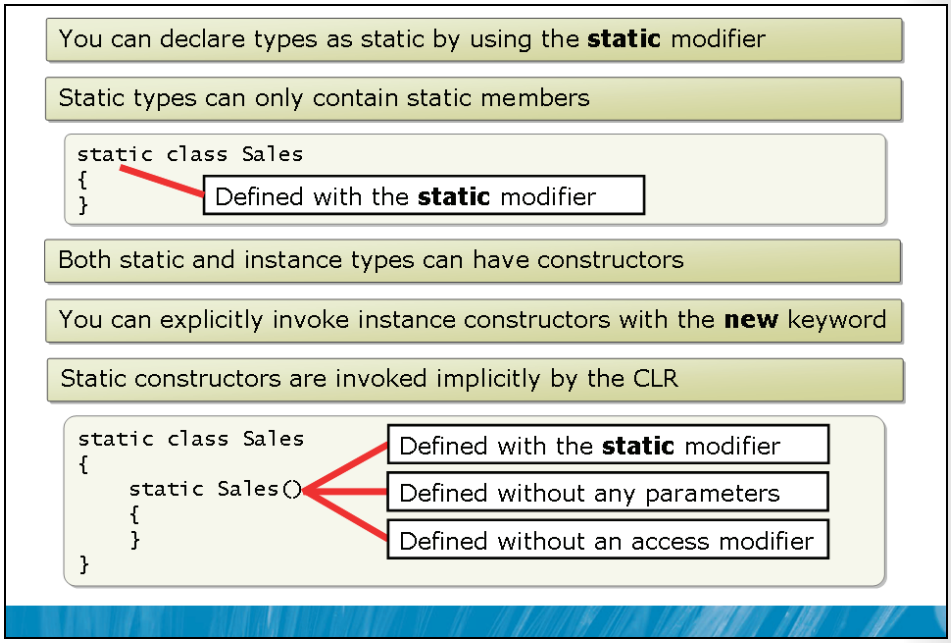
    public static int CalculateRemainingWorkYears(int age)
    {
        return (retirementAge - age);
    }
}
```

Additional Reading

For more information about the static modifier, see the static (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192933.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating Static Types and Using Static Constructors



Key Points

An instance type can contain both static and instance members, which at times can be very useful. However, if you develop a utility class that contains only static members, you can declare the type itself as static. To declare a type as static, you must use the **static** modifier in the type's declaration. The following code example shows how to create a static class.

```
static class Sales
{
    ...
}
```

If you declare a class as static, that class cannot contain instance members, such as fields, methods, and constructors. The class cannot contain an instance constructor; therefore, you cannot create objects of that type. However, static types can contain static constructors, which behave differently from instance constructors.

MCT USE ONLY. STUDENT USE PROHIBITED

When you use static types, you cannot explicitly invoke a constructor by using the **new** keyword. Typically, the common language runtime (CLR) implicitly invokes a static constructor before any code tries to access or invoke a static member in that type. You define a static constructor in a static type, as the following code example shows.

```
static class Sales
{
    static Sales()
    {

    }
}
```

When you use static constructors in types, you must follow some rules to avoid compilation errors. You must define only a single constructor that is prefixed with the **static** modifier. You cannot explicitly invoke a static constructor; therefore, the constructor does not need to be accessible outside the type, so it always uses the implicit **private** access modifier. When defining the signature for a static constructor, you cannot specify parameters. In addition, a static constructor can only reference other static members.

Implementing the Singleton Design Pattern

Static constructors have many uses; for example, you can define objects that adhere to the singleton software engineering pattern. The singleton pattern prescribes that your code should be restricted to creating one instance of a type. You then use the instance whenever you require an instance of that type in your application.

In the following code example, the **Sales** type implements the singleton pattern.

```
static class Sales
{
    static SaleData data = null;

    static Sales()
    {

        if (SaleData.WebServerConnectionExists())
        {
            data = SaleData.GetWebServerData();
        }
        else if (SaleData.LocalDatabaseConnectionExists())
        {
            data = SaleData.GetDatabaseData();
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        else
        {
            throw new NotSupportedException(
                "No data source could be found.");
        }
    }

    public static string[] GetAllSalesRegions()
    {
        throw new NotImplementedException();
    }
}
```

The **Sales** class contains a constructor that ensures that the **data** object is initialized before the **GetAllSalesRegions** method is invoked. The constructor achieves this with some conditional logic, which either initializes the **data** object or throws an exception.

A client application that consumes the **Sales** class can be certain that the **data** object has been initialized before it calls the **GetAllSalesRegions** method.

Question: Identify the errors in the class definition in the following code example.

```
static class Person
{
    int ageLimit;

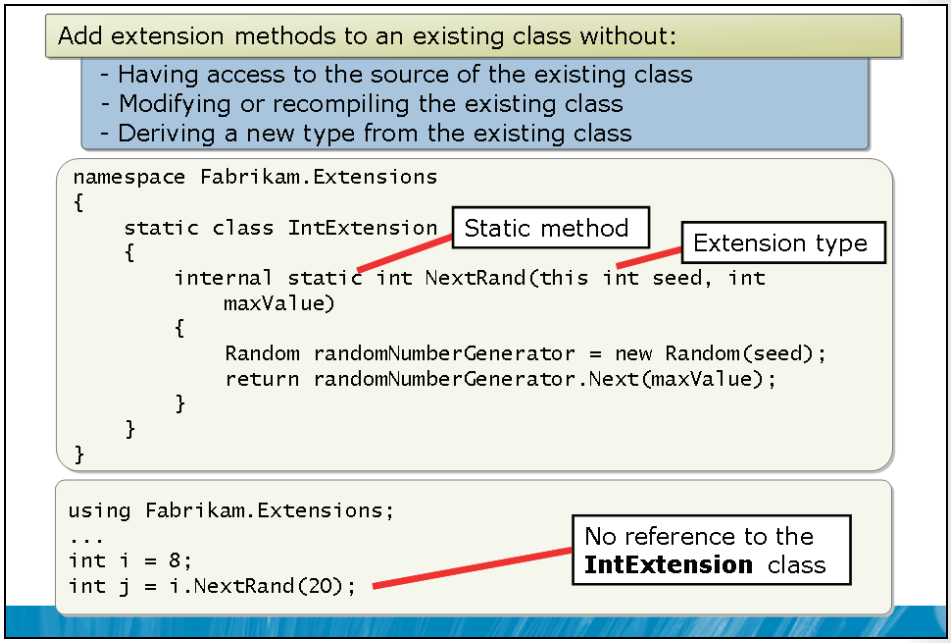
    private static Person(int ageLimit)
    {
        this.ageLimit = ageLimit;
    }

    public static string[] GetAllNames()
    {
        throw new NotImplementedException();
    }

}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Using Extension Methods



Key Points

The purpose of extension methods is to provide a way to extend existing types with your own custom functionality that does not affect or break other applications that already use these types. Before the introduction of extension methods, it was necessary to define a new class to wrap the existing functionality and provide the new required method. However, this can affect existing code and introduce breaking changes.

Defining Extension Methods

When you define an extension method, the first parameter in the method's signature indicates the type that is being extended and is prefixed with the **this** keyword to indicate that it is an extension method. At run time, the parameter is replaced with an instance of the type, and you can manipulate this instance in whatever way you require to implement the logic of the extension method.

It is common to define extension methods in static classes and create these classes in a specific namespace. However, extension methods must always be defined as static.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows how to define an extension method called **NextRand** for the **int** type. The method returns a random number that uses the existing **int** object as a seed. The method also accepts a parameter that represents a maximum value for the new random number.

```
namespace Fabrikam.Extensions
{
    static class IntExtension
    {
        internal static int NextRand(this int seed, int maxValue)
        {
            Random randomNumberGenerator = new Random(seed);
            return randomNumberGenerator.Next(maxValue);
        }
    }
}
```

Using Extension Methods

A client application uses an extension method in the same way that it uses a standard method; there is no visible difference. You can still invoke the method on an instance of an object.

The following code example shows how to invoke the **NextRand** method in the **Fabrikam.Extensions** namespace. Notice that the client code does not reference the **IntExtension** class that the **NextRand** method is declared in.

```
using Fabrikam.Extensions;

namespace Fabrikam.Client
{
    class Program
    {
        static void Main()
        {
            int i = 8;
            int j = i.NextRand(20);
        }
    }
}
```

Question: Briefly explain the difference between a standard method's signature and a signature for an extension method.

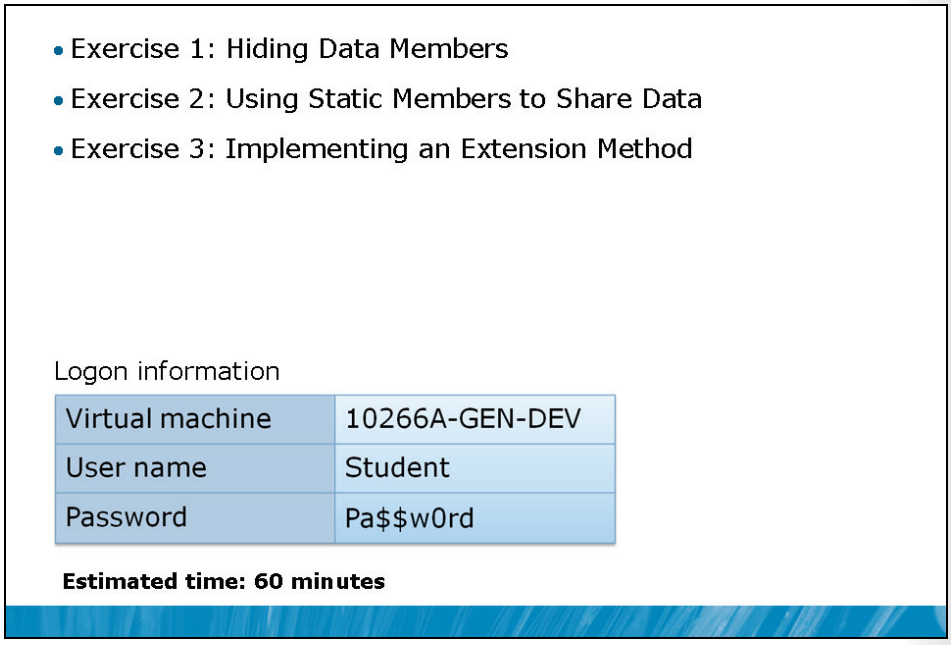
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about extension methods, see the Extension Methods (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192934.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Encapsulating Data and Methods



Objectives

After completing this lab, you will be able to:

- Hide data members in a type by using access modifiers.
- Use static members to share data in types.
- Use extension methods to add functionality to the **System.Int64** struct.

Introduction

In this lab, you will use encapsulation to hide information in a class. You will add static members and methods to a type to share data between instances of the type. Finally, you will add an extension method to a built-in type in the .NET Framework.

MCT USE ONLY. STUDENT USE PROHIBITED

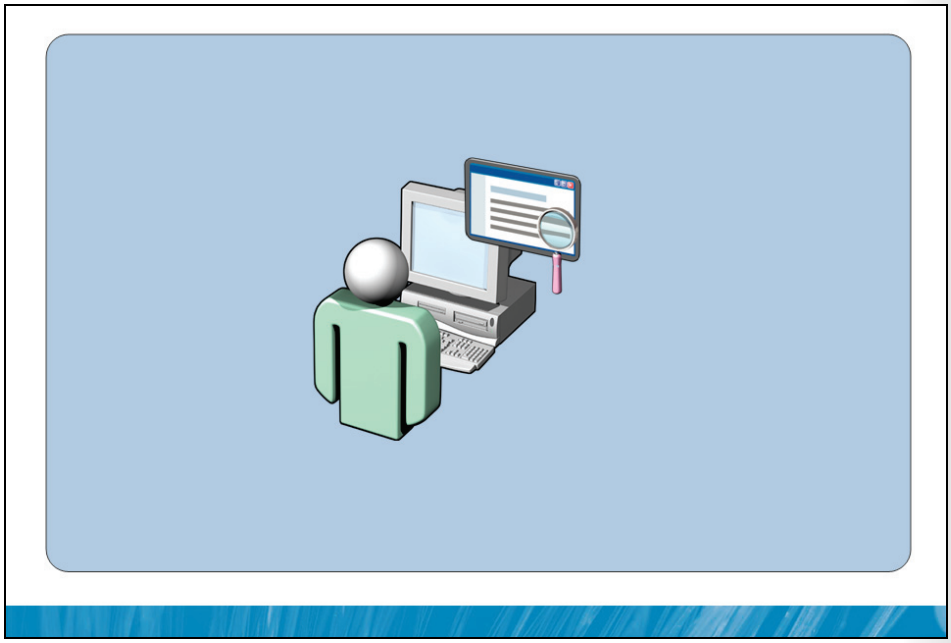
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

You are building an application that drives a machine that stress-tests girders for the construction of high-rise buildings, bridges, and other critical structures. You have defined types to support this application, but they currently expose all members publicly, which can cause problems. After they are created, the **girderMaterial**, **crossSection**, **lengthInMm**, **heightInMm**, and **widthInMm** members of a **StressTestCase** object should be immutable; this guarantees that the test case results that are reported in a test case object match the data for the test.

Exercise 1: Hiding Data Members

In this exercise, you will make the fields in the **StressTestCase** class private and verify that these fields are now inaccessible outside code in the class.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Declare fields in the **StressTestCase** class as private.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Build the project and correct errors.
4. Update unit tests to resolve errors.

▶ Task 1: Open the StressTesting solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio® 2010.
3. Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex1\Starter folder.

▶ Task 2: Declare fields in the StressTestCase class as private

1. Review the task list.
2. In the task list, locate the **TODO - Modify the StressTestCase class to make members private** task, and then double-click this task. This task is located in the **StressTestCase** class.
3. In the **StressTestCase** class, remove the **TODO - Modify the StressTestCase class to make members private** comment, and then modify each field definition to make all of the fields private.

▶ Task 3: Build the project and correct errors

1. Build the project, and then review the error list.

   The project should fail to build because the code in the **doTests_Click** method in the test harness project attempts to access the fields in the **StressTestCase** class that are now private.
2. Comment out the code that caused the errors that are shown in the error list.

▶ Task 4: Update unit tests to resolve errors

1. On the **Build** menu, click **Build Solution**. There should still be some errors.

   The remaining errors are located in the unit test project.
2. In the task list, locate the **TODO - Update unit tests to resolve errors** task, and then double-click this task. This task is located in the **StressTestCaseTest** unit test class.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the **StressTestCaseConstructorTest** method, comment out the five **Assert** statements that cause errors.
4. Update the method to verify that the constructed object contains the correct member values by performing the following tasks:



Hint: You cannot access the member data directly because you have just declared private members. The **ToString** method returns a string representation of the object, including the member data.

   a. Before you instantiate the **target** object, declare a new string named **expected** and populate the string with the following data that represents the expected results of the test.

```
Material: Composite, CrossSection: CShaped, Length: 5000mm, Height:
32mm, Width: 18mm, No Stress Test Performed
```

   b. At the end of the method, add an **Assert** statement that checks whether the **expected** string matches the output of the **target.ToString** method.
5. Update the **StressTestCaseConstructorTest1** method and resolve the errors by performing the following tasks:
   a. Comment out the five existing **Assert** statements.
   b. Before the method creates the **target** object, create a new string that contains the expected result from a default **StressTestCase** class. This string is the same as the string that the previous test expects.
   c. At the end of the method, add an **Assert** statement that checks whether the **expected** string matches the output of the **target.ToString** method.
6. Rebuild the solution and correct any errors.
7. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

Exercise 2: Using Static Members to Share Data

In this exercise, you will define a struct that holds a pair of private fields to record the total number of tests that are performed and the total number of failures. You will add a private static member to the **StressTestCase** class that is based on this struct. You will then modify the **PerformStressTest** method to increment the fields

MCT USE ONLY. STUDENT USE PROHIBITED

in this struct as appropriate. Finally, you will add a static method to the class that returns the value of this struct.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Create a struct to hold the number of successes and failures.
3. Modify the **StressTestCase** class to contain a **TestStatistics** object.
4. Display the statistics in the user interface.
5. Test the solution.
6. Examine and run unit tests for the **TestStatistics** class.

▶ Task 1: Open the StressTesting solution

- Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex2\Starter folder. This solution contains a copy of the **StressTestCase** class with the public properties made private.

▶ Task 2: Create a struct to hold the number of successes and failures

1. Review the task list.
2. In the task list, locate the **TODO - Create the TestStatistics struct** task, and then double-click this task. This task is located in the **StressTestCase** class.
3. Delete the **TODO - Create the TestStatistics struct** comment, and then define a new public struct named **TestStatistics**, which has the following private members:
   a. An integer named **numberOfTestsPerformed**.
   b. An integer named **numberOfFailures**.
4. Add a method to the **TestStatistics** struct named **IncrementTests**. The method should accept a Boolean parameter named *success*, but not return a value. Add code to the method to perform the following tasks:
   a. Increment the **numberOfTestsPerformed** member.
   b. If the *success* parameter is **false**, increment the **numberOfFailures** member.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Below the **IncrementTests** method, add a method named **GetNumberOfTestsPerformed**. This method should take no parameters and return an integer value. Add code to the method to return the value of the **numberOfTestsPerformed** member.
6. Below the **GetNumberOfTestsPerformed** method, add a method named **GetNumberOfFailures**. The method should take no parameters and return an integer value. Add code to the method to return the value of the **numberOfFailures** member.
7. Below the **GetNumberOfFailures** method, add an **internal** method named **ResetCounters**. The method should take no parameters and not return a value. Add code to the method to set both the **numberOfFailures** and the **numberOfTestsPerformed** members to zero.
8. Build the project and correct any errors.

▶ Task 3: Modify the StressTestCase class to contain a TestStatistics object

1. In the task list, locate the **TODO - Add a TestStatistics field and method to the StressTestCase class** task, and then double-click this task. This task is located in the **StressTestCase** class.
2. Delete the **TODO - Add a TestStatistics field and method the StressTestCase class** comment, and then declare a new private static member of type **TestStatistics** named **statistics**.
3. Below the **statistics** member declaration, add a public static method named **GetStatistics**. The method should take no parameters, but should return a **TestStatistics** object. Add code to the method to return the value of the **statistics** member.
4. Below the **GetStatistics** method, add a public static method named **ResetStatistics**. The method should take no parameters and should not return a value. Add code to the method to invoke the **ResetCounters** method on the **statistics** member.
5. In the task list, locate the **TODO - Update the PerformStressTest method to handle statistics** task, and then double-click this task. This method is located in the **StressTestCase** class.
6. Delete the **TODO - Update the PerformStressTest method to handle statistics** comment, and in the **PerformStressTest** method, add code to invoke the **IncrementTests** method on the **statistics** member when a test

MCT USE ONLY. STUDENT USE PROHIBITED

either passes or fails. If the test passes, specify the value **true** as the argument to the **IncrementTests** method. If the test fails, specify the value **false** as the argument to the **IncrementTests** method.

▶ Task 4: Display the statistics in the user interface

1. In the task list, locate the **TODO - Update the UI to display statistics** task, and then double-click this task. This task is located in the **MainWindow** class, at the end of the **doTests_Click** method.
2. At the end of the **doTests_Click** method, delete the comment and add code to perform the following tasks:
   a. Create a new **TestStatistics** object named **statistics**. Initialize the object with the value that is returned by calling the **StressTestCase**.**GetStatistics** method.
   b. In the **statisticsLabel1** label, display the message "Number of tests: <*tests*>, Failures: <*failures*>", where *tests* is the number of tests that were executed, and *failures* is the number of tests that failed.



Hint: Set the **Content** property of a **Label** control to display a message in that control.

   c. Invoke the **IncrementTests** method on the **statistics** object, and pass **true** as a parameter.
   d. Invoke the static **GetStatistics** method on the **StressTestCase** object, and store the result in the statistics variable.
   e. In the **statisticsLabel2** label, display the message "Number of tests: <*tests*>, Failures: <*failures*>", where *tests* is the number of tests that were executed, and *failures* is the number of tests that failed.



Note: This demonstrates the principle of passing or returning by value. When the code first calls the **GetStatistics** method, a copy of the value is returned from the **StressTestCase** object. Therefore, when the code calls the **IncrementTests** method, the update is performed on the copied value and not the original value. When the **GetStatistics** method is called for the second time, another copy of the original value is retrieved; therefore, both labels will display the same value.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Test the solution

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**, and then examine the statistics labels, which should both display the same values.
4. Close the MainWindow window, and then return to Visual Studio.

▶ Task 6: Examine and run unit tests for the TestStatistics class

1. In the task list, locate the **TODO - Examine and run unit tests** task, and then double-click this task. This task is located in the StressTestClass_TestStatisticsTest file.
2. Examine the **GetNumberOfFailuresTest** method.

   This method creates a new **TestStatistics** object named **target** and then invokes the **IncrementTests** method twice, passing **false** as the parameter. The method then retrieves the number of failures from the **TestStatistics** object and uses an **Assert** statement to verify that the value is correct.
3. Examine the **GetNumberOfTestsPerformed** method.

   This method creates a new **TestStatistics** object named **target** and then invokes the **IncrementTests** method three times. The method then retrieves the number of tests that were performed from the **TestStatistics** object and uses an **Assert** statement to verify that the value is correct.
4. Examine the **IncrementTestsTest** method.

   This method creates a **TestStatistics** object named **target** and then invokes the **IncrementTests** method on this object four times. The method then retrieves the number of tests that were performed from the **target** object and uses an **Assert** statement to verify that the value is correct.
5. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

Exercise 3: Implementing an Extension Method

In this exercise, you will add a long integer field to the **TestCaseResult** struct to hold information. You will update the **PerformStressTest** method in the **StressTestCase** class to populate this field with simulated results. To display the

data in this field as a binary string, you will add an extension method called **ToBinaryString** to the **System.Int64** struct.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Define a new extension method.
3. Modify the **TestCaseResult** struct to include a long field.
4. Modify the **PerformStressTest** method.
5. Display the failure data.
6. Test the solution.
7. Examine and run unit tests.

▶ Task 1: Open the StressTesting solution

- Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex3\Starter folder. This solution contains a copy of the solution from the previous exercise.

▶ Task 2: Define a new extension method

1. In the StressTest project, add a new public static class named **Extensions**, in a file named Extensions.cs:
   a. In Solution Explorer, right-click the **StressTest** project, point to **Add**, and then click **Class**.
   b. In the **Add New Item - StressTest** dialog box, in the **Name** box, type **Extensions** and then click **Add**.
   c. Modify the **Extensions** class definition. This class should be a **public static** class.
2. In the **Extensions** class, add a new public static extension method named **ToBinaryString**. The method should take a 64-bit integer parameter named *i* and return a string value.



Hint: To indicate that a method is an extension method, prefix the parameter with the **this** keyword.



Hint: You can use **long** as an alias for the **System.Int64** type.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the **ToBinaryString** method, add code to create a string that holds the binary representation of the 64-bit integer value that is passed in the i integer, and return this string.

▶ Task 3: Modify the TestCaseResult struct to include a long field

1. Review the task list.
2. In the task list, locate the **TODO - Modify the TestCaseResult struct** task, and then double-click this task. This task is located in the **TestCaseResult** struct.
3. In the **TestCaseResult** struct, delete the comment and add a public field of type **long** named **failureData**.

▶ Task 4: Modify the PerformStressTest method

1. In the task list, locate the **TODO - Update the PerformStressTest method** task, and then double-click this task. This task is located in the **StressTestCase** class, in the **PerformStressTest** method.
2. In the **PerformStressTest** method, delete the **TODO - Update the PerformStressTest method** comment, and then add code to update the **failureData** member of the **TestCaseResult** object with a random number to simulate the data that is retrieved from the stress-testing equipment.



Hint: Use the **Rand** member of the **Utility** static class to generate a random number. This method contains a method called **Next** that returns a random number in a specified range. Pass the value **int.MaxValue** as the parameter to the **Next** method to generate a random number between 0 and this value. The value **int.MaxValue** field specifies the maximum value that the integer type supports.

▶ Task 5: Display the failure data

1. In the task list, locate the **TODO - Update the UI to display the binary string** task, and then double-click this task. This task is located in the **MainWindow** class, in the **doTests_Click** method.
2. Modify the **doTests_Click** method to append the binary data that is contained in the **failureData** member to the failure information that is displayed in the user interface; append a space character followed by the result of the

MCT USE ONLY. STUDENT USE PROHIBITED

ToBinaryString method call to the end of the string that is added to the **resultList.Items** collection.

▶ Task 6: Test the solution

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**, and then verify that when an error occurs, binary data is displayed after the reason for the failure.
4. Close the MainWindow window, and then return to Visual Studio.

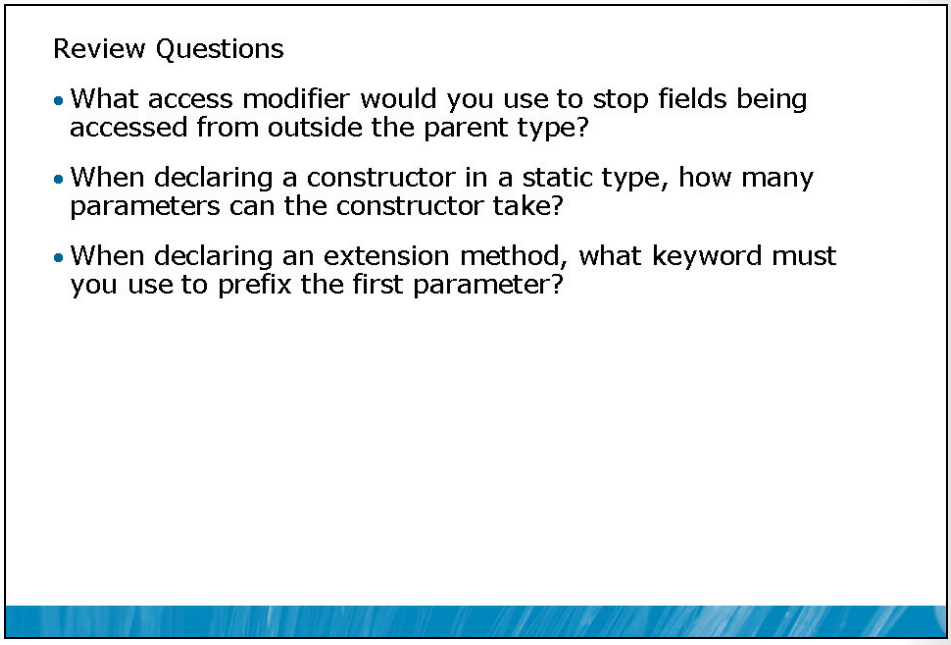
▶ Task 7: Examine and run unit tests

1. In the task list, locate the **TODO - Review and run unit tests** task, and then double-click this task. This task is located in the **ExtensionsTest** class.
2. Examine the **ToBinaryStringTest** method.

   This method creates a long variable, i, with the value 8 and then creates a string variable, expected, with the value "1000". The method then invokes the **ToBinaryString** extension method on the long variable i and stores the result in a string named **actual**. The method then uses an **Assert** statement to verify that the expected and **actual** values are the same. The method then updates the long variable i with the value 10266 and the expected variable with the binary representation "10100000011010". Next, it directly calls the **ToBinaryString** method, passes the long variable i as a parameter, and stores the result of the method call in the actual variable. The method uses a second **Assert** statement to verify that the expected and actual values are the same.
3. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

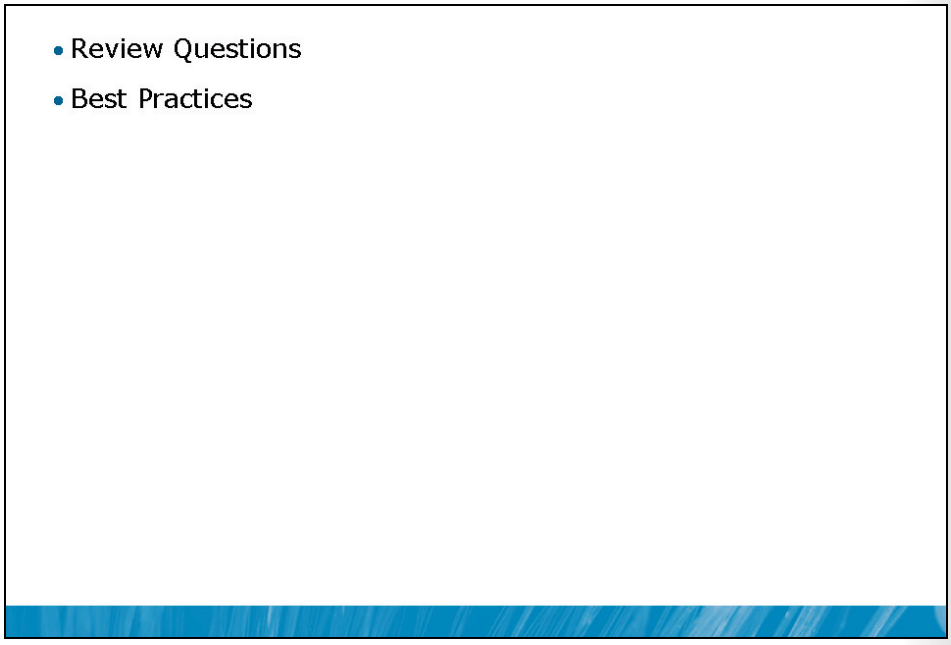


Review Questions

1. What access modifier would you use to stop fields being accessed from outside the parent type?
2. When declaring a constructor in a static type, how many parameters can the constructor take?
3. When declaring an extension method, what keyword must you use to prefix the first parameter?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. Briefly explain the purpose of encapsulation.
2. Which access modifier do you use to expose a method to a type in a different assembly?
3. How do you invoke a static constructor?

Best Practices Related to Encapsulating Data and Methods

Supplement or modify the following best practices for your own work situations:

- Do not expose the inner workings of your types with the **public** and **internal** access modifiers. If in doubt, use the **private** access modifier.
- If a type does not need to store instance data, declare the type as static.
- If you must add functionality to an existing type and do not want to derive a new type, use extension methods.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 8

Inheriting from Classes and Implementing Interfaces

Contents:

Lesson 1: Using Inheritance to Define New Reference Types 8-3

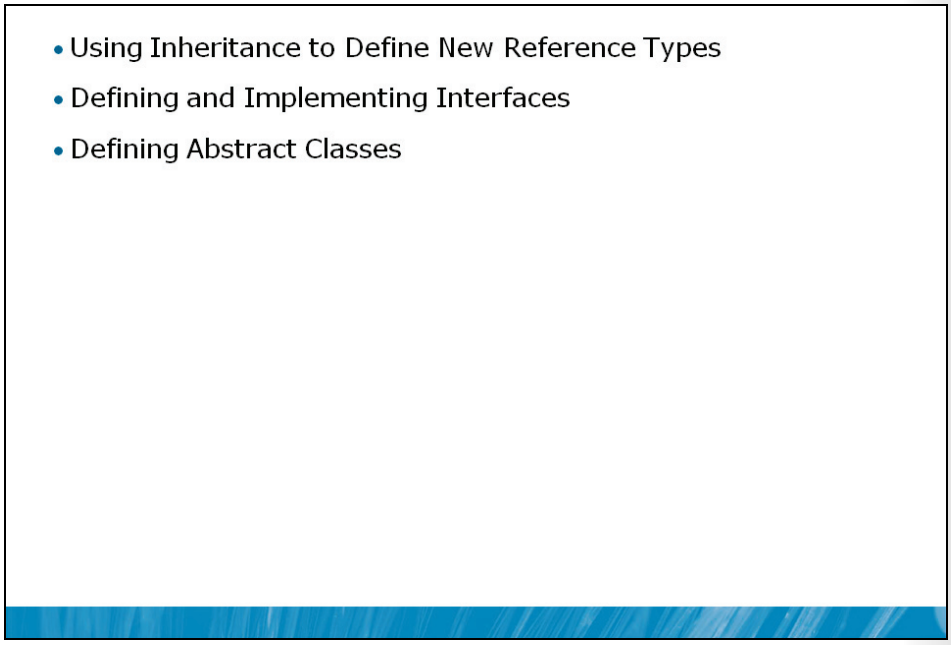
Lesson 2: Defining and Implementing Interfaces 8-27

Lesson 3: Defining Abstract Classes 8-45

Lab: Inheriting from Classes and Implementing Interfaces 8-56

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



This module introduces inheritance and interfaces in the Microsoft® .NET Framework, and how you can use them to simplify complex problems, reduce code duplication, and speed development.

Inheritance is a key concept in an object-oriented language. You can use inheritance, interfaces, and abstract classes to develop object hierarchies in your code. These object hierarchies can help reduce bugs by defining clear contracts for what a class will expose, and providing default implementations where you can sensibly abstract code into a base type.

Objectives

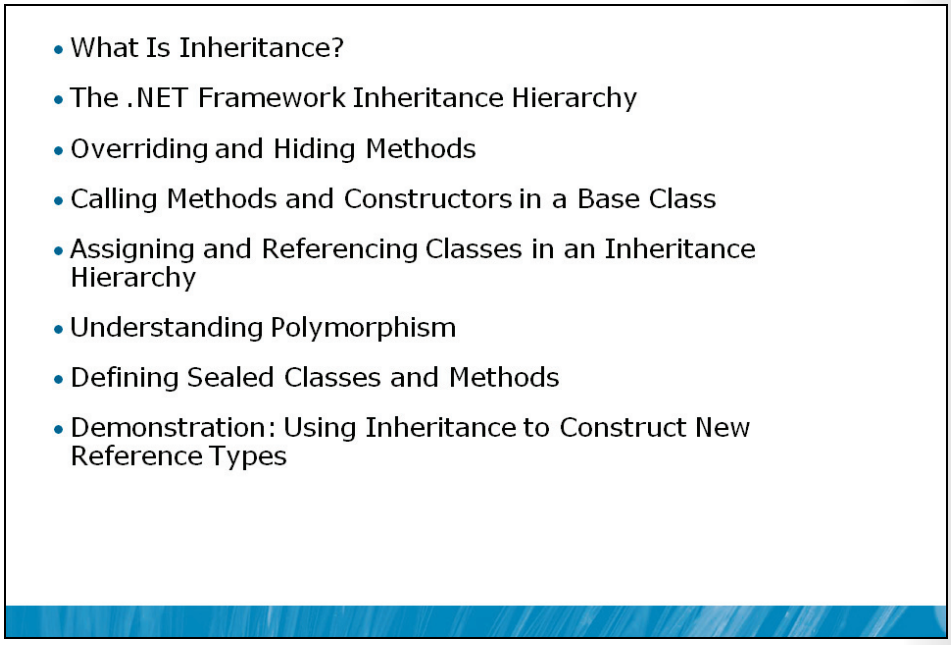
After completing this module, you will be able to:

- Use inheritance to define new reference types.
- Define and implement interfaces.
- Define abstract classes.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1

Using Inheritance to Define New Reference Types



This lesson describes inheritance in the .NET Framework and helps you understand how you can use inheritance to develop better code faster, and with fewer bugs.

Developing an object hierarchy is an important process. The object hierarchy should be well designed, and you should avoid code duplication. Understanding inheritance in the .NET Framework is fundamental to this process.

Objectives

After this lesson, you will be able to:

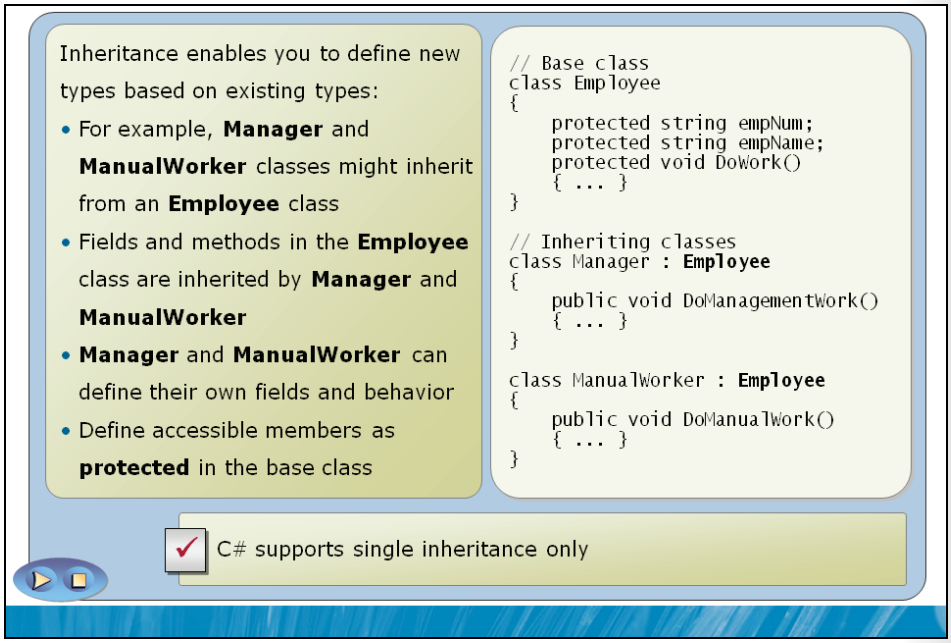
- Explain the purpose of inheritance and how it works.
- Describe the inheritance hierarchy of the .NET Framework.
- Override and hide methods.

MCT USE ONLY. STUDENT USE PROHIBITED

- Call methods and constructors in a base class.
- Assign references in an inheritance hierarchy.
- Explain how polymorphism works.
- Define sealed classes and methods.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Inheritance?



Key Points

Inheritance is a key concept in the world of object orientation. You can use inheritance as a tool to avoid repetition when you are defining different classes that have several features in common and are related to each other. Perhaps they are different subclasses of the same type, each with its own distinguishing feature–for example, managers and manual workers are all employees of a factory. If you were writing an application to simulate the factory, how would you specify that managers and manual workers have several features that are the same, but also other features that are different? For example, they all have an employee reference number and a name, but managers have different responsibilities and perform different tasks from manual workers.

As a solution, you might develop a class to represent an employee. This class can include fields to hold information that is common to all employees, such as the employee reference number and name. You can then develop a class to represent a manager and another class to represent a manual worker. Both the manager class and the manual worker class would also need to store the employee reference number and name. Rather than adding duplicate fields to each of these classes, you can simply specify that both of these classes inherit from the employee class, which

MCT USE ONLY. STUDENT USE PROHIBITED

already has fields to store these values. In addition, there might be some behavior that is common to managers and manual workers. You can implement this behavior as a method in the employee class. The manager and manual worker classes can then inherit this behavior. Using inheritance in this way reduces the need for code duplication, which reduces both development time and the risk of bugs being introduced. You can add additional, specific fields and methods to the manager and manual worker classes to model the different data and behaviors of these types.

Defining an Inherited Class

You specify that a class inherits from another class (called a base class) by using a colon and providing the name of the base class. Remember that all of the members in the base class are private by default. This means that they cannot be accessed by code in other classes, including classes that inherit from the base class. You can make the members of a base class visible to inheriting classes, but keep them hidden from other classes that are not part of the inheritance hierarchy by using the **protected** keyword, as the following code example shows.

```
[Visual C#]
// Base class
class Employee
{
    protected string empNum;
    protected string empName;
    protected void DoWork()
    {
        ...
    }
}
// Inheriting classes
class Manager : Employee
{
    public void DoManagementWork()
    {
        ...
    }
}

class ManualWorker : Employee
{
    public void DoManualWork()
    {
        ...
    }
}
```

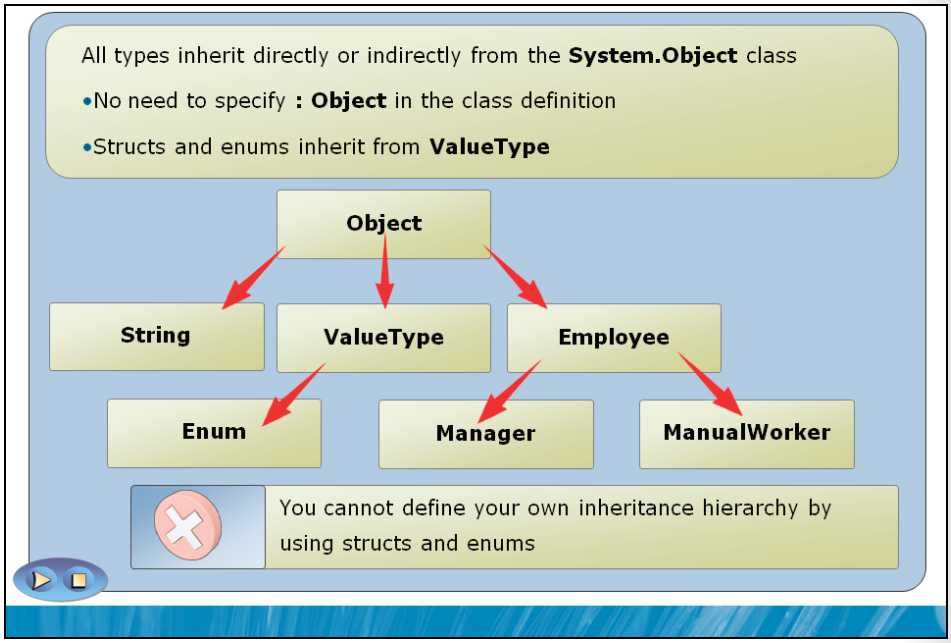
MCT USE ONLY. STUDENT USE PROHIBITED

Finally, C# supports single inheritance only. You cannot define a class that directly inherits from more than one base class.

Question: What accessor should you use to make class members accessible to child classes?

MCT USE ONLY. STUDENT USE PROHIBITED

The .NET Framework Inheritance Hierarchy



Key Points

In the .NET Framework, all types inherit either directly or indirectly from the **Object** class in the **System** namespace. The **Object** class provides functionality that is useful to all types, such as the **ToString** and **Equals** methods. When you create a new reference type such as a class, it inherits directly from the **Object** class and you do not need to specify this relationship as part of the class definition. When you inherit from another class, such as **Employee**, you automatically inherit all of the functionality of the **Object** class, too.

Value types such as structs inherit from the **System.ValueType** class, which in turn inherits from the **Object** class. Enum types inherit from the **System.Enum** class, which inherits from **ValueType**. However, unlike classes, this hierarchy is fixed and you cannot define your own custom inheritance hierarchy with value types; for example, you cannot explicitly specify that a struct inherits from another struct.

Question: What types inherit from the **Object** class?

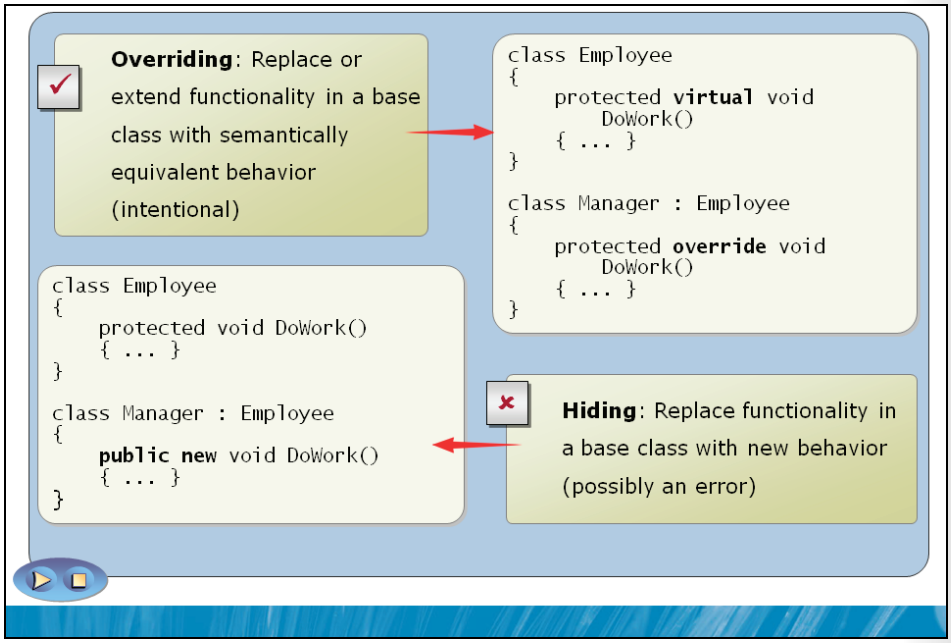
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about the **Object** class, see the Object Class page at http://go.microsoft.com/fwlink/?LinkId=192936.

MCT USE ONLY. STUDENT USE PROHIBITED

Overriding and Hiding Methods



Key Points

When you use inheritance to define a class, an inheriting class can provide its own methods. It is possible that some of these methods have the same names as those that are inherited from the base class. In these situations, you have to decide whether you want to override the inherited methods or hide them.

Overriding Methods

When you override a method, you provide an implementation that has the same meaning as the original method, but has an implementation that is specific to the class. For example, the **Object** class provides the **ToString** method, which returns a representation of an object as a string. However, the default implementation of the **ToString** method in the **Object** class simply returns the name of the type as a string. You might override this behavior in the **Employee** class to return a string that contains the name of the employee.

To override a method in a subclass, you use the **override** keyword, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Object
{
    public virtual string ToString()
    {
        // Return the type of the object as a string
        // (code not shown)

        ...;
    }
}

class Employee
{
    protected string empName;

    ...

    public override string ToString()
    {
        return string.Format("Employee: {0}", empName);
    }

}
```

You can only override methods that are marked as **virtual**, **override**, or **abstract** in the base class. Virtual methods typically provide a default implementation that inheriting classes are expected to replace with their own code. When you define a class that other classes might inherit from, you should decide which methods you will allow to be overridden in this way and declare them as virtual.

Note that when you override a virtual method, you cannot change the protection level of the method; if the method in the base class is protected, the override method must also be protected.

Hiding Methods

You can define methods in an inherited class that have the same name as methods in a base class, even if they are not marked as **virtual**, **abstract**, or **override**. However, this means that there is no relationship between your method and the original method, and your new method *hides* the original method. In this case, the C# compiler emits a warning (you might not be aware that a class that you are inheriting from has such a method, so you might want to change the name of your method to avoid this conflict), although your code still compiles.

If you are aware that you are hiding a method in a base class, you can turn the compiler warning off by marking the method with the **new** keyword, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Employee
{
    protected void DoWork()
    {
        ...
    }
}

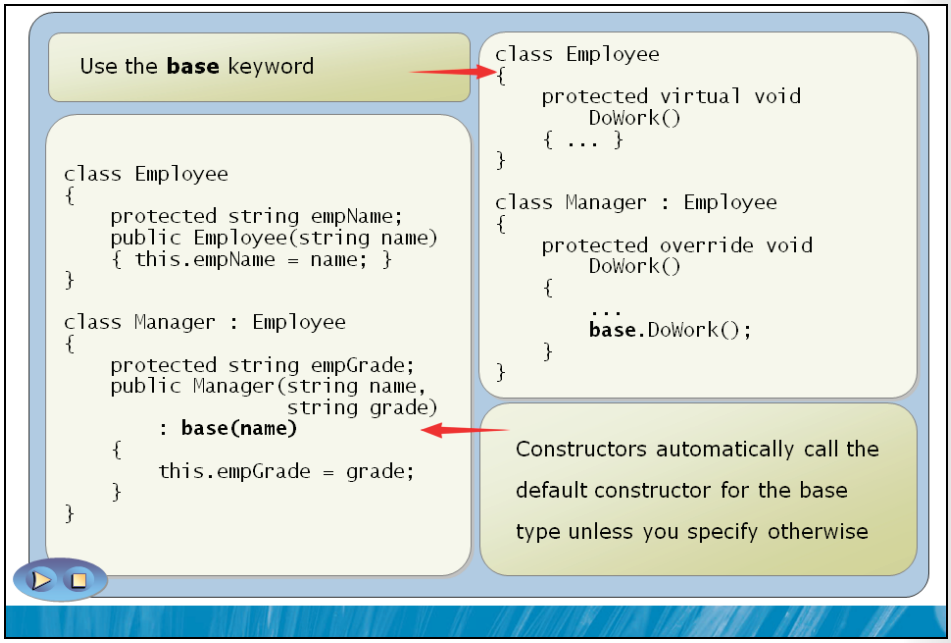
class Manager : Employee
{
    public new void DoWork()
    {
        // Hide the DoWork method in the base class
        ...
    }
    ...
}
```

When you hide a method, you can change the protection level. For example, you can hide a protected or private method in a base class with a public method that has the same name in an inheriting class. However, this practice is not recommended. Generally, it is better to override a method than to hide it—hiding is frequently an indication of poor design.

Question: What happens if you attempt to hide a method without using the **new** keyword?

MCT USE ONLY. STUDENT USE PROHIBITED

Calling Methods and Constructors in a Base Class



Key Points

An inheriting class can call methods in a base class by using the **base** keyword as the method prefix. This feature is useful when you are overriding methods because it enables you to provide your own functionality in addition to invoking the existing functionality that the base class defines. In effect, this extends methods. The following code example shows the **DoWork** method in the **Manager** class overriding the **DoWork** method in the **Employee** class from which it inherits, but calling the **DoWork** method in the **Employee** class at an appropriate point.

```
class Employee
{
    protected virtual void DoWork()
    {
        ...
    }
}

class Manager : Employee
{
    protected override void DoWork()
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        // Do processing specific to Managers
        ...
        // Call the DoWork method in the base class
        base.DoWork();
    }
    ...
}
```

Without the **base** keyword, the call to **DoWork** in the **Manager** class would simply call the **DoWork** method in the **Manager** class recursively.

Calling Base Class Constructors

In addition to the methods that it inherits, a derived class automatically contains all fields from the base class. These fields usually require initialization when an object is created. You typically perform this kind of initialization in a constructor. Remember that all classes have at least one constructor. (If you do not provide one, the compiler generates a default constructor for you.) It is good practice for a constructor in a derived class to call the constructor for its base class as part of the initialization. You can specify the **base** keyword to call a base class constructor when you define a constructor for an inheriting class, as the following code example shows.

```
class Employee
{
    protected string empName;

    public Employee(string name)  // constructor for base class
    {
        this.empName = name;
    }
    ...
}

class Manager : Employee
{
    protected string empGrade;

    public Manager(string name, string grade)
           : base(name) // calls Employee(name)
    {
        this.empGrade = grade;
    }
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

If you don't explicitly call a base class constructor in a derived class constructor, the compiler attempts to silently insert a call to the default constructor of the base class before executing the code in the derived class constructor. The following code example is an extract from the previous code example.

```
class Manager : Employee
{
    public Manager(string name, string grade)
    {
        ...
    }
    ...
}
```

The compiler will rewrite this code as the code in the following code example.

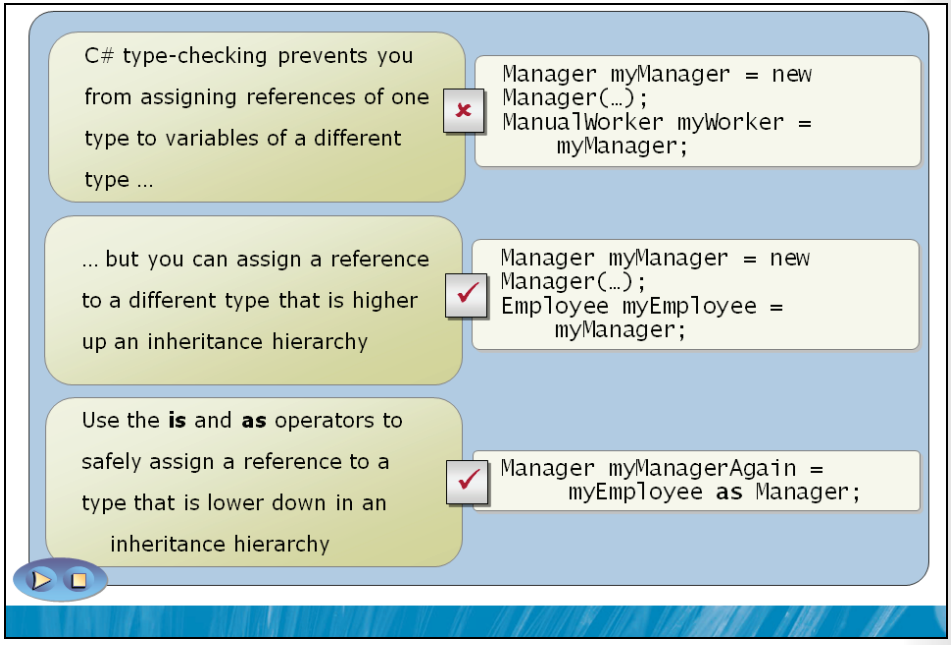
```
class Manager : Employee
{
    public Manager(string name, string grade)
        : base()
    {
        ...
    }
    ...
}
```

This works if the **Employee** class has a public default constructor. However, not all classes have a public default constructor (for example, remember that the compiler only generates a default constructor if you don't write any nondefault constructors), in which case forgetting to call the correct base class constructor results in a compile-time error.

Question: What happens if you do not call the constructor of a base class in the constructor for your class?

MCT USE ONLY. STUDENT USE PROHIBITED

Assigning and Referencing Classes in an Inheritance Hierarchy



Key Points

The type-checking rules of C# prevent you from assigning an object of one type to a variable that is declared as a different type. For example, given the definitions of the **Employee**, **Manager**, and **ManualWorker** classes that are shown in the following code example, the code that follows these definitions is illegal.

```
class Employee
{
    ...
}

class Manager : Employee
{
    ...
}

class ManualWorker : Employee
{
    ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
}
...

// Manager constructor expects a name and a grade
Manager myManager = new Manager("Fred", "VP");

ManualWorker myWorker = myManager;  // error - different types
```

However, it is possible to refer to an object from a variable of a different type as long as the type that you use is a class that is higher up the inheritance hierarchy. Therefore, the statements in the following code example are legal.

```
Manager myManager = new Manager("Fred", "VP");
Employee myEmployee = myManager;
// legal, Employee is the base class of Manager
```

This works because the inheritance hierarchy means that you can think of a **Manager** simply as a special type of **Employee**; it has everything that an **Employee** has with a few extra bits that you can define by any methods and fields that you add to the **Manager** class.

You can also make an Employee variable refer to a **ManualWorker** object. There is one significant limitation, however—when you refer to a **Manager** or **ManualWorker** object by using an Employee variable, you can access only methods and fields that are defined by the **Employee** class. Any additional methods that the **Manager** or **ManualWorker** classes define are not visible through the **Employee** class. This explains why you can assign almost anything to an Object variable. Remember that all classes inherit from **System.Object** directly or indirectly.

Converting Object References

Although you can assign a **Manager** object to an Employee variable, the converse is not true; you cannot unreservedly assign an **Employee** object to a Manager variable. This is because not all **Employee** objects are **Manager** objects; some might be **ManualWorker** objects. However, you can assign an **Employee** object to a Manager variable as long as you check that the **Employee** is really a **Manager** first, by using the **as** or **is** operators, or by using a cast.

The **as** operator checks that an object is a reference to a specified type. If it is, it returns a new reference by using this type, otherwise it returns null.

You can also use a cast to assign a reference of one type to a variable of a different type as long as the conversion is valid. However, you should use the **is** operator first to verify that the conversion will succeed. Like the **as** operator, the **is** operator

MCT USE ONLY. STUDENT USE PROHIBITED

checks that an object is a reference to a specified type and returns **true** if it is, and **false** if it is not. If you attempt to perform an invalid cast that the compiler does not detect at compile time, your code will throw an **InvalidCastException** at run time.

The following code example uses the **as** operator to check that **myEmployee** refers to a **Manager** object. If it does, the assignment results in **myManagerAgain** referring to the same **Manager** object as **myManager**. If **myEmployee** refers to some other type of **Employee**, such as a **ManualWorker**, the **as** operator returns null instead.

```
Manager myManager = new Manager("Fred", "VP");
Employee myEmployee = myManager; // myEmployee refers to a Manager
...
Manager myManagerAgain = myEmployee as Manager;
// OK - myEmployee is a Manager
...
ManualWorker myWorker = new ManualWorker("Bert");
myEmployee = myWorker;     // myEmployee now refers to a ManualWorker
...
myManagerAgain = myEmployee as Manager;
// returns null - myEmployee is a ManualWorker
```

Question: What exception is thrown if you attempt to perform an invalid cast?

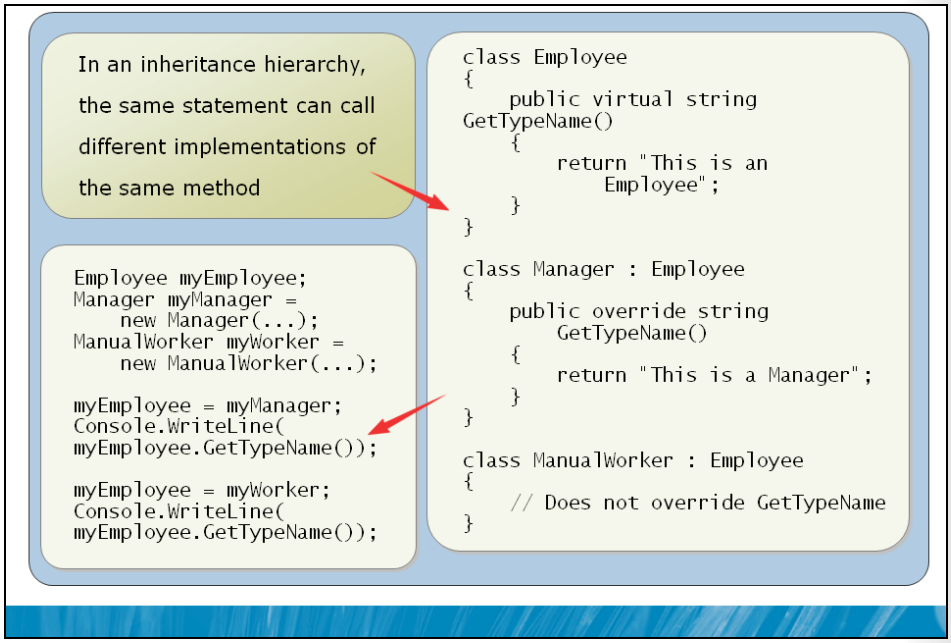
Additional Reading

For more information about the **as** operator, see the as (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192941.

For more information about the **is** operator, see the is (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192942.

MCT USE ONLY. STUDENT USE PROHIBITED

Understanding Polymorphism



Key Points

Virtual methods that are defined in classes that share an inheritance hierarchy enable you to call different versions of the same method, based on the type of the object, which is determined dynamically at run time. This is called *polymorphism*, and is a very powerful feature of object-oriented systems.

Consider the example classes in the following code example, which define a variation on the **Employee** hierarchy.

```
class Employee
{
    ...
    public virtual string GetTypeName()
    {
        return "This is an Employee";
    }
}

class Manager : Employee
{
    ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    public override string GetTypeName()
    {
        return "This is a Manager";
    }
}

class ManualWorker : Employee
{
    ...
    // Does not override GetTypeName
}
```

In this hierarchy, notice that the **override** keyword is used by the **GetTypeName** method in the **Manager** class, and the **ManualWorker** class does not have a **GetTypeName** method. In the following code example, what will be displayed by the two **Console.WriteLine** statements?

```
Employee myEmployee;
Manager myManager = new Manager(...);
ManualWorker myWorker = new ManualWorker(...);

myEmployee = myManager;
Console.WriteLine(myEmployee.GetTypeName()); // Manager
myEmployee = myWorker;
Console.WriteLine(myEmployee.GetTypeName()); // ManualWorker
```

You might expect them both to print "This is an Employee" because each statement calls the **GetTypeName** method on the myEmployee variable, which is an **Employee** reference. However, in the first case, **myEmployee** is actually a reference to a **Manager** object. The **GetTypeName** method is defined as virtual, so the runtime works out that it should call the **Manager.GetTypeName** method. Therefore, the statement actually prints the message "This is a Manager." The second **Console.WriteLine** statement calls **GetTypeName** on a **ManualWorker** object. However, the **ManualWorker** class does not have a **GetTypeName** method, so the default method in the **Employee** class is called, returning the string "This is an Employee."

Question: When you reference an object by its parent class, which version of a method is called: the version from the base class, or the overridden version in the child class?

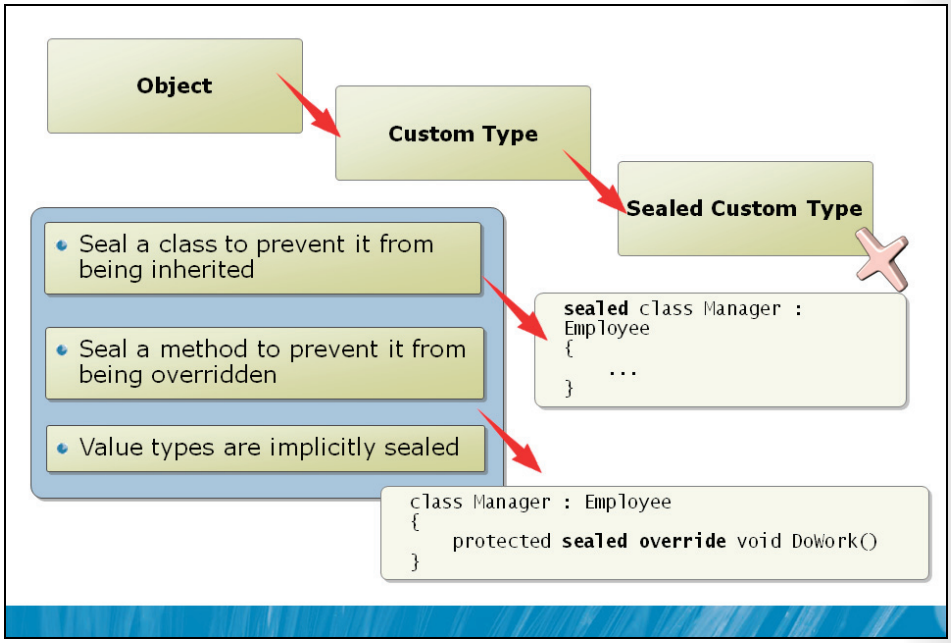
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about polymorphism, see the Polymorphism (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192937.

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Sealed Classes and Methods



Key Points

By default, when you define a class, other people who have access to the assembly that contains your class can inherit from it and add their own functionality. However, unless you consciously design a class with the intention of using it as a base class, it is extremely unlikely to function well as a base class. C# enables you to use the **sealed** keyword to prevent a class from being used as a base class if you decide that it should not be. The following code example declares the **Manager** class as **sealed**.

```
sealed class Manager : Employee
{
    ...
}
```

If any class attempts to use **Manager** as a base class, a compile-time error will be generated. Note that a sealed class cannot declare any virtual methods.

In the .NET Framework, all value types (structs and enums) are implicitly sealed.

MCT USE ONLY. STUDENT USE PROHIBITED

Sealing Methods

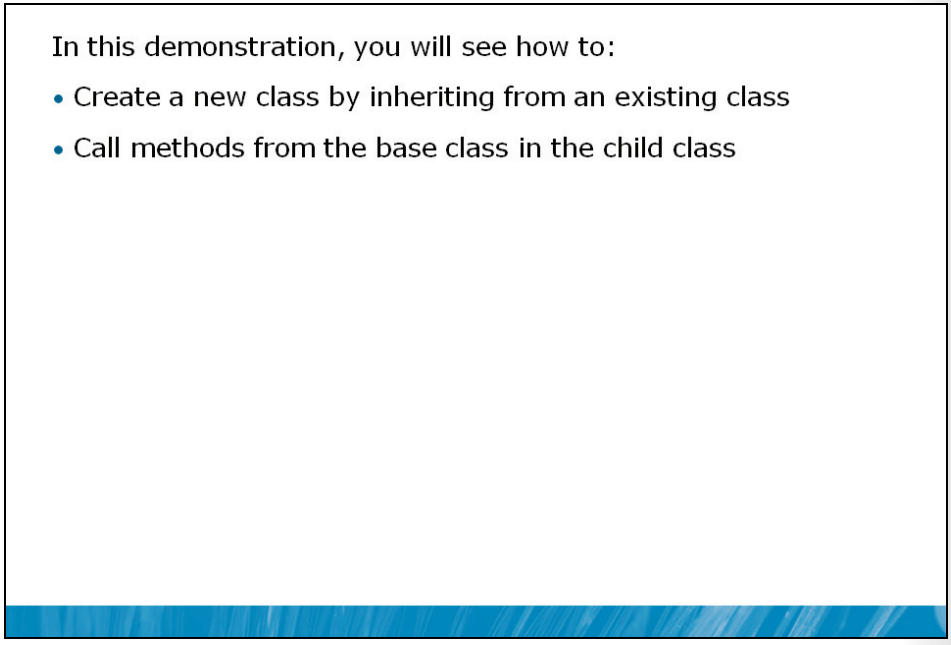
You can also use the **sealed** keyword to declare that an individual method in an unsealed class is sealed. This means that a derived class cannot then override the sealed method. You can seal only **override** methods, and you declare them as **sealed override**, as the following code example shows.

```
class Manager : Employee
{
    ...
    protected sealed override void DoWork()
    {
        ...
    }
}
```

Question: Can you define a class that inherits from the C# **int** type?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using Inheritance to Construct New Reference Types



Key Points

- Create a new class by inheriting from an existing class
- Call methods from the base class in the child class

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Microsoft Visual Studio 2010.
3. In Visual Studio 2010, open the **UsingInheritanceDemo** solution in the **E:\Demofiles\Mod8\Demo1\Starter\UsingInheritanceDemo** folder.
4. In the Program.cs file, add a new class called **Television.**

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Television
{
}
```

5. In the **Television** class, add a **protected virtual** method called **SetCurrentChannel**. The method should write a message to the console indicating that the television channel has been set.

   Your code should resemble the following code example.

```
class Television
{
      protected virtual void SetCurrentChannel()
      {
          Console.WriteLine("Channel set.");
      }
}
```

6. In the **Television** class, add a **protected** method called **TurnOn**. The method should write a message to the console indicating that the television is on.

   Your code should resemble the following code example.

```
class Television
{
      protected virtual void SetCurrentChannel()
      {
          Console.WriteLine("Channel set.").
      }
      protected void TurnOn()
      {
          Console.WriteLine("Television on.");
      }
}
```

7. In the Program.cs file, add a new class called **WidescreenTV** that inherits from the **Television** class:

   Your code should resemble the following code example.

```
class WidescreenTV : Television
{
}
```

8. Override the **SetCurrentChannel** method. The method should write a message to the screen indicating that the channel has been set on the widescreen television.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
class WidescreenTV : Television
{
      protected override void SetCurrentChannel()
      {
          Console.WriteLine("Widescreen channel set.");
      }
}
```

9. Add a constructor to the **WidescreenTV** class. The constructor should call the constructor of the base class, call the **TurnOn** method, call the **SetCurrentChannel** method, and then call the **SetCurrentChannel** method of the base class.

   Your code should resemble the following code example.

```
class WidescreenTV : Television
{
      protected override void SetCurrentChannel()
      {
          Console.WriteLine("Widescreen channel set.");
      }
      public WidescreenTV()
          : base()
      {
          TurnOn();
          SetCurrentChannel();
          base.SetCurrentChannel();
      }
}
```

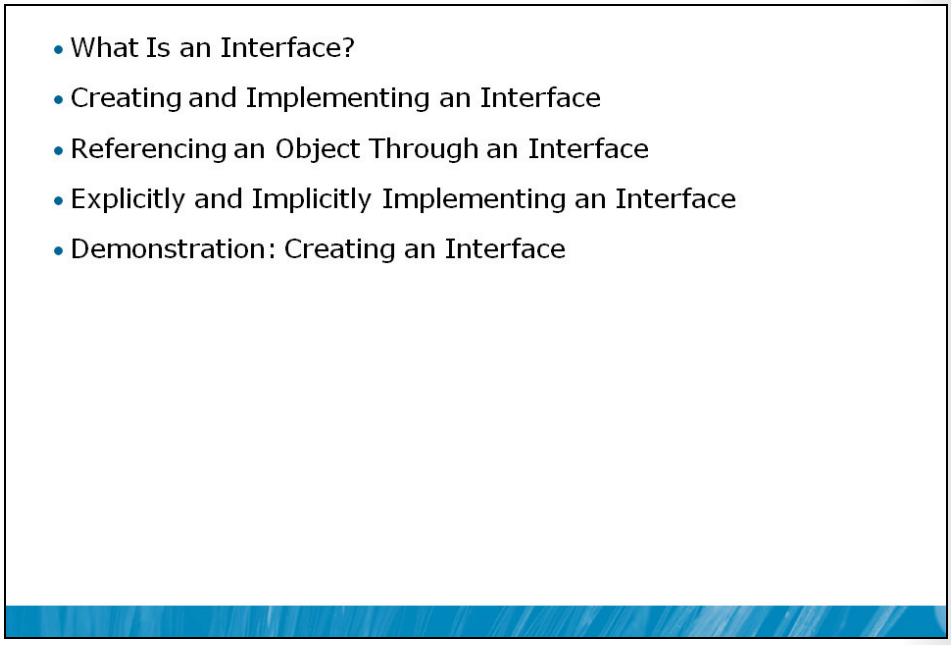
10. Uncomment the code in the **Program** class that creates an instance of the **WidescreenTV** class.
11. Run the application with debugging and verify that the following messages appear:

```
Television on.
Widescreen channel set.
Channel set.
```

Question: What tools does Visual Studio provide to help when you implement an interface?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Defining and Implementing Interfaces



This lesson introduces interfaces and describes how you can use them to standardize your code.

Interfaces enable you to define contracts explicitly, defining the methods that your objects expose. You can implement multiple interfaces to indicate that your code can perform several functions. For example, you can implement the **IComparable** interface to indicate that two objects can be compared for equality, and the **IDisposable** interface to indicate that an object can be cleaned up.

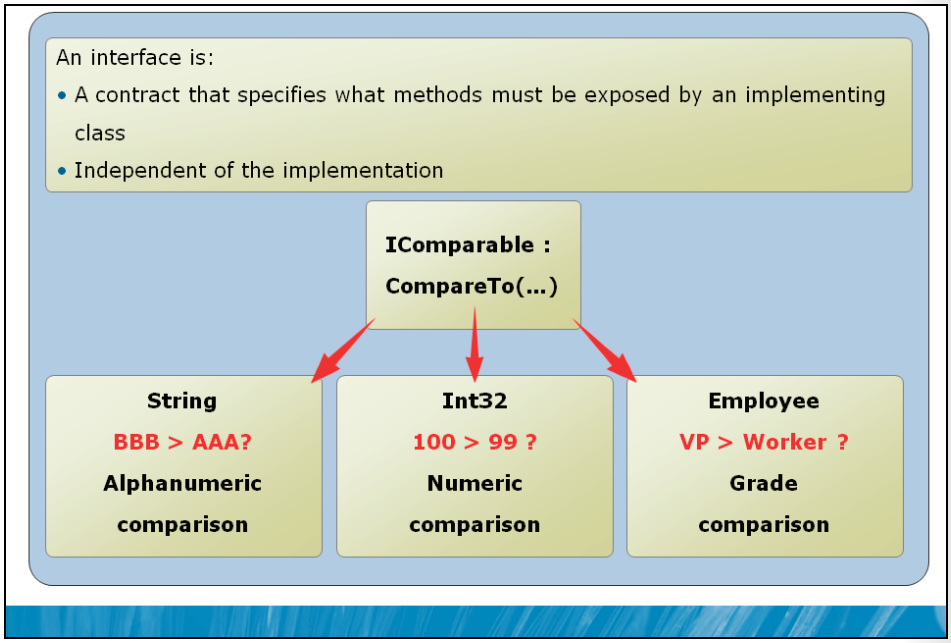
Objectives

After this lesson, you will be able to:

- Describe the purpose of interfaces.
- Create and implement an interface.
- Reference an object by using an interface.
- Explain explicit and implicit interface implementation.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Interface?



Key Points

Inheriting from a class is a powerful mechanism, but the real power of inheritance comes from implementing an interface. An interface does not contain any code or data; it just specifies the methods and properties that a class that inherits from the interface must provide. Using an interface enables you to completely separate the names and signatures of the methods of a class from the method's implementation.

Interfaces act as a contract; they guarantee that any class that implements the interface will expose the members that the interface specifies. This enables you to simplify development and standardize your code.

Interfaces enable you to specify functionality that a class should implement. How a class chooses to implement this functionality is the concern of the class and not a property of the interface. Different classes can implement the same interface in different ways, as long as they expose the set of methods that the interface defines.

For example, a frequent requirement in the .NET Framework is for classes to define a method that enables them to be compared, to determine their relative ordering. What it actually means to compare the value of two objects of a given class depends on the class itself; the **Employee** class might determine that

MCT USE ONLY. STUDENT USE PROHIBITED

employee objects should be ranked by grade (two employees with the same grade are equal, but an employee with a grade of "VP" might be considered superior to an employee with a grade of "Worker"), whereas the **String** class uses an alphanumeric comparison to determine the relative order of string values. To standardize the way in which objects of any given type can be compared, the **System** namespace in the .NET Framework class library defines the **IComparable** interface. This interface contains a single method called **CompareTo** that has the signature in the following code example.

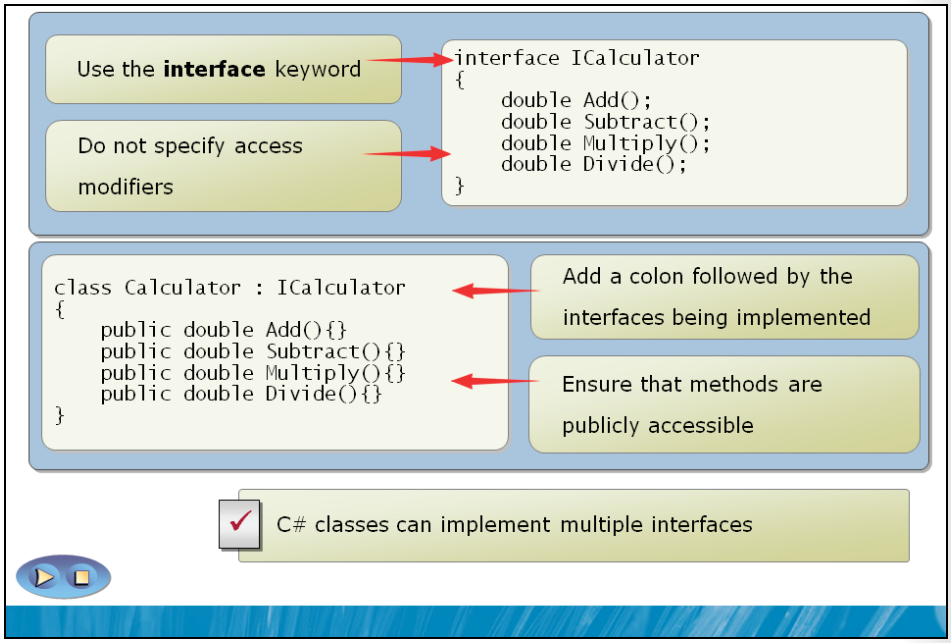
```
int CompareTo(Object obj)
```

A class that implements the **IComparable** interface must provide the **CompareTo** method. This method returns zero if the object that is specified as the parameter is considered equal to the object on which the **CompareTo** method was invoked; a value less than zero if the object is considered to be less than the object that was specified as the parameter; and a value greater than zero if the object is considered to be greater than the object that is specified as the parameter. It is up to the class that is implementing the interface to provide the actual logic for the **CompareTo** method.

Question: Can you add a default implementation of a method to an interface?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating and Implementing an Interface



Key Points

Syntactically, an interface is similar to a class except that you only declare methods and do not provide the code that implements them.

Defining an Interface

To define an interface, you use the **interface** keyword. Inside the interface, you declare methods exactly as in a class or a structure except that you never specify an access modifier (**public**, **private**, or **protected**), and you replace the method body with a semicolon, as the following code example shows.

```
interface ICalculator
{
    double Add();
    double Subtract();
    double Multiply();
    double Divide();
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Note that, in the example above, the name of the interface begins with an uppercase **I**. This is a common naming convention rather than an explicit requirement, but the .NET Framework documentation recommends that you adhere to this standard; all interfaces in the **System** namespace are prefixed in this way.

Implementing an Interface

To implement an interface, you declare a class or structure that inherits from the interface, and provides code for every method that the interface defines.



Note: Although you cannot create struct types that inherit from other struct types or classes, a struct type can implement an interface.

When you implement an interface, you must ensure that each method matches its corresponding interface method exactly, according to the following rules:

- The method names and return types match exactly.
- Any parameters (including **ref** and **out** keyword modifiers) match exactly.
- All methods implementing an interface must be publicly accessible. However, if you are using explicit interface implementation, the method should not have an access qualifier.

If there is any difference between the interface definition and its declared implementation, the class will not compile.

The following code example shows a class that implements the **ICalculator** interface.

```
class Calculator : ICalculator
{
    // The methods of the ICalculator interface return test data
    // in this code.
    #region ICalculator Members

    public double Add()
    {
        return 0;
    }

    public double Subtract()
    {
        return 0;
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    public double Multiply()
    {
        return 0;
    }

    public double Divide()
    {
        return 0;
    }

    #endregion
}
```

A class can only inherit from one other class; however, a class can implement multiple interfaces. If you want to specify that a class implements multiple interfaces, you separate each interface with a comma in the class declaration. When you implement more than one interface, you must ensure that you follow the rules above for every interface that your class implements, otherwise your code will not compile. For example, if you wanted to specify that the **Calculator** class implements the **IComparable** interface that was described in the previous topic you could change the class definition, as the following code example shows.

```
class Calculator : ICalculator, IComparable
{
    // Code to implement ICalculator.
    #region ICalculator Members

    public double Add()
    {
        return 0;
    }

    public double Subtract()
    {
        return 0;
    }

    public double Multiply()
    {
        return 0;
    }

    public double Divide()
    {
        return 0;
    }

    #endregion
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Code to implement IComparable.
    #region IComparable members

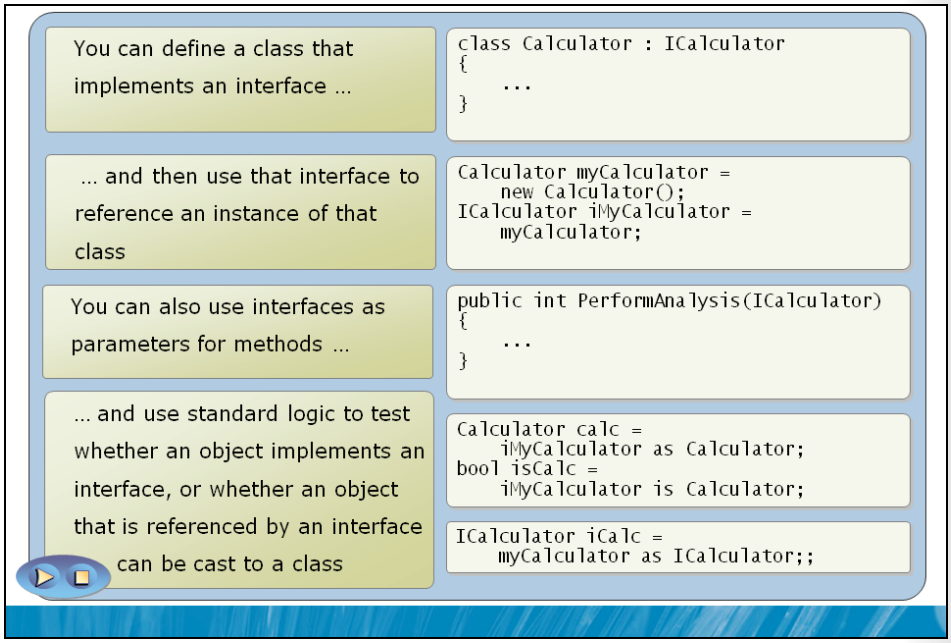
    public int CompareTo(Object obj)
    {
        ...
    }

    #endregion
}
```

Question: When you define an interface, do you add a method body?

MCT USE ONLY. STUDENT USE PROHIBITED

Referencing an Object Through an Interface



Key Points

In the same way that you can reference an object by using a variable that is defined as a class that is higher up an inheritance hierarchy, you can reference an object by using a variable that is defined as an interface that its class implements. Taking the example from the previous topic, you can reference a **Calculator** object by using an ICalculator variable, as the following code example shows.

```
Calculator myCalculator = new Calculator();
ICalculator iMyCalculator = myCalculator;
```

This works because all **Calculator** objects implement the **ICalculator** interface. However, the converse is not true, and you cannot assign an **ICalculator** object to a Calculator variable without casting it first to verify that it does reference a **Calculator** object and not some other class that also happens to implement the **ICalculator** interface. You can use the **is** and **as** keywords to check that an object implements an interface, or to check that an object that an interface references is an instance of a particular class.

MCT USE ONLY. STUDENT USE PROHIBITED

Passing Parameters as Interface References

The technique of referencing an object through an interface is useful because it enables you to define methods that can take different types as parameters, as long as the types implement a specified interface. For example, in the following code example, the **PerformAnalysis** method can take any argument that implements the **ICalculator** interface.

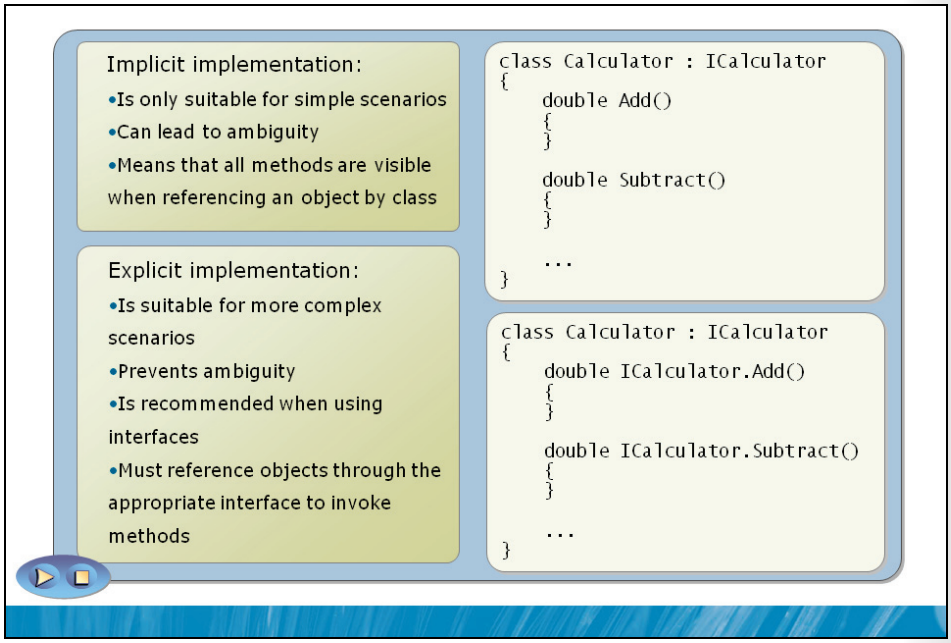
```
int PerformAnalysis(ICalculator calculator)
{
    ...
}
```

Note that, when you reference an object through an interface, you can invoke only methods that are visible through the interface.

Question: What happens if you attempt to cast an object to an interface that it does not implement?

MCT USE ONLY. STUDENT USE PROHIBITED

Explicitly and Implicitly Implementing an Interface



Key Points

The examples that you have seen so far have shown classes that implicitly implement an interface. If you revisit the **ICalculator** interface and the **Calculator** class, the **Calculator** class implements from the **ICalculator** interface. However, there is nothing in the implementation of any methods in the **Calculator** class that says that they are part of the **ICalculator** interface, as the following code example shows.

```
interface ICalculator
{
    double Add();
    double Subtract();
    double Multiply();
    double Divide();
}

class Calculator : ICalculator
{
    // The methods of the ICalculator interface return test data
    // in this code.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    #region ICalculator Members

    public double Add()
    {
        return 0;
    }

    public double Subtract()
    {
        return 0;
    }

    public double Multiply()
    {
        return 0;
    }

    public double Divide()
    {
        return 0;
    }

    #endregion
}
```

This might not be an issue in a simple situation, but suppose the **Calculator** class implemented multiple interfaces. There is nothing to prevent multiple interfaces specifying a method that has the same name, although they might have different semantics. For example, suppose you extended the **Calculator** class to implement the following **ITaxCalculator** interface, which includes methods for adding and subtracting tax amounts from a value. You might define the interface in the following code example.

```
interface ITaxCalculator
{
    double Add();
    double Subtract();
}
```

Now, if you implement this interface in the **Calculator** class, you have a problem, as the following code example shows.

```
class Calculator : ICalculator, ITaxCalculator
{
    public double Add()
    {
        return 0;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }

    public double Subtract()
    {
        return 0;
    }

    public double Multiply()
    {
        return 0;
    }

    public double Divide()
    {
        return 0;
    }
}
```

This is legal code, but are the **Add** and **Subtract** methods implementations of the methods in the **ICalculator** or **ITaxCalculator** interfaces? The answer as far as C# is concerned is both of these! By default, C# does not distinguish which interface the method is implementing, so the same method satisfies both interfaces. This also applies to the **Subtract** method. The problem is that the implementations of the **Add** and **Subtract** methods probably need to be different for both interfaces. The temptation is to attempt to rewrite the **Calculator** class as in the following code example.

```
class Calculator : ICalculator, ITaxCalculator
{
    ... // private fields not shown

    // This is the Add method for ICalculator
    public double Add()
    {
        return 0;
    }

    // This is the Subtract method for ICalculator
    public double Subtract()
    {
        return 0;
    }

    public double Multiply()
    {
        return 0;
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

```
    public double Divide()
    {
        return 0;
    }

    // This is the Add method for ITaxCalculator?
    public double Add()
    {
        return calculatedValue + taxAmount;
    }

    // This is the Subtract method for ITaxCalculator?
    public double Subtract()
    {
        return calculatedValue - taxAmount;
    }
}
```

However, this is no longer legal code because there are duplicate implementations of the **Add** and **Subtract** methods, which have the same signature.

Resolving Ambiguity by Using an Explicit Interface

To solve this problem and disambiguate which method is part of which interface implementation, you can implement interfaces explicitly. To do this, you specify which interface a method belongs to when you implement it, as the following code example shows.

```
class Calculator : ICalculator, ITaxCalculator
{
    ... // private fields not shown

    double ICalculator.Add()
    {
        return 0;
    }

    double ICalculator.Subtract()
    {
        return 0;
    }

    double ICalculator.Multiply()
    {
        return 0;
    }

    double ICalculator.Divide()
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        return 0;
    }

    double ITaxCalculator.Add()
    {
        return calculatedValue + taxAmount;
    }

    double ITaxCalculator.Subtract()
    {
        return calculatedValue - taxAmount;
    }
}
```

Now you can see that using the **Calculator Add** method returns 0, and adding tax (the **Add** method from the **ITaxCalculator** interface) returns a value that is dependent on the **calculatedValue** and **taxAmount** fields.

Invoking Methods Defined Through an Explicit Interface Implementation

Apart from prefixing the name of the method with the interface name, there is one other subtle difference in this syntax: the methods are not marked as **public**. You cannot specify the protection for methods that are part of an explicit interface implementation. This leads to another interesting phenomenon. If you create a Calculator variable in code, you cannot invoke either of the **Add** methods because they are not visible. As far as the **Calculator** class is concerned, they are both **private**. In fact, this makes sense. If the methods were visible through the **Calculator** class, which method would the code in the following code example invoke—the one for the **ICalculator** interface or the one for the **ITaxCalculator** interface?

```
Calculator calc= new Calculator();
...
double result = calc.Add();
```

So, how do you access these methods? The answer is that you must reference the **Calculator** object through the appropriate interface, as the following code example shows.

```
Calculator calc= new Calculator();
...
ICalculator calculator = calc;
double result = calculator.Add();
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
ITaxCalculator taxCalc = calc;
double tax = taxCalc.Add();
```

Although the methods are private to the class, they are publicly accessible through the interfaces, so this is legal code.

You should explicitly implement interfaces whenever possible.

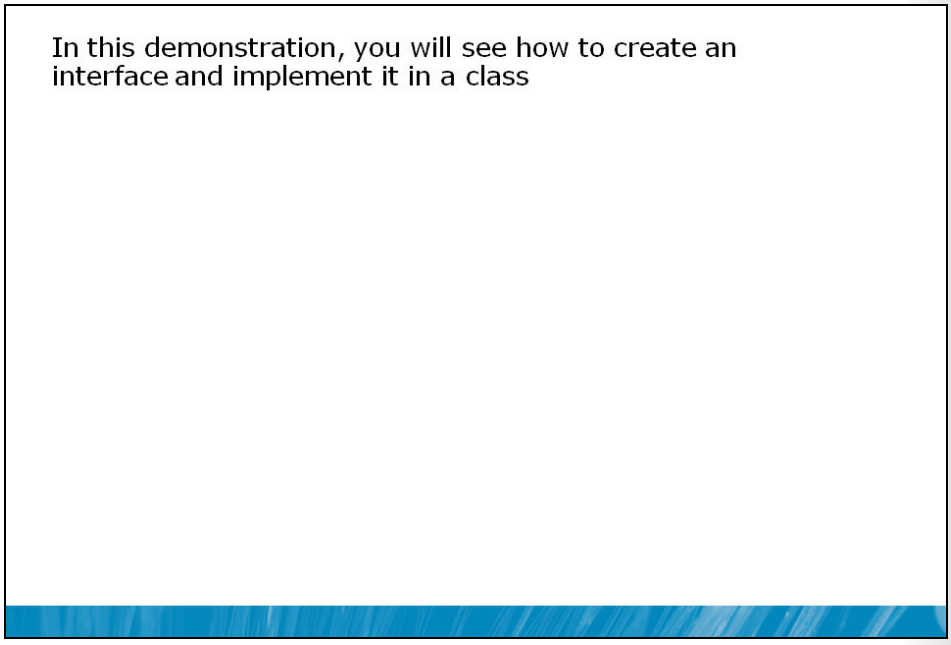
Question: What is the advantage of explicit implementation?

Additional Reading

For more information about how to explicitly implement interface members, see the Explicit Interface Implementation (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192938.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Creating an Interface



Key Points

In this demonstration, you will see how to create an interface and implement it in a class.

Demonstration Steps

1. Open Microsoft Visual Studio 2010.
2. In Visual Studio 2010, open the **CreatingAnInterfaceDemo** solution in the **E:\Demofiles\Mod8\Demo2\Starter\UsingInheritanceDemo** folder.
3. In the Program.cs file, add an interface called **ITelevision.**

   Your code should resemble the following code example.

```
interface ITelevision
{
}
```

4. Add a **TurnOn** method to the interface.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
interface ITelevision
{
      void TurnOn();
}
```

5. Add a **TurnOff** method to the interface.

   Your code should resemble the following code example.

```
interface ITelevision
{
      void TurnOn();
      void TurnOff();
}
```

6. Add an **IncreaseVolume** method to the interface.

   Your code should resemble the following code example.

```
interface ITelevision
{
      void TurnOn();
      void TurnOff();
      void IncreaseVolume();
}
```

7. Add a **DecreaseVolume** method to the interface.

   Your code should resemble the following code example.

```
interface ITelevision
{
      void TurnOn();
      void TurnOff();
      void IncreaseVolume();
      void DecreaseVolume();
}
```

8. In the Program.cs file, add a class called **Television** that implements the **ITelevision** interface.

   Your code should resemble the following code example.

```
class Television : ITelevision
{
}
```

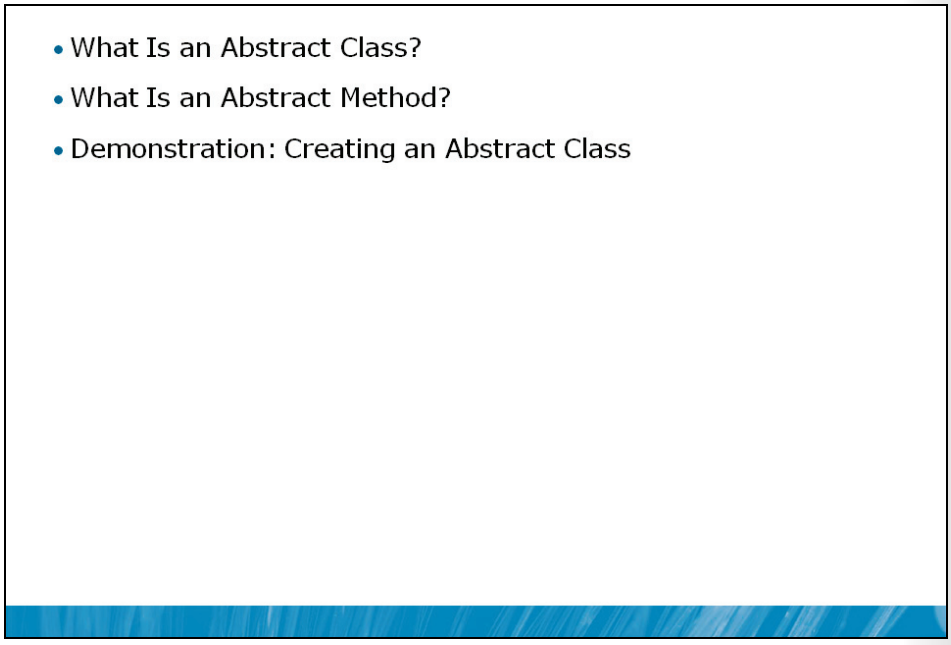
MCT USE ONLY. STUDENT USE PROHIBITED

9. In the definition of the **Television** class, implement the **ITelevision** interface by using the tools in Visual Studio.

Question: What is the recommended naming convention when defining interfaces?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Defining Abstract Classes



This lesson introduces abstract classes, and how you can use them to reduce code duplication, speed development, and reduce the risk of introducing bugs that duplicated code causes.

Abstract classes combine some of the properties of object inheritance with some of the properties of interfaces. You can use abstract classes to reduce code duplication by using default implementations of methods.

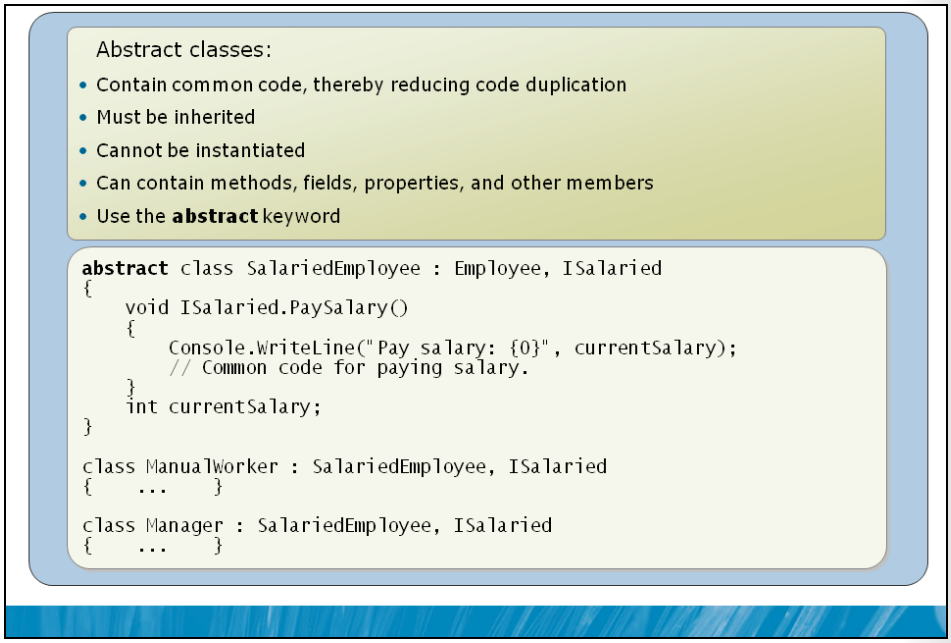
Objectives

After this lesson, you will be able to:

- Describe an abstract class.
- Describe an abstract method.
- Create an abstract class.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Abstract Class?



Key Points

An abstract class provides a mechanism to factor out common code that several related classes share into a single class.

For example, using the **Employees** class hierarchy that was shown earlier in this module, you might develop several classes to represent different types of employees, such as **Manager** and **ManualWorker**. Depending on the number of types of employees that you need to represent, the number of classes will increase. In situations such as this, it is quite common for some elements of these classes to have the same implementation. As an example, the **ManualWorker** and **Manager** classes in the following code example both implement the **ISalaried** interface and provide the **PaySalary** method, which is identical in both classes.

```
interface ISalaried
{
    void PaySalary();
}

class ManualWorker : Employee, ISalaried
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Code for paying salary.
    };
}

class Manager : Employee, ISalaried
{
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Same code as ManualWorker for paying salary.
    };
}
```

Duplication in code is a warning sign. If possible, you should refactor the code to avoid this duplication and reduce any maintenance costs. One way to achieve this refactoring is to put the common implementation into a new class that is created specifically for this purpose. In effect, you can insert a new class into the class hierarchy, as the following code example shows.

```
class SalariedEmployee : Employee, ISalaried
{
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}

class ManualWorker : SalariedEmployee, ISalaried
{
    ...
}

class Manager : SalariedEmployee, ISalaried
{
    ...
}
```

This is a good solution, but one thing is still not quite right: you can create instances of the **SalariedEmployee** class (and the **Employee** class for that matter). This doesn't really make sense. The **SalariedEmployee** class exists to provide a

MCT USE ONLY. STUDENT USE PROHIBITED

common default implementation. Its sole purpose is to be inherited from. The **SalariedEmployee** class is an abstraction of common functionality rather than an entity in its own right. You don't want developers to be able to instantiate this class.

To declare that creating instances of a class is not allowed, you can declare that the class is abstract by using the **abstract** modifier, as the following code example shows.

```
abstract class SalariedEmployee : Employee, ISalaried
{
    ...
    void ISalaried.PaySalary()
    {
        Console.WriteLine("Pay salary: {0}", currentSalary);
        // Common code for paying salary.
    }
    int currentSalary;
}
```

If you try to instantiate a **SalariedEmployee** object now, the code will not compile, as the following code example shows.

```
SalariedEmployee myEmployee =  new SalariedEmployee();  // Illegal
```

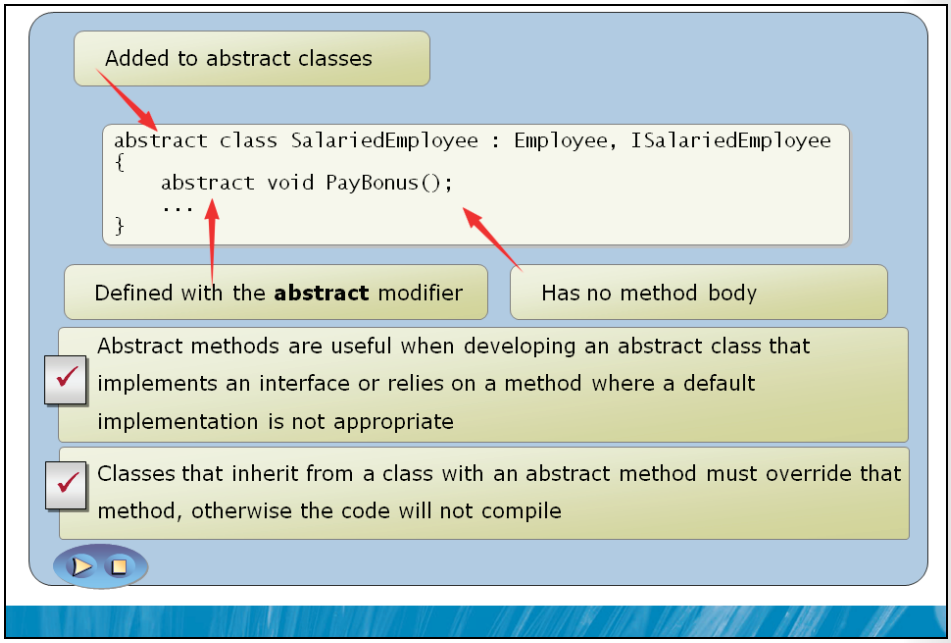
Question: Can a method in an abstract class contain a default implementation?

Additional Reading

For more information about the **abstract** modifier, see the abstract (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192940.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Abstract Method?



Key Points

An abstract class can contain abstract methods. An abstract method is similar in principle to a **virtual** method except that it does not contain a method body. A derived class *must* override this method. The following code example defines the **PayBonus** method in the **SalariedEmployee** class as an abstract method. All employees may share some methods, such as a **PaySalary** method, but they must provide their own implementation of the **PayBonus** method because the logic for paying bonuses may be different for different types of employee. An abstract method is useful if it does not make sense to provide a default implementation in the abstract class and you want to ensure that an inheriting class provides its own implementation of that method.

```
abstract class SalariedEmployee : Employee, ISalariedEmployee
{
    abstract void PayBonus();
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

When you define an abstract method, you use the **abstract** keyword and provide the method signature, but in a manner similar to defining methods in an interface, you replace the method body with a semicolon. Unlike an interface, you can add accessors to abstract methods.



Note: If you attempt to add an abstract method to a nonabstract class, your code will not compile.

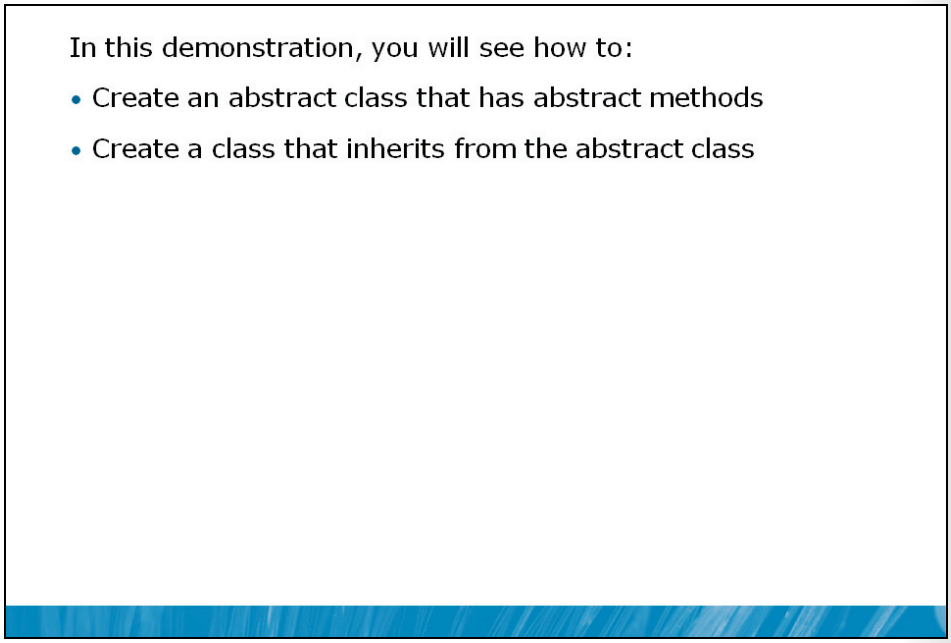
Question: Can an abstract method contain a default implementation?

Additional Reading

For more information about **abstract** and **sealed** class members, see the Abstract and Sealed Classes and Class Members (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192939.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Creating an Abstract Class



Key Points

- Create an abstract class that has abstract methods
- Create a class that inherits from the abstract class

Demonstration Steps

1. Open Microsoft Visual Studio 2010.
2. In Visual Studio 2010, open the **CreatingAnAbstractClassDemo** solution in the **E:\Demofiles\Mod8\Demo3\Starter\CreatingAnAbstractClassDemo** folder.
3. In the Program.cs file, add an abstract class called **Television**.

   Your code should resemble the following code example.

```
abstract class Television
{
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

4. Add a public **TurnOn** method to the class. The method should write a message to the console indicating that the television is on.

   Your code should resemble the following code example.

```
abstract class Television
{
      public void TurnOn()
      {
          Console.WriteLine("Television on.");
      }
}
```

5. Add a public **TurnOff** method to the class. The method should write a message to the console indicating that the television is off.

   Your code should resemble the following code example.

```
abstract class Television
{
      public void TurnOn()
      {
          Console.WriteLine("Television on.");
      }
      public void TurnOff()
      {
          Console.WriteLine("Television off.");
      }
}
```

6. Add a public abstract **IncreaseVolume** method to the class.

   Your code should resemble the following code example.

```
abstract class Television
{
      public void TurnOn()
      {
          Console.WriteLine("Television on.");
      }
      public void TurnOff()
      {
          Console.WriteLine("Television off.");
      }

      public abstract void IncreaseVolume();
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

7. Add a public abstract **DecreaseVolume** method to the class.

   Your code should resemble the following code example.

```
abstract class Television
{
      public void TurnOn()
      {
          Console.WriteLine("Television on.");
      }

      public void TurnOff()
      {
          Console.WriteLine("Television off.");
      }

      public abstract void IncreaseVolume();

      public abstract void DecreaseVolume();
}
```

8. In the Program.cs file, add a class called **WidescreenTV** that inherits from the abstract **Television** class.

   Your code should resemble the following code example.

```
class WidescreenTV : Television
{
}
```

9. Override the **IncreaseVolume** method. The method should write a message to the screen indicating that the volume has increased on the widescreen television.

   Your code should resemble the following code example.

```
class WidescreenTV : Television
{
      public override void IncreaseVolume()
      {
          Console.WriteLine("Volume increased (WidescreenTV).");
      }
}
```

10. Override the **DecreaseVolume** method. The method should write a message to the screen indicating that the volume has decreased on the widescreen television.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
class WidescreenTV : Television
{
      public override void IncreaseVolume()
      {
          Console.WriteLine("Volume increased (WidescreenTV).");
      }

      public override void DecreaseVolume()
      {
          Console.WriteLine("Volume decreased (WidescreenTV).");
      }
}
```

11. In the Program.cs file, add a class called **TV** that inherits from the abstract **Television** class.

    Your code should resemble the following code example.

```
class TV : Television
{
}
```

12. Override the **IncreaseVolume** method. The method should write a message to the screen indicating that the volume has increased on the television.

    Your code should resemble the following code example.

```
class TV : Television
{
      public override void IncreaseVolume()
      {
          Console.WriteLine("Volume increased (TV).");
      }
}
```

13. Override the **DecreaseVolume** method. The method should write a message to the screen indicating that the volume has increased on the television.

    Your code should resemble the following code example.

```
class TV : Television
{
      public override void IncreaseVolume()
      {
          Console.WriteLine("Volume increased (TV).");
      }
```

MCT USE ONLY. STUDENT USE PROHIBITED

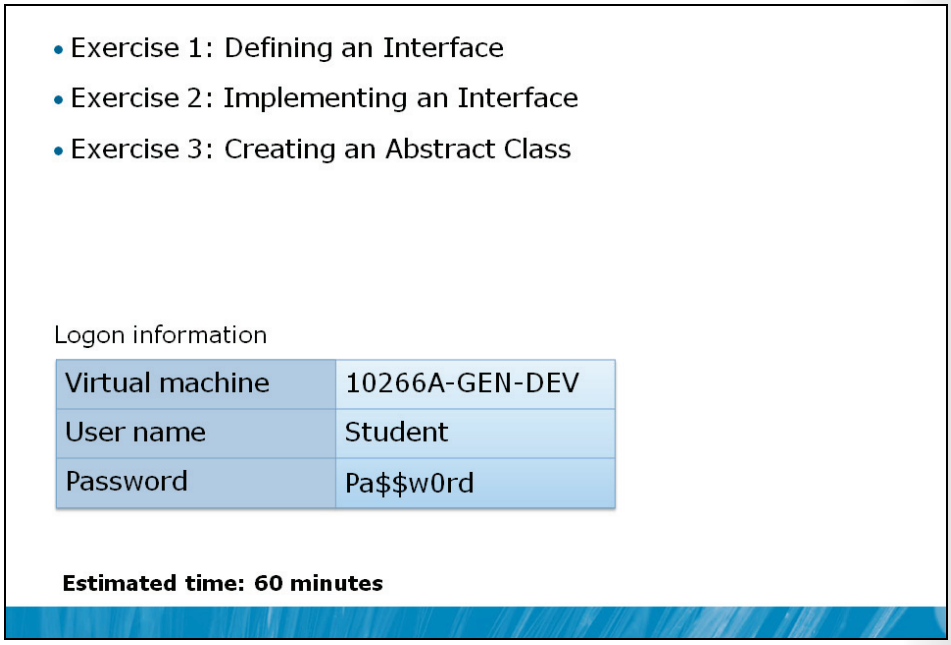
```
        public override void DecreaseVolume()
        {
            Console.WriteLine("Volume decreased (TV).");
        }
}
```

14. Uncomment the code in the **Program** class.
15. Run the application with debugging.

Question: Can you combine abstract and nonabstract (concrete) methods in an abstract class?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Inheriting from Classes and Implementing Interfaces



Objectives

After completing this lab, you will be able to:

- Define an interface.
- Implement an interface in a class.
- Create an abstract class and inherit from this abstract class.

Introduction

In this lab, you will define interfaces and create classes that implement them. You will then factor out common implementation code from the classes into methods in an abstract class and inherit from it.

MCT USE ONLY. STUDENT USE PROHIBITED

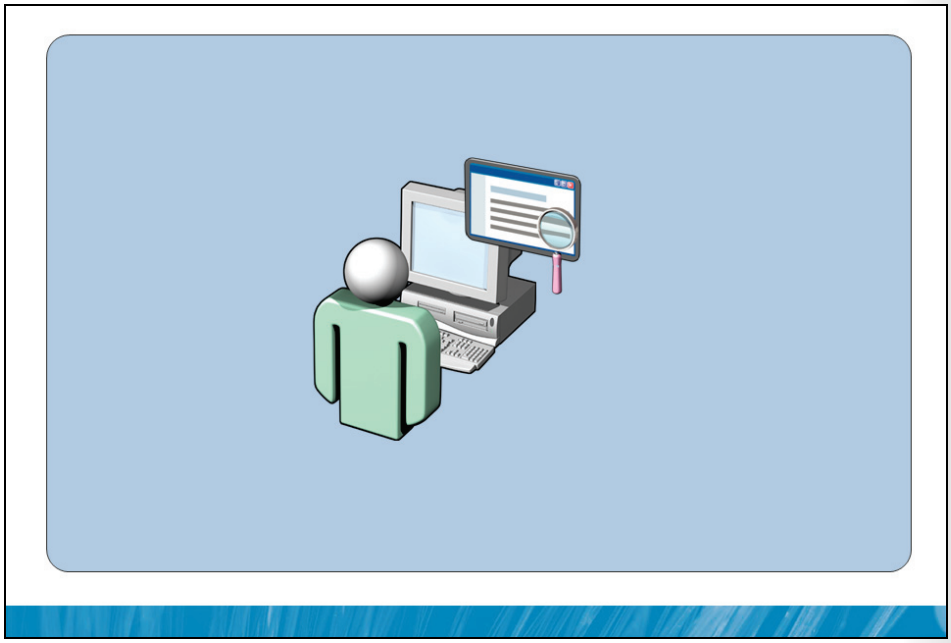
Lab Setup

For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data. These devices can be used to detect minuscule changes in objects over time. A measuring device monitors and measures one specific aspect of an object, such as its mass, its size in a given dimension (height, width, or length), or its distance from the measuring device. The data can be captured in metric or imperial units, and the device can convert the data that it has captured between the metric and imperial scales.

You have been asked to implement the software to drive these measuring devices.

Exercise 1: Defining an Interface

In this exercise, you will define an interface called **IMeasuringDevice** with the following public methods:

- **MetricValue**. This method will return a decimal that represents the metric value of the most recent measurement that was captured.
- **ImperialValue**. This method will return a decimal that represents the imperial value of the most recent measurement that was captured.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- **StartCollecting**. This method will start the device running. It will begin collecting measurements and record them.
- **StopCollecting**. This method will stop the device. It will cease collecting measurements.
- **GetRawData**. This method will retrieve a copy of all of the recent data that the measuring device has captured. The data will be returned as an array of integer values.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create the **IMeasuringDevice** interface.

▶ Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV machine as **Student** with the password **Pa$$w0rd**.
2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 8\Snippets folder
4. Open the Module8 solution in the E:\Labfiles\Lab 8\Ex1\Starter folder.

▶ Task 2: Create the IMeasuringDevice interface

1. Open the IMeasuringDevice code file.
2. In the **MeasuringDevice** namespace, declare the **IMeasuringDevice** interface. The **IMeasuringDevice** interface must be accessible to code in other assemblies.
3. Add a method named **MetricValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method.
4. Add a method named **ImperialValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method.
5. Add a method named **StartCollecting** with a no return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.

6. Add a method named **StopCollecting** with a no return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.
7. Add a method named **GetRawData** that returns an integer array return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.
8. Build the solution and correct any errors.

Exercise 2: Implementing an Interface

In this exercise, you will define the following enumeration:

- Units: Metric, Imperial

You will then define a class called **MeasureLengthDevice** that implements the **IMeasuringDevice** interface and drives a device that measures the length of an object. This class will also include the following private fields:

- **unitsToUse**: Units
- **dataCaptured**: integer array
- **mostRecentMeasure**: integer

You will provide a constructor to initialize the fields in the class (the user will specify a parameter that populates **unitsToUse**).

When the device starts running (when the **StartCollecting** method is called), the device will capture data and store it in the **dataCaptured** array (you will simulate this in the lab by using the code that is provided). This array has a finite, fixed size; when the device is full, it will wrap around and start to overwrite the oldest data. Each time that it takes a new measurement, the device copies this measurement to the **mostRecentMeasure** field. The **GetRawData** method will return the contents of the array. The **MetricValue** and **ImperialValue** methods will return the value in this field, converted according to the units that are specified in the **unitsToUse** field. If **unitsToUse** is **Metric**, **MetricValue** simply returns the data and **ImperialValue** performs a calculation to convert the data to imperial units. Similarly, if **unitsToUse** is **Imperial**, **ImperialValue** simply returns the data and **MetricValue** performs a calculation to convert the data to metric units.

The main tasks for this exercise are as follows:

1. Open the starter project.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Create the **Units** enumeration.
3. Create the **MeasureLengthDevice** class.
4. Update the test harness.
5. Test the **MeasureLengthDevice** class by using the test harness.

▶ Task 1: Open the starter project

- Open the Module8 solution in the E:\Labfiles\Lab 8\Ex2\Starter folder. This solution contains the completed interface from Exercise 1 and skeleton code for Exercise 2.

▶ Task 2: Create the Units enumeration

1. In Visual Studio, review the task list.
2. In the task list, double-click the task **TODO: Implement the Units enumeration**. This task is located in the UnitsEnumeration.cs file.
3. Remove the TODO comment in the UnitsEnumeration file and declare an enumeration named **Units**. The enumeration must be accessible from code in different assemblies.
4. Add the values **Metric** and **Imperial** to the enumeration.
5. Comment your code to make it easier for developers who use the enumeration.
6. Build the solution and correct any errors.

▶ Task 3: Create the MeasureLengthDevice class

1. In the task list, double-click the task **TODO: Implement the MeasureLengthDevice class**. This task is located in the MeasureLengthDevice.cs file.
2. Remove the TODO comment and add a **public** class named **MeasureLengthDevice**.
3. Modify the **MeasureLengthDevice** class declaration to implement the **IMeasuringDevice** interface.
4. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Bring the **DeviceControl** namespace into scope.

   The MeasuringDevice project already contains a reference to the DeviceController project. You are writing code to control a device. However, because the physical device is not available with this lab, the DeviceController project enables you to call methods that control an emulated device. The DeviceController project does not include a visual interface; to control the device, you must use the classes and methods that the project exposes. The DeviceController project is provided complete. You can review the code if you wish, but you do not need to modify it.

6. After the method stubs that the Implement Interface Wizard added in the **MeasureLengthDevice** class, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

   DeviceType is an enumeration that contains the values **LENGTH** and **MASS**. It is used to specify the type of measurement that the device records. It is defined in the DeviceController project.

7. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.LENGTH**.

8. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to the **StartCollecting** method to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the value in the **measurementType** field as the parameter to the **StartCollecting** method.

9. In the **StartCollecting** method, call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.

MCT USE ONLY. STUDENT USE PROHIBITED

10. Add the **GetMeasurements** method to the class, as shown in the following code example.



Note: A code snippet is available, called **Mod8GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();

        while (controller != null)
        {
            System.Threading.Thread.Sleep(timer.Next(1000, 5000));
            dataCaptured[x] = controller != null ?
                controller.TakeMeasurement() : dataCaptured[x];
            mostRecentMeasure = dataCaptured[x];

            x++;
            if (x == 10)
            {
                x = 0;
            }
        }
    });
}
```

The **GetMeasurements** method retrieves measurements from the emulated device. In this module, you will use the code in the **GetMeasurements** method to populate the **dataCaptured** array. This array acts as a fixed-length circular buffer, overwriting the oldest value each time a new measurement is taken. In a later module, you will modify this class to respond to events that the device raises whenever it detects a new measurement.

11. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.

12. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.

MCT USE ONLY. STUDENT USE PROHIBITED

13. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
14. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 25.4.
15. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 0.03937.
16. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
17. Build the solution and correct any errors.

▶ Task 4: Update the test harness

The test harness application for this lab is a simple Windows® Presentation Foundation (WPF) application that is designed to test the functionality of the **MeasureLengthDevice** class that you have just developed. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file by clicking the first **TODO: Add code to instantiate the device field** item in the task list. This task is located in the **createInstance_Click** method in the WPF window, and it runs when the user clicks the **Create Instance** button.
3. In the **createInstance_Click** method, replace both TODO comments with code to instantiate a field called **device** and set it to an instance of the **MeasureLengthDevice** class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureLengthDevice** constructor.
4. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Test the MeasureLengthDevice class by using the test harness

1. Set the Exercise2TestHarness project to be the default startup project.
2. Start the Exercise2TestHarness application.
3. Choose **Imperial**, and then click **Create MeasureLengthDevice Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.
4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **device** object that the **IMeasuringDevice** interface defines.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Raw Data**. You should see up to 10 values in the list box in the lower part of the window. This is the data that the device emulator has generated. It is stored in the **dataCaptured** array by the **GetMeasurements** method in the **MeasureLengthDevice** class. The **dataCaptured** array acts as a fixed-length circular buffer. Initially, it contains zero values, but as the device emulator reports measurements, they are added to this array. When the array is full, it wraps around and starts overwriting data, beginning with the oldest measurement.
7. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement. Note that a new measurement might have been taken since you clicked the **Get Raw Data** button.
8. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
9. Click **Stop Collecting**.
10. Choose **Metric**, and then click **Create MeasureLengthDevice Instance**. This action creates a new instance of the device emulator that uses metric measurements.
11. Click **Start Collecting**. This button starts the new **device** object.
12. Wait for 10 seconds.
13. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.

14. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
15. Click **Stop Collecting**.
16. Close the Exercise 2 Test Harness window.

Exercise 3: Creating an Abstract Class

In this exercise, you will define a class called **MeasureMassDevice**, which also implements the **IMeasuringDevice** interface. You will notice that, although the **MetricValue** and **ImperialValue** methods are implemented slightly differently from the **MeasureLength** class, the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods are identical. Code duplication is never a good thing, and can lead to maintenance difficulties. Consequently, you will create an abstract class called **MeasureDataDevice** that provides default implementations of the duplicated methods. Students will modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from this class.

The main tasks in this exercise are as follows:

1. Open the starter project.
2. Create the **MeasureMassDevice** class.
3. Update the test harness.
4. Test the **MeasureMassDevice** class by using the test harness.
5. Create the **MeasureDataDevice** abstract class.
6. Modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from the **MeasureDataDevice** abstract class.
7. Test the classes by using the test harness.

▶ Task 1: Open the starter project

- Open the Module8 solution in the E:\Labfiles\Lab 8\Ex3\Starter folder. This solution contains the completed interface from Exercise 2 and skeleton code for Exercise 3.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Create the MeasureMassDevice class

1. In Visual Studio, review the task list.
2. Open the MeasureMassDevice.cs file.
3. Replace the TODO comment with a **public** class named **MeasureMassDevice**.
4. Modify the **MeasureMassDevice** class declaration to implement the **IMeasuringDevice** interface.
5. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.
6. Bring the **DeviceControl** namespace into scope.

   The MeasuringDevice project already contains a reference to the DeviceController project. This project implements the **DeviceController** type, which provides access to the measuring device emulator.
7. After the method stubs that Visual Studio added, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

8. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.MASS**.
9. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the **measurementType** field as the parameter to the **StartDevice** method.
10. Add code to call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.

11. Add the **GetMeasurements** method to the class, as shown in the following code example.



Note: A code snippet is available, called **Mod8GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();

        while (controller != null)
        {
            System.Threading.Thread.Sleep(timer.Next(1000, 5000));
            dataCaptured[x] = controller != null ?
                controller.TakeMeasurement() : dataCaptured[x];
            mostRecentMeasure = dataCaptured[x];

            x++;
            if (x == 10)
            {
                x = 0;
            }
        }
    });
}
```

This is the same method that you defined for the **MeasureLengthDevice** class.

12. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.
13. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.
14. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
15. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException**

MCT USE ONLY. STUDENT USE PROHIBITED

exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 0.4536.

16. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 2.2046.
17. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
18. Build the solution and correct any errors.

▶ Task 3: Update the test harness

The test harness application in this lab is a modified version of the WPF application that you used in Exercise 2. It is designed to test the functionality of the **MeasureLengthDevice** and **MeasureMassDevice** classes. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

1. In Visual Studio, review the task list.
2. Open the MainWindow.xaml.cs file by using the first **TODO: Instantiate the device field by using the new MeasureMassDevice class** item in the task list.
3. In the **createInstance_Click** method, replace both TODO comments with code to instantiate the device field to an instance of the **MeasureMassDevice** class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureMassDevice** constructor.
4. Build the solution and correct any errors.

▶ Task 4: Test the MeasureMassDevice class by using the test harness

1. Set the Exercise3TestHarness project to be the default startup project.
2. Start the Exercise3TestHarness application.
3. Choose **Imperial**, choose **Mass Device**, and then click **Create Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **MeasureMassDevice** object.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement.
7. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
8. Click **Stop Collecting**.
9. Choose **Metric**, and then click **Create Instance**. This action creates a new instance of the device emulator that uses metric measurements.
10. Click **Start Collecting**. This button starts the new **device** object.
11. Wait for 10 seconds.
12. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
13. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
14. Click **Stop Collecting**.
15. Close the Exercise 3 Test Harness window.

▶ Task 5: Create the MeasureDataDevice abstract class

You have developed two classes, **MeasureLengthDevice** and **MeasureMassDevice**. Much of the functionality of these classes is common to both. This code duplication is unnecessary and risks introducing bugs. To reduce the code that is required and the risk of introducing bugs, you will create an abstract class that will contain the common functionality.

1. Open the MeasureDataDevice.cs file.
2. Remove the TODO comment and add an **abstract** class named **MeasureDataDevice**.
3. Modify the **MeasureDataDevice** class declaration to implement the **IMeasuringDevice** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Bring the **DeviceControl** namespace into scope.
5. In the **MeasureDataDevice** class, add a **public abstract** method named **MetricValue**. This method should return a **decimal** value, but not take any parameters.

   The implementation of the **MetricValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes. Declaring the **MetricValue** method as **abstract** forces child classes to implement this method.



Hint: Look at the code for the **MetricValue** method for the **MeasureLengthDevice** and **MeasureMassDevice** classes. You will observe that they are quite similar, apart from the conversion factors that are used, and you could factor this logic out into a method in the abstract **MeasureDataDevice** class. However, for the sake of this exercise, assume that these methods are totally different. The same note applies to the **ImperialValue** method that you will define in the next step.

6. In the **MeasureDataDevice** class, add a **public abstract** method with a **decimal** return type named **ImperialValue**.

   Like the **MetricValue** method, the implementation of the **ImperialValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes.
7. In the MeasureLengthDevice.cs file, locate and copy the code for the **StartCollecting** method, and then add this method to the **MeasureDataDevice** class.

   Visual Studio will warn you that the controller variable, the **measurementType** enumeration, and the **GetMeasurements** method are not defined. You will add these items to the **MeasureDataDevice** class in later steps in this task.
8. Copy the **StopCollecting** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

   Visual Studio will warn you that the controller variable is not defined.
9. Copy the **GetRawData** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

   Visual Studio will warn you that the dataCaptured variable is not defined.
10. Copy the **GetMeasurements** method from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

MCT USE ONLY. STUDENT USE PROHIBITED

Visual Studio will warn you that the dataCaptured, controller, and mostRecentMeasure variables are not defined.

11. Copy the five fields in the following table from the MeasureLengthDevice.cs file to the **MeasureDataDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

The warnings in the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods should disappear.

12. In the **MeasureDataDevice** class, modify the five fields that you added in the previous step to make them visible to classes that inherit from the abstract class.
13. Modify the declaration of the **measurementType** field so that it is no longer constant and not instantiated when it is declared.
14. Build the solution and correct any errors.

▶ Task 6: Modify the MeasureLengthDevice and MeasureMassDevice classes to inherit from the MeasureDataDevice abstract class

In this task, you will remove the duplicated code from the **MeasureLengthDevice** and **MeasureMassDevice** classes by modifying them to inherit from the **MeasureDataDevice** abstract class that you created in the previous task.

1. In the MeasureLengthDevice.cs file, modify the declaration of the **MeasureLengthDevice** class so that, in addition to implementing the **IMeasuringDevice** interface, it also inherits from the **MeasureDataDevice** class.
2. Remove the **StartCollecting** method from the **MeasureLengthDevice** class.
3. Remove the **StopCollecting** method from the **MeasureLengthDevice** class.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Remove the **GetRawData** method from the **MeasureLengthDevice** class.
5. Remove the **GetMeasurements** method from the **MeasureLengthDevice** class.
6. Remove the fields in the following table from the **MeasureLengthDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

7. Modify the constructor to set the **measurementType** field to **DeviceType.LENGTH**.
8. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
9. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
10. In the MeasureMassDevice.cs file, modify the declaration of the **MeasureMassDevice** class so that it inherits from the **MeasureDataDevice** class.
11. Remove the **StartCollecting** method from the **MeasureMassDevice** class.
12. Remove the **StopCollecting** method from the **MeasureMassDevice** class.
13. Remove the **GetRawData** method from the **MeasureMassDevice** class.
14. Remove the **GetMeasurements** method from the **MeasureMassDevice** class.
15. Remove the fields in the following table from the **MeasureMassDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private

MCT USE ONLY. STUDENT USE PROHIBITED

Name	Type	Accessor
controller	DeviceController	private
measurementType	DeviceType	private

16. Modify the constructor to set the **measurementType** field to **DeviceType.MASS**.
17. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
18. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
19. Build the solution and correct any errors.

▶ Task 7: Test the classes by using the test harness

In this task, you will check that the **MeasureLengthDevice** and **MeasureMassDevice** classes still work as expected.

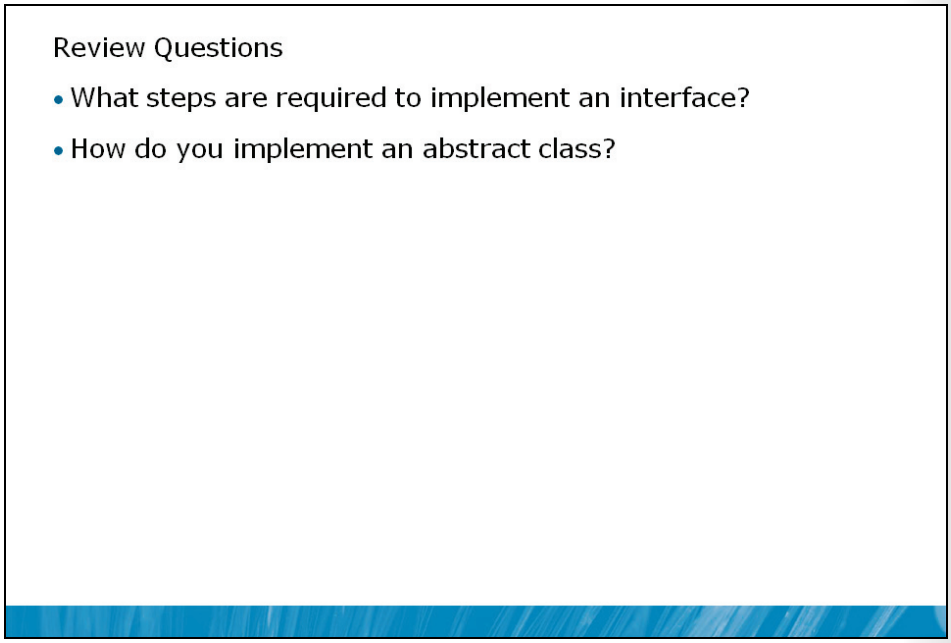
1. Start the Exercise3TestHarness application.
2. Choose **Imperial**, choose **Mass Device**, and then click **Create Instance**.
3. Click **Start Collecting**.
4. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
5. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
6. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
7. Click **Stop Collecting**.
8. Choose **Metric**, choose **Length Device**, and then click **Create Instance**.
9. Click **Start Collecting**. This button starts the new **device** object.
10. Wait for 10 seconds.
11. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.

MCT USE ONLY. STUDENT USE PROHIBITED

12. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
13. Click **Stop Collecting**.
14. Close the Exercise 3 Test Harness window.
15. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

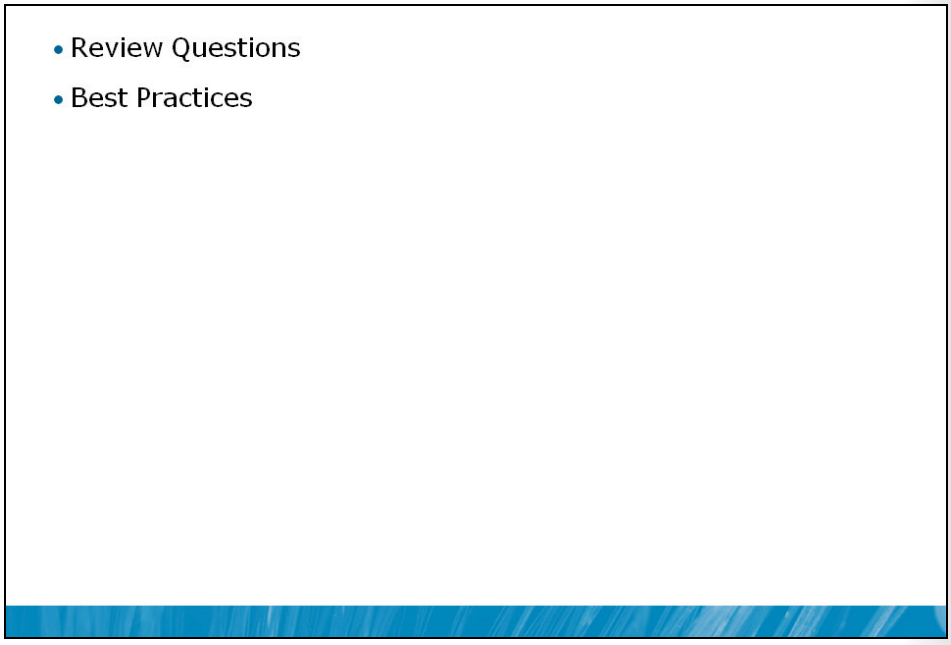


Review Questions

1. What steps are required to implement an interface?
2. How do you implement an abstract class?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What is the role of the object class in the .NET Framework?
2. What is the difference between implicit and explicit implementation of an interface?
3. What are the advantages of using an abstract class over an interface?

Best Practices Related to Inheritance

Supplement or modify the following best practices for your own work situations:

- Create an inheritance hierarchy where appropriate to reduce code duplication.
- Where appropriate, mark methods as **virtual** to enable child classes to override them.
- Where appropriate, mark methods as **sealed** to prevent child classes from overriding them.

MCT USE ONLY. STUDENT USE PROHIBITED

- Where appropriate, mark classes as **sealed** to prevent classes from inheriting from them.

Best Practices Related to Interfaces

Supplement or modify the following best practices for your own work situations:

- Use interfaces wherever possible as a contract that specifies what methods a class will expose.
- Prefix an interface with an uppercase I.
- Implement interfaces explicitly wherever possible.

Best Practices Related to Abstract Classes

Supplement or modify the following best practices for your own work situations:

- Use abstract classes to abstract common functionality and reduce code duplication.
- Use abstract methods to guarantee that an inheriting class overrides a method.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 9

Managing the Lifetime of Objects and Controlling Resources

Contents:

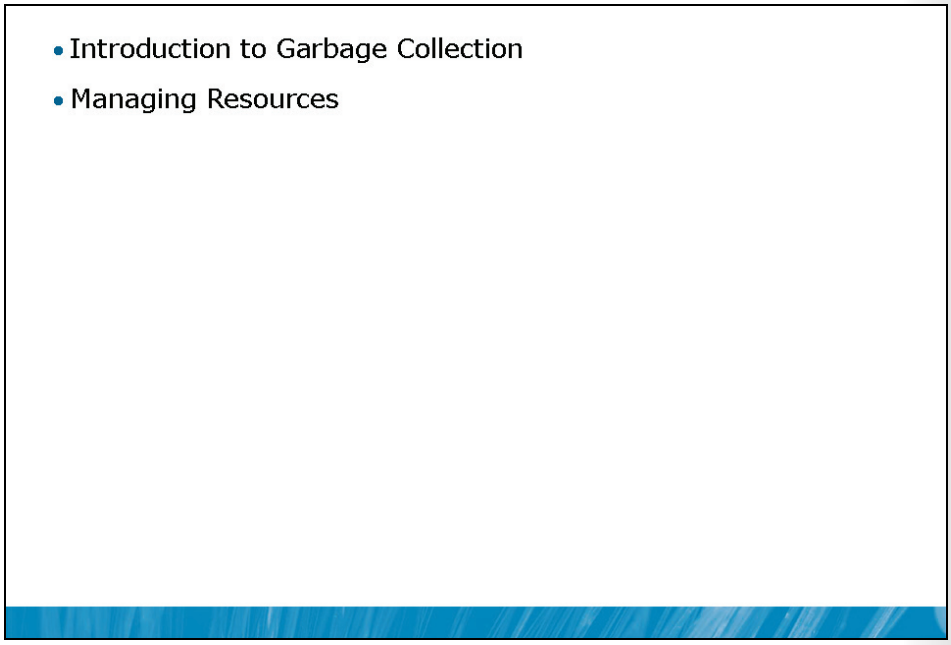
Lesson 1: Introduction to Garbage Collection 9-4

Lesson 2: Managing Resources 9-21

Lab: Managing the Lifetime of Objects and Controlling Resources 9-35

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



All applications use resources. When you build a Microsoft® Visual C#® application, resources fall into two broad categories: managed resources that are handled by the common language runtime (CLR) and unmanaged resources that are maintained by the operating system outside the scope of the CLR. A managed resource is typically an object based on a class defined by using a managed language, such as Visual C#. Examples of unmanaged resources include items implemented outside the Microsoft .NET Framework, such as Component Object Model (COM) components, file handles, database connections, and network connections.

Resource management is important in any applications that you develop. The NET Framework simplifies resource management by automatically reclaiming the resources by a managed object when it is no longer referenced by an application. Managed resources are handled by the .NET Framework garbage collector. However, unmanaged resources are not controlled by the garbage collector; you must take special steps to dispose of them properly and prevent them from being held longer than necessary.

MCT USE ONLY. STUDENT USE PROHIBITED

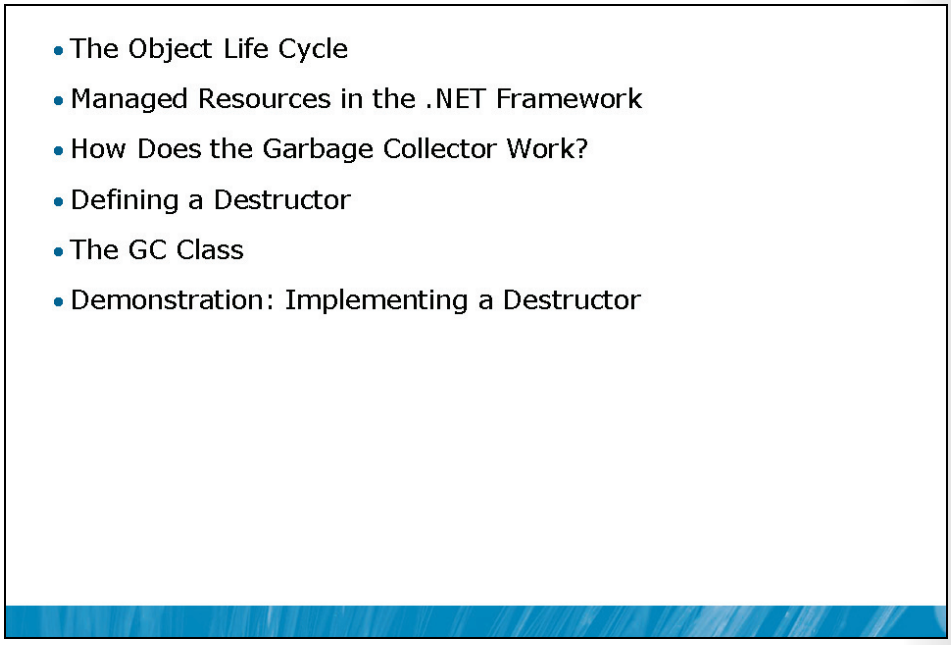
Objectives

After completing this module, you will be able to:

- Describe how garbage collection works in the .NET Framework.
- Manage resources effectively in an application.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Introduction to Garbage Collection



Every object that you create has a life cycle, from creation to destruction. When an object is destroyed, its state must be cleaned, and any managed resources used must be reclaimed. In the .NET Framework, the garbage collector performs these tasks.

This lesson introduces garbage collection in the .NET Framework.

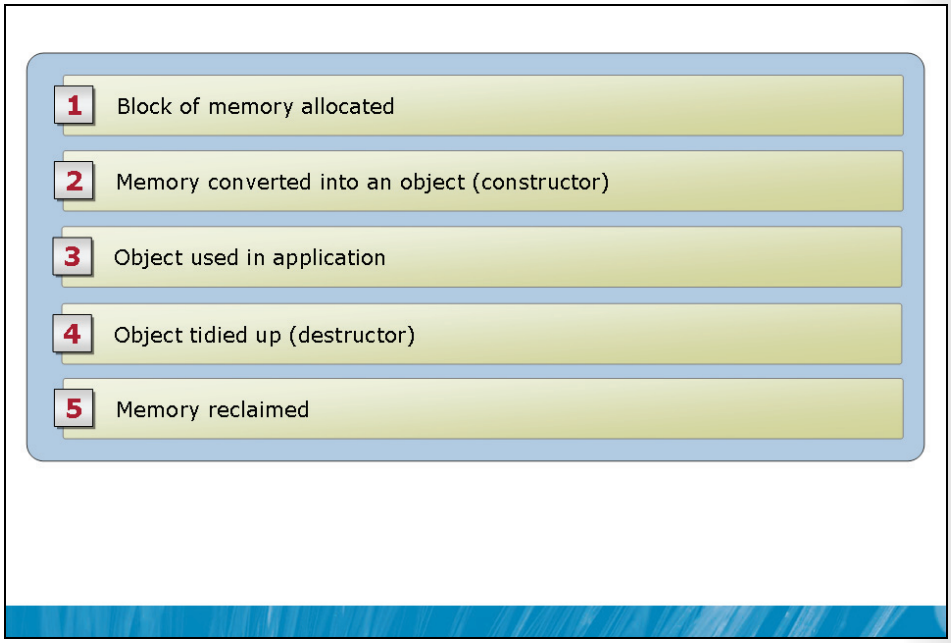
Objectives

After completing this lesson, you will be able to:

- Describe the life cycle of an object.
- Explain how memory is allocated for managed resources.
- Describe the operation of the garbage collector.
- Define a destructor.
- Explain the use of the **GC** class.

MCT USE ONLY. STUDENT USE PROHIBITED

The Object Life Cycle



Key Points

An object has several distinct stages in its life cycle, which starts at creation and ends in destruction. As a developer, the process that you use to create an object is very simple; you use the **new** keyword to instantiate the new object. However, the process that you use to create an object is not really this simple. When you create a new object, the following things happen:

1. A block of memory is allocated. This block of memory is big enough to hold the object.
2. The block of memory is converted to an object. The object is initialized.

You can control only the second of these two steps—converting the block of memory to an object. You can control this step by implementing a constructor.

The runtime handles the allocation of memory for managed objects; however, if you call unmanaged libraries, you may need to manually allocate memory for unmanaged objects you create.

MCT USE ONLY. STUDENT USE PROHIBITED

After an object is created, you can use the properties, methods, and other members.

Destroying an Object

When you have finished with an object, it can be destroyed.

You use destruction to reclaim any resources used by that object. Like creation, destruction is a two-phase process:

1. The object is cleaned up; for example, by releasing any unmanaged resources used by the application, such as file handles and database connections.
2. The memory used by the object is reclaimed.

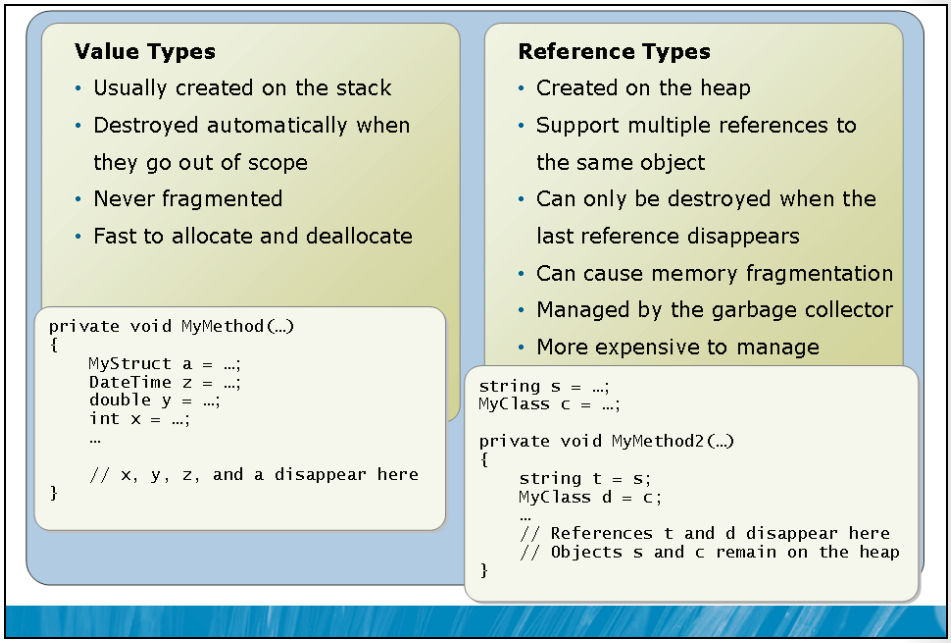
You can control only the first of these steps—cleaning up the object and releasing resources. You can control this step by implementing a destructor.

The CLR handles the release of memory used by managed objects; however, if you use unmanaged objects, you may need to manually release the memory used by these items.

Question: How can you control the creation phase for an object?

MCT USE ONLY. STUDENT USE PROHIBITED

Managed Resources in the .NET Framework



Key Points

The .NET Framework divides the items that a managed application can use into two broad categories: value types and reference types.

Managing Value Types

Value types are managed types that are usually created on the stack. The CLR manages the stack. When an object on the stack goes out of scope, the memory used by that object is reclaimed immediately. For example, at the end of a method, any variables based on value types (created on the stack) defined in that method are destroyed. The stack is a last in, first out (LIFO) structure. The CLR maintains a pointer to the top of the stack. When a value type variable is created, it is placed at the top of the stack and the stack pointer is moved up. When the variable goes out of scope, the stack pointer is moved back down again. In this way, new items overwrite old items and memory is reclaimed automatically; memory management is therefore a relatively inexpensive operation.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Managing Reference Types

Reference types are allocated memory from the heap. The heap is a block of memory also controlled by the CLR, separate from the stack. When you create an object, the CLR allocates memory for the object and creates a reference to it on the stack. Unlike a value type, a reference type can have several references to the same object. If you refer to the same object several times, the reference disappears when it goes out of scope, but other references to the same object that are still in scope remain valid. An object can be destroyed, its destructor run, and its resources reclaimed only when the final reference to the object disappears. Consequently, the lifetime of an object is not governed by the scope of any single reference to that object.

An important feature of the .NET Framework garbage collector is to monitor an object on the heap and determine when the last reference to that object has disappeared. The object can then be safely destroyed. Determining when an object has no references can be a time-consuming and expensive operation, so the garbage collector performs this task only when it needs to, typically when the amount of memory available on the heap falls below some threshold.

A second function of the garbage collector is to defragment the heap. If an application attempts to create an object for which there is currently insufficient contiguous empty space available on the heap, the garbage collector will attempt to move some existing objects around and compact the resulting free space into a chunk of memory big enough to hold the new object. Again, this can be a computationally expensive task.

Value Types on the Heap

Value types are usually created on the stack. However, there is one scenario where this does not occur. When you develop a class, you can use a value type as a field in that class. When you instantiate an object by using this class, the fields that constitute the object, including any value type fields, are allocated space on the heap. These value type fields remain active until the containing object is destroyed and the garbage collector reclaims the space used by these fields.

If you use a value type as a field in a struct, it will be stored as part of the struct. Structs are value types and as such are normally stored on the stack, unless they are elements inside a class. If the value type is an element inside a class, it is stored on the heap as described in the previous paragraph.

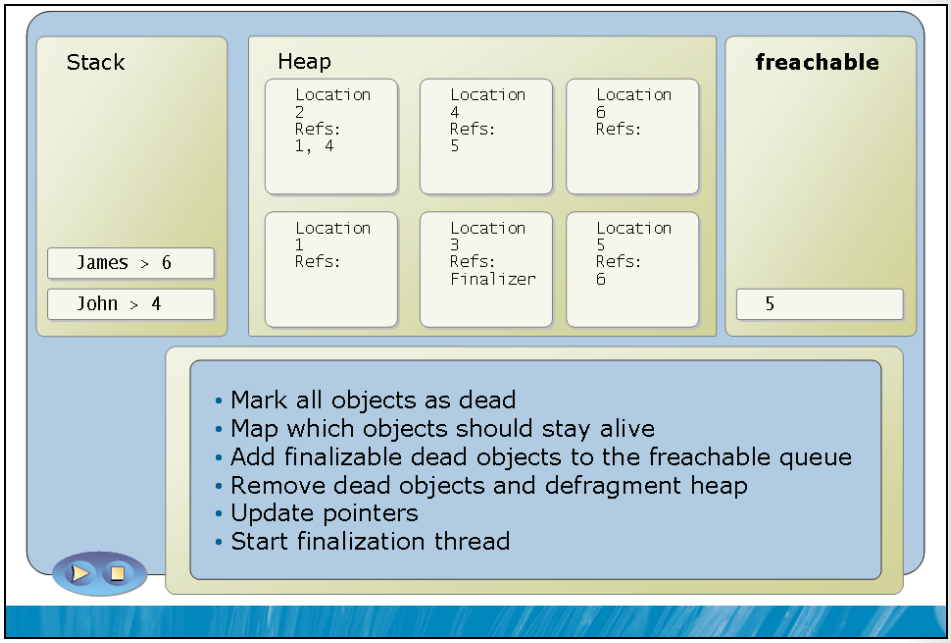
Question: Is the stack a first in, first out (FIFO), LIFO, first in, last out (FILO), or random access memory store?

Additional Reading

For more information about automatic memory management, see the 3.9 Automatic memory management page at http://go.microsoft.com/fwlink/?LinkId=192943.

MCT USE ONLY. STUDENT USE PROHIBITED

How Does the Garbage Collector Work?



Key Points

The garbage collector releases resources and memory for objects stored on the heap.

The garbage collector runs in its own thread and normally runs automatically, under well-defined circumstances. When the garbage collector runs, other threads in an application are halted because the garbage collector may move objects in memory and must update pointers to the correct addresses for these objects.

The garbage collector takes the following steps to reclaim resources:

1. It marks every object as dead; objects are considered dead unless proved otherwise.
2. It starts from objects referenced on the stack, marking referenced objects as alive. It performs this recursively; if an object that is already marked as alive references another object, that object is also marked as alive. The garbage collector includes logic to prevent infinite recursion, for example, where there is a circular reference between two objects.

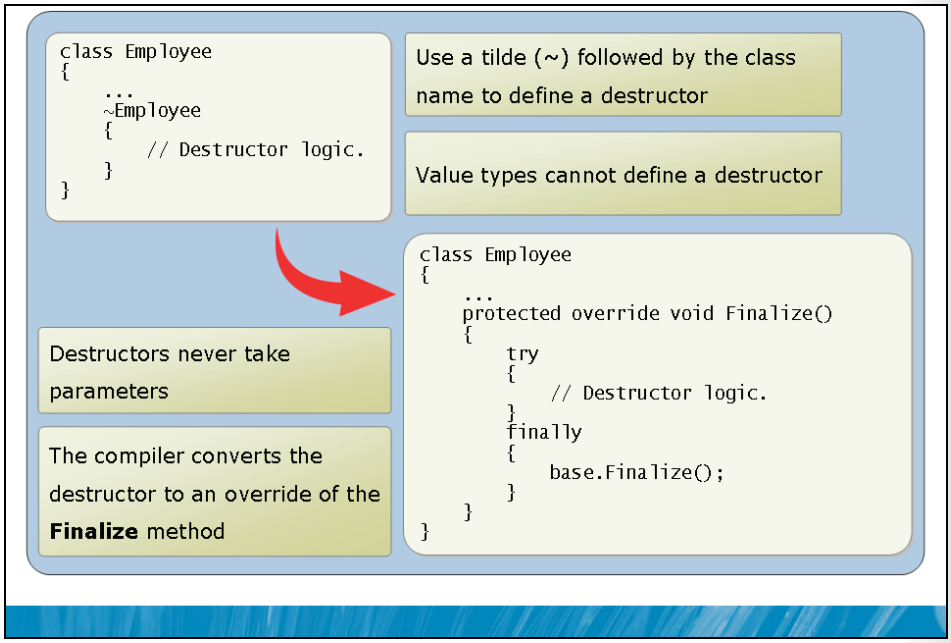
MCT USE ONLY. STUDENT USE PROHIBITED

3. It checks whether any of the objects that have been marked as dead have a destructor that must be run. Running the destructor is referred to as finalization. Any objects that require finalization are moved to a data structure maintained by the garbage collector called the **freachable** queue. The **freachable** queue stores pointers to objects that require finalization before their resources can be reclaimed.
4. Objects added to the **freachable** queue are marked as alive because there is now a valid reference to them; the destructor must be run before their memory can be reclaimed. Objects are normally added to the **freachable** queue only once.
5. Objects marked as alive are moved down the heap to form a contiguous block, defragmenting the heap. References to objects (on the stack and in other objects on the heap) moved by the garbage collector are updated.
6. Other threads resume.
7. On a separate thread, objects added to the **freachable** queue are finalized. After an object is finalized, the pointer to that object is removed from the **freachable** queue. Objects are not removed from memory until the next time the garbage collector runs.

Question: What is the purpose of the **freachable** queue?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining a Destructor



Key Points

You can add a destructor to a class to perform any additional application-specific cleanup that is necessary when your class is garbage collected.

Important: Defining a destructor adds an overhead to the process of garbage collecting an object; it must be added to the **freachable** queue so that the garbage collector can run the destructor. The garbage collector automatically manages the memory for managed objects, so you should implement a destructor only if an object references unmanaged resources that must be reclaimed when an object is destroyed.

Important: The point at which the destructor runs is not specified, and you cannot guarantee the order in which destructors for different objects will run. Therefore, do not make any assumptions or introduce dependencies between objects in a destructor.

To define a destructor, you add a tilde (~) followed by the name of the class. You then enclose the destructor logic in braces.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows the syntax for adding a destructor.

```
class Employee
{
    ...

    // Destructor
    ~Employee
    {
        // Destructor logic.
    }
}
```

The following restrictions apply to destructors:

- You cannot add a destructor to a struct or any other value type. Value types are stored on the stack, so garbage collection does not apply.
- You can never declare an access modifier for a destructor. Destructors are only ever called by the garbage collector, and you cannot call them from your code.
- You cannot declare a destructor that takes parameters. The garbage collector calls the destructor, and you have no control to pass parameters to it.

When you declare a destructor, the compiler automatically converts it to an override of the **Finalize** method of the object class; however, note that you cannot override the **Finalize** method yourself. You must declare a destructor, and the compiler performs the conversion.

The compiler will convert the destructor in the previous code example to the override of the **Finalize** method, as the following code example shows.

```
protected override void Finalize()
{
    try
    {
        // Destructor logic.
    }
    finally
    {
        base.Finalize();
    }
}
```

The compiler adds the logic from the destructor to a **try** block and then calls the **Finalize** method of the base class in the **finally** block. This ensures that the

MCT USE ONLY. STUDENT USE PROHIBITED

Finalize method of the base class is always called even if your code throws an exception.

When a class with a destructor is eligible for garbage collection, it is added to the **freachable** queue. The finalization thread then runs the finalization code. After an object is finalized, the reference is removed from the **freachable** queue and the object is eligible for garbage collection again. You should only define a destructor in classes that specifically require this functionality and omit them from all other classes to avoid this performance hit.

Question: Can you add a destructor to a struct?

Additional Reading

For more information about destructors, see the Destructors (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192944.

MCT USE ONLY. STUDENT USE PROHIBITED

The GC Class

Name	Description
Collect	Forces garbage collection
WaitForPendingFinalizers	Suspends the current thread until all of the objects in the **freachable** queue have been finalized
SuppressFinalize	Prevents finalization of the object passed as the parameter
ReRegisterForFinalize	Requests finalization for an object that has either already been finalized or had finalization suppressed
AddMemoryPressure	Informs the runtime that you must allocate a large block of unmanaged memory
RemoveMemoryPressure	Informs the runtime that you have released a large block of unmanaged memory

Key Points

Most of the time, you should let the garbage collector perform operations in its own time as directed by the CLR. However, under some circumstances, you may need to explicitly request that the garbage collector is invoked or modify the way in which it runs. To do this, you can use the **GC** class.

The **GC** class includes several static methods that you can call from your code. The following table includes some of the more frequently used methods exposed by the **GC** class.

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Notes
Collect	Forces garbage collection.	You should avoid using the **Collect** method in your code. If you force the garbage collector to run more often than necessary, it may have a negative performance impact on your application. The **Collect** method is asynchronous; when it returns, there is no guarantee that the garbage collection is complete, or even started, only that the garbage collector will run at the next suitable interval. `GC.Collect();`
WaitForPendingFinalizers	Suspends the current thread until all objects in the **freachable** queue have been finalized.	You can use this method if you specifically need to wait for all objects currently in the **freachable** queue to finalize. `GC.WaitForPendingFinalizers();`
SupressFinalize	Prevents finalization of the object passed as the parameter.	You should always call this method when you implement the dispose pattern. This method can improve performance by preventing finalization code from running twice. `GC.SuppressFinalize(this);`
ReRegisterForFinalize	Requests finalization for an object that has either already been finalized or had finalization suppressed.	You can use this method if you have suppressed finalization for an object, or an object has already been finalized but you require the runtime to finalize the object again. `GC.ReRegisterForFinalize(this);`

MCT USE ONLY. STUDENT USE PROHIBITED

Method	Description	Notes
AddMemoryPressure	Informs the runtime that you must allocate a large block of unmanaged memory.	This method informs the runtime that you are about to allocate a large block of memory and it will free resources where possible. When you use this method, you must specify how much memory you need to allocate. If you need to allocate several blocks of memory, you can call this several times in your application. You should call this method before allocating a large block of unmanaged memory. You should not use this method if you are creating managed objects. `GC.AddMemoryPressure(1000);`
RemoveMemoryPressure	Informs the runtime that you have released a large block of unmanaged memory.	This method informs the runtime that you have released a large block of memory and it will reduce the urgency with which it performs garbage collection. When you use this method, you must specify how much memory you have released. If you must release several blocks of memory, you can call this several times in your application. You should call this method after releasing a large block of unmanaged memory. You should not use this if you are destroying managed objects. You should always use the **AddMemoryPressure** and **RemoveMemoryPressure** methods together to ensure that you add and remove exactly the same amount of memory pressure. `GC.RemoveMemoryPressure(1000);`

MCT USE ONLY. STUDENT USE PROHIBITED

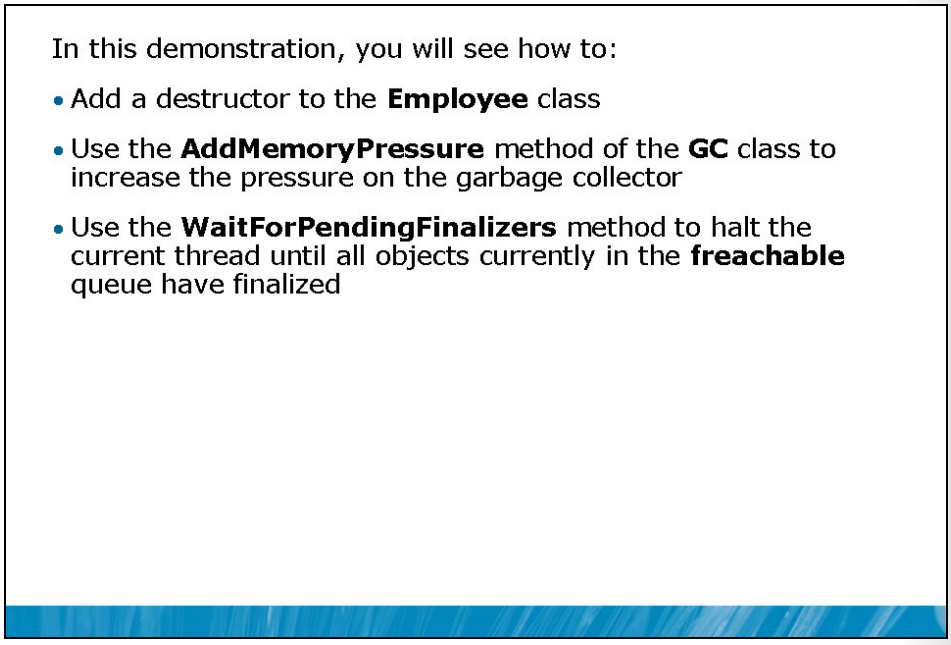
Question: How can you inform the runtime that you need to allocate a large block of unmanaged memory?

Additional Reading

For more information about the **GC** class, see the GC Members page at http://go.microsoft.com/fwlink/?LinkId=192945.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Implementing a Destructor



Key Points

- Add a destructor to the **Employee** class.
- Use the **AddMemoryPressure** method of the **GC** class to increase the pressure on the garbage collector.
- Use the **WaitForPendingFinalizers** method to halt the current thread until all objects currently in the **freachable** queue have finalized.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Start Microsoft Visual Studio® 2010:
3. Open the **Destructor Demo** solution in the **E:\Demofiles\Mod9\Demo1\Starter\Destructor Demo** folder.

MCT USE ONLY. STUDENT USE PROHIBITED

4. In the Employee.cs file, add a destructor to the **Employee** class. The destructor should write the current salary to the SalaryDetails.txt file.

   Your code should resemble the following code example.

```
~Employee()
{
    File.WriteAllText("SalaryDetails.txt", salary.ToString());
    Console.WriteLine("Employee finalized: {0}", name);
}
```

5. Run the application with debugging.
6. Notice that all of the employees are paid the same amount; the pay rises do not reflect the update made to the text file by the destructor. This is because the garbage collector only runs when it needs to, and this program does not create enough objects or use enough memory to cause a collection.
7. Stop debugging.
8. In the Program.cs file, uncomment the call to the **AddMemoryPressure** method.
9. Run the application with debugging.

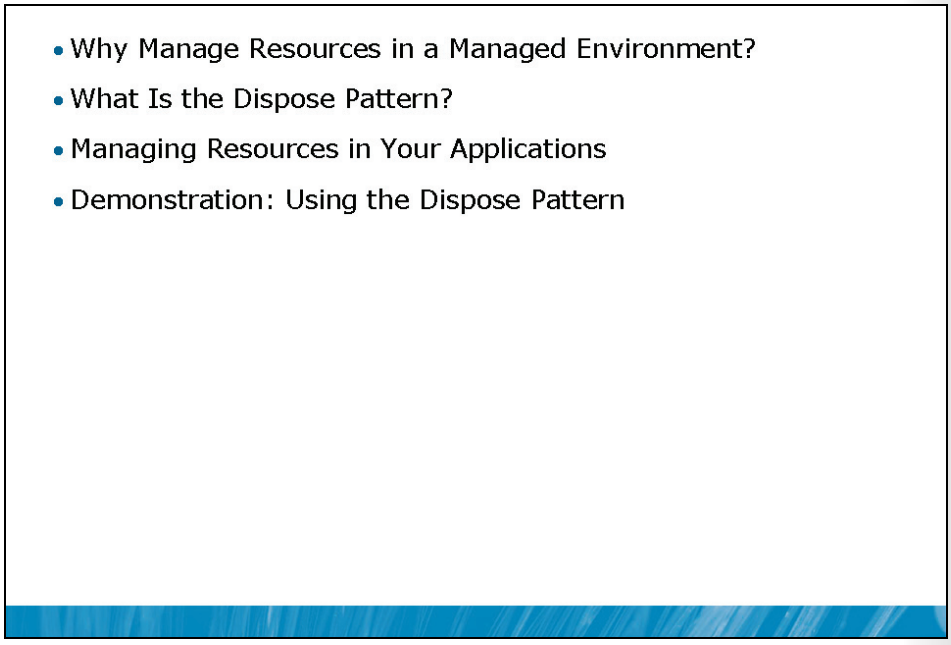
   Notice that the results have not changed—although the application now requires more memory, the application runs faster than the garbage collector finalizes the objects.
10. Stop debugging.
11. In the Program.cs file, uncomment the call to the **WaitForPendingFinalizers** method.
12. Run the application with debugging.

    Notice that the application now works as expected, because the application halts to wait for the finalizer to update the text file before continuing.

Question: How can you delay execution of the current thread until all objects in the finalization queue are finalized?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Managing Resources



The garbage collector automatically reclaims memory and resources for managed objects. However, if you use unmanaged resources in a class, you must take additional steps to ensure that they are released appropriately.

The dispose pattern is a design pattern that enables you to release unmanaged resources used by a class in a controlled and timely manner. Implementing the dispose pattern in your types will help to ensure that your applications perform well and do not retain unmanaged resources longer than necessary.

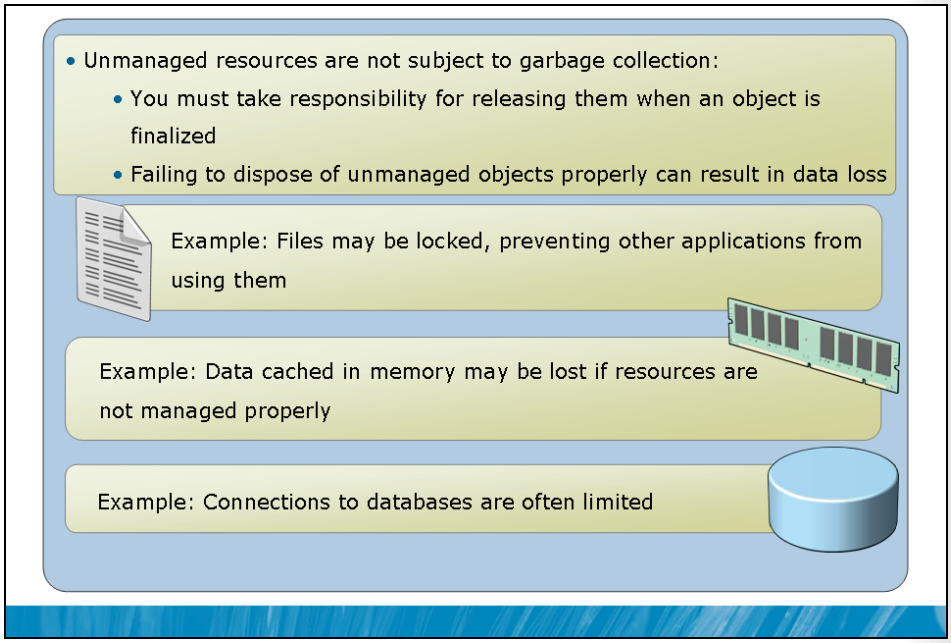
Objectives

After completing this lesson, you will be able to:

- Explain the need for resource management in a managed code.
- Describe the dispose pattern.
- Manage resources in applications.

MCT USE ONLY. STUDENT USE PROHIBITED

Why Manage Resources in a Managed Environment?



Key Points

The garbage collector is concerned with managed objects. It does not understand how to release the resources associated with unmanaged objects. If you reference an unmanaged resource in a class, when you remove the last reference to the class, the unmanaged object will not be destroyed, it will simply be orphaned. The operating system may not clean up the resource until your application terminates.

For example, the .NET Framework class library provides the **TextWriter** class that you can use to open a file on the local file system and write text to that file. The **TextWriter** class acts as a managed wrapper around text files, which are unmanaged resources controlled by the operating system.

When the **TextWriter** object opens a file, the operating system locks the file to ensure that no other processes can write to the same file. When you have finished using the **TextWriter** object in your code, you can remove all references to it. This action will destroy the managed **TextWriter** object, but it may not release the lock because it is part of an unmanaged resource that is not controlled by the garbage collector. You must take additional steps to release this lock; otherwise, if you then attempt to create another **TextWriter** object to write to the same file, it will fail.

MCT USE ONLY. STUDENT USE PROHIBITED

In addition to unmanaged locks, there are several other problems associated with incorrect resource management. For example, some unmanaged types use in-memory buffers to improve performance and only write to the underlying data source when either the buffer is full or the object is explicitly instructed to. If you fail to flush these buffers when an object is destroyed, the contents of these buffers may be lost. As an example, when you write data to a file by using the **TextWriter** class, the data may be buffered by the underlying file type. If you destroy the **TextWriter** object without releasing the resources associated with the file, the buffer may not be flushed correctly, and you may lose data. The **TextWriter** class provides the **Flush** method to write the contents of the buffer to the file system, which you can call to ensure that all data is written to the file before you destroy a **TextWriter** object.

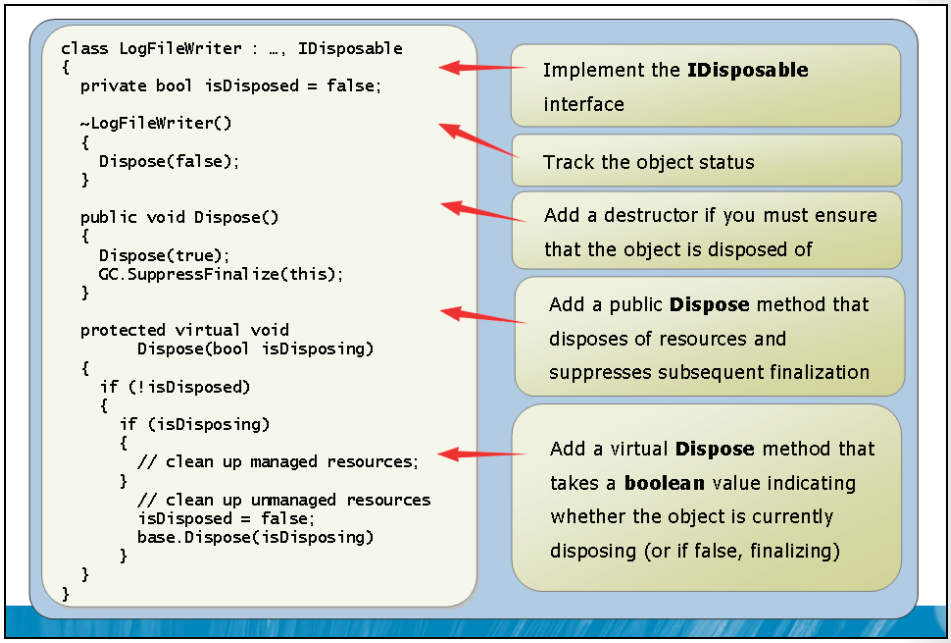
Database connections are another resource that are both expensive to maintain and often limited. Database servers frequently support only a limited number of concurrent connections. If you fail to release database connections when you have finished using them, the available database connections will soon deplete, and your application may throw unexpected exceptions when it tries to connect to the database.

If you manage resources correctly and ensure that all unmanaged resources are released when they are no longer required, you can prevent these problems.

Question: What types of resources may need to be managed correctly?

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the Dispose Pattern?



Key Points

The dispose pattern is a design pattern that frees resources used by an object. The .NET Framework provides the **IDisposable** interface, and objects that implement this interface should follow the dispose pattern.

The IDisposable Interface

The **IDisposable** interface defines a single method called **Dispose**, which takes no parameters. The **Dispose** method should release all of the unmanaged resources that an object owns. It should also release all resources owned by its base types by calling the **Dispose** method of the parent type.

Visual C# includes constructs that ensure that resources are released in a timely fashion for classes that implement the **IDisposable** interface.

Many of the classes in the .NET Framework that wrap unmanaged resources, such as the **TextWriter** class, implement the **IDisposable** interface. You should also implement the **IDisposable** interface when you create your own classes that reference unmanaged types.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Tracking Object Disposal

Calling the **Dispose** method does not destroy an object, and an object remains in existence when the **Dispose** method completes; the object is destroyed only after the final reference to it is removed and the garbage collector reclaims any remaining resources it is using. Therefore, when you implement the dispose pattern in a class, you should track the disposal status of the object and check whether the **Dispose** method has already been invoked and the resources released. A common technique is to add an **isDisposed Boolean** field to your class, set it in the **Dispose** method, and check it in every other method in your class. If methods in your class are called after disposal, this is normally a mistake, and you should throw an **ObjectDisposedException** exception.

The exception to this rule is the **Dispose** method itself. You should be able to run the **Dispose** method multiple times without throwing any exceptions or resulting in an inconsistent state. Your **Dispose** method should include logic to check the state of resources that are about to be released before it releases them. The following code example shows an example of a class that incorporates a **TextWriter** object, and that implements the **IDisposable** interface. This example uses the **Dispose** method to ensure that the **TextWriter** object is closed correctly and the underlying file resources are reclaimed.

```
class LogFileWriter : IDisposable
{
    private bool isDisposed = false;
    private TextWriter writer = ...;
    ...

    public void WriteDataToFile(...)
    {
        // Check that the current object has not been disposed of
        if (isDisposed)
            throw new ObjectDisposedException(...);
        ...
    }

    public void Dispose()
    {

        if (!isDisposed)
        {
            // Only close the TextWriter if it is not null
            // (in which case it has already been disposed)

            if (writer != null)
            {
                writer.Flush();
```

```
                writer.Close();
                writer = null;
            }

            // Indicate that the object has been disposed of and
            // resources have been released

            isDisposed = true;
        }
    }
}
```

Calling Dispose from a Destructor

If you must guarantee that the **Dispose** method is always invoked, you can include it as part of the finalization process performed by the garbage collector. To do this, you can add a destructor to your class and call the **Dispose** method. However, remember that finalization is a potentially expensive process, so you should implement this strategy only if you really must.

Disposing of Managed Resources Early

In some cases, you may want to dispose of managed resources in addition to unmanaged resources. This is typically the case if a managed resource is no longer required and is expensive to maintain, such as a large array. The garbage collector will reclaim this memory eventually when your object is destroyed, but you may be able to free the memory used by this array early by setting the reference to this array to null in the **Dispose** method. Note that this strategy does not guarantee that the memory used by the array will be collected early, only that it *may* be. The timing depends on the garbage collector.

If you invoke the **Dispose** method from a destructor, in addition to allowing an application to invoke the **Dispose** method manually, there is little point in trying to dispose of the managed resources more than once. In this case, the recommended approach is to overload the **Dispose** method and provide an implementation that takes a Boolean flag that indicates whether the **Dispose** method was called as part of the finalization process or directly by application code. The convention is to pass the value **true** if the **Dispose** method is called by an application and **false** if it is called by a destructor. The overloaded **Dispose** method should only dispose of managed resources if it was called directly (the parameter is **true**). When it is false, the managed resources will either have already been disposed of or will be about to be disposed of by the garbage collector anyway. In this case, the **Dispose** method should only attempt to release unmanaged resources. The public **Dispose** method, which takes no parameters and is defined as part of the **IDisposable** interface, can then simply call **Dispose(true)**, and the destructor can call **Dispose(false)**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

It is good practice to make the overloaded implementation of the **Dispose** method **protected** and **virtual**. In this way, it can only be accessed by code in the class and any child classes, but child classes can override it if they define additional resources that must be disposed of. The overloaded **Dispose** method should also invoke the **Dispose** method of any parent class if the parent class implements the dispose pattern.

The following code example shows a class that uses this strategy.

```
class LogFileWriter : ..., IDisposable
{
    private bool isDisposed = false;
    private TextWriter writer = ...;
    private int largeArray[] = ...;
    ...

    public void WriteDataToFile(...)
    {
        // Check that the current object has not been disposed of
        if (isDisposed)
            throw new ObjectDisposedException(...);
        ...
    }

    public void Dispose()
    {
        Dispose(true);
    }

    ~LogFileWriter()
    {
        Dispose(false);
    }

    protected virtual void Dispose(bool isDisposing)
    {
        if (!isDisposed)
        {
            if (isDisposing)
            {
                // Release managed resources only if Dispose
                // was called by the application
                largeArray = null;
                ...
            }

            // Always release unmanaged resources
            if (writer != null)
            {
```

```
                writer.Flush();
                writer.Close();
                writer = null;
            }

            // Indicate that the object has been disposed of and
            // resources have been released
            isDisposed = true;

            // Call Dispose in the parent class
            // (assuming that the parent provides this method)
            base.Dispose(isDisposing);
        }
    }
}
```

Suppressing Finalization

When you add a destructor to a class, by default, when the garbage collector disposes of resources, the objects are added to the **freachable** queue for finalization. In your public **Dispose** method, after you have released all of the necessary resources, you should call the **static SuppressFinalize** method of the **GC** class, passing in the current object. This call marks the object so that the garbage collector does not waste time running the finalization code for your object, which you have already cleaned up. The following code example shows this method.

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

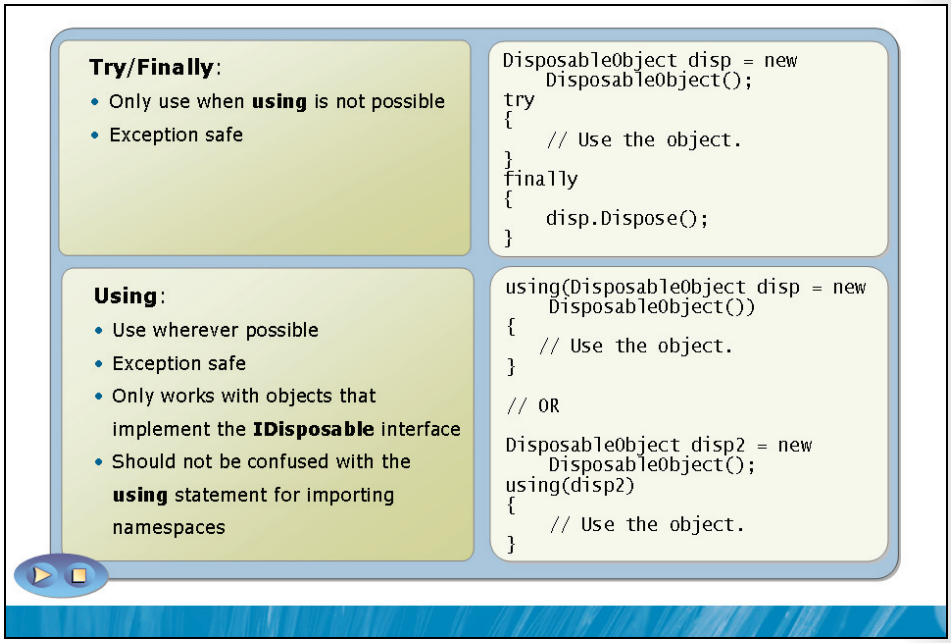
Question: What exception should you throw in your class if you attempt to use it after it has been disposed of?

Additional Reading

For more information about the **Dispose** method, see the Implementing a Dispose Method page at http://go.microsoft.com/fwlink/?LinkId=192946.

MCT USE ONLY. STUDENT USE PROHIBITED

Managing Resources in Your Applications



Key Points

Simply using types that implement the **IDisposable** interface is not sufficient to manage resources; you must remember to invoke the **Dispose** method in your code.

There are several approaches you can use to dispose of an object when you no longer require it:

- You can manually call the **Dispose** method at an appropriate point in your code, as the following code example shows.

```
LogFileWriter lfw = new LogFileWriter(...);
...

// Use the LogFileWriter object.

...
lfw.Dispose();
```

MCT USE ONLY. STUDENT USE PROHIBITED

- You can use a **try**/**finally** block and dispose of your object in the **finally** block, as the following code example shows.

```
LogFileWriter lfw;

try
{
    lfw = new LogFileWriter (...);
    ...
    // Use the LogFileWriter object.
    ...
}

finally
{
    if (lfw != null)
    {
        Lfw.Dispose();
    }
}
```

- You can use a **using** block (not to be confused with the **using** keyword for importing namespaces) to encapsulate your disposable object, as the following code example shows.

```
using (LogFileWriter lfw = new LogFileWriter (...))
{
    ...
    // Use the LogFileWriter object.
    ...
}
```

Using a **using** block (the third option) is the preferred way of guaranteeing that an object is disposed of when you have finished using it. When you add a **using** block to your code, the variables that you define in the **using** statement are accessible only in that block. 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 specified in the **using** statement.

To define a **using** block, you specify the **using** keyword, followed by brackets. Inside the brackets, you declare and initialize the variable that you must use in the block. You then add your code, enclosed in braces after the **using** statement.

The following code example defines a **using** block, which declares and initializes a *LogFileWriter* variable (as shown previously).

MCT USE ONLY. STUDENT USE PROHIBITED

```
using (LogFileWriter lfw = new LogFileWriter (...))
{
    ...
    // Use the LogFileWriter object.
    ...
}
```

The above example is functionally equivalent to the following code example.

```
{
    LogFileWriter lfw;

    try
    {
        lfw = new LogFileWriter (...);
        ...
        // Use the LogFileWriter object.
        ...

    }
    finally
    {
        if (lfw != null)
        {
            lfw.Dispose();
        }
    }
}
```

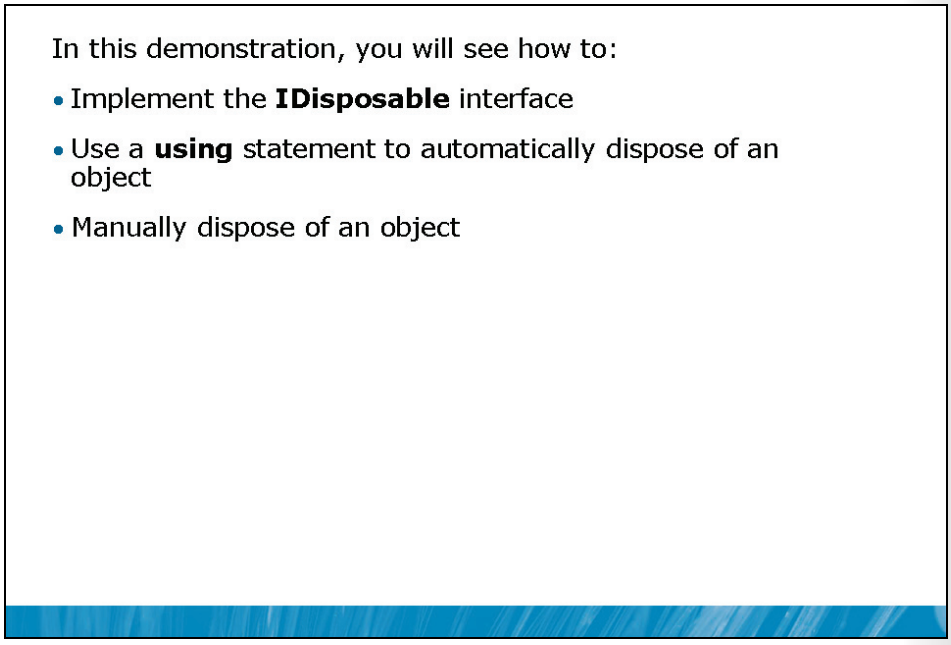
Notice the braces that enclose all of the code. These braces encapsulate the lfw variable and ensure that no other objects reference the **lfw** object. The object goes out of scope as soon as the flow of control leaves the block and therefore makes the object eligible for garbage collection.

If a **using** statement is not appropriate in your code (if the object does not implement the **IDisposable** interface, for example), using a **try**/**finally** block is an exception-safe approach to disposing of an object; if the code in the **try** block throws an exception, the **finally** block will still run calling the **Dispose** method. You should aim to use a **try**/**finally** block wherever a **using** statement cannot be used.

Question: If you do not dispose of an object when you have finished with it, will the runtime call the **Dispose** method automatically?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using the Dispose Pattern



Key Points

- Implement the **IDisposable** interface.
- Use a **using** statement to automatically dispose of an object.
- Manually dispose of an object.

Demonstration Steps

1. Start Visual Studio 2010.
2. Open the **Dispose Demo** solution in the **E:\Demofiles\Mod9\Demo2\Starter\Dispose** folder.
3. Modify the **Employee** class to implement the **IDisposable** interface. The code should follow best practices. Explain to students that although the call to **GC.SuppressFinalize** in the **Dispose** method is unnecessary at present, it is still worth including it in case a finalizer is added to the **Employee** class later.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Employee : IDisposable
{
        bool isDisposed = false;
        string name;

        public Employee(string Name)
        {
            name = Name;
        }

        public void PaySalary()
        {
            if (!isDisposed)
            {
                Console.WriteLine("Employee {0} paid.", name);
            }
            else
            {
                throw new ObjectDisposedException("Employee already
                    disposed.");
            }
        }

        public void Dispose()
        {
            Dispose(true);
        }

        protected virtual void Dispose(bool IsDisposing)
        {
            if (IsDisposing)
            {
                isDisposed = true;
                Console.WriteLine("Employee object disposed.");
            }
            GC.SuppressFinalize(this);
        }
    }
```

4. Run the application with debugging.
5. When the application pauses, press ENTER.

Question: What is the preferred construct for managing resources in an application?

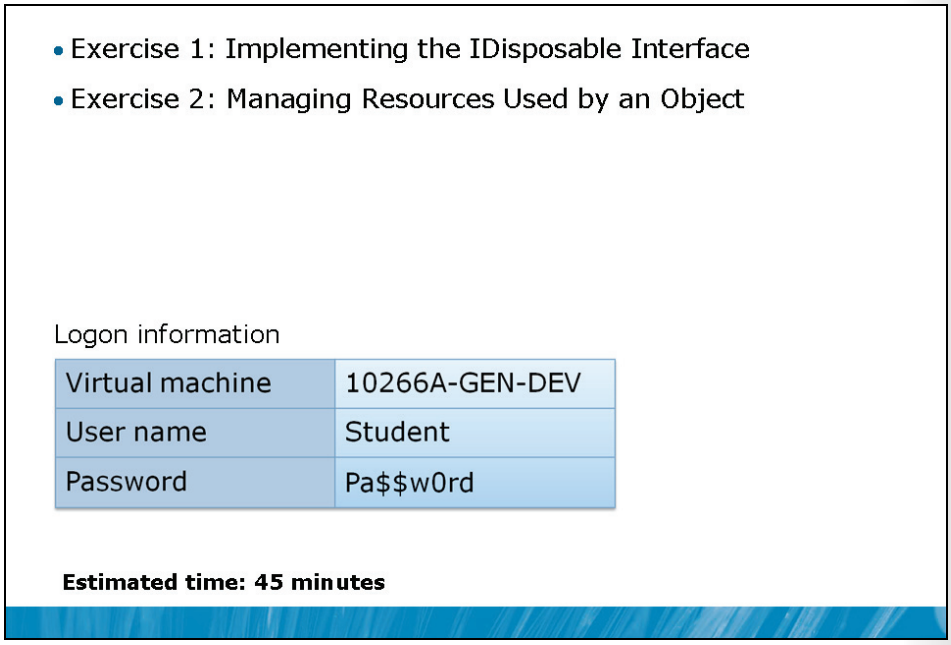
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about the **using** statement, see the using Statement (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192947.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Managing the Lifetime of Objects and Controlling Resources



Objectives

After completing this lab, you will be able to:

- Implement the **IDisposable** interface in a type.
- Ensure that resources associated with an object are reclaimed through a **using** statement.

Introduction

In this lab, you will define a type that implements the **IDisposable** interface and then reference objects of this type through a **using** statement to ensure that they are disposed of correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Setup

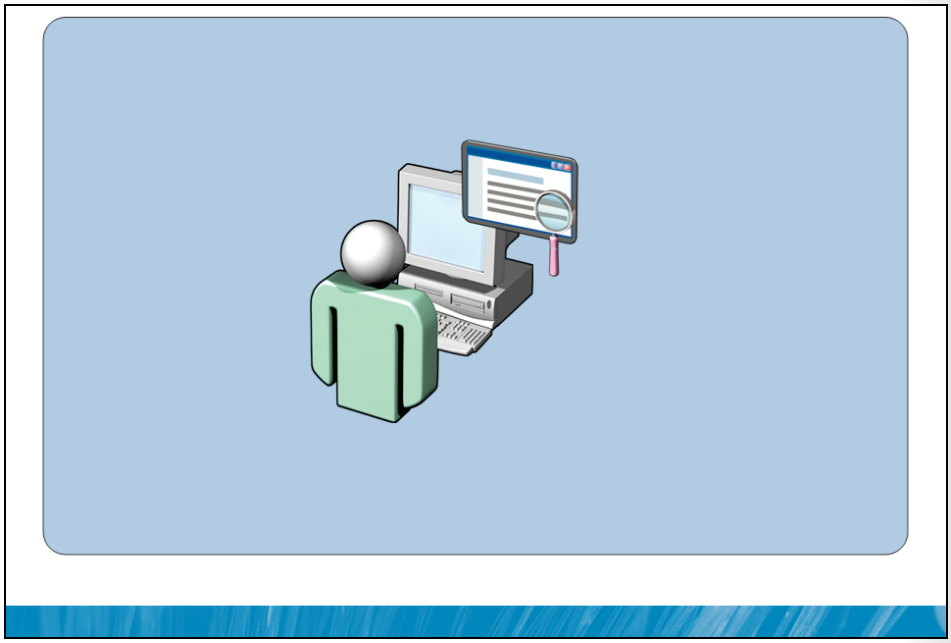
For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

- Start the 10266A-GEN-DEV virtual machine, and then log on by using the following credentials:
  - User name: **Student**
  - Password: **Pa$$w0rd**

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

Lab Scenario



The first version of the family of measuring devices produced by Fabrikam, Inc. recorded data to a local circular buffer on the device, implemented by using an array. However, this array has a fixed, finite size. If the user does not retrieve the data from the device sufficiently often, measurements will be overwritten and lost. You have been asked to develop the software to drive an enhanced version of these devices. The new version supports logging to a file and to the buffer in memory. This should prevent data loss.

Exercise 1: Implementing the IDisposable Interface

Scenario

In this exercise, you will create a new interface called **ILoggingMeasuringDevice** that extends the **IMeasuringDevice** interface and adds the following method:

- **GetLoggingFile**. This method will return the name of the file that the device logs data to.

You will modify the **MeasureDataDevice** abstract class and add the following private field:

MCT USE ONLY. STUDENT USE PROHIBITED

- **loggingFileName**. This field will contain the name of the file that the device will log data to.

You will implement the **GetLoggingFile** method in the abstract class to return the name of the file in the **loggingFileName** field.

In the **StartCollecting** method of the abstract class, you will add code to open the file and record measurements as they are written to the buffer. In the **StopCollecting** method, you will add code to close the file.

You will then extend the abstract class to implement the **IDisposable** interface. In the **Dispose** method, you will add code to ensure that the file is closed correctly and its contents are flushed to disk when the object is destroyed.

You will modify the constructor for the **MeasureMassDevice** class that inherits from the **MeasureDataDevice** abstract class and include a parameter that enables an application to specify a file name. The constructor will use this file name to populate the **loggingFileName** field.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create the **ILoggingMeasuringDevice** interface.
3. Modify the **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface.
4. Modify the **MeasureDataDevice** class to implement the **IDisposable** interface.
5. Modify the **MeasureMassDevice** class to use logging.

▶ Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 9\Snippets folder.
4. Open the Module9 solution in the E:\Labfiles\Lab 9\Ex1\Starter folder.

▶ Task 2: Create the ILoggingMeasuringDevice interface

In this task, you will develop the **ILoggingMeasuringDevice** interface. You will develop this new interface, which inherits from the existing **IMeasuringDevice**

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

interface, rather than editing the existing interface to ensure compatibility with existing code.

1. In Visual Studio, review the task list.
2. Open the ILoggingMeasuringDevice.cs file.
3. Remove the TODO comment and declare an interface named **ILoggingMeasuringDevice**. The interface must be accessible from code in different assemblies.
4. Modify the interface to inherit from the **IMeasuringDevice** interface.
5. Add a method named **GetLoggingFile** that returns a **string** value to the interface. The method should take no parameters. The purpose of this method is to return the file name of the logging file used by the device. Add an XML comment that summarizes the purpose of the method.
6. Build the solution and correct any errors.

▶ Task 3: Modify the MeasureDataDevice class to implement the ILoggingMeasuringDevice interface

In this task, you will modify the existing **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface. You will add code to enable logging and modify existing methods to use the logging functionality.

1. Open the MeasureDataDevice.cs file.
2. Remove the comment **TODO: Modify this class to implement the ILoggingMeasuringDevice interface instead of the IMeasuringDevice interface** above the **MeasureDataDevice** class. Modify the **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface instead of the **IMeasuringDevice** interface.
3. In the task list, locate the comment **TODO: Add fields necessary to support logging.** Double-click this item to go to the relevant line in the MeasureDataDevice.cs file.
4. Remove the TODO comment and add a **string** field named **loggingFileName**. This field must be accessible to classes that inherit from this class. This field will store the file name and path for the log file.
5. Add a **TextWriter** field named **loggingFileWriter**. This field should only be accessible to code in this class. You will use this object to write to a file.

6. In the task list, locate the comment **TODO: Add methods to implement the ILoggingMeasuringDevice interface.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.
7. Remove the TODO comment and add the **GetLoggingFile** method defined in the **ILoggingMeasuringDevice** interface. The method should take no parameters and return the value in the **loggingFileName** field.
8. In the task list, locate the comment **TODO: Add code to open a logging file and write an initial entry**. Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.
9. Remove the TODO comment and add the following code to instantiate the **loggingFileWriter** field. You can either type this code manually, or you can use the Mod9InstantiateLoggingFileWriter code snippet.

```
// New code to check the logging file is not already open.
// If it is already open then write a log message.
// If not, open the logging file.
if (loggingFileWriter == null)
{
    // Check if the logging file exists - if not create it.
    if (!File.Exists(loggingFileName))
    {
        loggingFileWriter = File.CreateText(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Created");
        loggingFileWriter.WriteLine("Collecting Started");
    }
    else
    {
        loggingFileWriter = new StreamWriter(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Opened");
        loggingFileWriter.WriteLine("Collecting Started");
    }
}
else
{
    loggingFileWriter.WriteLine
        ("Log file status checked - Already open");
    loggingFileWriter.WriteLine("Collecting Started");
}
```

The code checks whether the **loggingFileWriter** object has already been instantiated. If it has not, the code instantiates it by checking whether the file specified by the **loggingFileName** field already exists. If the file exists, the code opens the file; if it does not, the code creates a new file.

MCT USE ONLY. STUDENT USE PROHIBITED

10. In the task list, locate the comment **TODO: Add code to write a message to the log file.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.
11. Remove the TODO comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.
12. In the task list, locate the comment **TODO: Add code to log each time a measurement is taken.** Double-click this comment to go to the relevant line in the MeasureDataDevice.cs file.
13. Remove the TODO comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.
14. Build the solution and correct any errors.

▶ Task 4: Modify the MeasureDataDevice class to implement the IDisposable interface

In this task, you will modify the existing **MeasureDataDevice** class to implement the **IDisposable** interface. You will add code to ensure that the **TextWriter** object that writes messages to the log file is properly closed when an instance of the **MeasureDataDevice** class is disposed of.

1. At the top of the **MeasureDataDevice** class, remove the comment **TODO: Modify this class to implement the IDisposable interface**, and then modify the **MeasureDataDevice** class to implement the **IDisposable** interface in addition to the **ILoggingMeasuringDevice** interface.
2. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IDisposable** interface.
3. Move to the end of the **MeasureDataDevice** class. After the **Dispose** method added by the Implement Interface Wizard, add an overloaded **virtual void Dispose** method that implements the dispose pattern. This method should take a Boolean parameter called **disposing** and perform the following tasks:
    a. Check that the *disposing* parameter is set to **true**. If it is not, finish without disposing of anything.
    b. If the **loggingFileWriter** object is not null, write the message "Object disposed" to the logging file, flush the contents of the **loggingFileWriter** object, close it, and set the loggingFileWriter variable to **null**.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Locate the **Dispose** method, which takes no parameters, and then remove the default method body inserted by Visual Studio, which throws a **NotImplementedException** exception. Add statements that call the overloaded **Dispose** method and specify true as the parameter, and then suppress finalization for the current object.
5. Build the solution and correct any errors.

▶ Task 5: Modify the MeasureMassDevice class to use logging

In this task, you will modify the existing **MeasureMassDevice** class to set the **loggingFileName** field when the class is instantiated.

1. Open the MeasureMassDevice.cs file.
2. In the **MeasureMassDevice** class, remove the comment **TODO: Modify the constructor to set the log filename based on a string parameter**, and then modify the constructor to take a *string* parameter called *logFileName*. In the body of the constructor, set the **loggingFileName** field to the *logFileName* parameter. You should also update the XML comments for the constructor to describe the new parameter.
3. Build the solution and correct any errors.

Exercise 2: Managing Resources Used by an Object

Scenario

In this exercise, you will use a test harness application to test the disposal functionality that you added to the classes in the previous exercise. The test harness is a simple Windows® Presentation Foundation (WPF) application. Note that this application does not include exception handling or necessarily follow best practices for implementing a graphical user interface.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Test the logging functionality by using the test harness.
3. Modify the test harness to dispose of objects correctly.
4. Verify that the object is disposed of correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the starter project

- Open the Module9 solution from the E:\Labfiles\Lab 9\Ex2\Starter folder. This solution contains the completed code from Exercise 1 and skeleton code for Exercise 2.

▶ Task 2: Test the logging functionality by using the test harness

1. Run the Exercise2 Test Harness application.
2. Click **Get Measurements**. This action causes the application to pause for 20 seconds while some measurements data is generated and then display this data. This pause is necessary because the application waits for measurement data from the emulated device.

   Note that the measurement data is logged to the E:\Labfiles\Lab 9 \LogFile.txt file by default.
3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.
4. Using Notepad, open the LogFile.txt file in the E:\Labfiles\Lab 9 folder.
5. Review the contents of the LogFile.txt file.

   The file is empty. Although the application has retrieved values from the emulated device and written them to the log file, the **TextWriter** object caches data in memory and writes to the underlying file system when it is either flushed or closed. When you closed the application, you disposed of the **TextWriter** object without flushing its in-memory cache to the log file, which is why the file is empty.
6. Close Notepad.
7. Run the Exercise2 Test Harness application again, click **Get Measurements**, and then wait for the data to appear.
8. After the application populates the text boxes with data from the emulated device, click **Get Measurements** again.

   The application will throw an unhandled **IOException** exception. The exception is thrown because each time you click **Get Measurements**, you create a new instance of the **MeasureMassDevice** class. Each instance of the **MeasureMassDevice** class creates its own instance of the **TextWriter** class to log measurements. The test harness does not currently dispose of the **MeasureMassDevice** objects after the code run by the **Get Measurements** button completes. This means that the object is not closed and therefore

retains its lock on the log file. When you attempt to create a second instance of the **MeasureMassDevice** class that uses the same log file, this instance cannot access the file because it is still in use by the first instance.

9. Stop the Exercise 2 application.

▶ Task 3: Modify the test harness to dispose of objects correctly

1. In Visual Studio, open the MainWindow.xaml.cs file in the Exercise2 Test Harness project.
2. In the **createInstance_Click** method, remove the **TODO: Modify this method** comment in the MainWindow.xaml.cs file. Modify the **createInstance_Click** method to ensure that the **device** field is disposed of when the method completes by using a **using** block.
3. Build the solution and correct any errors.

▶ Task 4: Verify that the object is disposed of correctly

1. Run the Exercise2 Test Harness application.
2. Click **Get Measurements**, and then wait until the data appears.
3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.
4. Open Notepad and examine the log file.
5. Review the contents of the log file.

   The file now contains the values displayed on the form and status messages generated when the file is opened and closed. When the code for the **Get Measurements** button completes, it now disposes of the **MeasureMassDevice** instance, which forces the **TextWriter** object to flush its in-memory cache to the file, and then closes the **TextWriter** object.
6. Close Notepad
7. Run the Exercise2 Test Harness application again.
8. Click **Get Measurements**, and then wait for the data to appear.
9. In the Exercise 2 window, click **Get Measurements** again. The application will pause for another 20 seconds.

MCT USE ONLY. STUDENT USE PROHIBITED

This time, the application does not throw an exception. This is because the resources are properly disposed of each time you click **Get Measurements**. When you close the **TextWriter** object, you release the lock on the file, and a new instance of the **TextWriter** class can now use the same log file without throwing an exception.

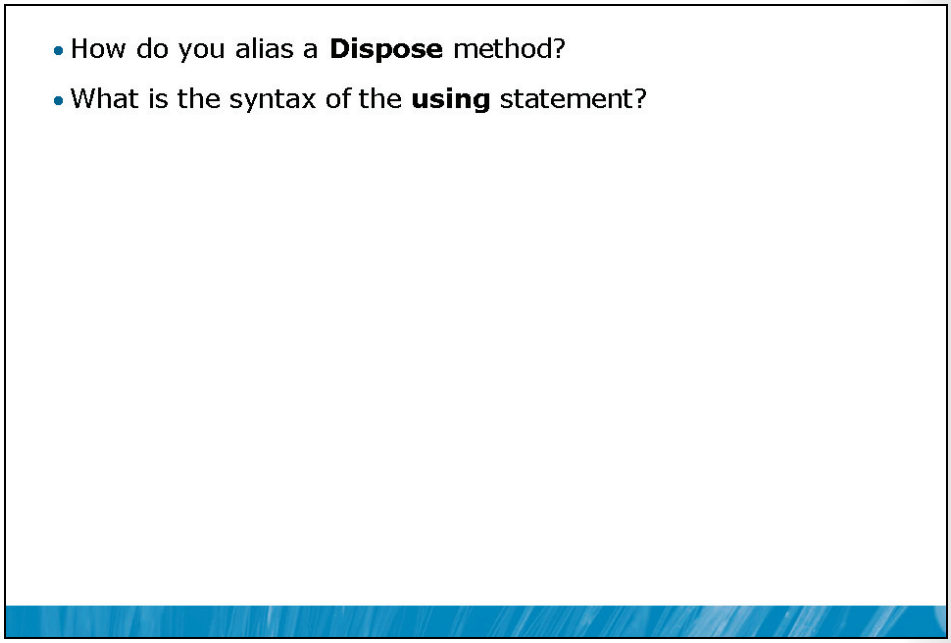
10. Open Notepad and examine the log file.
11. Review the contents of the log file.

    The file contains the most recent values displayed on the form.

12. Close Notepad.
13. Close the Exercise 2 window.
14. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

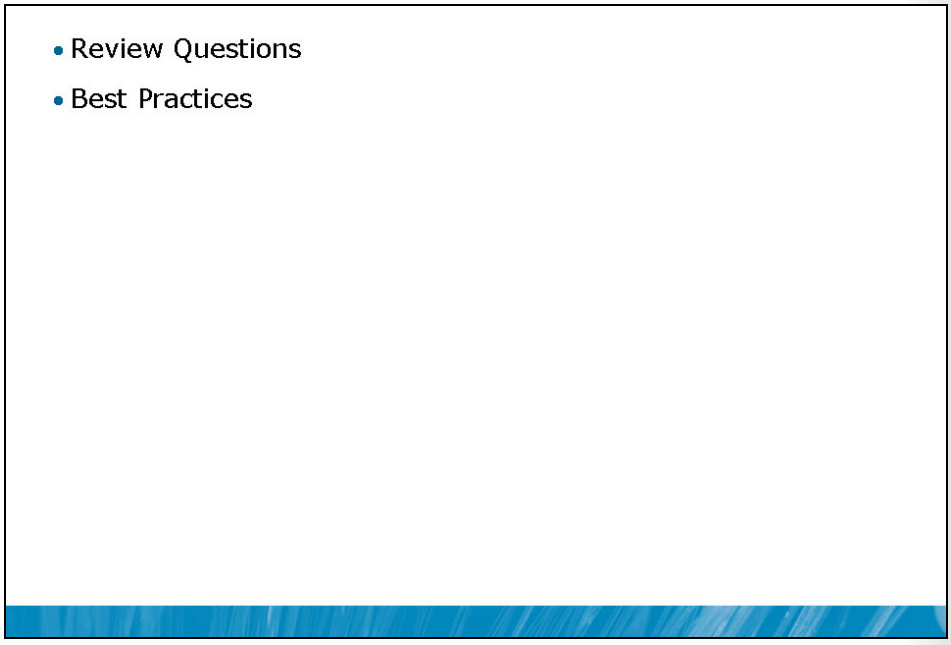


Review Questions

1. How do you alias a **Dispose** method?
2. What is the syntax of the **using** statement?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What methods are defined in the **IDisposable** interface?
2. Where would an instance of the **System.String** class be stored: on the heap or on the stack?
3. Should you add a destructor to every class you develop?

Best Practices Related to Disposing of Unmanaged Objects

Supplement or modify the following best practices for your own work situations:

- You should implement the dispose pattern whenever your code uses unmanaged resources.
- You should use a **using** statement to ensure disposal of objects wherever possible.

MCT USE ONLY. STUDENT USE PROHIBITED

- Where a **using** statement is not appropriate, you should ensure exception-safe disposal of objects by using a **try**/**finally** block; you should release resources in the **finally** block.

MCT USE ONLY. STUDENT USE PROHIBITED