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

Module 10

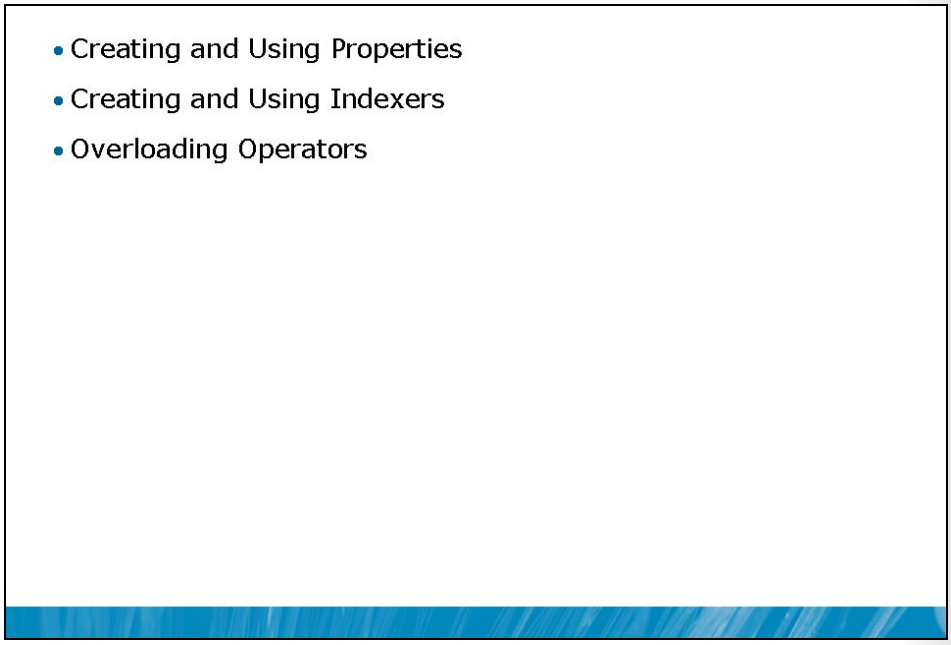
Encapsulating Data and Defining Overloaded Operators

Contents:

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

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



Nearly every application you develop will require you to develop at least one type to represent some entity. Types typically expose methods and data. A simple approach to exposing data is to make the fields used by your class public; however, this is often bad practice—or at least is not the most secure, efficient, or natural technique.

For example, providing an array-like syntax may be a better approach when accessing data in a class that stores a collection of data. Similarly, if a class exposes a member that should have only read-only access, exposing a field publicly provides both read and write access. This module will introduce you to properties and indexers. These are elements of Microsoft® Visual C#® that enable you to encapsulate data and expose data appropriately and efficiently.

Another syntax you will commonly use is that associated with operators. For example, it is intuitive to write 2 + 3 and expect that the result will be 5. Similarly, you will probably expect "*Hello*"+ "*World*" to return the concatenated string "*HelloWorld*". Many operators have well-defined behavior for the built-in Visual C# types, but you can also define operators for your own types. This module describes how to implement operators for your types by using overloading.

MCT USE ONLY. STUDENT USE PROHIBITED

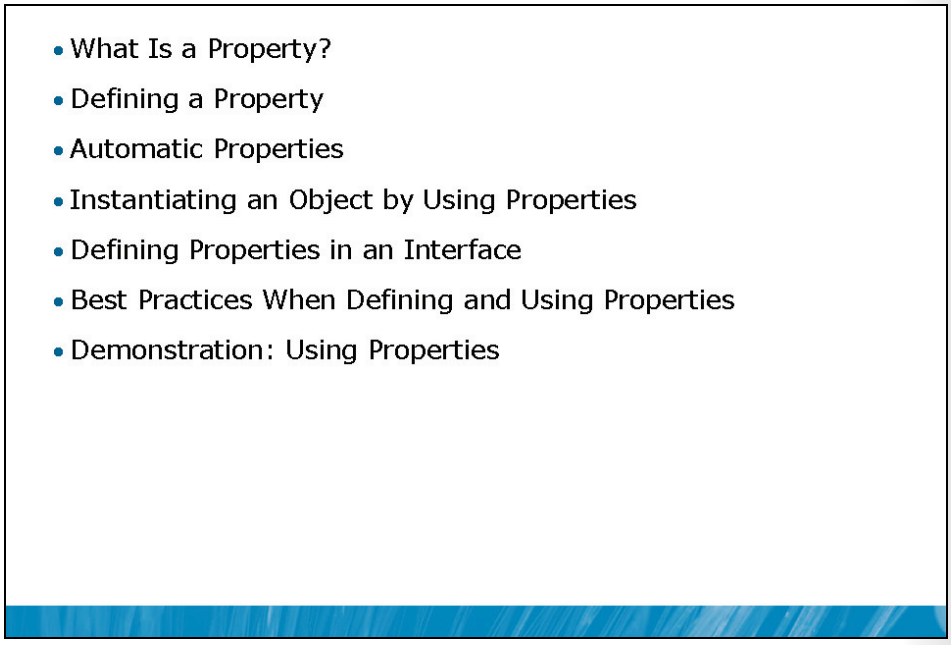
Objectives

After completing this module, you will be able to:

- Explain how properties work and use them to encapsulate data.
- Describe how to use indexers to provide access to data through an array-like syntax.
- Describe how to use operator overloading to define operators for your own types.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Creating and Using Properties



You can use properties to provide controlled access to the data in a type. This lesson introduces you to properties and shows you how to define them in your types. It also explains why you should use this approach to encapsulate data.

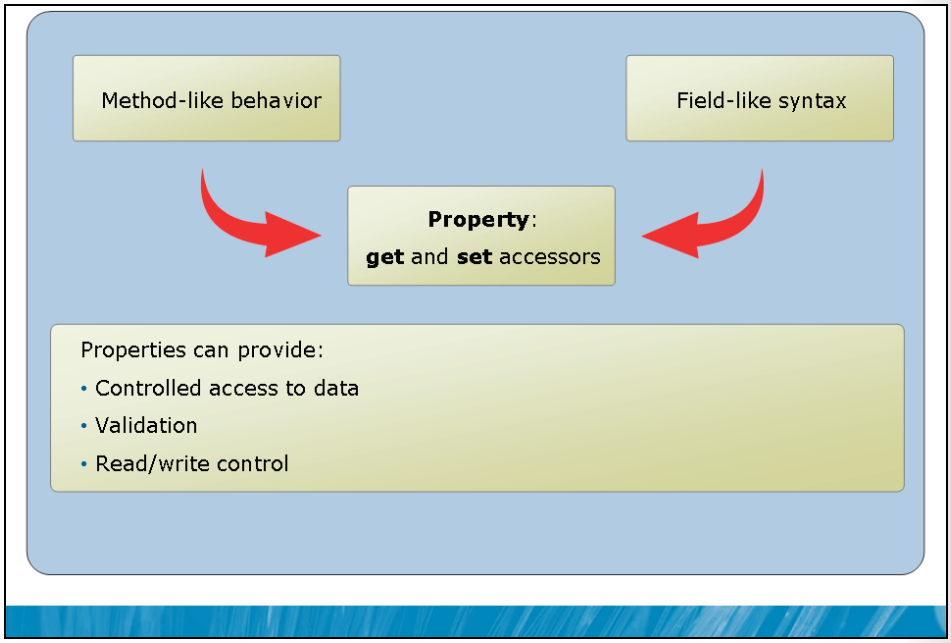
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of properties.
- Implement properties.
- Explain automatic properties.
- Instantiate an object by using properties.
- Define properties in an interface.
- Describe the best practices relating to properties.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Property?



Key Points

A property is a cross between a field and a method. You use field-like syntax to access a property. However, the behavior of a property is more like a method.

A property can contain two elements:

- A **get** accessor, which an application can use to read the property value.
- A **set** accessor, which an application can use to change the property value.

Properties are a common way of encapsulating data exposed by your class. Normally a property is mapped to a private field in your type. The field stores the data, and the **get** and **set** accessors of the property provide a mechanism for accessing that field. You are not obliged to provide both a **get** and a **set** accessor, so properties have the advantage that you can control whether to make a property read-only, write-only, or make the property readable and writeable which you cannot do by exposing a field.

Another advantage of using a property is the ability to validate data. If you expose a field in your type, any other type can read or write to that field. As long as the data

MCT USE ONLY. STUDENT USE PROHIBITED

is of the right type, any value can be assigned to that field. This is not always logical; sometimes you may need to restrict the range of acceptable values for a field in your type. With a property, you can add logic to the **set** accessor to check that a value falls in the expected range before updating the private field.

Although properties normally map to private fields, there is no requirement for them to do so. The **get** accessor of a property can return a calculated value, a constant value, or perform any other operation applicable to your application. Properties will often include additional logic; for example, if you update a file name by using a property, the property may check whether the file is currently in use and, if necessary, rename the file or open a new file according to the requirements of the application.

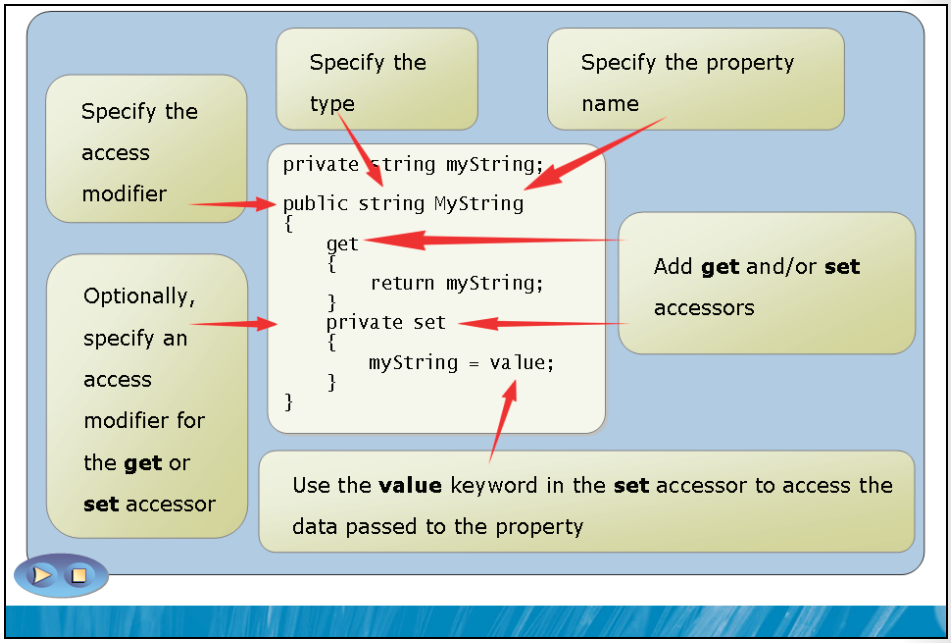
Question: How does the behavior of a method differ from a property?

Additional Reading

For more information about properties, see the Properties (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192948.

MCT USE ONLY. STUDENT USE PROHIBITED

Defining a Property



Key Points

A property has a type and a name, in much the same way as a field. However, the logic for a property is defined by the **get** and **set** accessors.

The **get** accessor, like a method, can include any code; however, it must return an object of the type specified by the property or throw an exception. The **set** accessor does not have to perform any function–although normally, you update a private field to perform some operation based on the value passed to the property. You do not specify a parameter for the **set** accessor; a **set** accessor always takes one parameter of the type exposed by the property. You can access the object passed as a parameter to a **set** accessor by using the **value** keyword.

The following code example shows how to define a simple property that provides access to a private field. The **get** keyword introduces a code block that defines the code that runs when an application reads the property. The **set** keyword defines the code block for the logic that runs when an application assigns a value to the property.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private string myString;

public string MyString
{
    get
    {
        return this.myString;
    }
    set
    {
        this.myString = value;
    }
}
```

To define a read-only property, you simply omit the **set** accessor. Similarly, to define a write-only property, do not implement a **get** accessor.

Defining Property Accessibility

When you define a property, you specify the access modifier for that property. The access modifier that you specify for a property is inherited by the **get** and **set** accessors. You can override the access modifier for either the **get** or **set** accessor; however, you cannot make an accessor more accessible than the containing property. For example, you cannot make the **get** accessor public if the property is **private**.

The following code example shows how to modify the accessibility level at the accessor level.

```
public string MyString
{
    get
    {
        return this.myString;
    }
    private set
    {
        myString = value;
    }
}
```

Using a Property in a Consuming Class

You use a property in a consuming class by using the dot notation in the same way as you access a public field. The following code example shows how to access the **MyString** property from the previous code example. Internally, the Visual C#

MCT USE ONLY. STUDENT USE PROHIBITED

compiler converts all attempts to read the property into calls to the **get** accessor and changes all attempts to write the property into calls to the **set** accessor.

```
MyObject theClass = new MyObject;

// Setting the string – calls the set accessor
theClass.MyString = "Property set.";

// Getting the string – calls the get accessor
Console.WriteLine(theClass.MyString);
```



Note: You can define static properties, but they can only access static data.

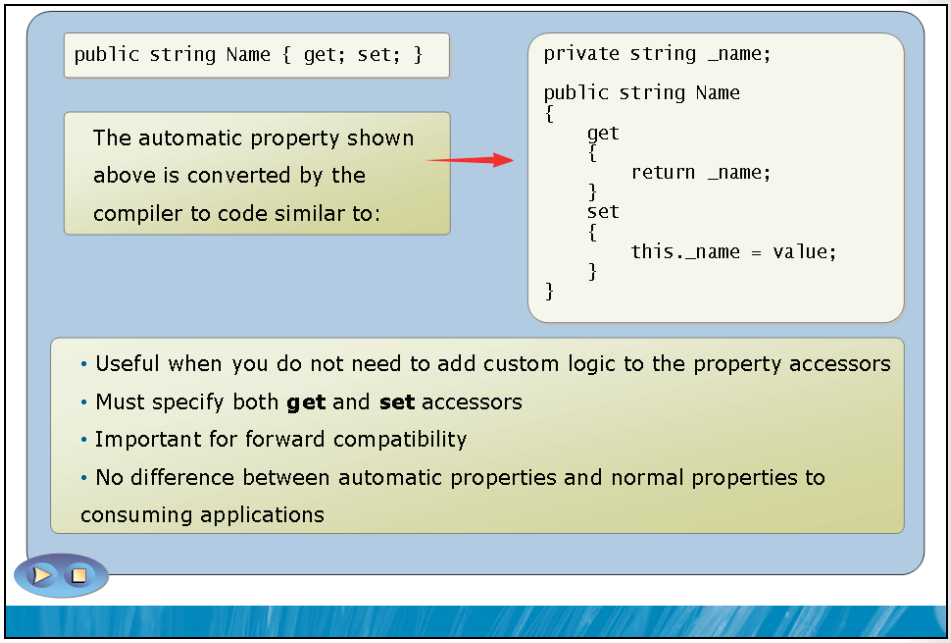
Question: How can you enable write access to a property to other types in the same assembly, but read access to a property from a class in any assembly?

Additional Reading

For more information about using properties, see the Using Properties (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192949.

MCT USE ONLY. STUDENT USE PROHIBITED

Automatic Properties



Key Points

When you develop a new type, you may include a data field that you want to expose to applications. If no additional processing or validation is required on that field, it may be tempting to simply expose the field publicly instead of adding a property to provide access to that field.

In this case, exposing a field may not seem like a problem. However, remember that you cannot add code to prevent invalid values in a field but you can in a property. Whether you need to add validation or other logic to a property when you originally develop a type does not mean that will always be the case. The requirements of your type may change over the lifetime of the application.

From a developer's perspective, using a property is exactly the same as using a field; however, this is not true to the compiler. The compiler converts code that accesses a property into a method call to the **get** accessor, and it similarly converts writing to a property to a method call to the **set** accessor. This has implications for existing applications if you must convert a field to a property at a later date; any application that used the type with the value exposed as a field must be recompiled

MCT USE ONLY. STUDENT USE PROHIBITED

with the data exposed through a property. If this type is in an assembly used by a number of applications, you may need to rebuild and redeploy a lot of installations.

You can avoid this extra work by simply exposing the data through a property when you originally develop the type. Any future changes to the type can then be made without the need to recompile applications that consume your type.

Where you must expose a field, and are tempted to simply make the field **public** rather than writing a property to get and set the field, you can use automatic properties.

Automatic properties provide a simple inline syntax that converts a field to a property. To use automatic properties, you simply add curly braces that contain both **set** and **get** accessors, each followed by a semicolon, as the following code example shows.

```
public string Name { get; set; }
```

When you use an automatic property, the compiler creates a private field and automatically generates code to read and write this field, as the following code example shows.

```
private string _name;

public string Name
{
    get
    {
        return this._name;
    }
    set
    {
        this._name = value;
    }
}
```



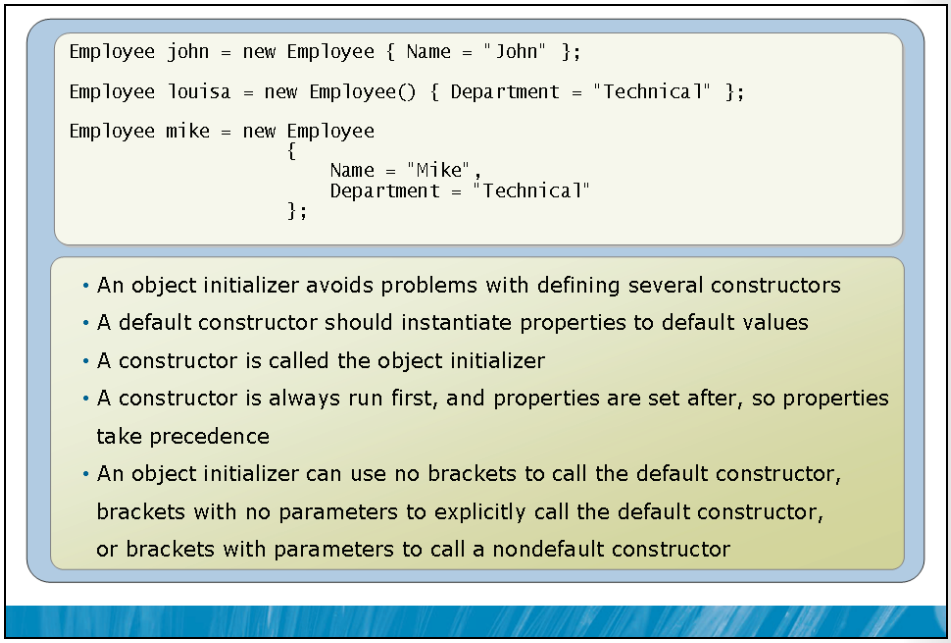
Note: Automatic properties always define both a **get** and **set** accessor. Automatic properties are intended for use where otherwise you would simply expose a public field. If you require more specific control over the data, you must write the property manually. It does not make any difference to consuming classes if you change from an automatic property to a manual property in a later build of your code; they are completely interchangeable, unlike properties and fields.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What is the benefit of using an automatic property compared to exposing a public field?

MCT USE ONLY. STUDENT USE PROHIBITED

Instantiating an Object by Using Properties



Key Points

You have previously seen how to use a constructor to instantiate an object and initialize its fields. You can declare several constructors, with different signatures, to enable other developers to set various combinations of fields in your type to appropriate values; however, this approach is problematic if you have more than a small number of fields or several properties of the same type.

The following code example shows a simple class with several constructors.

```
class Employee
{
    private string name;
    private string department;

    // Initialize both fields
    public Employee(string empName, string empDepartment)
    {
        this.name = Name;
        this.department = Department;
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Initialize name only
    public Employee(string empName)
    {
        this.name = empName;
    }

    // Initialize department only
    public Employee(string empDepartment)
    {
        this.department = empDepartment
    }
    ...
}
```

The intention of the constructors is to enable an application to specify a value for the employee name, department name, or both when it creates a new **Employee** object. However, this code will not compile because the compiler cannot distinguish between the two constructors that take a single *string* parameter. If you attempt to instantiate an **Employee** object by using the code shown in the following code example, the compiler does not know which constructor to use.

```
// Is "Fred" the name of an employee or a department?
Employee myEmployee = new Employee("Fred");
```

You can resolve this problem by using properties to initialize the object when you instantiate it. This syntax is known as an object initalizer. With an object initializer, you create a new object by using a constructor, but you specify the values to assign to properties after the constructor has completed by using property name/value assignment pairs separated by commas and enclosed in curly braces.

The following code example shows how to define a class that supports object initializers and how to create an object by using them.

```
class Employee
{
    // Default constructor.
    public Employee()
    {
        ...
    }

    // Constructor that sets the grade of an employee.
    public Employee(int grade)
    {
        ...
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Expose Name and Department as automatic properties.
    public string Name { get; set; }
    public string Department { get; set; }
    ...
}

// Instantiating an object and setting a single property.
Employee louisa = new Employee() { Department = "Technical" };

// Instantiating an object and setting a single property.
// You do not have to add the brackets to use the default constructor.
Employee john = new Employee { Name = "John" };


// Instantiating an object and setting a multiple properties.
// Separate properties with a comma.
Employee mike = new Employee
                {
                    Name = "Mike",
                    Department = "Technical"
                };
```

In the first example, (**louisa**), the default constructor is used to create the **Employee** object. After the object is created and the constructor has finished, the value "**Technical**" is assigned to the **Department** property. Note that if you use the default constructor, you can omit the brackets (), as the second example (**john**) and the third example (**mike**) illustrate.

If the **Employee** class has a nondefault constructor, you can invoke that together with an object initializer, as the following code example shows. This code example uses the constructor that sets the grade of an employee.

```
Employee antony =
    new Employee(2) { Name = "Antony", Department = "Management" };
```

When you use an object initializer, the constructor logic runs first, and then the properties are set to the values specified in the object initializer. This means that if you set a property in a constructor, and then set the same property in the object initializer, the value from the object initializer will overwrite the value set by the constructor.



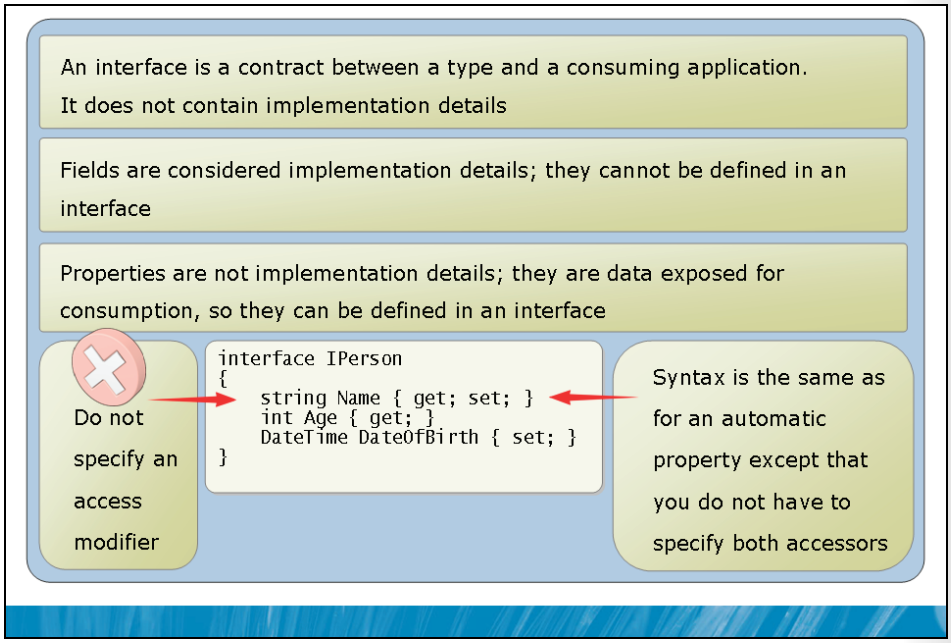
Hint: You should only define constructors that set any required properties to default values. Classes that consume your type can then override those properties in an object initializer.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: Why is it important to instantiate required properties to default values in the constructor?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Properties in an Interface



Key Points

An interface defines a contract that specifies the methods that a class should implement. An interface can also define properties. However, the implementation details of these properties (such as the fields they reference, if any) are the responsibility of the class.

To add a property to an interface, you use the same syntax as an automatic property, except you cannot specify an access modifier. The following code example shows properties added to an interface.

```
interface IPerson
{
    string Name { get; set; }
    int Age { get; }
    DateTime DateOfBirth { set; }
}
```

Classes that implement an interface that includes properties can implement the properties implicitly or explicitly.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows the **IPerson** interface implemented implicitly.

```
class Person : IPerson
{
    public string Name
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    public int Age
    {
        get { throw new NotImplementedException(); }
    }

    public DateTime DateOfBirth
    {
        set { throw new NotImplementedException(); }
    }
}
```

The following code example shows the **IPerson** interface implemented explicitly.

```
class Person : IPerson
{
    string IPerson.Name
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }

    int IPerson.Age
    {
        get { throw new NotImplementedException(); }
    }

    DateTime IPerson.DateOfBirth
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        set { throw new NotImplementedException(); }
    }
}
```

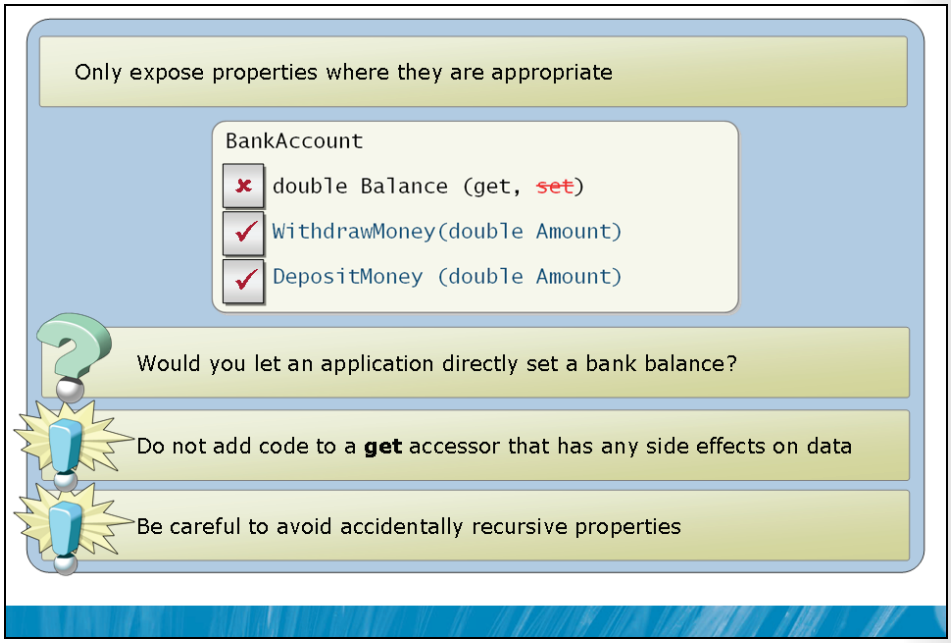
Question: When should you add a property to an interface?

Additional Reading

For more information about defining properties in an interface, see the Interface Properties (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192950.

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices When Defining and Using Properties



Key Points

Properties provide an excellent framework for exposing data from types you develop; however, if you do not use properties appropriately, you risk introducing bugs or simply exposing properties that enable consuming classes to perform undesirable behavior. You can mitigate the risks by following some best practices.

Using Properties Appropriately

It would be easy to say you should always expose a property for every field in types that you develop; however, this is not necessarily good practice. You should carefully consider whether exposing a property is appropriate to the types of operations an application can perform on a data item.

For example, if you are developing a type to represent a bank account, a field in the class can represent the balance of the account. It may be tempting to provide a property that enables an application to read and write the account balance, but this does not reflect the real-world operations that a bank typically implements; a bank enables you to deposit some money to increase your balance and to take money out (subject to any necessary overdraft constraints) rather than letting you directly

MCT USE ONLY. STUDENT USE PROHIBITED

set the balance of your account. Consequently, it is more appropriate to provide **Deposit** and **Withdraw** methods.

When you design types in your application, you should remember to design those types to expose the functionality required for the specific application. You should not expose every field as a property unless there is a good reason for exposing the field.

Do Not Implement Get Accessors With Side Effects

A **get** accessor should simply retrieve a value and return that value to the consuming application. When you implement a **get** accessor, retrieving the value should not impact the value or any other data stored by the type. The only exception to this rule is when you write applications that must adhere to security restrictions. In this case, you can add logic to the **get** accessor to log access or to further restrict access according to business requirements.

Use Naming Conventions

The convention when wrapping a field is to use a name that varies from the field only in the case of the initial letter. For example, a field called **myData** is typically encapsulated in a property called **MyData**. However, it is very easy to write code that calls itself recursively, such as in the following code example.

```
int myData;

public int MyData
{
    get
    {
        return MyData;
    }
    ...
}
```

The code will compile; however, there is a bug in this code. It will cause an infinite loop because the **MyData** property calls itself recursively. Bugs such as this can be difficult to spot. If you allow an application with a bug such as this to run for long enough, you will eventually get an **OutOfMemoryException** exception.

Question: When would you add logic to a **get** accessor that performs functionality other than to return the data?

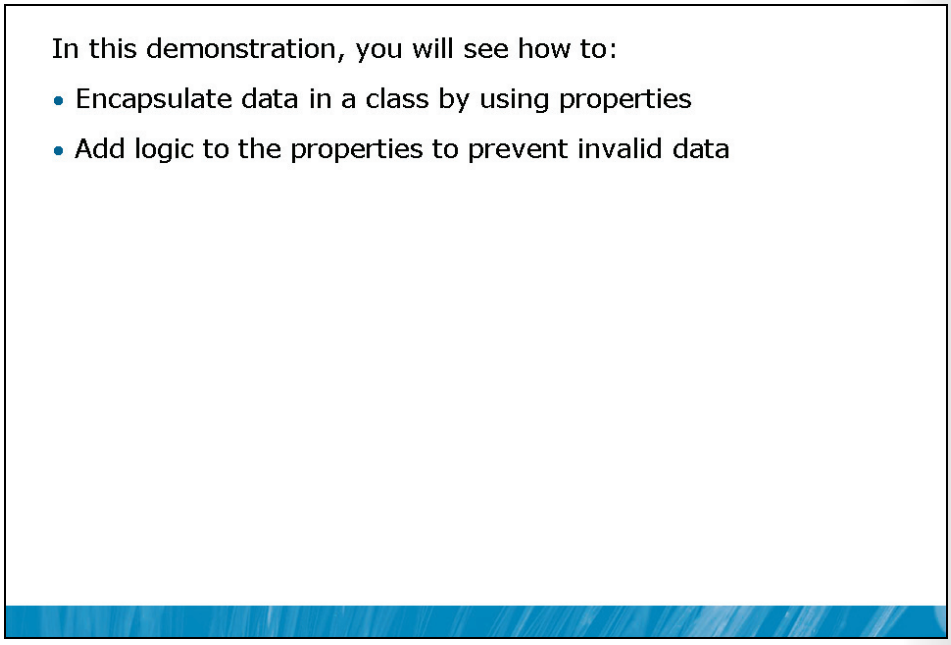
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about choosing between properties and methods, see the Choosing Between Properties and Methods page at http://go.microsoft.com/fwlink/?LinkId=192951.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using Properties



Key Points

- Convert a field to an automatic property.
- Create a new property to provide controlled access to data in a field.
- Test the properties by using a test harness.

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 **UsingPropertiesDemo** solution in the **E:\Demofiles\Mod10\Demo1\Starter\UsingPropertiesDemo** folder.
4. Open the Employee.cs file, and then review the **Employee** class. Notice the publicly exposed fields and the constructor that sets the **Name** field based on the parameter, and the **Salary** and **Department** fields to default values.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Convert the **Name** field to a property by using automatic properties:
   - Modify the following line of code.

```
public string Name;
```

   Change it to the following line of code.

```
public string Name { get; set; }
```

6. Convert the **Department** field to a property by using automatic properties:
   - Modify the following line of code.

```
public string Department;
```

   Change it to the following line of code.

```
public string Department { get; set; }
```

7. Convert the **public Salary** field to a **private** field and rename it **salary**:
   - Modify the following line of code.

```
public int Salary;
```

   Change it to the following line of code.

```
private int salary;
```

8. Uncomment the commented **Salary** property, and then explain how it ensures that an employee can never have a negative salary.
9. Open the Program.cs file, and then review the **Employee** class.
10. Uncomment all of the code up to and including the first occurrence of the following code.

```
Console.ReadLine();
```

    Notice how the **julie** object is created by using the constructor, and explain that the properties are subsequently set by using the dot notation.

    Notice how the **james** object is created by using named properties. Emphasize that these named properties are set after the constructor is run, so they take precedence over the default values set by the constructor.

MCT USE ONLY. STUDENT USE PROHIBITED

11. Uncomment the remaining code in the file.

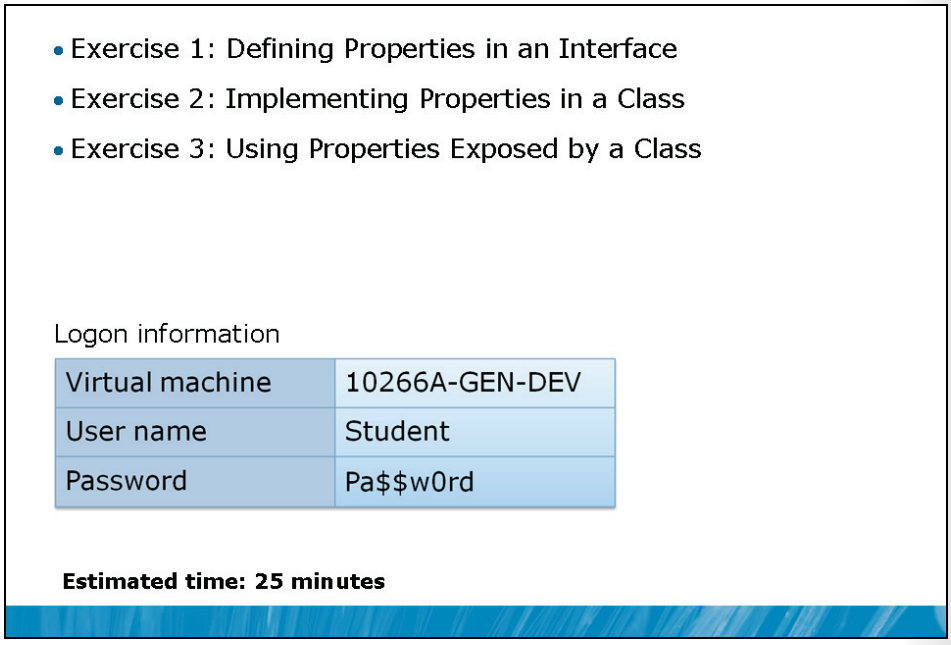
    Notice that the code attempts to set James' salary to a negative value. Remind students that the property prevented negative values.

12. Run the application without debugging.
13. When the application pauses, highlight that the application has worked as expected, and the two employees' details are displayed correctly, and then press ENTER.
14. When the application pauses, highlight that the application has worked as expected, and James' salary has been set to 0 instead of a negative value, and then press ENTER.
15. Close Visual Studio.

Question: If you set a property in a constructor, and you use named properties to set the same property when you instantiate the object, which takes precedence: the value from the constructor or the named property?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab A: Creating and Using Properties



Objectives

After completing this lab, you will be able to:

- Define properties in an interface.
- Implement properties in a class.
- Use properties exposed by a class.

Introduction

In this lab, you will define properties in an interface and then implement these properties in a class. You will also use a test application to verify that the properties behave as expected.

MCT USE ONLY. STUDENT USE PROHIBITED

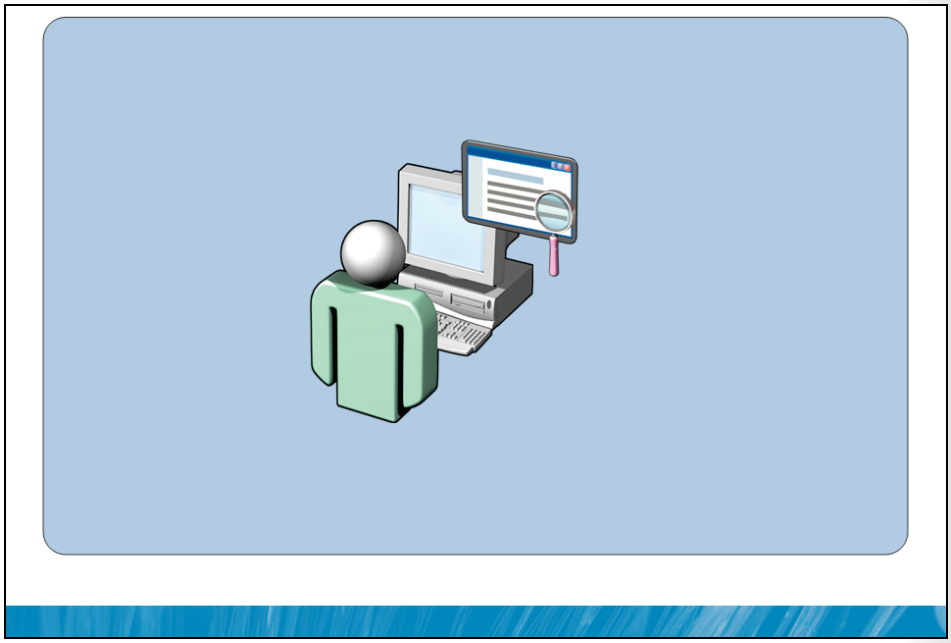
Lab Setup

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



You have been asked to enhance the functionality of the software that drives a number of the scientific devices produced by Fabrikam, Inc.

The software for the measuring devices developed in the previous labs must be improved and simplified by using properties to provide controlled access to the private data members of the **MeasureDataDevice** abstract class. In this way, other developers can write software to manipulate the data exposed by these devices in a variety of ways. Consequently, these developers will no longer be restricted by the limited set of access methods that this class currently provides.

In this lab, you will modify the **IMeasuringDevice** interface and add the following properties:

- **UnitsToUse**: A read-only property based on the Units enumeration that exposes the **unitsToUse** field.
- **DataCaptured**: A read-only integer array property that exposes the **dataCaptured** field.
- **MostRecentMeasure**: A read-only integer property that exposes the **mostRecentMeasure** field.

MCT USE ONLY. STUDENT USE PROHIBITED

- **LoggingFileName**: A read/write string property that exposes the **loggingFileName** field.

You will leave the existing methods in the **IMeasuringDevice** interface intact, because the updated software has to support older applications that still use these methods.

You will modify the **MeasureDataDevice** abstract class from the previous lab and implement the properties. The property **set** accessor for the **LoggingFileName** property will close the existing logging file (if it is open) and then open a new file with the specified name. The remaining properties will simply return the value of the underlying field. You will test the new functionality by using the **MeasureMassDevice** class.

Exercise 1: Defining Properties in an Interface

Scenario

In this exercise, you will define an interface called **IMeasuringDeviceWithProperties** with the following public properties:

- **UnitsToUse**. This read-only property will return the units used by the emulated device.
- **DataCaptured**. This read-only property will return a copy of all of the recent data that the measuring device has captured.
- **MostRecentMeasure**. This read-only property will return the most recent measurement taken by the device.
- **LoggingFileName**. This read/write property will return and update the name of the logging file used by the device.

The **IMeasuringDeviceWithProperties** interface will inherit from the **IMeasuringDevice** interface; classes that implement the new interface will always be required to implement the **IMeasuringDevice** interface.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add properties to the **IMeasuringDeviceWithProperties** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ 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 10\Snippets folder.
4. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab A\Ex1\Starter folder.

▶ Task 2: Add properties to the IMeasuringDeviceWithProperties interface

1. In Visual Studio, review the task list.
2. Open the IMeasuringDeviceWithProperties.cs file.
3. Remove the comment **TODO: Add properties to the interface.**.
4. Add a read-only property to the interface of type **Units** called **UnitsToUse**.
5. Add a read-only property to the interface of type **int[]** called **DataCaptured**.
6. Add a read-only property to the interface of type **int** called **MostRecentMeasure**.
7. Add a read/write property to the interface of type **string** called **LoggingFileName**.
8. Build the solution and correct any errors.

Exercise 2: Implementing Properties in a Class

Scenario

In this exercise, you will modify the existing **MeasureDataDevice** class (which currently implements the **IMeasuringDevice** interface) to implement the **IMeasuringDeviceWithProperties** interface. When you implement the **LoggingFileName** property, you will implement logic in the **set** accessor that checks whether the log file is open, and if it is open, closes the file and opens a new log file with the updated name.

The main tasks for this exercise are as follows:

1. Open the starter project.

2. Update the **MeasureDataDevice** class to implement the **IMeasuringDeviceWithProperties** interface.

▶ Task 1: Open the starter project



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **IMeasuringDeviceWithProperties** interface successfully, proceed directly to **Task 2: Update the MeasureDataDevice class to implement the IMeasuringDeviceWithProperties interface**.

- Open the Module10 solution in the E:\Labfiles\Lab 10\Lab A\Ex2\Starter folder. This solution contains a completed version of the **IMeasuringDeviceWithProperties** interface.

▶ Task 2: Update the MeasureDataDevice class to implement the IMeasuringDeviceWithProperties interface

1. In Visual Studio, review the task list.
2. Open the MeasureDataDevice.cs file.
3. Remove the comment **TODO: Implement the IMeasuringDeviceWithProperties interface.**.
4. Modify the class declaration to implement the **IMeasuringDeviceWithProperties** interface instead of the **ILoggingMeasuringDevice** interface.

   The **IMeasuringDeviceWithProperties** interface inherits from the **ILoggingMeasuringDevice** interface, so modifying the declaration will not break compatibility with existing applications; the class can still be cast as an instance of the **ILoggingMeasuringDevice** interface.
5. Remove the comment **TODO: Add properties specified by the IMeasuringDeviceWithProperties interface.**.

   You will use the Implement Interface Wizard in the next step to add the properties.
6. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDeviceWithProperties** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

7. Locate the **UnitsToUse** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **UnitsToUse** property to return the **unitsToUse** field.
8. Locate the **DataCaptured** property **get** accessor, and then remove the default that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **DataCaptured** property to return the **dataCaptured** field.
9. Locate the **MostRecentMeasure** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **MostRecentMeasure** property to return the **mostRecentMeasure** field.
10. Locate the **LoggingFileName** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **LoggingFileName** property to return the **loggingFileName** field.
11. Modify the **set** accessor of the **LoggingFileName** property as shown in the following code example.



Note: A code snippet is available, called Mod10LoggingFileNamePropertySetAccessor, that you can use to add this code.

```
if (loggingFileWriter == null)
{
    // If the file has not been opened, simply update the file name.
    loggingFileName = value;
}
else
{
    // If the file has been opened, close the current file first,
    // and then update the file name and open the new file.
    loggingFileWriter.WriteLine("Log File Changed");
    loggingFileWriter.WriteLine("New Log File: {0}", value);
    loggingFileWriter.Close();

    // Now update the logging file and open the new file.
    loggingFileName = value;

    // Check whether the logging file exists—if not, create it.
    if (!File.Exists(loggingFileName))
    {
        loggingFileWriter = File.CreateText(loggingFileName);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            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");
        }
        loggingFileWriter.WriteLine("Log File Changed Successfully");
    }
```

The **set** accessor for the **LoggingFileName** property checks whether the log file is currently open. If the log file has not been opened, the **set** accessor simply updates the local field. However, if the log file has been opened, the accessor closes the current log file and opens a new log file with the new file name in addition to updating the local field.

12. Build the solution and correct any errors.

Exercise 3: Using Properties Exposed by a Class

Scenario

In this exercise, you will use a test harness application to test the functionality of the **MeasureDataDevice** class you developed in the previous exercise.

The main tasks for this exercise are as follows:

1. Add the test harness to the solution.
2. Update the test harness.
3. Test the properties by using the test harness.

▶ Task 1: Add the test harness to the solution

The test harness application for this lab is a simple Windows® Presentation Foundation (WPF) application that is designed to test the functionality of the **MeasureDataDevice** class that you have just modified. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

MCT USE ONLY. STUDENT USE PROHIBITED

1. Add the test harness to the solution. The test harness is a project called Exercise3TestHarness, located in the E:\Labfiles\Lab 10\Lab A\Ex3 \Starter\Exercise3TestHarness folder.
2. Set the Exercise3TestHarness project as the startup project for the solution.

▶ Task 2: Update the test harness

1. In Visual Studio, review the task list.
2. Review the user interface for the test application.

   The test harness application includes functionality to enable you to test the properties you developed in the previous exercise. The **Start Collecting** button creates a new instance of the **MeasureMassDevice** object and starts collecting measurements from the emulated device. The application includes text boxes that display the output from the application. It also includes an **Update** button to enable you to update the file name of the log file. Finally, the test harness includes a button to stop the collection of measurements from the emulated device and dispose of the object.
3. Open the MainWindow.xaml.cs file.



Note: In the following steps, you will store values in the **Text** property of **TextBox** controls in the WPF window. This is a **string** property. In some of the steps, you may need to call the **ToString** method to convert the property to a **string**.

4. Remove the comment **TODO: Add code to set the unitsBox to the current units**.
5. Locate the following line of code.

```
unitsBox.Text = "";
```

6. Update the code you located in the previous step to set the **Text** property of the **unitsBox** object to the **UnitsToUse** property of the **device** object.
7. Remove the comment **TODO: Add code to set the mostRecentMeasureBox to the value from the device.**.
8. Locate the following line of code.

```
mostRecentMeasureBox.Text = "";
```

MCT USE ONLY. STUDENT USE PROHIBITED

9. Update the code you located in the previous step to set the **Text** property of the **mostRecentMeasureBox** object to the **MostRecentMeasure** property of the **device** object.
10. Remove the comment **TODO: Update to use the LoggingFileName property**.
11. Locate the following line of code.

```
loggingFileNameBox.Text =
    device.GetLoggingFile().Replace(labFolder, "");
```

12. Update the code you located in the previous step to set the **Text** property of the **loggingFileNameBox** object to the **LoggingFileName** property of the **device** object. Your code should call the **Replace** method of the **string** class in the same way as the code you are updating.
13. Remove the comment **TODO: Update to use the DataCaptured property**.
14. Locate the following line of code.

```
rawDataValues.ItemsSource = device.GetRawData();
```

15. Update the code you located in the previous step to set the **ItemsSource** property of the **rawDataValues** object to the **DataCaptured** property of the **device** object.
16. In the **updateButton_Click** method, remove the comment **TODO: Add code to update the log file name property of the device** and add code to set the **LoggingFileName** property of the **device** object to the concatenation of the **labFolder** field and the **Text** property of the **loggingFileNameBox** box.
17. Build the solution and correct any errors.

▶ Task 3: Test the properties by using the test harness

1. Start the Exercise3TestHarness application.
2. Click **Start Collecting**. This action causes the application to pause for 10 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.
3. Using Windows Explorer, move to the **E:\Labfiles\Lab 10\Lab A** folder, and then verify that the default logging file, LogFile.txt, has been created.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Return to the Exercise3TestHarness window. Wait at least a further 10 seconds to ensure that the emulated device has generated some additional values before you perform the following steps.
5. Change the log file to **LogFile2.txt**, and then click **Update**.

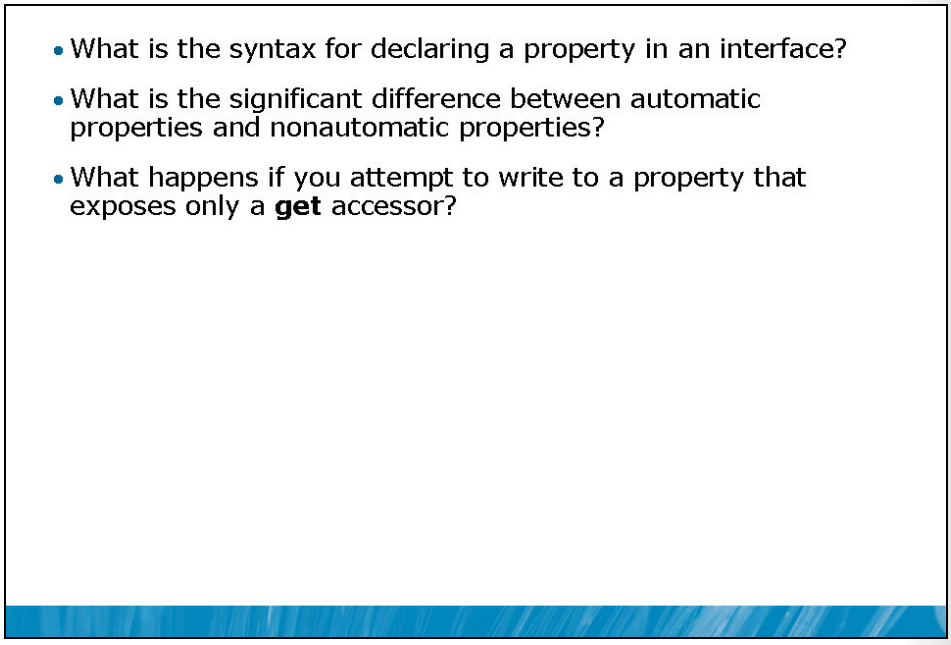
   The **Update** button calls the code you added to set the **LoggingFileName** property of the device; because the device is running, and therefore logging values to the log file, the code will close the current log file and open a new one with the name you specified.
6. Wait at least 10 seconds to ensure that the emulated device has generated some additional values before you perform the following steps.
7. Using Windows Explorer, move to the **E:\Labfiles\Lab 10\Lab A** folder, and then verify that the new logging file, LogFile2.txt, has been created.
8. Return to the Exercise3TestHarness window, and then click **Stop Collecting / Dispose Object**.
9. Close the Exercise3TestHarness window.
10. Close Visual Studio.
11. Using Notepad, open the LogFile.txt file in the E:\Labfiles\Lab 10\Lab A folder.
12. Review the contents of the LogFile.txt file.

    The file includes the values originally displayed in the test harness in addition to some not displayed. The file then indicates that the log file has changed and gives the name of the new log file.
13. Open the LogFile2.txt file in the E:\Labfiles\Lab 10\Lab A folder.
14. Review the contents of the LogFile2.txt file.

    The file indicates that the log file has changed successfully. The file then includes any measurements taken after the log file changed and finally indicates that collecting stopped and the object was disposed of.
15. Close Notepad.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

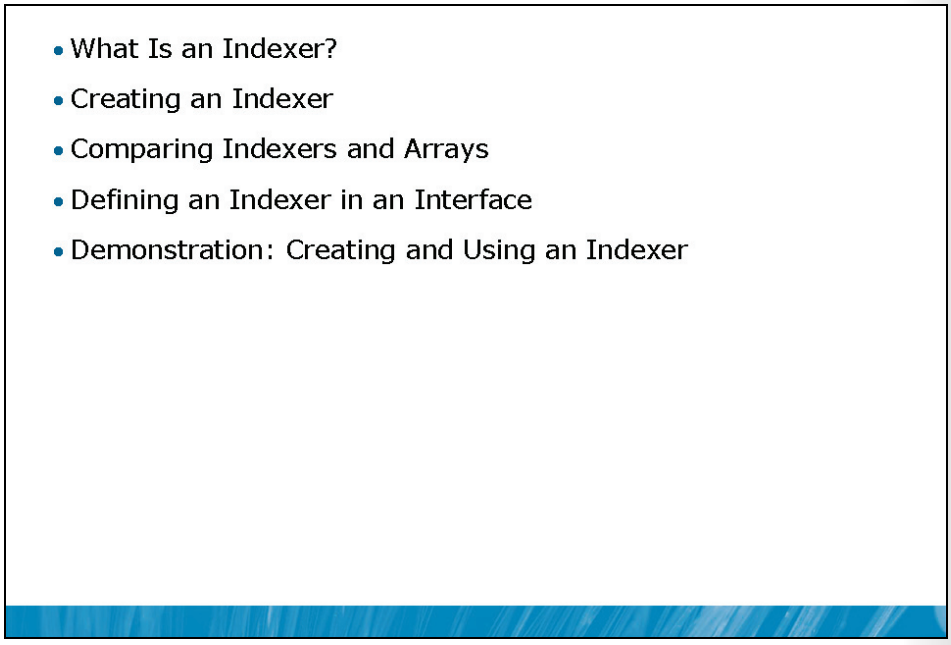


Review Questions

1. What is the syntax for declaring a property in an interface?
2. What is the significant difference between automatic properties and nonautomatic properties?
3. What happens if you attempt to write to a property that exposes only a get accessor?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Creating and Using Indexers



A property typically provides access to a single item in a type. However, some types are inherently multivalued, such as an array or a collection. Similarly, an item may contain subelements that you want to provide easy access to. For example, you can think of a string as a set of characters, and you may need to provide access to the individual characters in a string field through a property.

The most natural syntax for accessing elements in a set is to use array-like notation, and you can provide this access by defining indexer properties.

This lesson introduces you to indexers and describes how you can use indexers to encapsulate data in your applications.

Objectives

After completing this lesson, you will be able to:

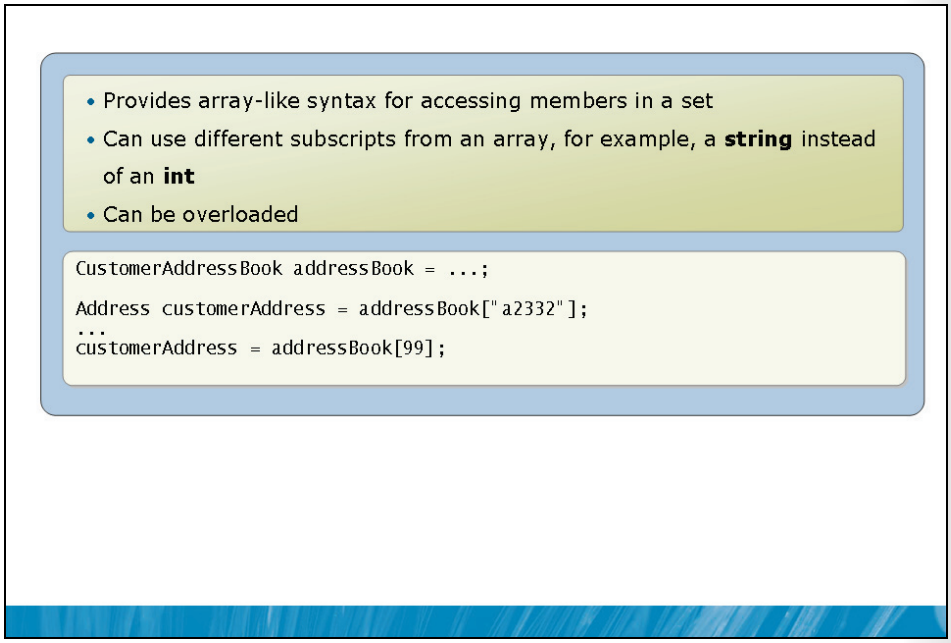
- Describe the purpose of an indexer.
- Implement an indexer.

MCT USE ONLY. STUDENT USE PROHIBITED

- Access data in your applications by using a type that exposes an indexer.
- Describe the differences between an indexer and an array.
- Define an indexer in an interface.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Indexer?



Key Points

An indexer provides a mechanism for encapsulating a set of values, in the same way that a property encapsulates a single value. You use an indexer to access a single value in a set of values, but you use **get** and **set** accessors to control how the value is retrieved or set based on a subscript passed as a parameter to the indexer. The **get** and **set** accessors use a property-like syntax.

Accessing an indexer uses the same syntax as accessing an array. However, with indexers, you have more flexibility. For example, with an indexer, you can use a noninteger type as the subscript instead of an integer normally used to access an array.

The following code example shows the use of a simple indexer for a type called **CustomerAddressBook**. This type provides an indexer that enables an application to retrieve the address of a customer by specifying the ID of that customer. The customer ID is held as a string.

MCT USE ONLY. STUDENT USE PROHIBITED

```
CustomerAddressBook addressBook = ...;
// Use an indexer to find the address of a customer.
Address customerAddress = addressBook["a2332"];
```

A type can define overloaded indexers that take different types of parameters. For example, the **CustomerAddressBook** type could also provide an indexer that retrieves a customer address based on an integer reference number, as the following code example shows.

```
// Find the address of the customer with the specified reference.
Address customerAddress = addressBook[99];
```

In addition to defining indexers that take different parameters, indexers can also return different types; they do not have to return an instance of the type that defines the indexer.

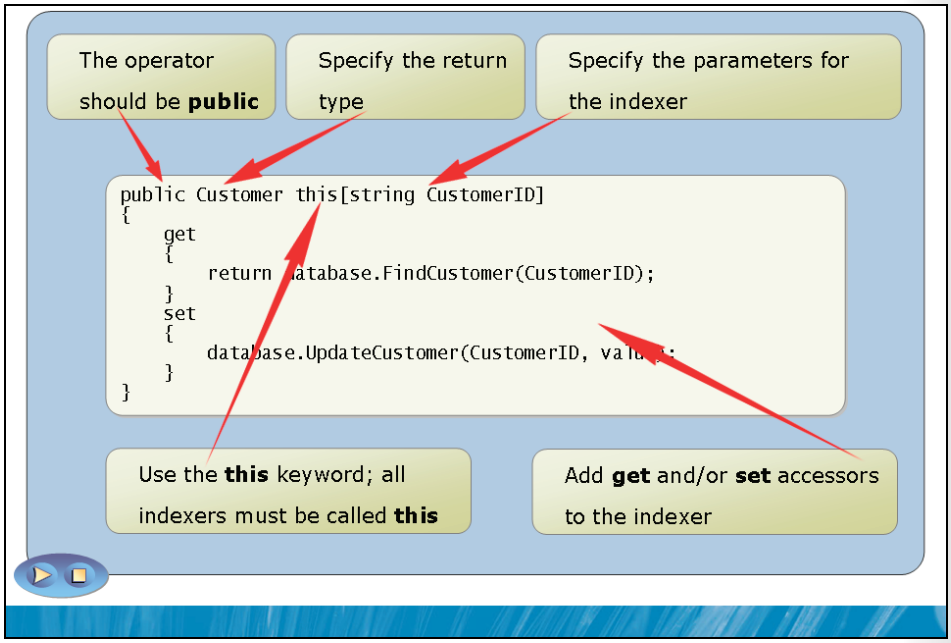
Question: When may you want to add an indexer to a type?

Additional Reading

For more information about the comparison between properties and indexers, see the Comparison Between Properties and Indexers (*C# Programming Guide*) page at http://go.microsoft.com/fwlink/?LinkId=192952.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating an Indexer



Key Points

Writing an indexer is a cross between writing a property and using an array.

You use syntax reminiscent of properties to specify the type and **get** and **set** accessors, but the name of the indexer is always *this*. You specify the types and names of parameters by using array-like notation in square brackets.

Like a property, an indexer can also be read-only (it only has a **get** accessor) or write-only (it only has a **set** accessor).

You can access the indexer parameters by name in the accessors, and in the **set** accessor, you can use the **value** keyword to access the value passed to the indexer.

Parameters passed to an indexer are only intended to be used to locate the data item to set or get. In the **get** accessor, you return the item found at this location, and in the **set** accessor, you store the data specified by the *value* parameter at this location.

The following code example shows a simple indexer that enables an application to find the address of a customer given the customer ID, or update the address. The address is stored in a database, accessed through the database variable.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class AddressBook
{
    public Address this[string CustomerID]
    {
        get
        {
            return database.FindCustomer(CustomerID);
        }
        set
        {
            database.UpdateCustomer(CustomerID, value);
        }
    }
    ...
}
```



Important: Ensure that you incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value.



Note: You cannot define static indexers.

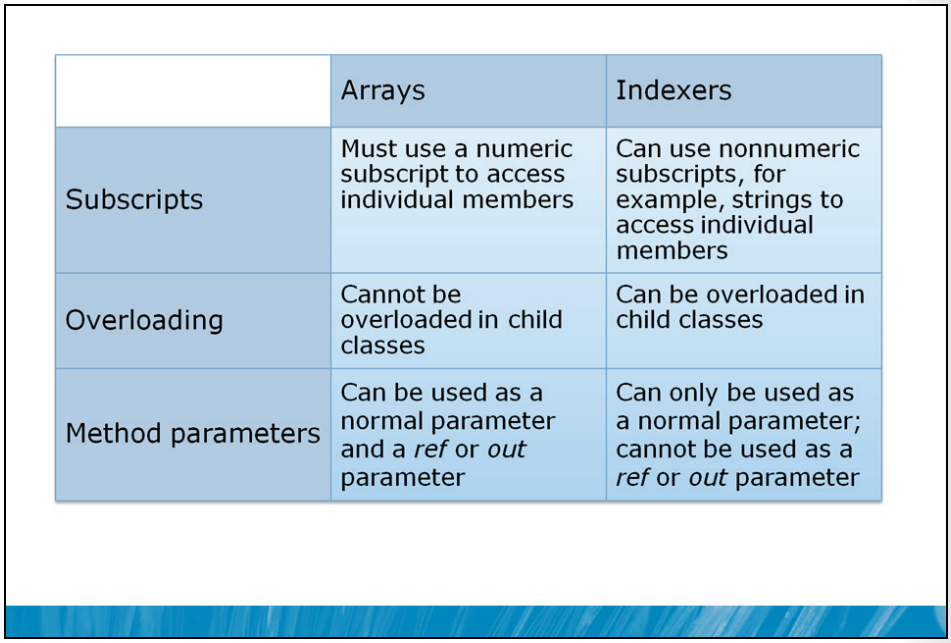
Question: What information should you use as parameters for an indexer?

Additional Reading

For more information about using indexers, see the Using Indexers (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192953.

MCT USE ONLY. STUDENT USE PROHIBITED

Comparing Indexers and Arrays



	Arrays	Indexers
Subscripts	Must use a numeric subscript to access individual members	Can use nonnumeric subscripts, for example, strings to access individual members
Overloading	Cannot be overloaded in child classes	Can be overloaded in child classes
Method parameters	Can be used as a normal parameter and a *ref* or *out* parameter	Can only be used as a normal parameter; cannot be used as a *ref* or *out* parameter

Key Points

To use an indexer, you use a similar syntax to that of an array; however, there are several important differences between an indexer and an array.

Indexer Subscripts

When you use an array, you access members of that array by using a numeric subscript. For example, you can access the fifth element in an array and use syntax similar to *myArray[4]* (assuming a zero-based index). With arrays, you can only use numeric subscripts. An indexer gives you greater flexibility because you can use nonnumeric subscripts.

Overloading an Indexer

You cannot overload an array; the implementation is defined by the runtime, and all classes that inherit from your class cannot change the behavior of that array. However, you have complete control over the behavior of an indexer, and classes that inherit from your class can override the indexer and provide their own implementation.

MCT USE ONLY. STUDENT USE PROHIBITED

Using an Indexer As a Parameter

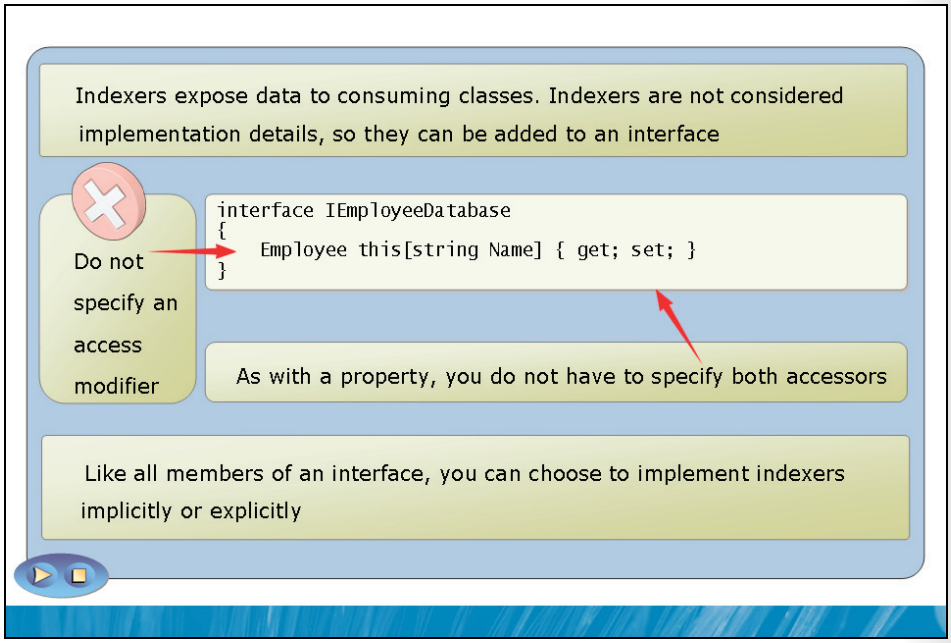
The previous two differences are benefits of using an indexer instead of an array. They are both true because when you use an indexer, you effectively call a method in your class (although this is handled by the compiler).

When you call a method that takes a *ref* or *out* parameter, you must pass a pointer to a memory location to the method. Items in an array can be mapped directly to memory locations, so they can be used as a parameter to a method that takes a *ref* or *out* parameter. Indexers do not map directly to memory locations, so you cannot use an indexer as a *ref* or *out* parameter, although you can pass them as value parameters.

Question: Should you use an indexer or an array if you must pass a value to a method by reference?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining an Indexer in an Interface



Key Points

You can specify an indexer in an interface. Any class that implements the interface is then required to implement that indexer. To specify an indexer in an interface, you add the indexer, without an access modifier, specifying **get**, **set**, or both accessors. You replace the body of the accessors with a semicolon.

The following code example shows an indexer in an interface.

```
interface IEmployeeDatabase
{
    Employee this[string Name] { get; set; }
}
```

You can implement an indexer in a class that implements the interface implicitly or explicitly.

The following code example shows a class implicitly implementing an interface with an indexer.

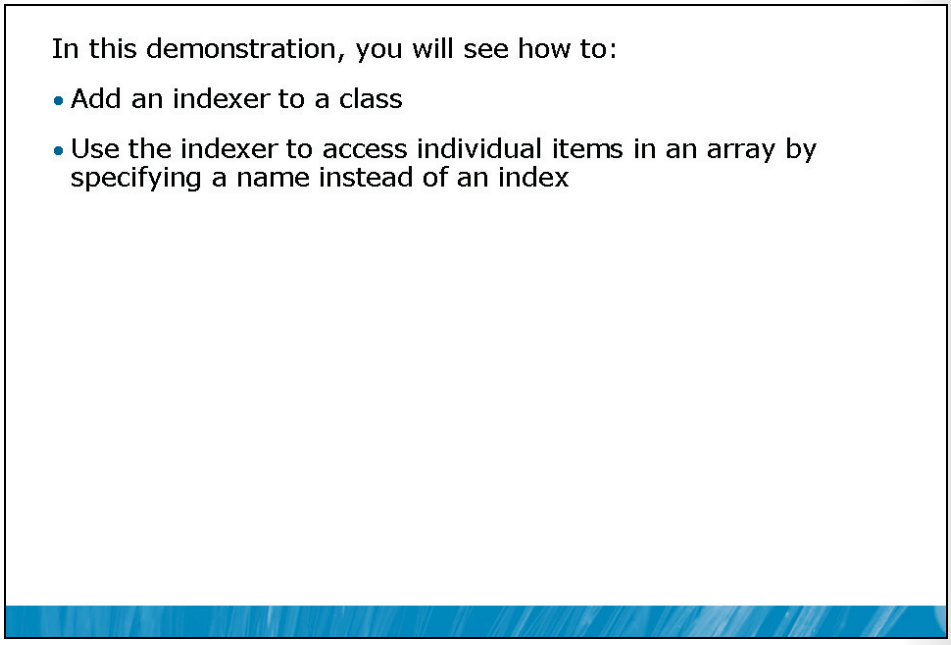
MCT USE ONLY. STUDENT USE PROHIBITED

```
class EmployeeDatabase : IEmployeeDatabase
{
    public Employee this[string Name]
    get
    {
        ...
        return employee;
    }
    set
    {
        ...
    }
}
```

Question: How can you use interfaces to add more than one indexer that takes the same parameters to a class?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Creating and Using an Indexer



Key Points

- Add an indexer to a class to enable access to individual records in a class simulating a database.
- Use the indexer by using a test harness.

Demonstration Steps

1. Start Visual Studio.
2. Open the **CreatingAndUsingAnIndexerDemo** solution in the **E:\Demofiles\Mod10\Demo2\Starter\CreatingAndUsingAnIndexerDemo** folder.
3. Open the EmployeeDatabase.cs file, and then review the **EmployeeDatabase** class.

   Notice that the class stores an array of **Employee** objects.

MCT USE ONLY. STUDENT USE PROHIBITED

Notice the **AddToDatabase** method, and then explain how it adds **Employee** objects to the array and increments a pointer to the top of the array.

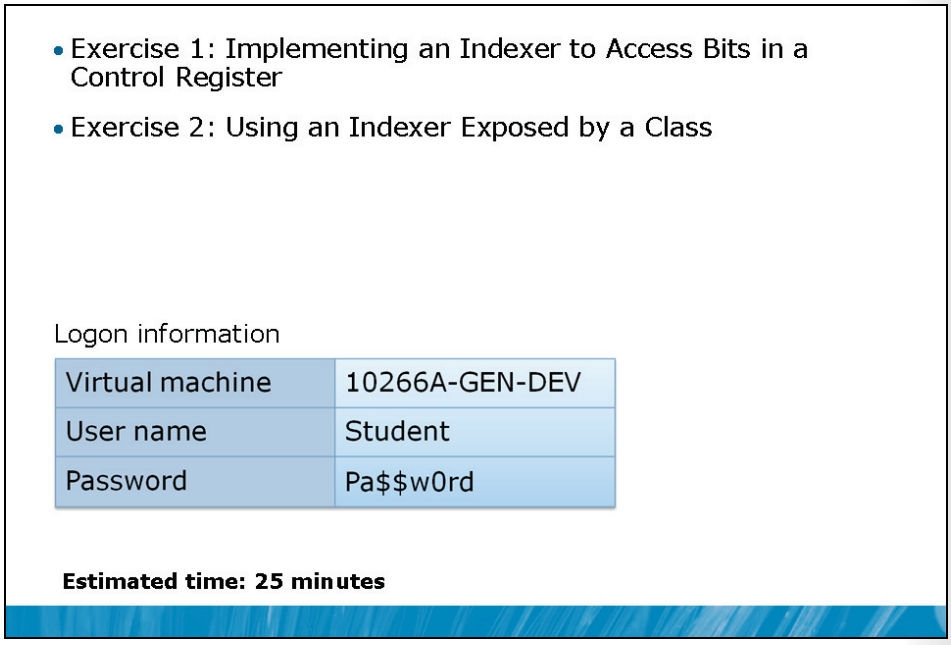
4. Uncomment the indexer that returns an **Employee** object. Notice how the indexer takes a *string* parameter called **Name** and iterates through each employee in the array until it finds one with a matching **Name** property. It then returns that value. If it does not find a match after iterating over the entire array, it returns **null**.
5. Open the Program.cs file, and, uncomment the commented code, and then explain how this code uses the indexer to retrieve **Employee** instances by specifying the employee name.
6. Run the application without debugging.

   Notice that the application runs as expected, and the details of the two employees retrieved from the database are displayed correctly, and then press ENTER.
7. Close Visual Studio.

Question: Can you develop more than one indexer with the same set of parameters?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab B: Creating and Using Indexers



Objectives

After completing this lab, you will be able to:

- Implement an indexer to provide access to items in a class.
- Use an indexer to query and modify data.

Introduction

In this lab, you will add an indexer to a class. You will then use a test application to verify that the indexer functions correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

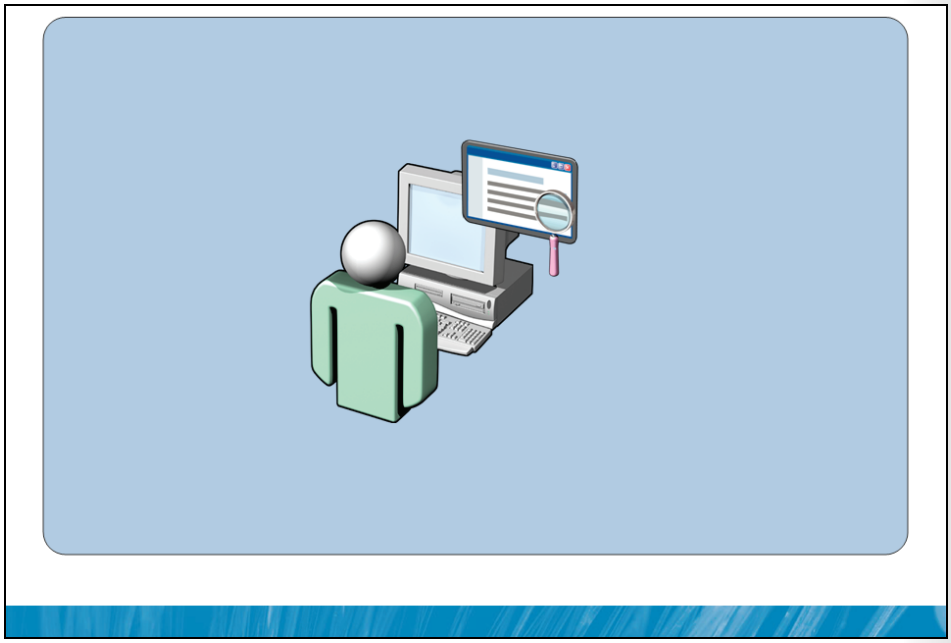
Lab Setup

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

Lab Scenario



The software that drives some devices provides access to the control registers that these devices use internally. You have previously seen how to display the data in these registers by converting the integer data held in them into binary strings. You have now been asked to provide read/write access to the individual bits in a register.

In this lab, you will define a new structure called **ControlRegister** that contains the following members:

- **registerData**: A private integer field representing the value of the control register.
- **RegisterData**: A read/write property that exposes the **registerData** field.
- An indexer that provides read/write access to the individual bits in the **registerData** field by using array-like notation. For example, if **DeviceRegister** is an instance of the **ControlRegister** structure, the statement **DeviceRegister[2] = 1** will set bit 2 of the **registerData** field to the value 1, and the statement **x = DeviceRegister[3]** will return the value of bit 3 in the

MCT USE ONLY. STUDENT USE PROHIBITED

registerData field. The indexer must ensure that all of the values assigned are either 0 or 1.

In this lab, you will use binary operators to access bits in a control register. You will use the left-shift operator (**<<**), the right-shift operator (**>>**), the **NOT** operator (**~**), the **AND** operator (**&**), and the **OR** operator (**|**).

The following code example shows how to use the **AND** operator and the left-shift operator to check whether the fifth bit is 0 or 1 in a control register.

```
registerData & (1 << index)

If registerData = 3 and index = 5:

1                           : 0 0 0 0 0 0 0 1
1 << 5                      : 0 0 1 0 0 0 0 0

registerData                : 0 0 0 0 0 0 1 1

registerData & (1 << 5)     : 0 0 0 0 0 0 0 0

The result is 0 so the bit was 0. If the fifth bit in the register was
1 the result would have been a value other than 0.
```

Exercise 1: Implementing an Indexer to Access Bits in a Control Register

Scenario

In this exercise, you will add an indexer to a **ControlRegister** class that represents a control register. The class will store the value of the control register in an integer field, and you will use binary operators to retrieve and update the bits in the register.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add an indexer to the **ControlRegister** class.

▶ 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. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab B\Ex1\Starter folder.

▶ Task 2: Add an indexer to the ControlRegister class

1. In Visual Studio, review the task list.
2. Open the ControlRegister.cs file.
3. Remove the comment **TODO: Add an indexer to enable access to individual bits in the control register** and add a **public** indexer to the class. The indexer should take an **int** called **index** as the parameter and return an **int**.
4. Add a **get** accessor to the indexer. In the **get** accessor, add code to determine whether the bit specified by the *index* parameter in the **registerData** object is set to 1 or 0 and return the value of this bit.



Hint: Use the logical **AND** operator (**&**) and the left-shift operator (**<<**) to determine whether the result of left-shifting the value in the **registerData** object by the value of the **index** object is zero or non-zero. If the result is zero, return 0; otherwise, return 1. You can use the following code example to assist you with this step.

```
// Incomplete—Use this as part of your solution.
(registerData & (1 << index)) != 0
```

5. Add a **set** accessor to the indexer. In the **set** accessor, add code to verify that the parameter specified is either 1 or 0. Throw an **ArgumentException** exception with the message "Argument must be 1 or 0" if it is not one of these values.
6. In the **set** accessor, if **value** is 1, add code to set the bit specified by the **index** object in the **registerData** field to 1; otherwise, set this bit to 0.



Hint: Use the compound assignment operators **|=** and **&=** to set a specified bit in an integer value to 1 or 0. Use the expression **(1 << index)** to determine which bit in the integer value to set.

7. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Using an Indexer Exposed by a Class

Scenario

In this exercise, you will use a test harness to access bits in the **ControlRegister** class that you implemented in the previous exercise.

The main tasks for this exercise are as follows:

1. Add the test harness to the solution.
2. Update the test harness.
3. Test the **ControlRegister** class by using the test harness.

▶ Task 1: Add the test harness to the solution

The test harness application for this lab is a simple console application that is designed to test the functionality of the **ControlRegister** class to which you have added an indexer. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class you have developed.

1. Add the test harness to the solution. The test harness is a project called Exercise2TestHarness, located in the E:\Labfiles\Lab 10\Lab B\Ex2 \Starter\Exercise2TestHarness folder.
2. Set the Exercise2TestHarness project as the startup project for the solution.

▶ Task 2: Update the test harness

1. In Visual Studio, review the task list.
2. Open the Program.cs file.
3. Remove the TODO comment.
4. Add code to create a new instance of the **ControlRegister** class called **register**.
5. Add code to set the **RegisterData** property of the **register** object to **8**.
6. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: A code snippet is available, called Mod10WriteRegisterData, that you can use to add this code.

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

7. Add a statement to write the message "Set Bit 1 to 1" to the console.
8. Add a statement to set the bit at index **1** in the **register** object to **1**.
9. Add code to write a blank line to the console.
10. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: You can use the Mod10WriteRegisterData code snippet to add this code.

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

11. Add a statement to write the message "Set Bit 0 to 1" to the console.
12. Add code to set the bit at index **0** in the **register** object to **1**.
13. Add code to write a blank line to the console.
14. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: You can use the Mod10WriteRegisterData code snippet to add this code.

MCT USE ONLY. STUDENT USE PROHIBITED

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

15. Build the solution and correct any errors.

▶ Task 3: Test the ControlRegister class by using the test harness

1. Start the Exercise2TestHarness application.
2. Verify that the output from the console appears correctly. The output should resemble the following code example.

```
RegisterData : 8
Bit 0: 0
Bit 1: 0
Bit 2: 0
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0

Set Bit 1 to 1

RegisterData : 10
Bit 0: 0
Bit 1: 1
Bit 2: 0
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0

Set Bit 0 to 1

RegisterData : 11
Bit 0: 1
Bit 1: 1
Bit 2: 0
```

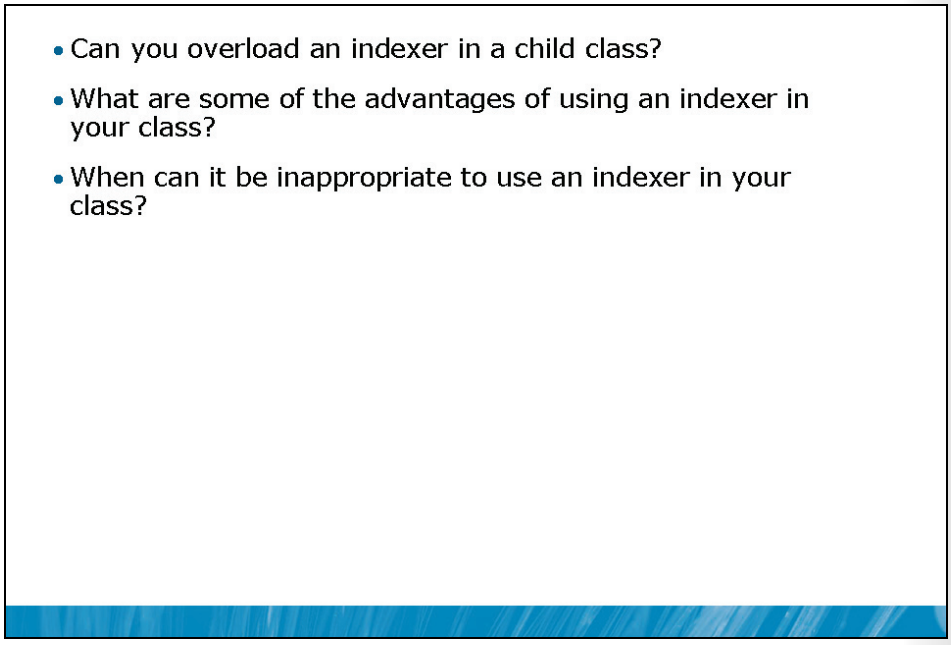
MCT USE ONLY. STUDENT USE PROHIBITED

```
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0
```

3. Close the Exercise2TestHarness window.
4. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

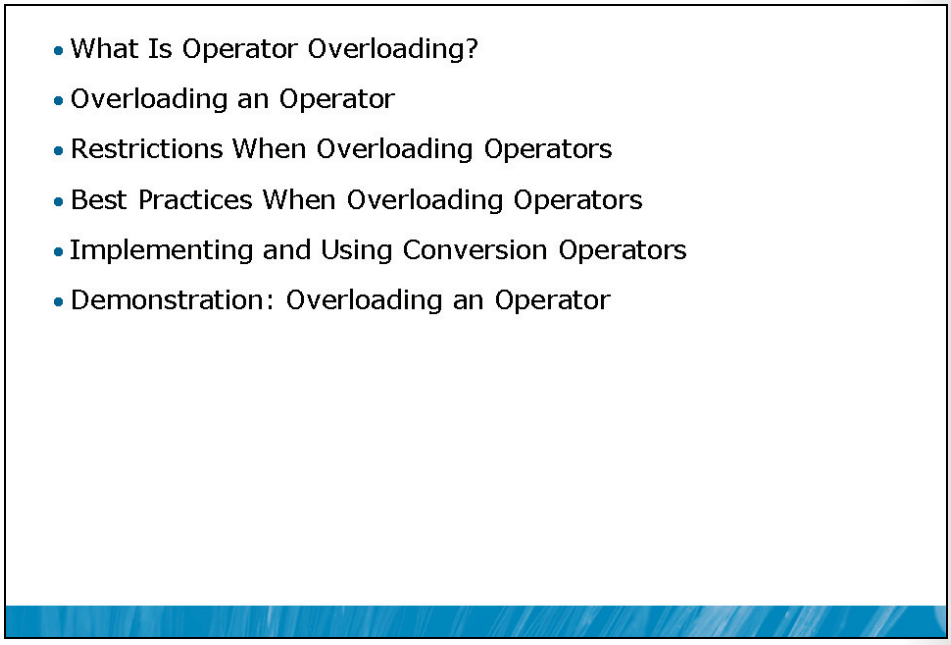


Review Questions

1. Can you overload an indexer in a child class?
2. What are some of the advantages of using an indexer in your class?
3. When can it be inappropriate to use an indexer in your class?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Overloading Operators



Many of the built-in types defined by Visual C# provide operators to enable you to perform some common operations on them. For example, Visual C# defines operators such as **+**, **-**, *****, and **/**, which have a well-defined behavior over numeric data. However, you have also seen that the **+** operator can work on the string type, when its behavior is quite different; the **+** operator for strings concatenates strings together. This is an example of an overloaded operator.

You can implement overloaded operators for your own types. This lesson shows you how to define and implement operator overloading. It also describes some best practices you should follow when you define operators for your types.

Objectives

After completing this lesson, you will be able to:

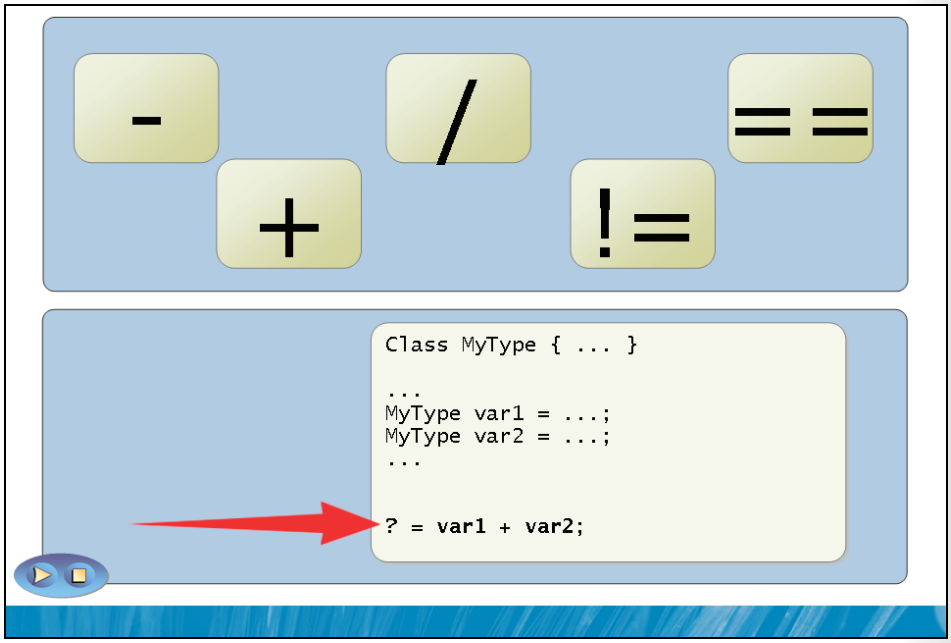
- Describe how operator overloading works.
- Define an overloaded operator.
- Explain the restrictions when overloading operators.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain the best practices for operator overloading.
- Describe how to implement and use conversion operators.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Operator Overloading?



Key Points

Visual C# includes several operators that enable you to perform common operations on objects. You can use these operators to construct expressions. The exact behavior of each of the operators is dependent on the type of the object that you perform the operation on.

An operator is a special method that takes a set of parameters and returns a value. When you invoke an operator, the operands are passed as parameters to this method, and the value returned by the method is used as the result of the operator. When you overload an operator, you provide your own implementation of this method.

Visual C# defines three categories of operators that you can overload:

- *Unary operators*. These operators include **!**, **++**, **--**, **+**, and **-**. When you overload these operators, you specify a single parameter that must be of the same type as the class that defines the operator.

MCT USE ONLY. STUDENT USE PROHIBITED

- *Binary operators*. These operators include *, /, +, -, and %. When you overload these operators, you specify two parameters, at least one of which must be of the same type as the class that defines the operator.
- *Conversion operators*. You can use these operators to change data from one type to another. When you overload these operators, you specify a single parameter that contains the data that you want to convert from. This data can be any valid type.

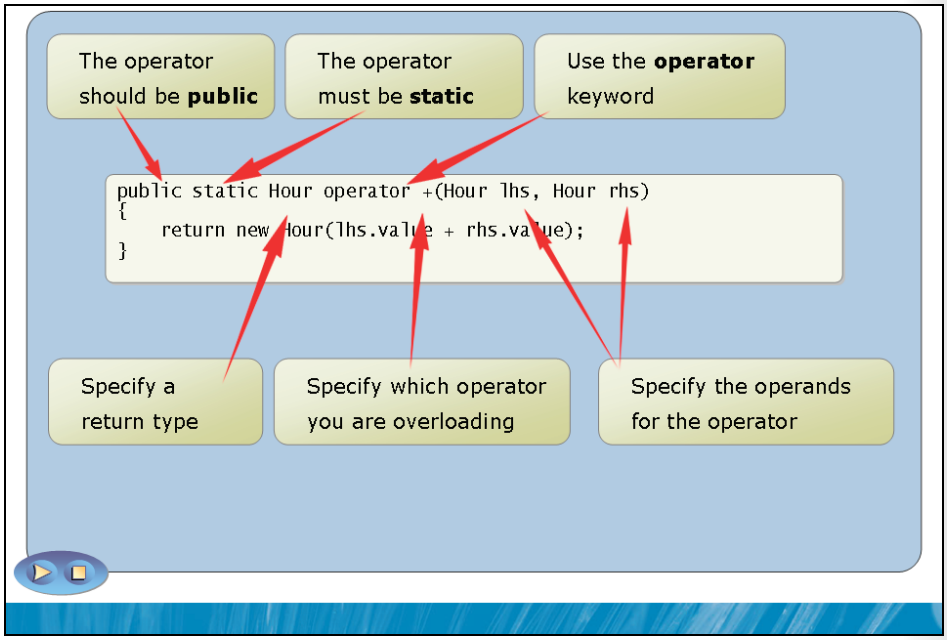
You cannot overload all of the operators defined by Visual C#. The following table summarizes which operators you can and cannot overload.

Operators	Ability to be overloaded
+, **-**, **!**, **~**, **++**, **--**, **true**, **false**	These unary operators can be overloaded.
+, **-**, *****, **/**, **%**, **&**, **|**, **^**, **<<**, **>>**	These binary operators can be overloaded.
==, **!=**, **<**, **>**, **<=**, **>=**	These comparison operators can be overloaded.
&&, **||**	The conditional logical operators cannot be overloaded, but they are evaluated by using **&** and **|**, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
()	The cast operator cannot be overloaded, but you can define new conversion operators as described later in this module.
+=, **-=**, ***=**, **/=**, **%=**, **&=**, **|=**, **^=**, **<<=**, **>>=**	Assignment operators cannot be overloaded, but **+=**, for example, is evaluated by using **+**, which can be overloaded.
=, **.**, **?:**, **->**, **new**, **is**, **sizeof**, **typeof**	These operators cannot be overloaded.

Question: If you overload the **+** operator in a type, does the compiler automatically generate an equivalent **−** operator?

MCT USE ONLY. STUDENT USE PROHIBITED

Overloading an Operator



Key Points

To define your own operator behavior, you must overload a selected operator. You use method-like syntax with a return type and parameters, but the name of the method is the keyword **operator** together with the symbol for the operator that you are overloading. For example, to overload the **+** operator, you define a method called **operator +**.

For example, the following code example shows a user-defined structure named **Hour** that defines a binary **+** operator to add together two instances of **Hour**.

```
struct Hour
{
    public Hour(int initialValue)
    {
        this.value = initialValue;
    }

    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    ...

    private int value;
}
```

Notice the following points about the **operator +** method:

- All operators must be **public**.
- All operators must be **static**. Operators are never polymorphic and cannot use the **virtual**, **abstract**, **override**, or **sealed** modifier.



Tip: When declaring highly stylized functionality (such as operators), it is useful to adopt a naming convention for the parameters. For example, developers often use **lhs** and **rhs** (acronyms for left-hand side and right-hand side, respectively) for binary operators.

When you use the **+** operator on two expressions of type **Hour**, the Visual C# compiler automatically converts your code to a call to your **operator +** method. Take the following code example as an example.

```
Hour Example(Hour a, Hour b)
{
    return a + b;
}
```

The Visual C# compiler converts the previous code into code that resembles the following code example (this is pseudocode and not legal Visual C# syntax).

```
Hour Example(Hour a, Hour b)
{
    return Hour.operator +(a,b); // pseudocode
}
```

There is one final rule that you must follow when declaring an operator: at least one of the parameters must always be of the containing type. In the preceding **operator +** method example for the **Hour** class, one of the parameters, *a* or *b*, must be an **Hour** object. In this example, both parameters are **Hour** objects.

Operators follow the usual overloading rules, and you can overload an operator as many times as you want in a class as long as the Visual C# compiler can distinguish between each overload (the signatures must be unique in the class). For example, you can define an additional implementation of the **operator +** method to add an integer (a number of hours) to an **Hour** object—the first

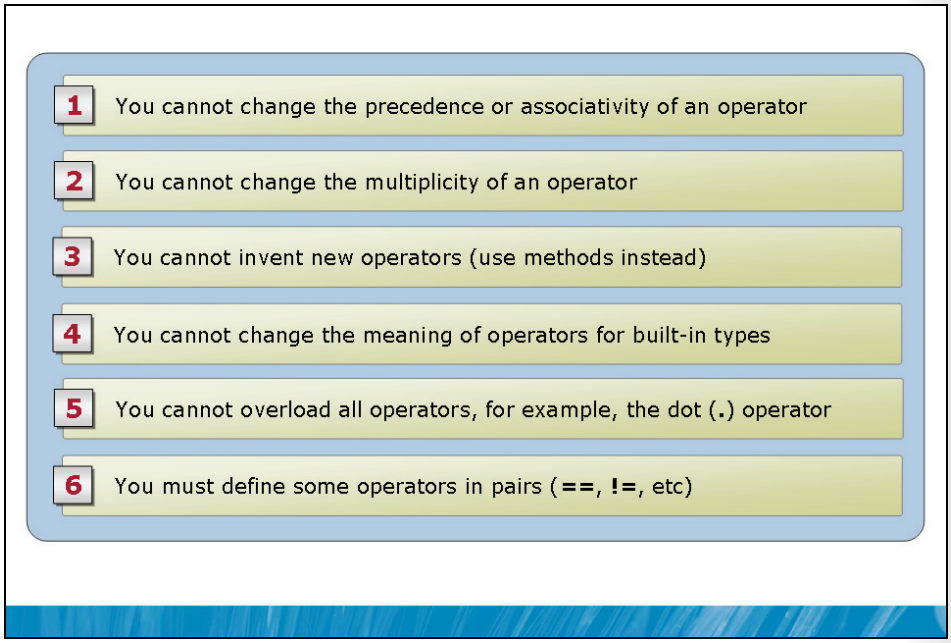
MCT USE ONLY. STUDENT USE PROHIBITED

parameter can be an **Hour** object and the second parameter can be an integer object.

Question: Does the first operand of an overloaded operator have to be the containing type?

MCT USE ONLY. STUDENT USE PROHIBITED

Restrictions When Overloading Operators



Key Points

When you overload an operator, you can completely control how an operation is performed; however, there are some rules that apply to operators that you cannot change:

- You cannot change the precedence or associativity of an operator. The precedence and associativity are based on the operator symbol (for example, **+**) and not on the type (for example, **int**) on which the operator symbol is being used. Hence, the expression **a + b * c** is always the same as **a + (b * c)**, regardless of the types of **a**, **b**, and **c**.
- You cannot change the multiplicity (the number of operands) of an operator. For example, * (the symbol for multiplication) is a binary operator (has two operands). If you declare a * operator for your own type, it must be a binary operator. Similarly, ++ is a unary operator (takes one operand); if you declare ++ in your type, it must be a unary operation.
- You cannot invent new operator symbols. For example, you cannot create a new operator symbol, such as **, for raising one number to the power of

MCT USE ONLY. STUDENT USE PROHIBITED

another number. If you must perform an operation for which there is no operator, you must create a method instead.

- You cannot change the meaning of operators when applied to built-in types. For example, the expression **1 + 2** has a predefined meaning, and you cannot override this meaning. In fact, when you define an operator in your type, at least one of the operands for that operation must be the containing type, so you cannot define an operation where all of the operands are built-in types.

In addition, you must implement the comparison operators in pairs. For example, if you overload the **>** operator, you must also overload the **<** operator. If you overload the **==** operator, you must also overload the **!=** operator.



Note: If you define the **==** operator and the **!=** operator in a class, you should also override the **Equals** and **GetHashCode** methods inherited from **System.Object** (or **System.ValueType** if you are creating a structure). The **Equals** method should exhibit exactly the same behavior as the **==** operator. (You should define one in terms of the other.) The **GetHashCode** method is used by other classes in the Microsoft .NET Framework (for example, when you use an object as a key in a hash table).

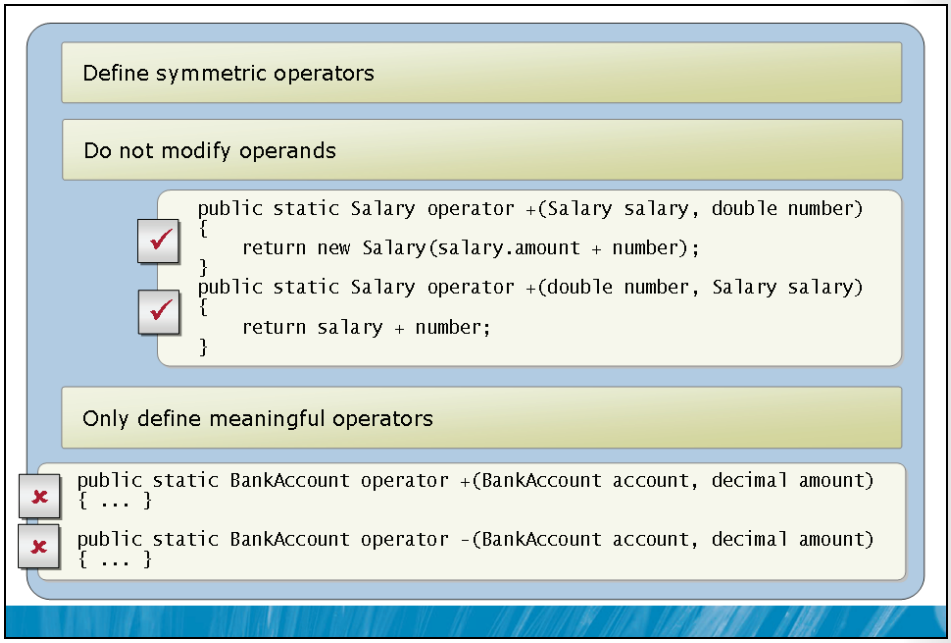
Question: How can you change the multiplicity of an operator?

Additional Reading

For more information about using the **Equals** method, see the Object.Equals Method (Object) page at http://go.microsoft.com/fwlink/?LinkId=192954.

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices When Overloading Operators



Key Points

When you define overloaded operators for your types, you should adhere to the following best practices where possible:

- Do not modify the operands.
- Define symmetric operators.
- Define only meaningful operators.

Not Modifying Operands

An operator should never change the values of either of its operands. If any of these operands are reference types, using the operator will change the value of the operand in addition to returning a result (the operator causes a side effect). For example, in the following code example, the **Salary** class provides the **+** operator to add a **decimal** value to the **amount** field in the class and returns the updated **Salary** object as the result.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class Salary
{
    private decimal amount;
    public decimal Amount
    {
        get { return this.amount; }
    }

    public Salary(decimal amt)
    {
        this.amount = amt;
    }
    ...

    public static Salary operator +(Salary salary, decimal number)
    {
        salary.amount += number;
        return salary;
    }
}
```

This is a poor implementation, because the **+** operator changes the value of the first operand. When the **+** operator completes, the **amount** field in the first operand has the same value as the result. In the following code example, the value in **newSalary.Amount** is 109, but the value in **salary.Amount** is also 109 when it would be expected to have remained at 99 by most users.



Note: If **Salary** is a struct rather than a class, this side effect will not occur, because the *Salary* parameter will be passed to the **+** operator by value rather than by reference.

```
Salary salary = new Salary(99);
Salary newSalary = salary + 10;
Console.WriteLine("{0} {1}", salary.Amount, newSalary.Amount);

Output
------
109 109
```

Instead, you should return a new object that contains the new value, as the following code example shows.

```
class Salary
{
    private decimal amount;
    public decimal Amount
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        get { return this.amount; }
    }

    public Salary(decimal amt)
    {
        this.amount = amt;
    }
    ...

    public static Salary operator +(Salary salary, decimal number)
    {
        return new Salary(salary.Amount + number);
    }
}
```

Defining Symmetric Operators

When you define a binary operator, you should avoid imposing an order on the operands. If an operator is commutable, the order in which you specify the operands should not make any logical difference. In the **Salary** class example, the expressions **salary + 99** and **99 + salary** should have the same result.

Remember that when you overload an operator, the first operand is used as the first parameter and the second operand is used as the second parameter. Therefore, to support both forms of addition, the **Salary** class must provide the two implementations of the **+** operator shown in the following code example. The compiler will not automatically add symmetric operators for you.

```
public static Salary operator +(Salary salary, decimal number)
{
    return new Salary(salary.Amount + number);
}

public static Salary operator +(decimal number, Salary salary)
{
    // Call the first operator – avoid code duplication.
    return salary + number;
}
```

Note that the second implementation of the + operator simply invokes the first by switching the two operands over. This is good practice because it ensures that the logic defining the operation is held in a single operator and consequently is easier to maintain.

MCT USE ONLY. STUDENT USE PROHIBITED

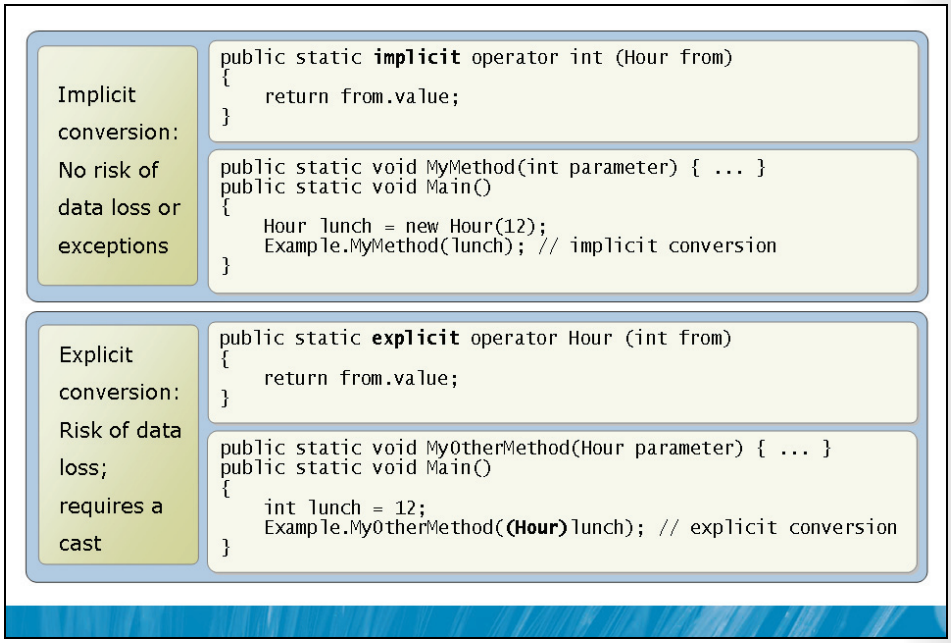
Defining Only Meaningful Operators

As with properties, you should define operators only where it is natural and meaningful to do so. For example, you should probably not define the + or – operators on a class that models a bank account to add or remove funds from the account. This is because banks generally perform many additional checks when they add or remove funds and it is not a simple addition or subtraction operation. Instead, you should provide **Deposit** and **Withdraw** methods that can encapsulate these checks in a more meaningful manner.

Question: Why should you always return a new object rather than update one of the operands?

MCT USE ONLY. STUDENT USE PROHIBITED

Implementing and Using Conversion Operators



Key Points

A conversion operator converts an expression from one type to another.

A conversion can be either implicit or explicit. Implicit conversions occur when changing an expression from one type to a more specific type with no loss of precision. This is called a *widening* conversion. An operator that implements a widening conversion can be invoked automatically by the compiler without requiring any additional intervention by a programmer. For example, in the following code example, the statement that assigns an integer expression to a double variable should always succeed without losing data because the **int** has a smaller scale and precision than the **double** type.

```
int i = 99;
double d = i;  // Safe, widening conversion from int to double
```

An explicit conversion occurs when changing from a type to a less specific type where there is the risk of data loss. This is called a *narrowing* conversion. Because of the potential loss of data, narrowing conversions are not performed automatically, but require the programmer to specify a cast. Assigning a double

MCT USE ONLY. STUDENT USE PROHIBITED

value to an integer is an example of a narrowing conversion that requires a cast, as the following code example shows.

```
double d = 99.9;
int i = (int)d;  // Data loss, narrowing conversion from double to int
```

In this example, the value store in **d** is truncated as part of the conversion, so the result stored in **i** is 99.

Defining Conversion Operators

The syntax for declaring a user-defined conversion operator is similar to that for declaring an overloaded operator. A conversion operator must be **public** and must also be **static**. The name of a conversion operator is either **implicit** (if it implements a widening conversion) or **explicit** (if it implements a narrowing conversion).

The following code example shows a conversion operator that allows an **Hour** object to be implicitly converted to an **int**. This is a safe conversion because all hours have an equivalent integer value.

```
struct Hour
{
    ...
    public static implicit operator int (Hour from)
    {
        return from.value;
    }

    private int value;
}
```

You declare the type you are converting from (**Hour**) as the single parameter and the type you are converting to (**int**) after the **operator** keyword.

When you declare a conversion operator, you must specify whether it is an implicit or an explicit conversion operator by using the **implicit** and **explicit** keywords.

You can invoke an implicit conversion operator without requiring a cast as the following code example shows.

```
class Example
{
    public static void MyOtherMethod(int parameter) { ... }
    public static void Main()
    {
        Hour lunch = new Hour(12);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        Example.MyOtherMethod(lunch); // implicit conversion
    }
}
```

The following code example shows an explicit conversion operator that converts an **int** object to an **Hour** object. Notice that the return type is now **Hour** and the parameter is an **int**. This is a narrowing operation because not all integer values represent valid hours. The conversion operator builds an **Hour** object by using the remainder after dividing the integer parameter by 24, as the following code example shows.

```
struct Hour
{
    ...
    public Hour(int hr)
    {
        this.value = hr % 24;
    }

    public static explicit operator Hour (int from)
    {
        return new Hour(from);
    }

    private int value;
}
```

When should you declare a conversion operator as explicit or implicit? If a conversion is always safe, does not run the risk of losing information, and cannot throw an exception, it can be defined as an implicit conversion. Otherwise, it should be declared as an explicit conversion.

Symmetric Operators and Conversions

Conversion operators provide you with an alternative way to resolve the problem of providing symmetric operators. For example, suppose you define the **+** operator to enable you to add **Hour** objects to **Hour** objects, and **Hour** objects to **int** values. Instead of providing three versions of the **+** operator (**Hour** + **Hour**, **Hour** + **int**, and **int** + **Hour**) for the **Hour** structure, you can provide a single version of **operator+** that takes two *Hour* parameters and an implicit **int** to **Hour** conversion operator, as the following code example shows.

```
struct Hour
{
    public Hour(int hr)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        this.value = hr % 24;
    }

    public static Hour operator +(Hour lhs, Hour rhs)
    {
        return new Hour(lhs.value + rhs.value);
    }

    public static implicit operator Hour (int from)
    {
        return new Hour (from);
    }

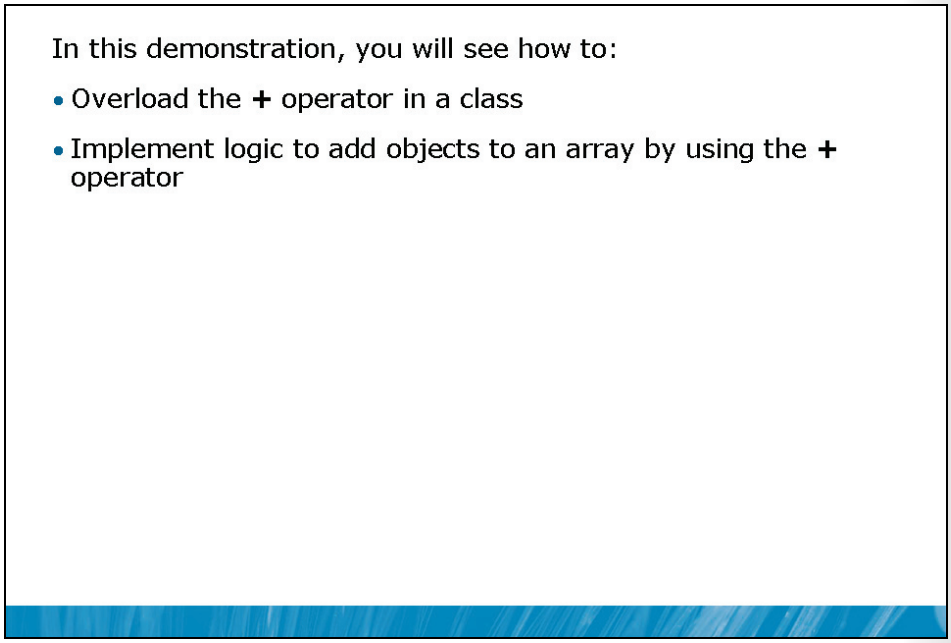
    ...
    private int value;
}
```

If you add an **Hour** object to an **int** object (in either order), the Visual C# compiler automatically converts the **int** object to an **Hour** object and then calls the **+** operator with two **Hour** arguments.

Question: When should you use an explicit conversion?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Overloading an Operator



Key Points

- Add a **+** operator to a class.
- Use the **+** operator in a consuming class.
- Use the **+=** syntax in a consuming class.

Demonstration Steps

1. Start Visual Studio.
2. Open the **OverloadingAnOperator** solution in the **E:\Demofiles\Mod10\Demo3\Starter\OverloadingAnOperator** folder.
3. Open the EmployeeDatabase.cs file, and then review the **EmployeeDatabase** class.

   Notice that the class stores an array of **Employee** objects and is the same as in the previous demonstration.

MCT USE ONLY. STUDENT USE PROHIBITED

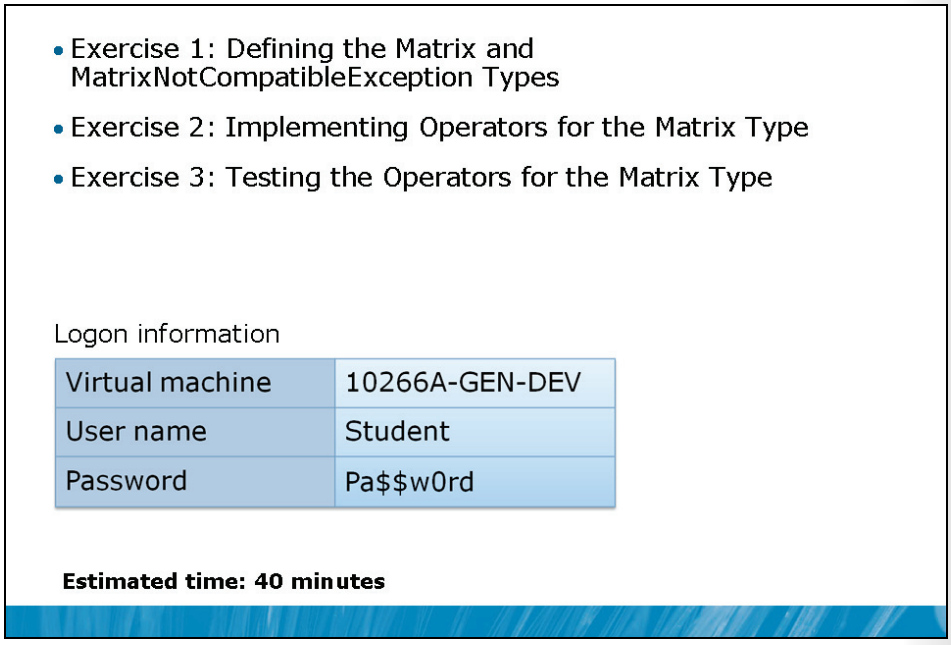
4. Uncomment the **+** operator that returns an **EmployeeDatabase** object. Notice how the **+** operator takes an **EmployeeDatabase** object and an **Employee** object as parameters, adds the **Employee** object to the database, and then returns a reference to the database.
5. Open the Program.cs file.
6. In the Program.cs file, and uncomment the commented code.

   Notice how this code adds several **Employee** objects to the database by using both the **+** syntax and the **+=** syntax.
7. Run the application without debugging.
8. Close Visual Studio.

Question: When can you use the **+=** syntax to abbreviate an addition operation?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab C: Overloading Operators



Objectives

After completing this lab, you will be able to:

- Define a new type that models a matrix.
- Implement operators for the matrix type.
- Use operators defined by the matrix type.

Introduction

In this lab, you will create a new type that models square matrices. You will implement the addition, subtraction, and multiplication operators for this type and test that these operators function correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

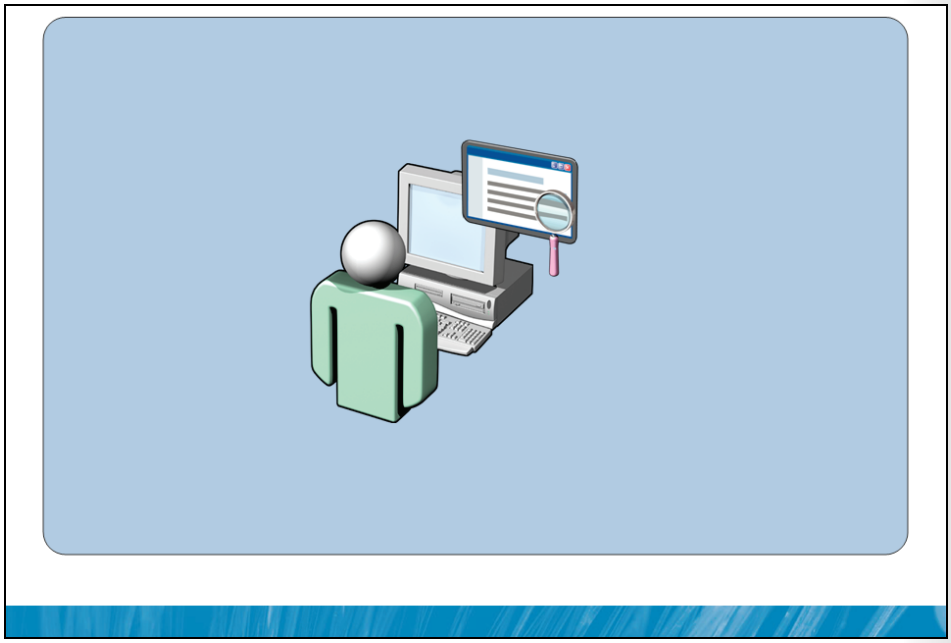
Lab Setup

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

Lab Scenario



Some of the engineering devices produced by Fabrikam, Inc. must perform calculations that involve matrices. You have been asked to implement a new, reusable type that can perform simple matrix operations.

In this lab, you will create a new type called **Matrix**. This type will implement a simple $n \times n$ square matrix. The value of n will be specified in the constructor, and the data for the matrix will be held in a two-dimensional array. The **Matrix** type will provide read/write access to the data in the array through an array property.

You will implement the following operators for the **Matrix** type:

- The * operator will perform matrix multiplication. It will return a new matrix that is the product of multiplying with another matrix provided as an argument.
- The + operator will perform matrix addition. It will return a new matrix that is the result of adding to another matrix provided as an argument.
- The - operator will perform matrix subtraction. It will return a new matrix that is the result of subtracting another matrix provided as an argument.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

All operators will perform error-checking to ensure that the matrices are compatible.

To add matrices, you add each element in one matrix to the corresponding element in the other. To add two matrices, y and z, you calculate each element x[a, b] in the result matrix by adding element y[a, b] to z[a, b]. Subtracting matrices is similar; for each element x[a, b] in the result matrix, calculate y[a, b] − z[a, b].

To multiply matrices, you calculate the sum of the products of the values in each row in one matrix and the values in each column in the other. To calculate each element x[a, 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 x[3,2] 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.

Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types

Scenario

In this exercise, you will define a **Matrix** class to represent a matrix. You will add an indexer to the class to enable access to individual data items in the matrix, and you will override the **ToString** method to return a formatted string that represents the matrix. You will also define a **MatrixNotCompatibleException** exception class. The **MatrixNotCompatibleException** class will be used when an operator is applied to two matrices that are incompatible; for example, because they are not the same size. The **MatrixNotCompatibleException** class will include fields exposed as read-only properties to reference the matrices on which the operation was performed. The fields will be set by using a constructor.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create a **Matrix** class.
3. Create a **MatrixNotCompatibleException** exception class.

▶ 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. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab C\Ex1\Starter folder.

▶ Task 2: Create a Matrix class

1. In Visual Studio, review the task list.
2. Open the Matrix.cs file.
3. Remove the comment **TODO: Add the Matrix class** and add a **public Matrix** class to the **MatrixOperators** namespace.
4. Add a two-dimensional array of integers named **data** to the **Matrix** class.
5. Add a **public** constructor to the **Matrix** class. The constructor should take a single integer parameter called *size* and initialize the **data** array to a square array by using the value passed to the constructor as the size of each dimension of the array.
6. After the constructor, add the following code to add an indexer to the class. You can either type this code manually, or you can use the Mod10MatrixClassIndexer code snippet.

```
public int this[int RowIndex, int ColumnIndex]
{
    get
    {
        if (RowIndex > data.GetUpperBound(0) ||
            ColumnIndex > data.GetUpperBound(0))
        {
            throw new IndexOutOfRangeException();
        }

        else
        {
            return data[RowIndex, ColumnIndex];
        }

    }


    set
    {
        if (RowIndex > data.GetUpperBound(0) ||
            ColumnIndex > data.GetUpperBound(0))
        {
            throw new IndexOutOfRangeException();
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        else
        {
            data[RowIndex, ColumnIndex] = value;
        }

    }
}
```

The indexer takes two parameters, one that indicates the row, and another that indicates the column. The indexer checks that the values are in range for the current matrix (that they are not bigger than the matrix) and then returns the value of the indexed item from the data array.

7. After the indexer, add the following code to override the **ToString** method of the **Matrix** class. You can either type this code manually, or you can use the Mod10MatrixClassToStringMethod code snippet.

```
public override string ToString()
{
    StringBuilder builder = new StringBuilder();

    // Iterate over every row in the matrix.
    for (int x = 0; x < data.GetLength(0); x++)
    {
        // Iterate over every column in the matrix.
        for (int y = 0; y < data.GetLength(1); y++)
        {
            builder.AppendFormat("{0}\t", data[x, y]);
        }
        builder.Append(Environment.NewLine);
    }

    return builder.ToString();
}
```

8. Build the solution and correct any errors.

▶ Task 3: Create a MatrixNotCompatibleException exception class

1. In Visual Studio, review the task list.
2. If it is not already open, open the Matrix.cs file.
3. Remove the comment **TODO: Add the MatrixNotCompatibleException exception class** and add a **public MatrixNotCompatibleException** class to the **MatrixOperators** namespace.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Modify the **MatrixNotCompatibleException** class to inherit from the **Exception** class.
5. Add a field of type **Matrix** called **firstMatrix** to the **MatrixNotCompatibleException** class and instantiate it to **null**.
6. Add a field of type **Matrix** called **secondMatrix** to the **MatrixNotCompatibleException** class and instantiate it to **null**.
7. Add a property of type **Matrix** called **FirstMatrix** to the **MatrixNotCompatibleException** class, and then add a **get** accessor that returns the **firstMatrix** field.
8. Add a property of type **Matrix** called **SecondMatrix** to the **MatrixNotCompatibleException** class, and then add a **get** accessor that returns the **secondMatrix** field.
9. Add the following constructors to the **MatrixNotCompatibleException** class. You can either type this code manually, or you can use the Mod10MatrixNotCompatibleExceptionClassConstructors code snippet.

```
public MatrixNotCompatibleException()
    : base()
{
}

public MatrixNotCompatibleException(string message)
    : base(message)
{
}

public MatrixNotCompatibleException(string message,
    Exception innerException)
    : base(message, innerException)
{
}

public MatrixNotCompatibleException(SerializationInfo info,
    StreamingContext context)
    : base(info, context)
{
}
```

10. Add a constructor to the **MatrixNotCompatibleException** class. The constructor should take two **Matrix** objects and a **string** object as parameters. The constructor should use the **string** object to call the **base** constructor and instantiate the **matrix1** and **matrix2** fields by using the *Matrix* parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

11. Build the solution and correct any errors.

    At the end of the exercise, your code should resemble the following code example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace MatrixOperators
{
    public class Matrix
    {
        int[,] data;

        public Matrix(int Size)
        {
            data = new int[Size, Size];
        }

        public int this[int RowIndex, int ColumnIndex]
        {
            get
            {
                if (RowIndex > data.GetUpperBound(0) ||
                    ColumnIndex > data.GetUpperBound(0))
                {
                    throw new IndexOutOfRangeException();
                }
                else
                {
                    return data[RowIndex, ColumnIndex];
                }
            }
            set
            {
                if (RowIndex > data.GetUpperBound(0) ||
                    ColumnIndex > data.GetUpperBound(0))
                {
                    throw new IndexOutOfRangeException();
                }
                else
                {
                    data[RowIndex, ColumnIndex] = value;
                }
            }
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        public override string ToString()
        {
            StringBuilder builder = new StringBuilder();

            // Iterate over every row in the matrix.
            for (int x = 0; x < data.GetLength(0); x++)
            {
                // Iterate over every column in the matrix.
                for (int y = 0; y < data.GetLength(1); y++)
                {
                    builder.AppendFormat("{0}\t", data[x, y]);
                }
                builder.Append(Environment.NewLine);
            }

            return builder.ToString();
        }
    }

    public class MatrixNotCompatibleException : Exception
    {
        Matrix firstMatrix = null;
        Matrix secondMatrix = null;

        public Matrix FirstMatrix
        {
            get
            {
                return firstMaxtrix;
            }
        }

        public Matrix SecondMatrix
        {
            get
            {
                return secondMaxtrix;
            }
        }

        public MatrixNotCompatibleException()
            : base()
        {
        }

        public MatrixNotCompatibleException(string message)
            : base(message)
        {
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        public MatrixNotCompatibleException(string message,
            Exception innerException)
            : base(message, innerException)
        {
        }

        public MatrixNotCompatibleException(SerializationInfo info,
            StreamingContext context)
            : base(info, context)
        {
        }

        public MatrixNotCompatibleException(Matrix matrix1,
            Matrix matrix2, string message)
            : base(message)
        {
            firstMatrix = matrix1;
            secondMatrix = matrix2;
        }
    }
}
```

Exercise 2: Implementing Operators for the Matrix Type

Scenario

In this exercise, you will add addition, subtraction, and multiplication operators to the **Matrix** class. The operators you add in this exercise will operate only when the two operands are matrices of the same size. You will ensure that the operands are the same size–if they are not, you will throw a **MatrixNotCompatibleException** exception.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add an addition operator to the **Matrix** class.
3. Add a subtraction operator to the **Matrix** class.
4. Add a multiplication operator to the **Matrix** class.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the starter project



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **Matrix** and **MatrixNotCompatibleException** types successfully, proceed directly to **Task 2: Add an addition operator to the Matrix class**.

- Open the Module10 solution in the E:\Labfiles\Lab 10\Lab C\Ex2\Starter folder.

▶ Task 2: Add an addition operator to the Matrix class

1. In Visual Studio, review the task list.
2. Open the Matrix.cs file.
3. Replace the comment **TODO Add an addition operator to the Matrix class** with an overload of the **+** operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.
4. Add code to the **+** operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new **MatrixNotCompatibleException** exception, by using the matrices and the message "Matrices not the same size" as parameters.
5. If both matrices are the same size, add code that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as either of the source matrices.
6. Add code to iterate over every item in the first matrix. For each item in the first matrix, calculate the sum of this item and the corresponding item in the second matrix, and store the result in the corresponding position in the **newMatrix** matrix.



Hint: Use a **for** loop to iterate over the rows in the first matrix and a nested **for** loop to iterate over the columns in each row.

7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the **+** operator.
8. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Add a subtraction operator to the Matrix class

1. In Visual Studio, review the task list.
2. If it is not already open, open the Matrix.cs file.
3. Replace the comment **TODO Add a subtraction operator to the Matrix class** with an overload of the **-** operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.
4. Add code to the **-** operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new **MatrixNotCompatibleException** exception, by using the matrices and the message "Matrices not the same size" as parameters.
5. If both matrices are the same size, add code that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as either of the source matrices.
6. Add code to iterate over every item in the first matrix. For each item in the first matrix, calculate the difference between this item and the corresponding item in the second matrix, and store the result in the corresponding position in the **newMatrix** matrix.
7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the **-** operator.
8. Build the solution and correct any errors.

▶ Task 4: Add a multiplication operator to the Matrix class

1. In Visual Studio, review the task list.
2. If it is not already open, open the Matrix.cs file.
3. Replace the comment **TODO Add a multiplication operator to the Matrix class** with an overload of the ***** operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.
4. Add code to the ***** operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new **MatrixNotCompatibleException** exception, by using the matrices and the message "Matrices not the same size" as parameters.

5. Add code to the conditional block that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as the source matrices.
6. Add code to iterate over every item in the first matrix and calculate the product of the two matrices, storing the result in the **newMatrix** matrix. Remember that to calculate each element $x_{a,b}$ in **newMatrix**, 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.
7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the * operator.
8. Build the solution and correct any errors.

Exercise 3: Testing the Operators for the Matrix Type

Scenario

In this exercise, you will use a test harness to test the operators in the **Matrix** class that you developed in the previous exercise.

The main tasks for this exercise are as follows:

1. Add the test harness to the solution.
2. Add code to test the operators in the **Matrix** class.
3. Test the matrix operators by using the test harness.

▶ Task 1: Add the test harness to the solution

The test harness application for this lab is a simple console application that is designed to test the functionality of the **Matrix** class. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class you have developed.

1. Add the test harness to the solution. The test harness is a project called Exercise3TestHarness, located in the E:\Labfiles\Lab 10\Lab C\Ex3 \Starter\Exercise3TestHarness folder.
2. Set the Exercise3TestHarness project as the startup project for the solution.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Add code to test the operators in the Matrix class

1. In Visual Studio, review the task list.
2. Open the Program.cs file.
3. Review the **Main** method.

   This method creates two 3×3 square matrices called **matrix1** and **matrix2** and populates them with sample data. The method then displays their contents to the console by using the **ToString** method.
4. Remove the TODO comment.
5. Add a statement to write the message "Matrix 1 + Matrix 2:" to the console.
6. Add a statement to create a new **Matrix** object called **matrix3** and populate it with the sum of the **matrix1** and **matrix2** objects.
7. Add code to write the contents of the **matrix3** matrix to the console, followed by a blank line.
8. Add a statement to write the message "Matrix 1 - Matrix 2:" to the console.
9. Add code to create a new **Matrix** object called **matrix4** and populate it with the difference between the **matrix1** and **matrix2** objects (subtract **matrix2** from **matrix1**).
10. Add code to write the contents of the **matrix4** matrix to the console, followed by a blank line.
11. Add a statement to write the message "Matrix 1 × Matrix 2:" to the console.
12. Add code to create a new **Matrix** object called **matrix5** and populate it with the product of the **matrix1** and **matrix2** objects.
13. Add code to write the contents of the **matrix5** matrix to the console, followed by a blank line.
14. Build the solution and correct any errors.

▶ Task 3: Test the matrix operators by using the test harness

1. Start the Exercise3TestHarness application.
2. Verify that the output from the console appears correctly. The output should resemble the following.

```
Matrix 1:
1     2       3
4     5       6
7     8       9

Matrix 2:
9     8       7
6     5       4
3     2       1

Matrix 1 + 2:
10    10      10
10    10      10
10    10      10

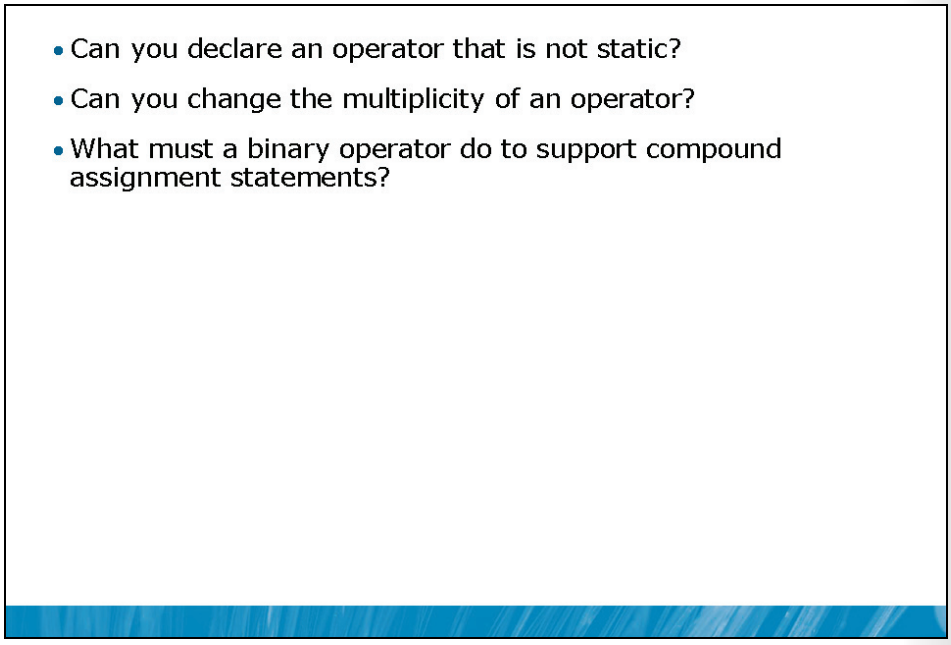
Matrix 1 - 2:
-8    -6      -4
-2    0       2
4     6       8

Matrix 1 x 2:
30    24      18
84    69      54
138   114     90
```

3. Close the console window.
4. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

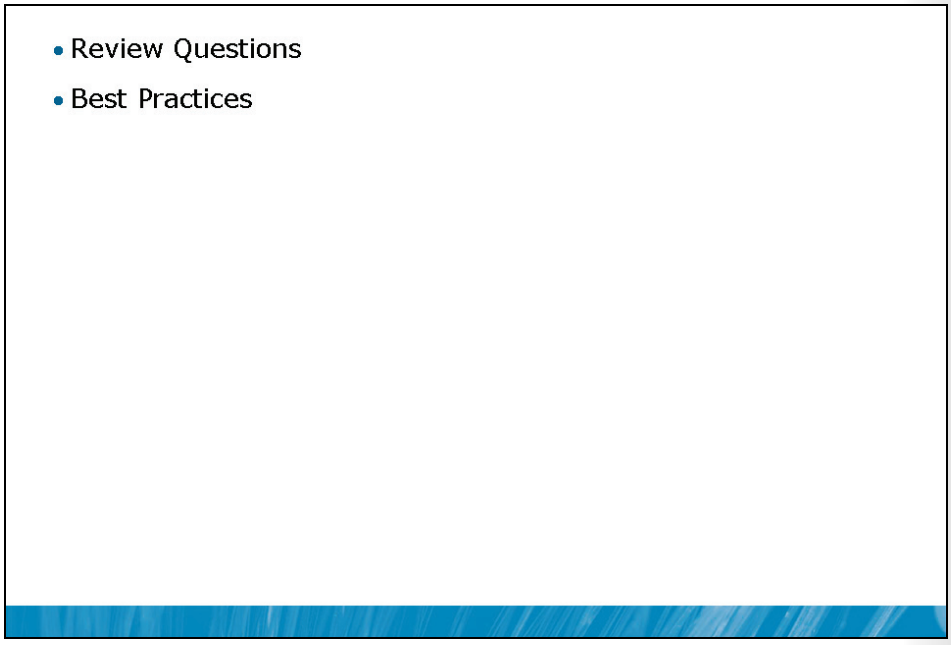


Review Questions

1. Can you declare an operator that is not static?
2. Can you change the multiplicity of an operator?
3. What must a binary operator do to support compound assignment statements?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. If you are developing a new type and must expose data, how can you expose the data as a property with minimal extra effort?
2. You must develop an application to represent a set of data. You must expose individual members of the data to consuming classes. How can you expose individual members in a dataset to consuming classes, and are there any disadvantages to the approach?
3. You have overloaded the == operator in a type you are developing. As required by the compiler, and to comply with best practices, you are also going to implement the != operator. Should you implement the != operator from scratch, or should you use the == operator that you have already defined and negate the result?

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices Related to Properties

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

- Use properties only when a property is appropriate, but do not expose data unnecessarily.
- Use automatic properties instead of making a field public unless there is a very good reason not to.

Best Practices Related to Indexers

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

- Use an indexer to access a data member that is part of a set. An indexer is not a method: if you are writing too much code in an indexer, consider whether it would be better implemented as a method.

Best Practices Related to Operators

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

- Implement symmetric operators for commutable operations.
- Do not modify the value of operands in an operator.
- Define only meaningful operators.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 11

Decoupling Methods and Handling Events

Contents:

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

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



In this course, you have seen how you can call a method to perform an operation by using the name of the method. Sometimes you will need to call a method determined dynamically at run time, or you may want to call code that is not available when you develop your type. For example, you may need to call code that other developers have written for applications that consume your type. You can decouple an operation from the method that implements it and write code to determine at run time which method should implement the operation.

This module explains how to decouple an operation from the method that implements it and 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.

Objectives

After completing this module, you will be able to:

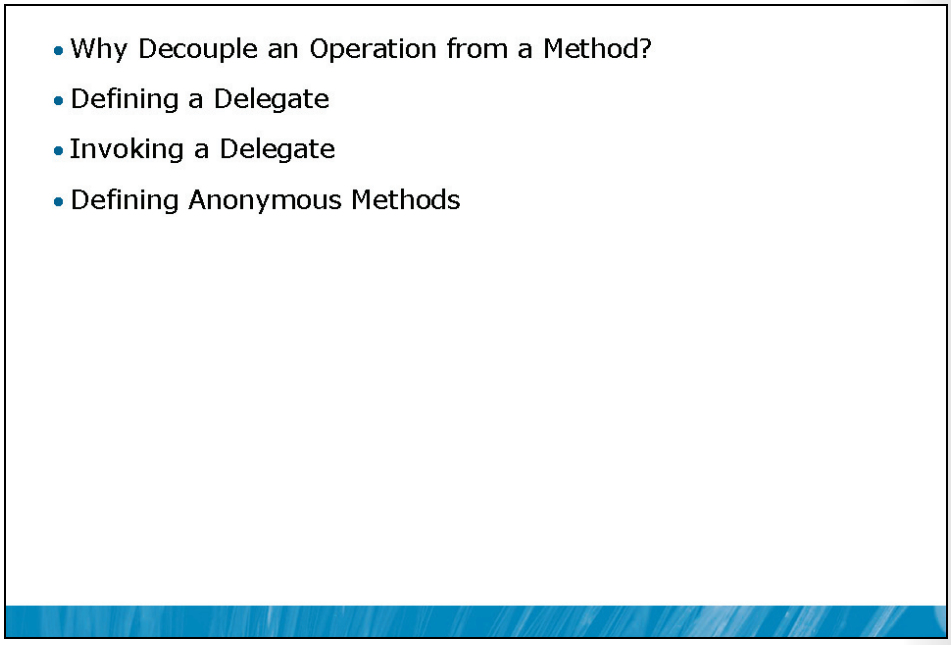
- Describe the purpose of delegates, and explain how to use a delegate to decouple an operation from the implementing method.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain the purpose of lambda expressions, and describe how to use a lambda expression to define an anonymous method.
- Explain the purpose of events, and describe how to use events to report that something significant has happened in a type that other parts of the application need to be aware of.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Declaring and Using Delegates



You can use delegates to decouple an operation from the methods that implement the operation. This lesson explains how to use delegates as an indirect mechanism to invoke one or more methods and how to use anonymous methods to implement a decoupled operation.

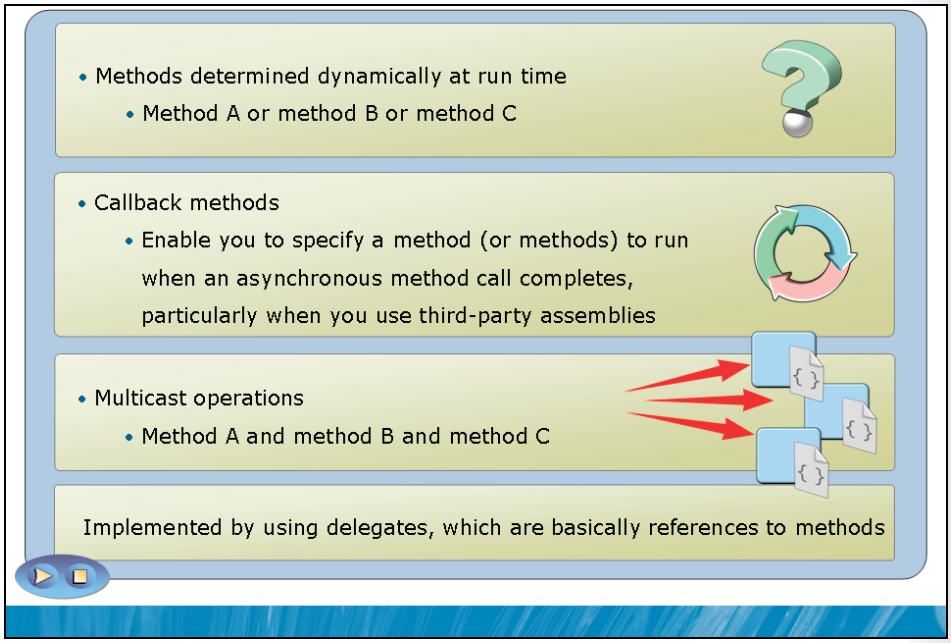
Objectives

After completing this lesson, you will be able to:

- Describe why you may want to decouple an operation from the method that implements it, and explain how you decouple an operation in Microsoft® Visual C#®.
- Explain how to define a delegate.
- Invoke methods through a delegate synchronously and asynchronously.
- Describe and define anonymous methods.

MCT USE ONLY. STUDENT USE PROHIBITED

Why Decouple an Operation from a Method?



Key Points

When you develop applications by using Visual C#, in most cases, you invoke methods explicitly, by name. The logical operation and the physical implementation of that operation in the form of a method are tightly coupled. However, there are times when this approach is not suitable. For example, you may develop a framework that can invoke different methods to perform an operation depending on criteria determined dynamically when the application runs. One way to implement this functionality is to use a series of **if** or **switch** statements, but this is a static approach and depends on the various methods being available when the code is written. A more extensible approach is to use delegates.

Another scenario concerns callback methods. Third-party vendors who provide assemblies with methods that you can invoke asynchronously frequently enable you to specify a method in your code to run when their method has completed. These third-party vendors are unlikely to know in advance the names of all of your methods, so instead they can provide a delegate that you can associate with one or more of your methods.

MCT USE ONLY. STUDENT USE PROHIBITED

To put it simply, a delegate is a reference to a method. A delegate defines the signature of the method; for example, a delegate may specify that a method takes two *string* parameters and returns an integer. At run time, you can associate a delegate with any method that matches this signature. To call the method, you invoke the delegate.

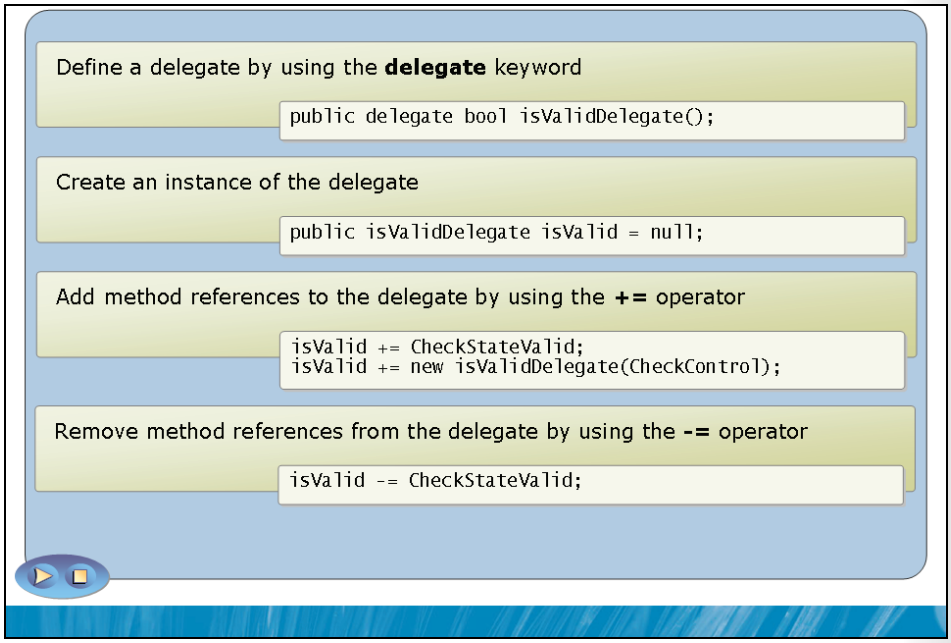
Multicast Operations

Another common use for decoupling an operation from a method is multicast operations. If an operation is multicast, several methods can implement the same operation. You can add references to all of the implementing methods to the delegate, and when the delegate is invoked, the runtime will invoke each method in turn. When you use a multicast operation, the implementing methods are called in sequence according to when they were added to the delegate. However, note that if one of the methods throws an unhandled exception, there is no guarantee that the application will call subsequent methods.

Question: If you develop a class library and want to enable developers who write consuming applications to run code after an asynchronous method call completes, how can you provide this functionality?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining a Delegate



Key Points

A delegate is a reference to one or more methods, so when you define a delegate, you must specify the signature of the delegate. Methods that the delegate refers to must have matching signatures. You define a delegate by using the **delegate** keyword. When you define a delegate, you create a new type; this is the same as using the **class** or **struct** keywords. You can specify an access modifier for a delegate in the same way as you can for any type.

Note: When you use the **delegate** keyword to define a new delegate type, the Visual C# compiler converts the new delegate type to inherit from the **MulticastDelegate** class. You cannot explicitly inherit from the **MulticastDelegate** class—you must use the **delegate** keyword.

The following code example shows a simple delegate.

```
public delegate bool isValidDelegate();
```

MCT USE ONLY. STUDENT USE PROHIBITED

The preceding code example declares a delegate named **IsValidDelegate**. The delegate was made visible to other classes by using the **public** access modifier, and it returns a **bool** value and takes no parameters. To use the **IsValidDelegate** delegate, you must initialize it. The following code example illustrates this.

```
public isValidDelegate isValid = null;
```

After you have initialized the delegate, you can refer it to one or more methods. You can use the compound assignment operators **+=** and **-=** to add or remove method references from a delegate. The following code example shows how to add method references to the **isValid** instance of the **isValidDelegate** delegate class.

```
// An instance method that happens to appear in the same class.
void bool CheckStateValid()
{
    ...
}

// An instance method that happens to appear in the same class.
public void bool CheckControl()
{
    ...
}

...

// Adding a method reference by using the += operator.
// Implicitly calls the delegate constructor.
isValid += CheckStateValid;

// Adding a method reference by using the += operator.
// Explicitly calls the delegate constructor.
isValid += new isValidDelegate(CheckControl);
```

The following code example shows how to remove a method reference from the **isValid** delegate instance.

```
// Removing a method reference by using the -= operator.
isValid -= CheckStateValid;
```

Question: Which of the following are valid scopes to define a delegate: the namespace scope, the class scope, or the method scope?

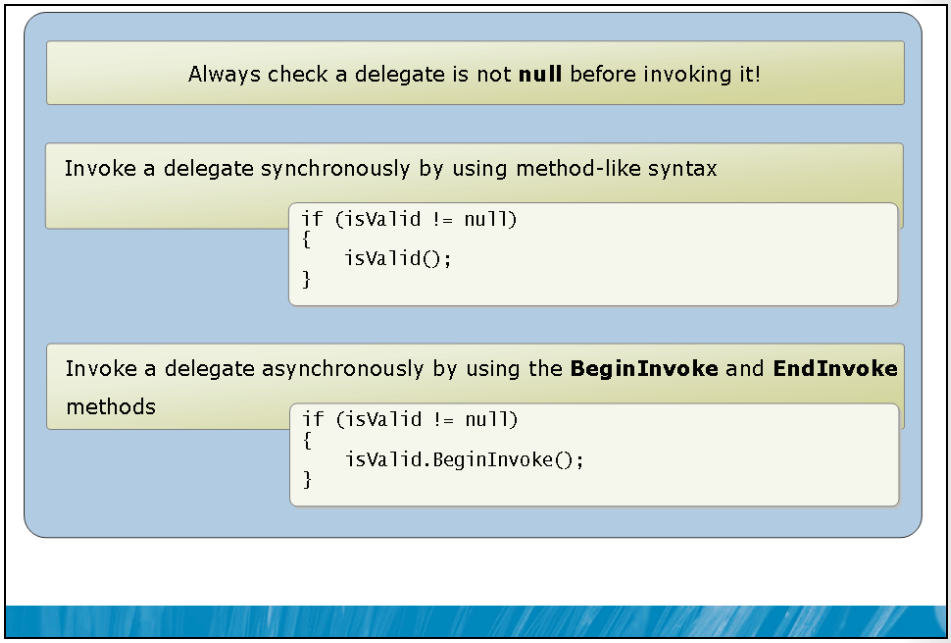
MCT USE ONLY. STUDENT USE PROHIBITED

Additional Reading

For more information about delegates, see the Delegates (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192955.

MCT USE ONLY. STUDENT USE PROHIBITED

Invoking a Delegate



Key Points

After you define a delegate and create an instance of that delegate, you can invoke it in code.

You can invoke a delegate in the same way as you call a method, by using the delegate instance name followed by any parameters in parentheses. There is one important difference between invoking a delegate and calling a method: a delegate may not reference any methods and may therefore be null. You should always check that a delegate is not null before you invoke it. When a delegate does not reference any methods, it will resolve to null; when the delegate has references to one or more methods, it will not resolve to null.

The following code example shows how to invoke the **isValid** delegate.

```
if(isValid != null)
{
    isValid();
}
```

MCT USE ONLY. STUDENT USE PROHIBITED



Warning: When you invoke a delegate synchronously, if it is multicast, each of the implementing methods is called in order. If your delegate has a return type, the value from the last implementing method is returned to the invoking application. The return values from any other implementing methods are ignored.

Invoking a Delegate Asynchronously

In addition to invoking a delegate synchronously, you can invoke a delegate asynchronously. If you want to invoke a delegate asynchronously, it must reference only a single method; you cannot invoke a multicast delegate asynchronously.

The **delegate** type supports the asynchronous programming design pattern through the **BeginInvoke** and **EndInvoke** methods. An application can call the **BeginInvoke** method to run the method that a delegate references asynchronously and use the **EndInvoke** method to capture any data that the delegated method returns.

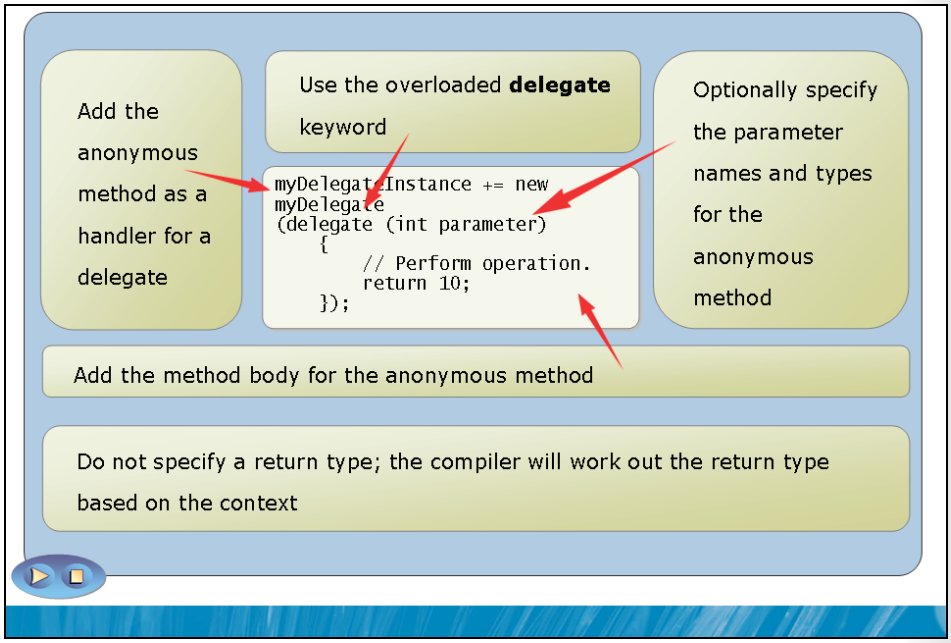
Question: Why should you always check that a delegate is not null before you invoke it?

Additional Reading

For more information about asynchronous programming, see the Asynchronous Programming Overview page at http://go.microsoft.com/fwlink/?LinkId=192956.

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Anonymous Methods



Key Points

When you use a delegate, it must reference a method. However, you may frequently find that all you want to do is refer a delegate to a block of code, and it adds overhead to your programming effort to create a method to encompass this block of code (you must think up a meaningful name for the method). To deal with situations such as this, Visual C# enables you to define anonymous methods. An anonymous method is a method that does not have a name, just a type, a parameter list, and a method body. You can only use an anonymous method with a delegate; you can only invoke the method through the delegate.

Defining an Anonymous Method

To define an anonymous method, you use the **delegate** keyword. After the **delegate** keyword, you can optionally provide the list of parameters in parentheses. If you provide the list of parameters, you must provide the types for those parameters. Finally, you provide the method body. You must enclose the method body in braces like a named method.

The following code example shows how to define an anonymous method.

MCT USE ONLY. STUDENT USE PROHIBITED

```
// Define a delegate type.
delegate int myDelegate(int number);

// Create an instance of the delegate type.
myDelegate myDelegateInstance = null;

public void addAnonymousMethodsToDelegate()
{
    // Add an anonymous method to the delegate.
    // Do not specify any parameters.
    myDelegateInstance += new myDelegate( delegate
    {
        // Perform operation.
        return 5;
    });

    // Add an anonymous method to the delegate.
    // Specify parameters; the parameters must match
    // the signature of the delegate.
    myDelegateInstance += new myDelegate(delegate (int parameter)
    {
        // Perform operation.
        return 10;
    });

    // Invoke the delegate.
    int returnedValue = myDelegateInstance(2);

    // returnedValue = 10 (as second method is called last).
}
```

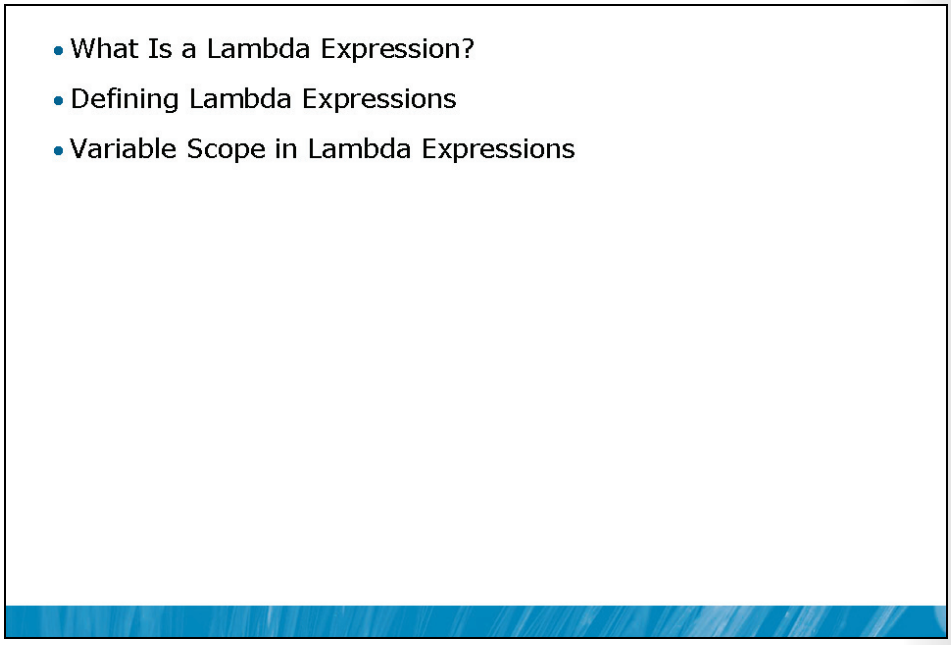
Question: If you specify parameters for an anonymous method, do you need to specify the types of those parameters?

Additional Reading

For more information about anonymous methods, see the Anonymous Methods (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192961.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Using Lambda Expressions



Lambda expressions provide a technique for implementing anonymous methods that is more succinct than you can achieve by using delegates. This lesson explains how to use lambda expressions. It also describes some of the advantages of lambda expressions compared with anonymous methods.

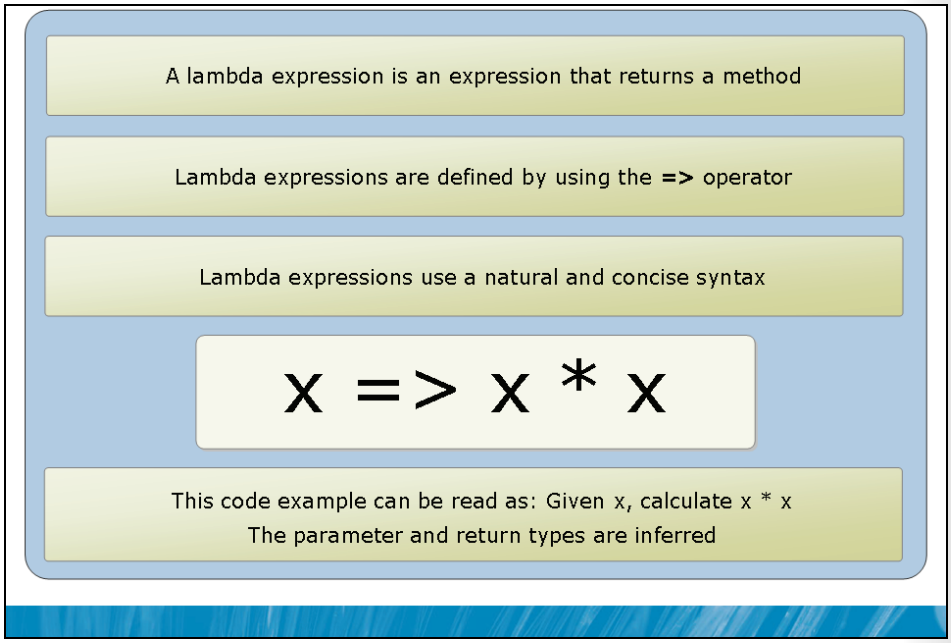
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of lambda expressions, and define lambda expressions.
- Define lambda expressions that take parameters.
- Describe the scope of variables in a lambda expression.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Lambda Expression?



Key Points

A lambda expression is an expression that returns a method. These expressions evolved from the world of functional programming, but they are particularly useful for defining anonymous but strongly typed methods. They have a sound mathematical foundation, and they are widely used throughout Visual C#, especially when defining Language-Integrated Query (LINQ) expressions.

Note: LINQ is described in detail in a later module.

A lambda expression consists of a set of parameters and a body. The body defines a function that may return a value; if it does, the Visual C# compiler infers the return type from the definition of the method body.

Defining a Lambda Expression

You define a lambda expression by using the => operator. The following code example shows a very simple lambda expression.

MCT USE ONLY. STUDENT USE PROHIBITED

```
x => x * x
```

You can read this lambda expression as "Given x, calculate x * x." The type of x does not have to be specified, because it will be inferred when the lambda expression is used. In addition, the return type is also inferred.

You can reference the lambda expression from a delegate, as the following code example shows.

```
delegate int MyDelegate(int a);
...
MyDeletage myDelegateInstance = null;
myDelegateInstance += new MyDelegate(x => x * x);
```

In this example, the type of x and the return type in the lambda expression are determined by the delegate, which takes an integer parameter and returns an integer. You can abbreviate the statement that assigns the lambda expression to **myDelegateInstance**, as the following code example shows.

```
delegate int MyDelegate(int a);
...
MyDeletage myDelegateInstance = null;
myDelegateInstance += x => x * x;
```

This syntax is natural and concise.

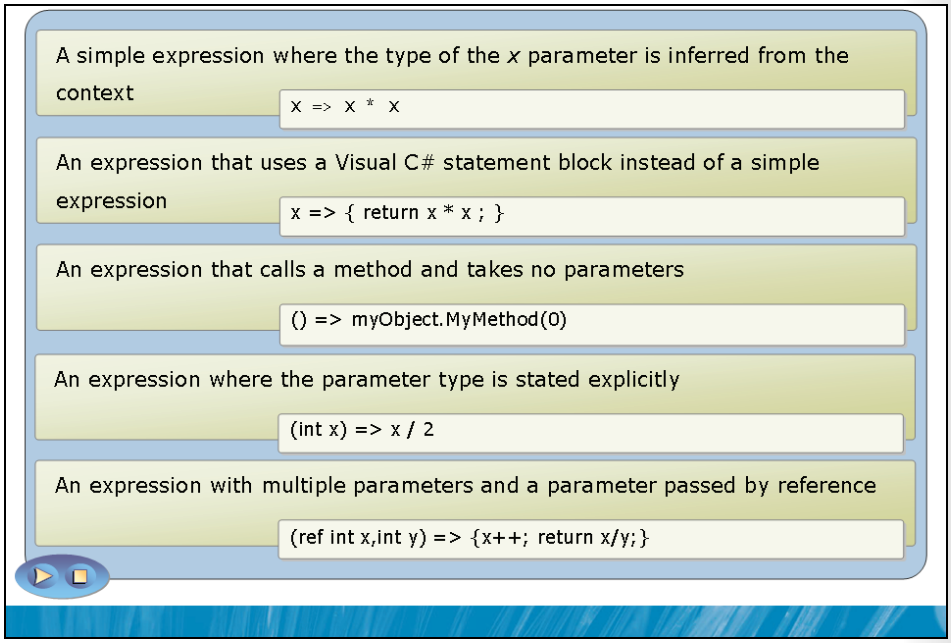
When you invoke the delegate, you specify values for any parameters that the lambda expression takes and the body of the lambda expression runs. You can capture any return value in the same way as calling a regular method. The following code example shows how to invoke a delegate based on the simple lambda expression and delegate shown previously.

```
Console.WriteLine(myDelegateInstance(10)); // Displays the value 100
```

Question: Can you define a lambda expression without using a delegate to reference the expression?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Lambda Expressions



Key Points

Lambda expressions can take several subtly different forms. Lambda expressions were originally part of a mathematical notation called the Lambda Calculus that provides a notation for describing functions. Although the Visual C# language has extended the syntax and semantics of the Lambda Calculus in its implementation of lambda expressions, many of the original mathematical principles still apply.

The body of a lambda expression can be a simple expression, as shown in the previous topic, or it can be a block of Visual C# code that defines a method body enclosed in braces. If you define a method body, you can make use of any Visual C# programming constructs.

Note: Although you can use any Visual C# code you like in a lambda expression, it is not considered good practice to define code that modifies data that your application uses elsewhere. Purists refer to this as programming with side effects, and these side effects can often be the cause of subtle bugs that are difficult to track down.

MCT USE ONLY. STUDENT USE PROHIBITED

A lambda expression can take more than one parameter, in which case you specify a parameter list that is enclosed in parentheses, as the following code example shows.

```
delegate int AddDelegate(int a, int b)
...
AddDelegate myAddDelegate = null;
myAddDelegate += (x, y) => x + y;
```

A lambda expression can also take zero parameters. In this case, specify an empty parameter list. The following code examples show many of the different forms of lambda expression available in Visual C#.

```
// A simple expression that returns the square of its parameter.
// The type of parameter x is inferred from the context.
x => x * x


// Semantically the same as the preceding
// expression, but using a Visual C# statement block as
// a body rather than a simple expression.
x => { return x * x ; }

// A simple expression that returns the value of the
// parameter divided by 2.
// The type of parameter x is stated explicitly.
(int x) => x / 2

// Calling a method.
// The expression takes no parameters.
// The expression may or may not
// return a value.
() => myObject.MyMethod(0)

// Multiple parameters; the compiler
// infers the parameter types.
// The parameter x is passed by value, so
// the effect of the ++ operation is
// local to the expression.
(x, y) => { x++; return x / y; }

// Multiple parameters
// with explicit types.
// Parameter x is passed by
// reference, so the effect of
// the ++ operation is permanent.
(ref int x, int y) { x++; return x / y; }
```

MCT USE ONLY. STUDENT USE PROHIBITED

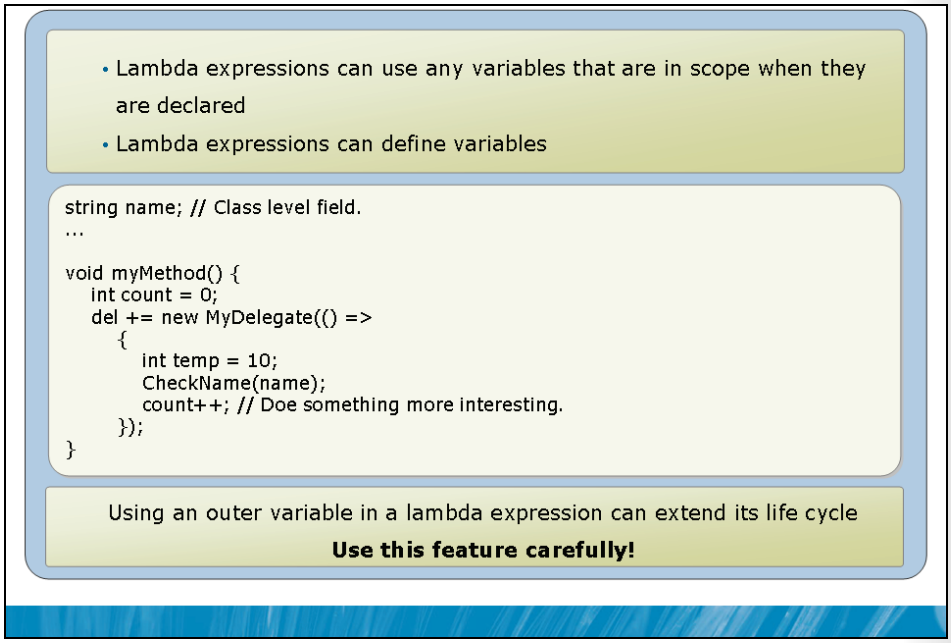
Question: When can you omit the parentheses that surround the parameters in a lambda expression?

Additional Reading

For more information about lambda expressions, see the Lambda Expressions (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192957.

MCT USE ONLY. STUDENT USE PROHIBITED

Variable Scope in Lambda Expressions



Key Points

When you define a lambda expression, you can define variables in the body of that expression. The scope of these variables is the lambda expression. When the lambda expression completes, the variable goes out of scope. Code that does not appear in the lambda expression cannot access variables that are defined in that expression.

When you define a lambda expression, you can also access variables defined outside that expression. However, the lifetime of these variables is extended until the lambda expression itself goes out of scope, which may have an impact on garbage collection.

The following code example shows how a lambda expression that is assigned to a delegate in a method can make use of variables that are defined in that method.

```
MyDelegate del = null;
...
void myMethod()
{
    int count = 0;
```

MCT USE ONLY. STUDENT USE PROHIBITED

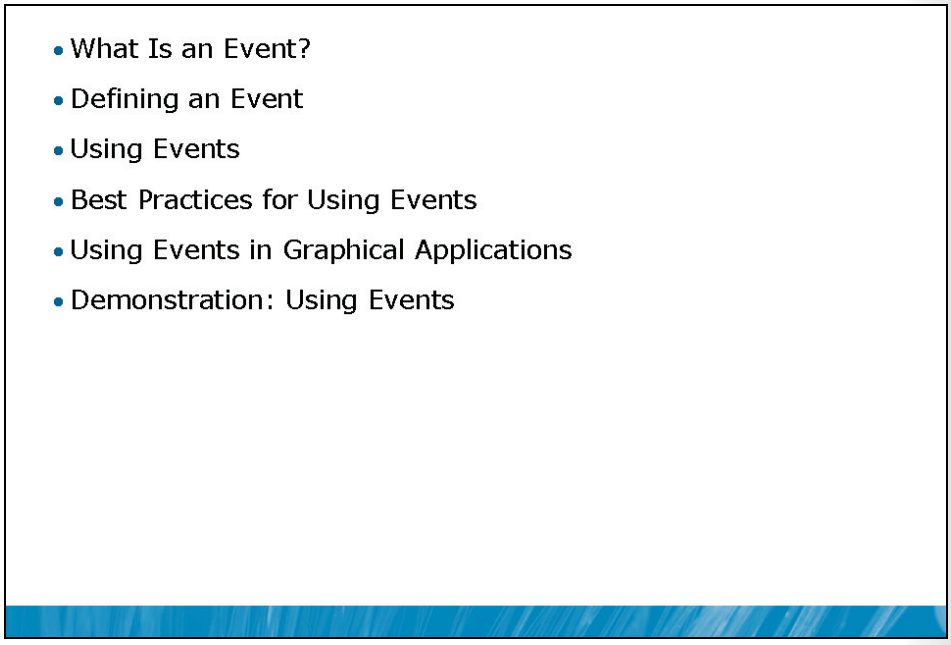
```
    del += new MyDelegate(() =>
        {
            count++;
            // Perform operation using count variable.
            ...
        });
}
```

In the preceding code example, if the **myMethod** method did not instantiate the **del** delegate, when the method completed normally, the count variable would become eligible for garbage collection. However, if any references to the **del** delegate exist when the method completes, the count variable will remain. When all references to the **del** delegate are removed, the garbage collector will mark the count variable for collection. In this case, the data is an integer field that uses very few resources; however, a lambda expression can use any object of any type that is in scope when the expression is defined. If the object is large and uses a lot of resources, this may not be desirable or intended. You should consider carefully whether it is strictly necessary before you reference variables that are defined outside the scope of a lambda expression.

Question: If you reference an open database connection in a lambda expression, which would normally go out of scope at the end of the method that defines the expression, what happens to the database connection when the method completes?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3
Handling Events



Events enable you to indicate that something significant has happened in your application and other elements in your application may need to be informed of this occurrence. For example, when a user clicks a button in a graphical application, you typically want to run a block of code that performs some action associated with the button click, such as saving data to a file. This lesson explains how to use events and describes some of the best practices you should follow when you implement events in your applications.

Objectives

After completing this lesson, you will be able to:

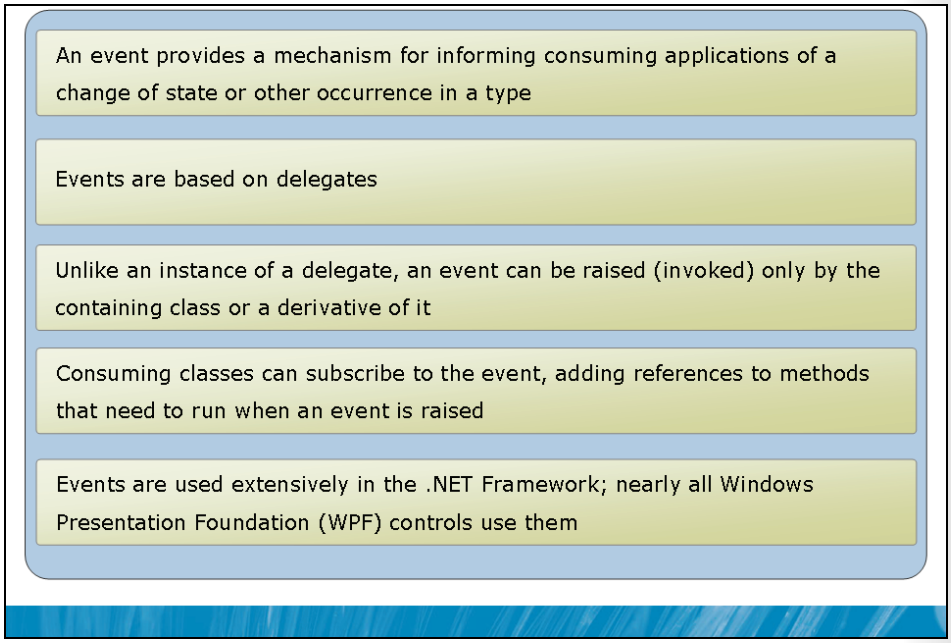
- Describe the purpose of an event.
- Define an event.
- Raise and subscribe to an event.

MCT USE ONLY. STUDENT USE PROHIBITED

- Explain some of the best practices associated with using events.
- Describe the use of events in graphical applications.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Event?



Key Points

In Visual C#, events are very similar to delegates. In fact, events are based on delegates, although semantically, events have a subtly different purpose.

A type uses an event to indicate a significant occurrence and arrange for a delegate to be called. Whereas any object that has access to a delegate can invoke that delegate, only the type that defines an event can trigger that event.

When a type defines an event, other types can define a method that matches the signature of the delegate that is associated with the event. These other types can then subscribe to the event and specify that this method should be run when the event is raised.

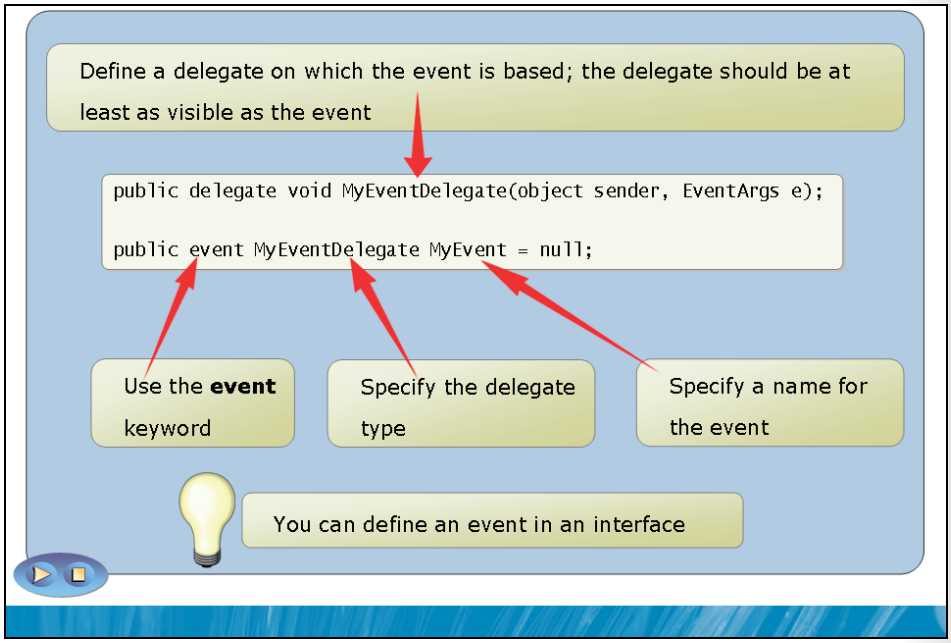
Classes in the Microsoft .NET Framework use events extensively, including nearly all Windows® Presentation Foundation (WPF) controls. You should use events wherever you must develop a type that needs to inform consuming classes about a change in state.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What is the difference between a publicly exposed instance of a delegate and a publicly exposed event?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining an Event



Key Points

Events are based on delegates, so before you can define an event, you must define a delegate. There are conventions that specify the standard signature for an event delegate; you should try to follow the conventions wherever possible. The standard convention for defining an event delegate is that the delegate should not return a value and should take two parameters. The first parameter should be of type **object**, named *sender*. The second parameter should be a derivative from the **EventArgs** class from the **System** namespace, named *e*. If your event does not need to pass any event arguments when the event is invoked, you can use the **EventArgs** class directly.

The following code example shows a typical delegate for use with an event.

```
public delegate void MyEventDelegate(object sender, EventArgs e);
```

To define an event, you use the **event** keyword and specify the delegate. It is common practice to initialize the event to the **null** value because this enables the type to determine whether the event has any subscribers prior to raising it (when a method subscribes to the event, the event will no longer be null). The following

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

code example shows how to define an event named **MyEvent** based on the **MyEventDelegate** delegate.

```
public event MyEventDelegate MyEvent = null;
```



Note: A delegate is a type and can be at the namespace or class level; however, an event is a class member and can only be at the class level. An event at the class level can use a delegate at any level, including delegates that exist in other assemblies. This is common practice if you want to use some of the delegates defined in the .NET Framework.

When you define an event, you should normally make the event **public** so that consuming classes can see the event. When you define an event, remember that the delegate must be at least as visible as the event—otherwise, consuming classes will not be able to create instances of the delegate and therefore will not be able to subscribe to your event.



Note: Unlike a delegate, you can define an event in an interface.

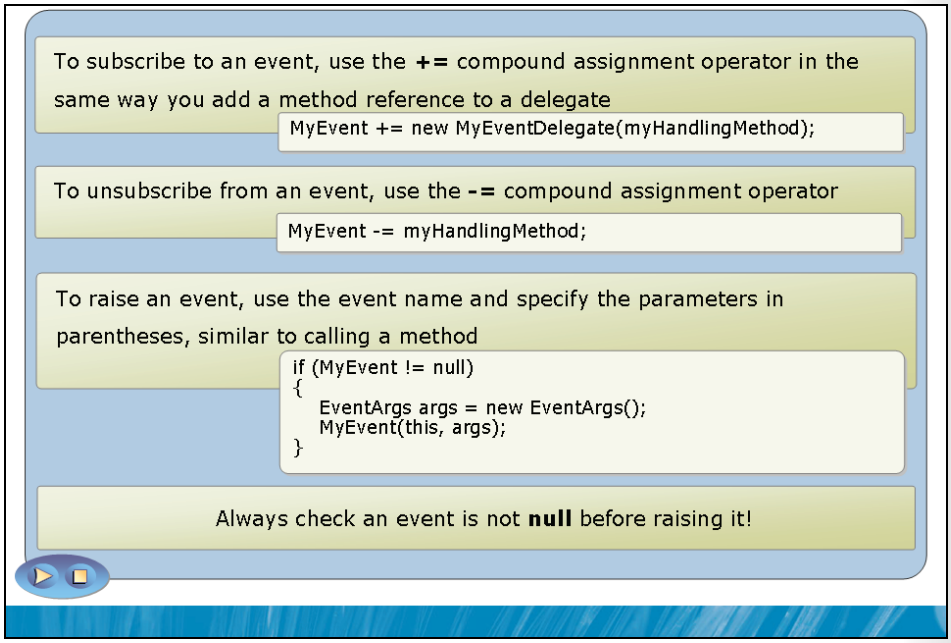
Question: What is the result of defining a **public** event based on a **private** delegate?

Additional Reading

For more information about events, see the Events (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192958.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Events



Key Points

After you have defined an event, you can use the event in your application. You can subscribe to the event in consuming types and applications and raise the event in your type.

Subscribing to an Event

To subscribe to an event, you use the **+=** compound assignment operator in the same way that you add a method reference to a delegate.

The following code example shows how to add a subscriber to the **MyEvent** event.

```
MyEvent += new MyEventDelegate(myHandlingMethod);
```

In addition to subscribing to an event, you can unsubscribe from an event. To unsubscribe from an event, you use the **-=** compound assignment operator.

The following code example shows how to remove a subscriber from the **MyEvent** event.

MCT USE ONLY. STUDENT USE PROHIBITED

```
MyEvent -= myHandlingMethod;
```

Raising an Event

To raise an event, you use the event name and specify the parameters in parentheses, similar to calling a method. When you raise an event, you should first check that the event is not null. If there are no subscribers, you should not attempt to raise the event. If you attempt to raise an event that has no subscribers, the code will throw a **NullReferenceException** exception.

The following code example shows how to raise an event.

```
if (MyEvent != null)
{
    EventArgs args = new EventArgs();
    MyEvent(this, args);
}
```



Note: Best practices state that you should use an **On** method when you raise an event. For more information about **On** methods, see "Best Practices for Using Events" later in this module.

Question: Can consuming classes raise an event?

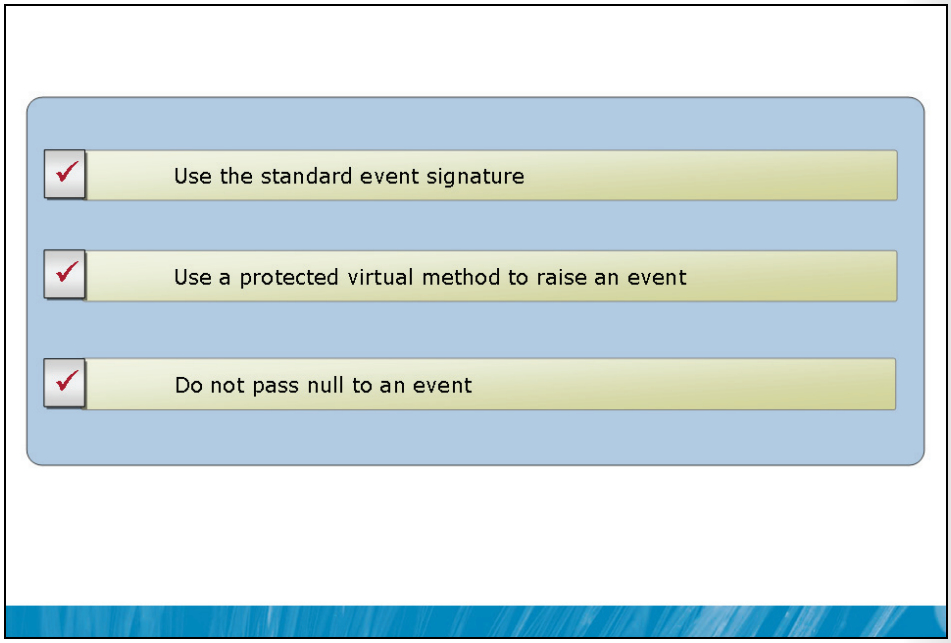
Additional Reading

For more information about how to publish events, see the How to: Publish Events that Conform to .NET Framework Guidelines (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192959.

For more information about how to hook up events, see the How to: Hook Up Events By Using IntelliSense (C#) page at http://go.microsoft.com/fwlink/?LinkId=192962.

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices for Using Events



Key Points

When you develop a type that uses events, you should adhere to best practices as closely as possible. Adhering to best practices will help to ensure that other developers will require minimal additional knowledge to create types that consume your type. The best practices also help to ensure that any changes that you make in the future will not cause breaking changes to any applications that already use your type.

Best Practice: Use the Standard Event Signature

When you define the delegate for an event, you should use the standard event signature. An event should specify **void** as the return type and also specify two parameters: the first parameter should be an **object** called *sender* and the second parameter should be a type that derives from the **EventArgs** class called *e*. You should define your own class that inherits from the **EventArgs** class instead of using the **EventArgs** class, unless you are sure that you will never need to pass any information in the event arguments in the future—in which case, you can use the **EventArgs** class. When you develop a class to hold the event arguments for your type, it is normal to name the class *MyName***EventArgs**, where you replace *MyName*

MCT USE ONLY. STUDENT USE PROHIBITED

with either the name of the event or a name related to the type of the arguments that are passed. Naming the type based on the data it represents rather than the event for which it is being developed can lead to reusable types and is therefore often better.

Best Practice: Use a Protected Virtual Method to Raise an Event

When you define an event in your type, define a protected virtual method that raises the event. The method should be named according to the name of the event, but prefixed with the word On; for example, the method to raise the **MyEvent** event would be named **OnMyEvent**. The **On** event should take the same parameters as the event and contain logic; for example, it should check that the event is not null before it raises the event. When you need to raise the event in your type, you can then use the **On** method instead of raising the method directly. This reduces code duplication because you no longer need to check that the event is not null every time you raise the event.

The **On** method should be protected and virtual. This enables any classes that derive from your type to modify the process when an event is raised; for example, by adding validation to a property in the event arguments and throwing an exception if the value is not valid. By using a protected virtual method to raise the exception if the argument does not pass validation checks you can modify the method in any child classes if the validation rules change. If you use an **On** method, the new validation will apply to occurrences where you raise the event in your type and any occurrences where the event is raised in the child class.

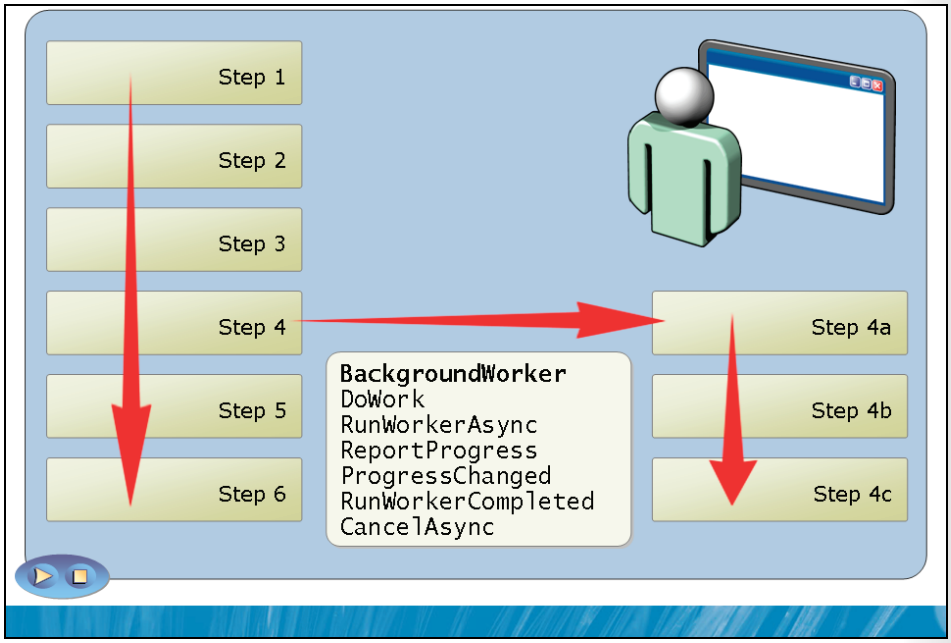
Best Practice: Do Not Pass Null Values to an Event

When you raise an event, it can be tempting to use null as a parameter. You should avoid passing null and always pass instantiated objects as parameters. For the *sender* parameter, you should normally use the **this** keyword. You should not omit the *sender* parameter because consuming applications will often depend on this parameter.

Question: What is the naming convention for methods that encapsulate the logic associated with raising an event?

MCT USE ONLY. STUDENT USE PROHIBITED

Using Events in Graphical Applications



Key Points

Graphical applications use a large number of events. Graphical applications typically involve a lot of user interaction; when users click a button, they expect the application to respond immediately. When you develop a graphical application, you can use events to respond immediately to users when they interact with the interface. For example, a button exposes a **Click** event, and you can add event handlers by subscribing to the **Click** event; in the event handler, you add code to handle the click event and perform the appropriate logic.

Threading in .NET Framework Applications

When you write a simple application, you use a single thread. All of your code runs sequentially on that thread. If a process takes a long time to complete, the application will pause while it waits for that process to finish. You will often need to develop applications that do not hang while long-running processes complete. In the .NET Framework, you solve this problem by using threads. You can use several threads in your applications. Each thread can run concurrently with other threads, which means that the application no longer needs to wait while a long-

MCT USE ONLY. STUDENT USE PROHIBITED

running process completes. The long-running process can be performed on a different thread, and the application can continue with other tasks.

The User Interface and Threading

Graphical applications in the .NET Framework use threading like every other application. In the .NET Framework, all interaction with the user interface (UI) must be performed on a single thread, often referred to as the UI thread. The UI thread is the only thread that can update the UI or respond to events that the UI raises. You can use other threads with the UI thread to ensure that your graphical application remains responsive; however, you must marshal data between the threads to ensure thread safety. You can manually implement this logic, or you can use the **BackgroundWorker** thread.

Using the BackgroundWorker Class to Implement Multithreading

The **BackgroundWorker** class enables you to run code on another thread. The **BackgroundWorker** class includes events that indicate the progress of the thread. A UI can subscribe to these events and update the screen by using the UI thread. This enables you to build a UI that remains responsive and up to date.



Note: Although the **BackgroundWorker** class works well with UIs, you are not restricted to using it in this environment. You can employ a **BackgroundWork** object whenever you need to run code on a separate thread and report the progress of that code.

To use the **BackgroundWorker** class, perform the following steps:

1. Create an instance of the **BackgroundWorker** class.
2. Add a handler for the **DoWork** event. The event handler for the **DoWork** event should contain the code that you need to perform on a separate thread. Optionally, you can use the **ReportProgress** method of the **BackgroundWorker** object to report the status of the operation.
3. Optionally, add a handler for the **ProgressChanged** event. The **ProgressChanged** event handler runs on the UI thread so you can add code to update the UI. The **ProgressChanged** event is raised whenever you call the **ReportProgress** method.
4. Optionally, add a handler for the **RunWorkerCompleted** event. This event is raised when the method you associated with the **DoWork** event completes. The handler for the **RunWorkerCompleted** event runs on the UI thread, so it can update the UI.

MCT USE ONLY. STUDENT USE PROHIBITED

5. If you use the **ProgressChanged** method, set the **WorkerReportsProgress** property to **true**.
6. Call the **RunWorkerAsync** method of the **BackgroundWorker** object. This method raises the **DoWork** event and starts the background thread running by using the method that you specified to subscribe to this event.

The **BackgroundWorker** class also provides a method called **CancelAsync**. This method sets a Boolean property called **CancellationPending** to **true** and requests that the method running as a result of the **DoWork** event is terminated. The method that the **DoWork** event runs should periodically check the **CancellationPending** property, and if it is **true**, the method should finish.

The following code example shows how you can use the **BackgroundWorker** class in the **Click** event handler of a button to run a long-running process without freezing the UI. The example also shows how you can respond to the **ProgressChanged** event to keep the UI of the application updated.

```
using System.ComponentModel;
...

private void button1_Click(object sender, RoutedEventArgs e)
{

    BackgroundWorker bw = new BackgroundWorker();

    // Alternatively you can omit the parameter types.

    bw.DoWork += ((object doWorkSender, DoWorkEventArgs doWorkArgs) =>
        {
            bw.ReportProgress(10);
            // Perform long running process.
            // Use the doWorkArgs.Argument property.
            bw.ReportProgress(50);
            // Continue long running process.
            bw.ReportProgress(90);
        });

    bw.ProgressChanged += ((object progressChangedSender,
        ProgressChangedEventArgs progressChangedArgs) =>
        {
            // Update label in UI with progress.
            statusBox.Content =
                progressChangedArgs.ProgressPercentage.ToString();
        });

    bw.RunWorkerCompleted += ((object runWorkerCompletedSender,
        RunWorkerCompletedEventArgs runWorkerCompletedArgs) =>
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        {
            // Alternatively update the user interface.

            MessageBox.Show("Complete");
        });

    bw.WorkerReportsProgress = true;
    bw.RunWorkerAsync();
}
```

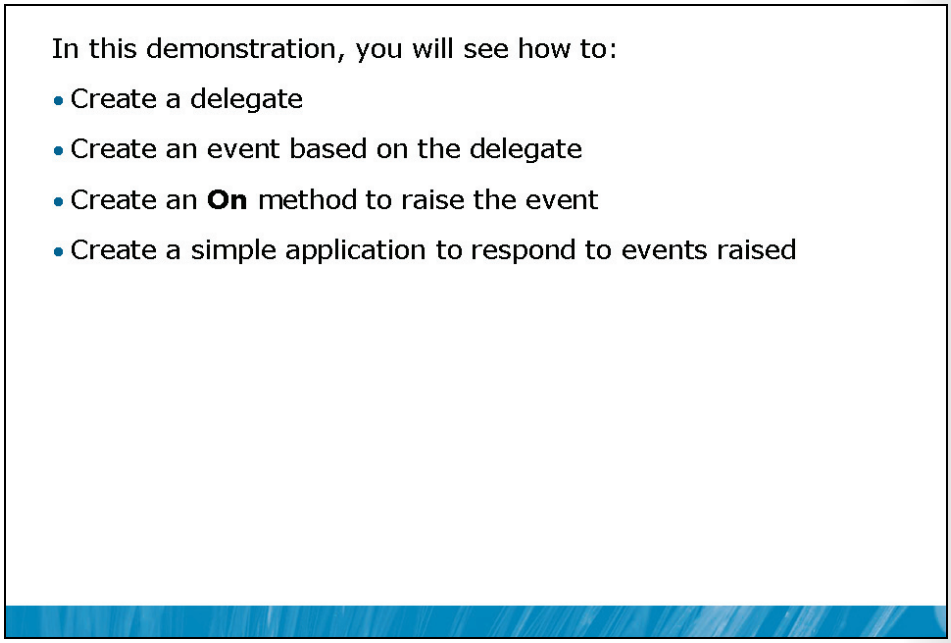
Question: Can you add code to the **DoWork** event handler to update the user interface directly?

Additional Reading

For more information about how to run an operation in the background, see the How to: Run an Operation in the Background page at http://go.microsoft.com/fwlink/?LinkId=192960.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using Events



Key Points

In this demonstration, you will see how to:

- Create a delegate.
- Create an event based on the delegate.
- Create an **On** method to raise the event.
- Create a simple application to respond to events raised.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Start Microsoft Visual Studio® 2010.
3. Open the Module 11, Demo1 starter solution.
4. Open the Heartbeat.cs file, and then review the **Heartbeat** class.

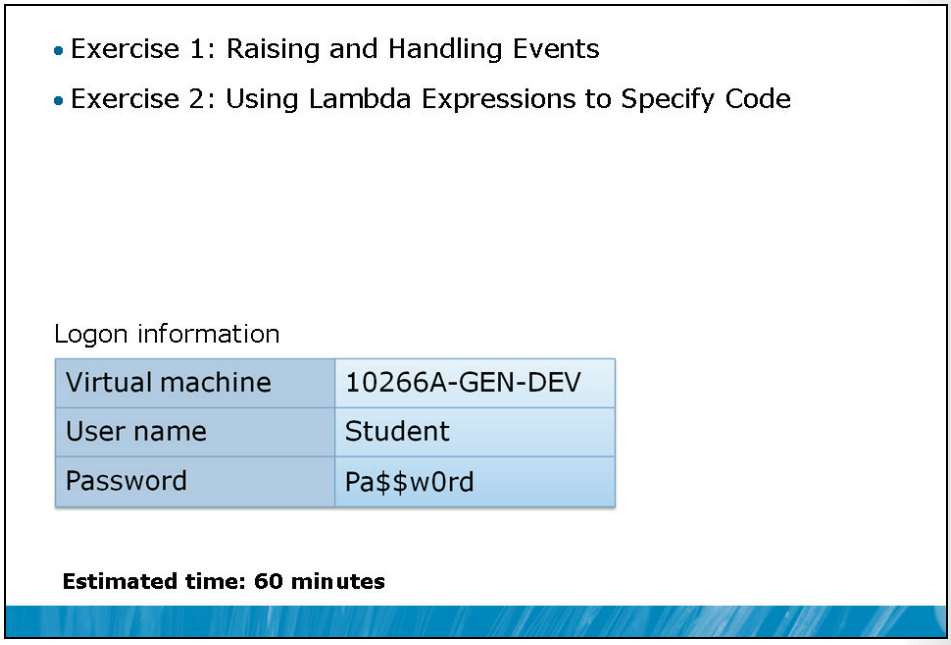
MCT USE ONLY. STUDENT USE PROHIBITED

5. Uncomment the **HeartbeatEventArgs** class. This class inherits from the **EventArgs** class and defines a read-only property for the heartbeat count.
6. Uncomment the code that creates a new delegate named **BeatDelegate**. You will use this delegate for an event to inform consuming classes each time the application increments the heartbeat counter.
7. Uncomment the code that defines an event named **Beat** that is based on the **BeatDelegate** delegate. Your class will raise this event each time the heartbeat count is incremented.
8. Uncomment the **OnBeat** method. This method includes logic to check that the **Beat** event is not null (which would indicate that the event had no subscribers) before raising the event. This is best practice, and this method can be overridden in child classes.
9. Uncomment the code in the **Start** method that raises the **Beat** event by using the **OnBeat** method. You use the **this** keyword as the first parameter and create a new instance of the **HeartbeatEventArgs** class by using the current count as the second parameter.
10. Open the MainWindow.xaml.cs file, and then review the event handlers for the **Click** events.
11. Uncomment the code in the **button1_Click** method that adds an event handler for the **Beat** event of the **beat** object.
12. Uncomment the **beat_Beat** method. This method handles the **Beat** event by displaying a message box each time that the event is raised. Note the use of the property from the custom event arguments class.
13. Run the application.
14. Click **Start**.
15. Verify that the application works correctly. Highlight the message boxes when they appear (they should appear every three seconds).
16. Close the application.
17. Close Visual Studio.

Question: Why should you use the **protected** modifier instead of the **public** modifier as the access modifier for an **On** method?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Decoupling Methods and Handling Events



Objectives

After completing this lab, you will be able to:

- Raise an event and handle it by using a delegate.
- Use lambda expressions to abstract methods and actions.

Introduction

In this lab, you will define and raise events and handle them by using delegates. You will use lambda expressions to specify actions to perform and will run these actions by invoking the lambda expressions.

MCT USE ONLY. STUDENT USE PROHIBITED

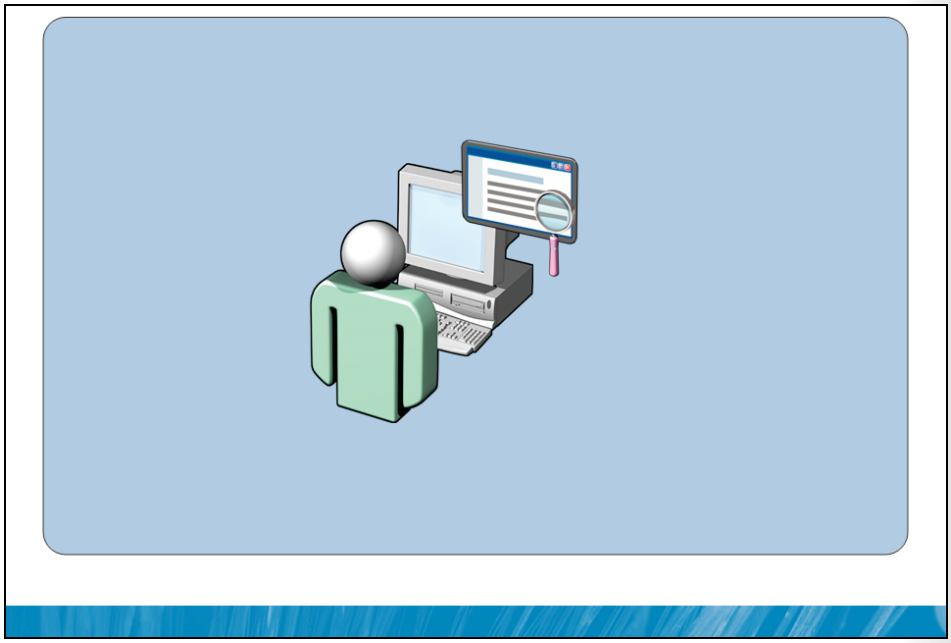
Lab Setup

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



You have been asked to add further features to the measuring devices that log measurement data. The measuring devices take new measurements when they detect a change in the object being measured. These changes may occur at any time. You have been asked to modify the software that drives these devices to trigger an event each time a new measurement is taken. It must be possible to pause the data collection process from the client application, stop receiving measurements, and then later restart the collection process.

The rate at which new measurements are received is variable; therefore, it is not easy to tell whether the device is still functioning. You have been asked to add heartbeat functionality to the devices that fires an event on a regular basis to notify client applications that the device is still working. The heartbeat event should also return a datetime stamp to the client application. The heartbeat interval should be set when the **MeasureDataDevice** object is created.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 1: Raising and Handling Events

In this exercise, you will modify the **IMeasuringDevice** interface and add an event called **NewMeasurementTaken**. This event will be triggered whenever the device detects a change and takes a new measurement.

You will modify the **MeasureDataDevice** abstract class from the previous lab and implement this event. The **NewMeasurementTaken** event will occur after the device has populated the internal buffer with the new measurement and logged it.

You will use a **BackgroundWorker** component to poll for new measurements. The polling for new measurements will take place in the **DoWork** event, and the **ProgressReported** event will raise the **NewMeasurementTaken** event to notify the client application that a new measurement has been taken.

You will start the background thread running by using the **RunWorkerAsync** method, and the device will support cancellation of the background thread by using the **CancelWorkerAsync** method.

You will test the new functionality by using an existing WPF application that creates an instance of the **MeasureMassDevice** class and trapping the events that it raises by using a delegate. The WPF application should be able to pause and then restart the the **MeasureMassDevice** class.

The main tasks for this exercise are as follows:

1. Open the Events solution.
2. Create a new interface that extends the **IMeasuringDevice** interface.
3. Add the **NewMeasurementTaken** event to the **MeasureDataDevice** class.
4. Add a **BackgroundWorker** member to the **MeasureDataDevice** class.
5. Add the **GetMeasurements** method to the **MeasureDataDevice** class.
6. Implement the **dataCollector_DoWork** method.
7. Implement the **dataCollector_ProgressChanged** method.
8. Call the **GetMeasurements** method to start collecting measurements.
9. Call the **CancelAsync** method to stop collecting measurements.
10. Dispose of the **BackgroundWorker** object when the **MeasureDataDevice** object is destroyed.
11. Update the UI to handle measurement events.
12. Implement the **device_NewMeasurementTaken** event-handling method.

MCT USE ONLY. STUDENT USE PROHIBITED

13. Disconnect the event handler.
14. Test the solution.

▶ Task 1: Open the Events 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 Events solution in the E:\Labfiles\Lab 11\Ex1\Starter folder.

▶ Task 2: Create a new interface that extends the IMeasuringDevice interface

1. In the MeasuringDevice project, add a new interface named **IEventEnabledMeasuringDevice** in a file named IEventEnabledMeasuringDevice.cs.



Note: Creating a new interface that extends an existing interface is good programming practice, because it preserves the structure of the original interface for backward compatibility with preexisting code. All preexisting code can reference the original interface, and new code can reference the new interface and take advantage of any new functionality.

2. Modify the interface definition so that the **IEventEnabledMeasuringDevice** interface extends the **IMeasuringDevice** interface.
3. In the **IEventEnabledMeasuringDevice** interface, add an event named **NewMeasurementTaken** by using the base **EventHandler** delegate.
4. Build the application to enable Microsoft IntelliSense® to reflect your changes.

▶ Task 3: Add the NewMeasurementTaken event to the MeasureDataDevice class

1. Review the task list.
2. Locate the **TODO - Modify the class definition to implement the extended interface** task, and then double-click this task. This task is located in the MeasureDataDevice class file.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Remove the **TODO - Modify the class definition to implement the extended interface** comment, and then modify the class definition to implement the **IEventEnabledMeasuringDevice** interface instead of the **IMeasuringDevice** interface.
4. In the task list, locate the **TODO - Add the NewMeasurementTaken event** task, and then double-click this task. This task is located at the end of the **MeasureDataDevice** class.
5. Remove the **TODO - Add the NewMeasurementTaken event** comment, and then declare an event named **NewMeasurementTaken** by using the same signature as the interface.
6. Below the event, remove the **TODO - Add an OnMeasurementTaken method** comment, and then add a protected virtual method named **OnNewMeasurementTaken**. The method should accept no parameters and have a void return type. The **MeasureDataDevice** class will use this method to raise the **NewMeasurementTaken** event.
7. In the **OnNewMeasurementTaken** method, add code to check that there is a subscriber for the **NewMeasurementTaken** event; if so, raise the event. The signature of the **EventHandler** delegate defines two parameters: an *object* parameter that indicates the object that raised the event and an *EventArgs* parameter that provides any additional data that is passed to the event handler. Set the *object* parameter to **this** and the *EventArgs* parameter to **null**.



Note: It is good programming practice to check that there are subscribers for an event before you raise it. If an event has no subscribers, the related delegate is null, and the .NET Framework runtime will throw an exception if the event is raised.

▶ Task 4: Add a BackgroundWorker member to the MeasureDataDevice class

1. In the task list, locate the **TODO - Declare a BackgroundWorker to generate data** task, and then double-click this task. This task is located near the top of the **MeasureDataDevice** class.
2. Remove the **TODO - Declare a BackgroundWorker to generate data** comment, and then add a private **BackgroundWorker** member named **dataCollector** to the class.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Add the GetMeasurements method to the MeasureDataDevice class

The **GetMeasurements** method will initialize the **dataCollector BackgroundWorker** member to poll for new measurements and raise the **NewMeasurementTaken** event each time it detects a new measurement.

1. In the task list, locate the **TODO - Implement the GetMeasurements method** task, and then double-click this task.
2. Remove the **TODO - Implement the GetMeasurements method** comment, and then add a new private method named **GetMeasurements** to the class. This method should take no parameters and not return a value.
3. In the **GetMeasurements** method, add code to perform the following actions:
   a. Instantiate the **dataCollector BackgroundWorker** member.
   b. Specify that the **dataCollector BackgroundWorker** member supports cancellation.
   c. Specify that the **dataCollector BackgroundWorker** member reports progress while running.



Hint: Set the **WorkerSupportsCancellation** and **WorkerReportsProgress** properties.

4. Add the following code to instantiate a **DoWorkEventHandler** delegate that refers to a method called **dataCollector_DoWork**. Attach the delegate to the **DoWork** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_DoWork** method when the **DoWork** event is raised.



Hint: Use IntelliSense to generate a code stub for the **dataCollector_DoWork** method. To do this, type the first part of the line of code, up to the **+=** operators, and then press the TAB key twice. Visual Studio uses a built-in code snippet to complete the line of code and then add a method stub. You can do this each time you hook up an event handler to an event by using the **+=** compound assignment operator.

```
...
    dataCollector.WorkerReportsProgress = true;
    dataCollector.DoWork +=
        new DoWorkEventHandler(dataCollector_DoWork);
} ...
```

5. Using the same technique as in the previous step, instantiate a **ProgressChangedEventHandler** delegate that refers to a method called **dataCollector_ProgressChanged**. Attach this delegate to the **ProgressChanged** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_ProgressChanged** method when the **ProgressChanged** event is raised.
6. Add code to start the **dataCollector BackgroundWorker** object running asynchronously.

▶ Task 6: Implement the dataCollector_DoWork method

1. Underneath the **GetMeasurements** method, locate the **dataCollector_DoWork** method.

   This method was generated during the previous task. It runs on a background thread, and its purpose is to collect and store measurement data.
2. In the **dataCollector_DoWork** method, remove the statement that raises the **NotImplementedException** exception and add code to perform the following actions:
   a. Instantiate the **dataCaptured** array with a new integer array that contains 10 items.
   b. Define an integer i with an initial value of zero. You will use this variable to track the current position in the **dataCaptured** array.
   c. Add a **while** loop that runs until the **dataCollector**.**CancellationPending** property is **false**.
3. In the **while** loop, add code to perform the following actions:
   a. Invoke the **controller**.**TakeMeasurement** method, and store the result in the **dataCaptured** array at the position that the integer **i** indicates. The **TakeMeasurement** method of the **controller** object blocks until a new measurement is available.
   b. Update the **mostRecentCapture** property to contain the value in the **dataCaptured** array at the position that the integer **i** indicates.
   c. If the value of the disposed variable is **true**, terminate the **while** loop. This step ensures that the measurement collection stops when the **MeasureDataDevice** object is destroyed.
4. Add code to the **while** loop after the statements that you added in the previous step to perform the following actions:

MCT USE ONLY. STUDENT USE PROHIBITED

a. Check whether the **loggingFileWriter** property is null.

b. If the **loggingFileWriter** property is not null, call the **loggingFileWriter**.**Writeline** method, passing a string parameter of the format "Measurement - *mostRecentMeasure*" where *mostRecentMeasure* is the value of the mostRecentMeasure variable.



Note: The **loggingFileWriter** property is a simple **StreamWriter** object that writes to a text file. This property is initialized in the **StartCollecting** method. You can use the **WriteLine** method to write to a **StreamWriter** object.

5. Add a line of code to the end of the **while** loop to invoke the **dataCollector**.**ReportProgress** method, passing zero as the parameter.

   The **ReportProgress** method raises the **ReportProgress** event and is normally used to return the percentage completion of the tasks assigned to the **BackgroundWorker** object. You can use the **ReportProgress** event to update progress bars or time estimates in the UI. In this case, because the task will run indefinitely until canceled, you will use the **ReportProgress** event as a mechanism to prompt the UI to refresh the display with the new measurement.

6. Add code to the end of the **while** loop to perform the following actions:

   a. Increment the integer **i**.

   b. If the value of the integer is greater than nine, reset **i** to zero.

      You are using the integer i as a pointer to the next position to write to in the **dataCaptured** array. This array has space for 10 measurements. When element 9 is filled, the device will start to overwrite data beginning at element 0.

▶ Task 7: Implement the dataCollector_ProgressChanged method

1. Locate the **dataCollector_ProgressChanged** method.

   This method was generated during an earlier task. It runs when the **ProgressChanged** event is raised. In this exercise, this event occurs when the **dataCollector_DoWork** method takes and stores a new measurement.

2. In the event handler, delete the exception code, and then invoke the **OnNewMeasurementTaken** method, passing no parameters.

MCT USE ONLY. STUDENT USE PROHIBITED

The **OnNewMeasurementTaken** method raises the **NewMeasurementTaken** event that you defined earlier. You will modify the UI to subscribe to this event, so that when it is raised, the UI can update the displayed information.

▶ Task 8: Call the GetMeasurements method to start collecting measurements

1. In the task list, locate the **TODO - Call the GetMeasurements method** task, and then double-click this task. This task is located in the **StartCollecting** method.
2. Remove the **TODO - Call the GetMeasurements method** comment, and add a line of code to invoke the **GetMeasurements** method.

▶ Task 9: Call the CancelAsync method to stop collecting measurements

1. In the task list, locate the **TODO - Cancel the data collector** task, and then double-click this task. This task is located in the **StopCollecting** method.
2. Remove the **TODO - Cancel the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **CancelAsync** method to stop the work performed by the **dataCollector BackgroundWorker** object.

▶ Task 10: Dispose of the BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - Dispose of the data collector** task, and then double-click this task. This task is located in the **Dispose** method of the **MeasureDataDevice** class.
2. Remove the **TODO - Dispose of the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **Dispose** method to dispose of the **dataCollector** instance.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 11: Update the UI to handle measurement events

1. In the task list, locate the **TODO - Declare a delegate to reference NewMeasurementEvent** task, and then double-click this task. This task is located in the code behind the MainWindow.xaml window.
2. Remove the comment and add code to define a delegate of type **EventHandler** named **newMeasurementTaken**.
3. In the **startCollecting_Click** method, remove the comment **TODO - use a delegate to refer to the event handler**, and add code to initialize the **newMeasurementTaken** delegate with a new **EventHandler** delegate that is based on a method named **device_NewMeasurementTaken**. You will create the **device_NewMeasurementTaken** method in the next task.



Note: You cannot use IntelliSense to automatically generate the stub for the **device_NewMeasurementTaken** method, as you did in earlier tasks.

4. In the **startCollecting_Click** method, remove the **TODO - Hook up the event handler to the event** comment, and add code to connect the **newMeasurementTaken** delegate to the **NewMeasurementTaken** event of the **device** object. The **device** object is an instance of the **MeasureMassDevice** class, which inherits from the **MeasureDataDevice** abstract class.



Hint: To connect a delegate to an event, use the **+=** compound assignment operator on the event.

▶ Task 12: Implement the device_NewMeasurementTaken event-handling method

1. In the task list, locate the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** task, and then double-click this task.
2. Remove the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** comment, and add a private event-handler method named **device_NewMeasurementTaken**. The method should not return a value, but should take the following parameters:
   a. An **object** object named **sender**.

b. An **EventArgs** object named **e**.

3. In the **device_NewMeasurementTaken** method, add code to check that the **device** member is not null. If the **device** member is not null, perform the following tasks:

   a. Update the **Text** property of the **mostRecentMeasureBox** text box with the value of the **device.MostRecentMeasure** property.



Hint: Use the **ToString** method to convert the value that the **device.MostRecentMeasure** property returns from an integer to a string.

   b. Update the **Text** property of the **metricValueBox** text box with the value that the **device.MetricValue** method returns.

   c. Update the **Text** property of the **imperialValueBox** text box with the value that the **device.ImperialValue** method returns.

   d. Reset the **rawDataValues.ItemsSource** property to **null**.

   e. Set the **rawDataValues.ItemsSource** property to the value that the **device.GetRawData** method returns.



Note: The final two steps are both necessary to ensure that the data-binding mechanism that the **Raw Data** box uses on the WPF window updates the display correctly.

▶ Task 13: Disconnect the event handler

1. In the task list, locate the **TODO - Disconnect the event handler** task, and then double-click this task. This task is located in the **stopCollecting_Click** method, which runs when the user clicks the **Stop Collecting** button.

2. Remove the **TODO - Disconnect the event handler** comment, and add code to disconnect the **newMeasurementTaken** delegate from the **device.NewMeasurementTaken** event.



Hint: To disconnect a delegate from an event, use the **-=** compound assignment operator on the event.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 14: Test the solution

1. Build the project and correct any errors.
2. Start the application.
3. Click **Start Collecting**, and verify that measurement values begin to appear in the **Raw Data** box.

   The **MeasureMassDevice** object used by the application takes metric measurements and stores them, before raising the **NewMeasurementTaken** event. The event calls code that updates the UI with the latest information. Continue to watch the **Raw Data** list box to see the buffer fill with data and then begin to overwrite earlier values.
4. Click **Stop Collecting**, and verify that the UI no longer updates.
5. Click **Start Collecting** again. Verify that the **Raw Data** list box is cleared and that new measurement data is captured and displayed.
6. Click **Stop Collecting**.
7. Close the application, and then return to Visual Studio.

Exercise 2: Using Lambda Expressions to Specify Code

In this exercise, you will declare a new delegate type and a new **EventArgs** type to support the **HeartBeat** event. You will modify the **IMeasuringDevice** interface and the **MeasureDataDevice** class to generate the heartbeat by using a **BackgroundWorker** object. You will specify the code to run on the new thread by using a lambda expression.

In the **ReportProgress** event handler, you will specify the code to notify the client application with another lambda expression.

You will handle the **HeartBeat** event in the WPF application by using a lambda expression.

The main tasks for this exercise are as follows:

1. Open the Events solution.
2. Define a new **EventArgs** class to support heartbeat events.
3. Declare a new delegate type.
4. Update the **IEventEnabledMeasuringDevice** interface.

5. Add the **HeartBeat** event and **HeartBeatInterval** property to the **MeasureDataDevice** class.
6. Use a **BackgroundWorker** object to generate the heartbeat.
7. Call the **StartHeartBeat** method when the **MeasureDataDevice** object starts running.
8. Dispose of the **heartBeatTimer BackgroundWorker** object when the **MeasureDataDevice** object is destroyed.
9. Update the constructor for the **MeasureMassDevice** class.
10. Handle the **HeartBeat** event in the UI.
11. Test the solution.

▶ Task 1: Open the Events solution

- Open the Events solution in the E:\Labfiles\Lab 11\Ex2\Starter folder.



Note: The Events solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1; however, it includes an updated task list to enable you to complete this exercise.

▶ Task 2: Define a new EventArgs class to support heartbeat events

1. In the MeasuringDevice project, add a new code file named HeartBeatEvent.cs.
2. In the code file, add a **using** directive to bring the **System** namespace into scope.
3. Define a new class named **HeartBeatEventArgs** in the **MeasuringDevice** namespace. The class should extend the **EventArgs** class.



Note: A custom event arguments class can contain any number of properties; these properties store information when the event is raised, enabling an event handler to receive event-specific information when the event is handled.

4. In the **HeartBeatEventArgs** class, add a read-only automatic **DateTime** property named **TimeStamp**.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Add a constructor to the **HeartBeatEventArgs** class. The constructor should accept no arguments, and initialize the **TimeStamp** property to the date and time when the class is constructed. The constructor should also extend the base class constructor.

▶ Task 3: Declare a new delegate type

- Below the **HeartBeatEventArgs** class, declare a **public delegate** type named **HeartBeatEventHandler**. The delegate should refer to a method that does not return a value, but that has the following parameters:

  a. An **object** parameter named *sender*.

  b. A **HeartBeatEventArgs** parameter named *args*.

▶ Task 4: Update the IEventEnabledMeasuringDevice interface

1. In the task list, locate the **TODO - Define the new event in the interface** task and then double-click this task. This task is located in the **IEventEnabledMeasuringDevice** interface
2. Remove this comment and add an event called **HeartBeat** to the interface. The event should specify that subscribers use the **HeartBeatEventHandler** delegate type to specify the method to run when the event is raised.
3. Remove the **TODO - Define the HeartBeatInterval member in the interface** comment, and then add a read-only integer property called **HeartBeatInterval** to the interface.

▶ Task 5: Add the HeartBeat event and HeartBeatInterval property to the MeasureDataDevice class

1. In the task list, locate the **TODO - Add the HeartBeatInterval property** task, and then double-click this task. This task is located in the **MeasureDataDevice** class.
2. Remove the **TODO - Add the HeartBeatInterval property** comment, and add a protected integer member named **heartBeatIntervalTime**.
3. Add code to implement the public integer property **HeartBeatInterval** that the **IEventEnabledMeasuringDevice** interface defines. The property should return the value of the **heartBeatInterval** member when the **get** accessor method is

MCT USE ONLY. STUDENT USE PROHIBITED

called. The property should have a **private set** accessor method to enable the constructor to set the property.

4. Remove the **TODO - Add the HeartBeat event** comment, and add the **HeartBeat** event that the **IEventEnabledMeasuringDevice** interface defines.
5. Remove the **TODO - add the OnHeartBeat method to fire the event** comment, and add a protected virtual void method named **OnHeartBeat** that takes no parameters.
6. In the **OnHeartBeat** method, add code to perform the following actions:
   a. Check whether the **HeartBeat** event has any subscribers.
   b. If the event has subscribers, raise the event, passing the current object and a new instance of the **HeartBeatEventArgs** object as parameters.

▶ Task 6: Use a BackgroundWorker object to generate the heartbeat

1. Remove the **TODO - Declare the BackgroundWorker to generate the heartbeat** comment, and then define a private **BackgroundWorker** object named **heartBeatTimer**.
2. Remove the **TODO - Create a method to configure the BackgroundWorker using Lambda Expressions** comment, and declare a private method named **StartHeartBeat** that accepts no parameters and does not return a value.
3. In the **StartHeartBeat** method, add code to perform the following actions:
   a. Instantiate the **heartBeatTimer BackgroundWorker** object.
   b. Configure the **heartBeatTimer** object to support cancellation.
   c. Configure the **heartBeatTimer** object to support progress notification.
4. Add a handler for the **heartBeatTimer DoWork** event by using a lambda expression to define the actions to be performed. The lambda expression should take two parameters (use the names *o* and *args*). In the lambda expression body, add a **while** loop that continually iterates and contains code to perform the following actions:
   a. Use the static **Thread.Sleep** method to put the current thread to sleep for the length of time that the **HeartBeatInterval** property indicates.
   b. Check the value of the **disposed** property. If the value is **true**, terminate the loop.

MCT USE ONLY. STUDENT USE PROHIBITED

c. Call the **heartBeatTimer**.**ReportProgress** method, passing zero as the parameter.



Note: Use the **+=** compound assignment operator to specify that the method will handle the **DoWork** event, define the signature of the lambda expression, and then use the **=>** operator to denote the start of the body of the lambda expression.

5. Add a handler for the **heartBeatTimer**.**ReportProgress** event by using another lambda expression to create the method body. In the lambda expression body, add code to call the **OnHeartBeat** method, which raises the **HeartBeat** event.
6. At the end of the **StartHeartBeat** method, add a line of code to start the **heartBeatTimer BackgroundWorker** object running asynchronously.

▶ Task 7: Call the StartHeartBeat method when the MeasureDataDevice object starts running

1. In the task list, locate the **TODO - Call StartHeartBeat() from StartCollecting method** task, and then double-click this task. This task is located in the **StartCollecting** method.
2. Remove this comment, and add a line of code to invoke the **StartHeartBeat** method.

▶ Task 8: Dispose of the heartBeatTimer BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - dispose of the heartBeatTimer BackgroundWorker** task, and then double-click this task. This task is located in the **Dispose** method.
2. Remove the comment and add code to check that the **heartBeatTimer BackgroundWorker** object is not null. If the **heartBeatTimer** object is not null, call the **Dispose** method of the **BackgroundWorker** object.

   You have now updated the **MeasureDataDevice** abstract class to implement event handlers by using lambda expressions. To enable the application to benefit from these changes, you must modify the **MeasureMassDevice** class, which extends the **MeasureDataDevice** class.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 9: Update the constructor for the MeasureMassDevice class

1. Open the MeasureMassDevice class file.
2. At the start of the class, modify the signature of the constructor to take an additional **integer** value named **heartBeatInterval**.
3. Modify the body of the constructor to store the value of the **HeartBeatInterval** member in the **heartBeatInterval** member.
4. Below the existing constructor, remove the **TODO − Add a chained constructor that calls the previous constructor** comment, and add a second constructor that accepts the following parameters:
   a. A **Units** instance named **deviceUnits**.
   b. A **string** instance named **logFileName**.
5. Modify the new constructor to implicitly call the existing constructor. Pass a value of **1000** as the *heartBeatInterval* parameter value.

▶ Task 10: Handle the HeartBeat event in the UI

1. In the task list, locate the **TODO - Use a lambda expression to handle the HeartBeat event in the UI** task, and then double-click this task. This task is located in the **startCollecting_Click** method in the code behind the MainWindow window in the Monitor project.
2. Remove the comment, and add a lambda expression to handle the **device**.**HeartBeat** event. The lambda expression should take two parameters (name them *o* and *args*). In the body of the lambda expression, add code to update the **heartBeatTimeStamp** label with the text "HeartBeat Timestamp: *timestamp*" where timestamp is the value of the **args.TimeStamp** property.



Hint: Set the **Content** property of a label to modify the text that the label displays.

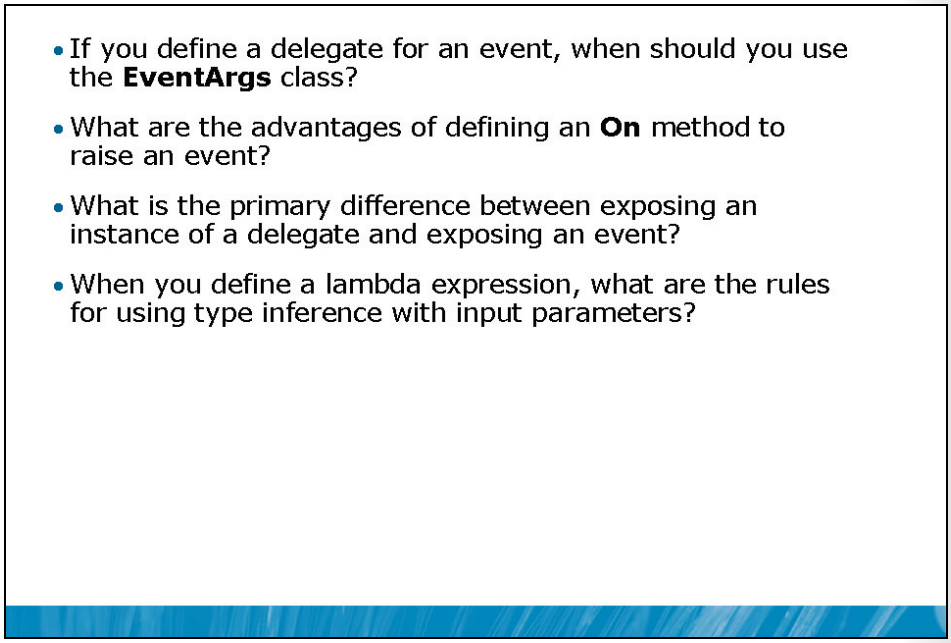
▶ Task 11: Test the solution

1. Build the project and correct any errors.
2. Start the application.
3. Click **Start Collecting**, and verify that values begin to appear as before. Also note that the **HeartBeat Timestamp** value now updates once per second.

4. Click **Stop Collecting**, and verify that the **RawData** list box no longer updates. Note that the timestamp continues to update, because your code does not terminate the timestamp heartbeat when you stop collecting.
5. Click **Dispose Object**, and verify that the timestamp no longer updates.
6. Close the application, and then return to Visual Studio.
7. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

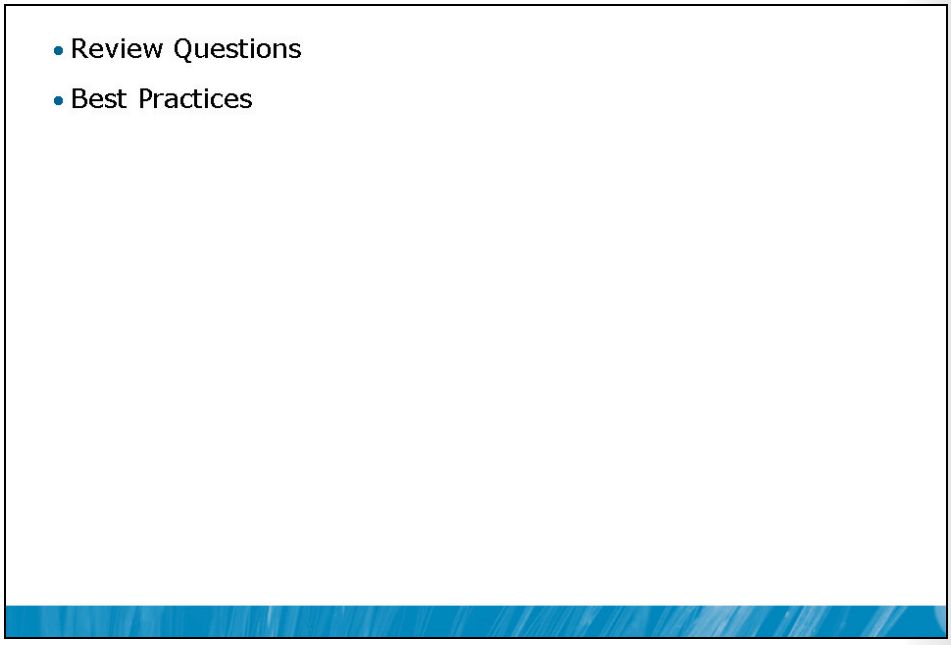


Review Questions

1. If you define a delegate for an event, when should you use the **EventArgs** class?
2. What are the advantages of defining an **On** method to raise an event?
3. What is the primary difference between exposing an instance of a delegate and exposing an event?
4. When you define a lambda expression, what are the rules for using type inference with input parameters?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. When might it be inappropriate to use a lambda expression?
2. How can you invoke a method asynchronously if it only natively supports being called synchronously?
3. Can lambda expressions use variables declared outside the lambda expression?

Best Practices Related to Using Delegates

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

- Use the delegate types defined in the .NET Framework instead of developing custom delegate types wherever possible.
- Use delegates to invoke synchronous methods asynchronously where appropriate; however, you should not omit asynchronous methods from a type

MCT USE ONLY. STUDENT USE PROHIBITED

where you can implement the asynchronous version of a method more efficiently than using the delegate syntax.

Best Practices Related to Using Lambda Expressions

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

- Only use a lambda expression if you use the method only once. If you are writing duplicate lambda expressions, you should normally use a named method instead.
- Do not change an object's state in a lambda expression. Wherever possible, you should write lambda expressions that do not have side effects.
- Avoid referencing variables defined outside the scope of the lambda expression.

Best Practices Related to Using Events

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

- Use the standard event signature.
- Use a protected virtual method to raise an event.
- Do not pass null as a parameter when you raise an event.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 12

Using Collections and Building Generic Types

Contents:

Lesson 1: Using Collections 12-4

Lab A: Using Collections 12-22

Lesson 2: Creating and Using Generic Types 12-28

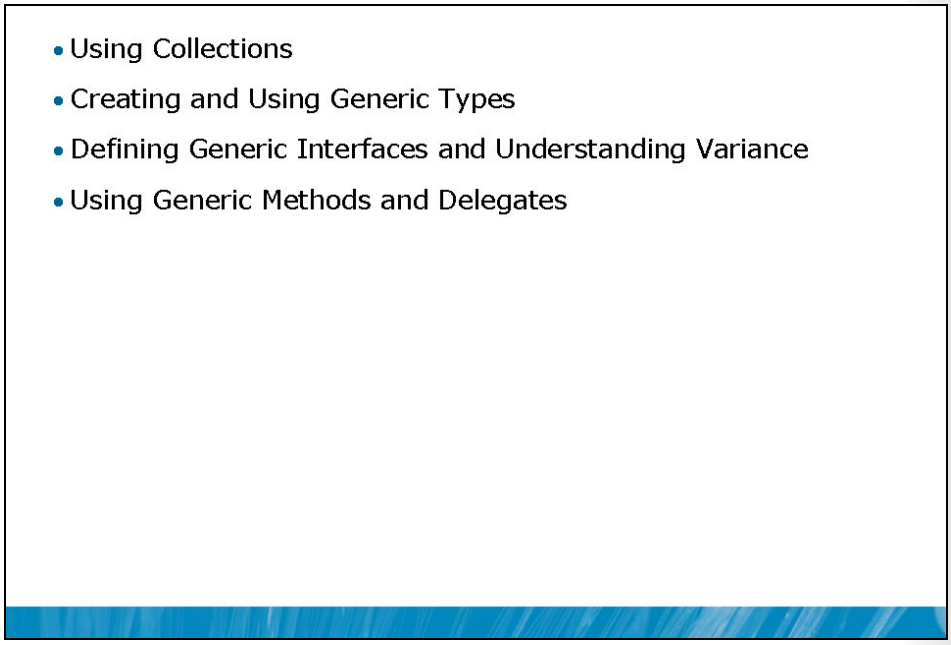
Lesson 3: Defining Generic Interfaces and Understanding Variance 12-42

Lesson 4: Using Generic Methods and Delegates 12-56

Lab B: Building Generic Types 12-69

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



In this course, you have seen how to develop types for use in your application, and created instances of types by using a named instance, or by storing several instances in an array. When you develop an application, you will often want to create multiple instances of a type. However, you may not want to specifically name each instance and you may need more flexibility than you can achieve by using arrays. In this module, you will learn about collection classes and how you can use them with greater flexibility than a simple array.

The basic collection classes introduce a new problem. Classes that act on other types are often not type-safe. For example, many collection classes frequently use the **Object** type to store items, and must then be cast back to their original type before they can be used. It is the programmer's responsibility to ensure that the correct casts are performed, and it is easy to introduce errors by casting an item to the wrong type. This module introduces generics and how you can use generic classes to maintain type integrity and avoid issues that are associated with a lack of type safety.

MCT USE ONLY. STUDENT USE PROHIBITED

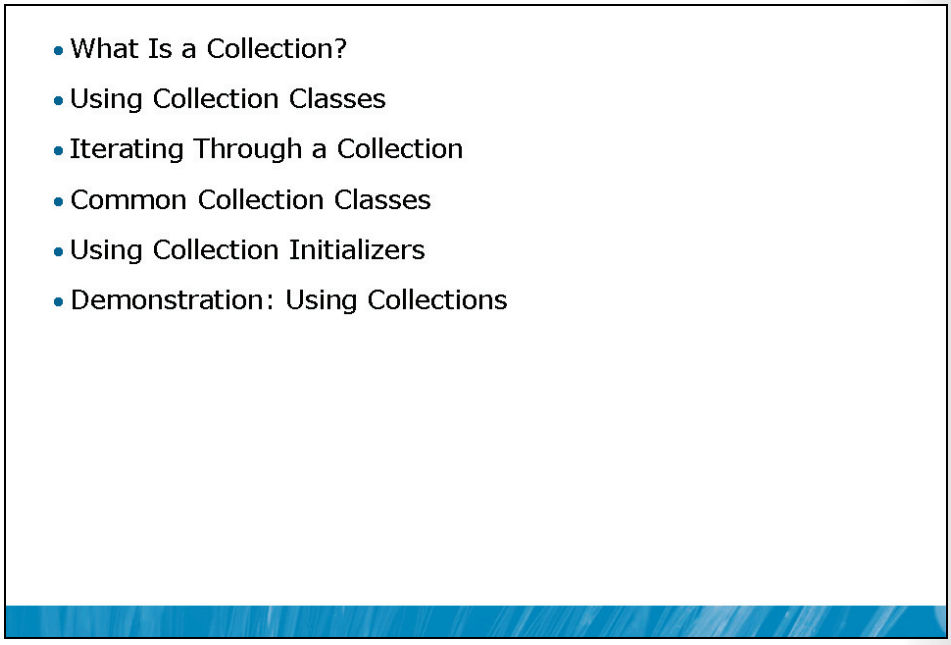
Objectives

After completing this module, you will be able to:

- Use collection classes.
- Define and use generic types.
- Define generic interfaces and explain the concepts of covariance and contravariance.
- Define and use generic methods and delegates.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Using Collections



You have already seen how to use arrays to aggregate data and store multiple related instances of objects. Arrays are useful, but they have their limitations. For example, they are difficult to resize after you have created them, and the way in which you access an item in an array may not reflect the way in which the real world works. Collections are more flexible. They can dynamically resize themselves as you add or remove data, and they provide a range of strategies that you can use to access the items that they hold.

This lesson introduces you to collections and describes how you can use them in your applications.

Objectives

After completing this lesson, you will be able to:

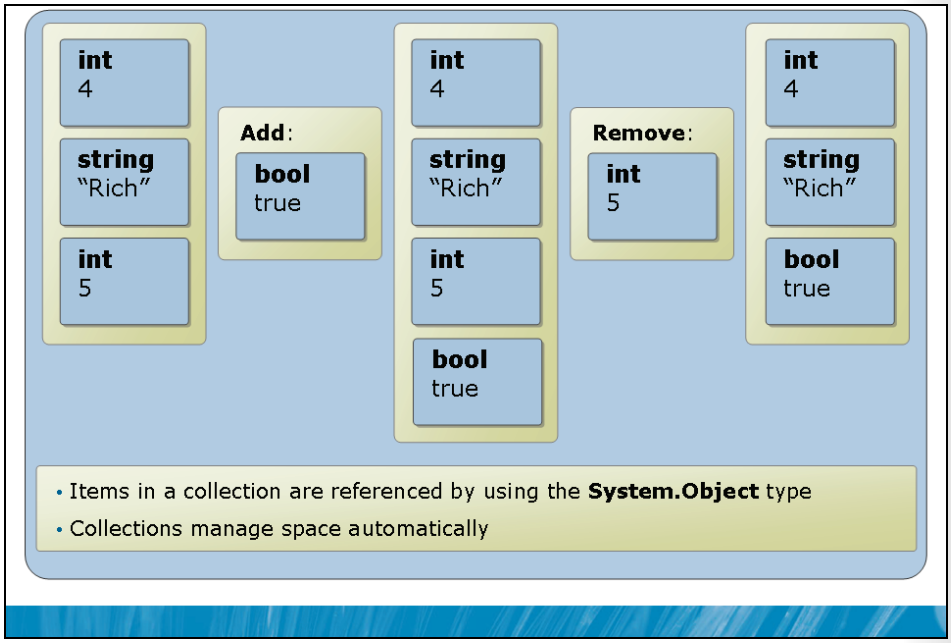
- Describe the purpose of a collection.
- Describe how collection classes work.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe the common collection classes in the Microsoft® .NET Framework.
- Explain collection initializers.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Collection?



Key Points

A collection is a type that aggregates objects; it acts as a container for a set of objects. You can create an instance of a collection class and add objects to the collection. You can then access those items by using methods that the collection class provides. Collection classes may appear to be similar to arrays, but, in fact, a collection class serves a different purpose to an array and in most circumstances is much more flexible.

When you define an array, you specify the type of data that the array stores and the size of the array. An array is type-safe, but has a big limitation; when you define an array, you must specify how many items the array can hold. This quantity may not be easy to determine in advance. If you specify too large an array, you will use too much memory; if you specify too small an array, you will run out of space. Arrays work well when you know exactly how many values you need to store.

Collections are much more flexible. When you create an instance of a collection class, you do not need to specify the size of the collection. The collection dynamically grows and shrinks according to the volume of data that it stores. This dynamic space management makes collections very powerful; if you used arrays

MCT USE ONLY. STUDENT USE PROHIBITED

instead of collections and needed the array to resize, you would need to write significantly more code to manage the array size. The disadvantage of arrays is the overhead that is associated with performing dynamic memory management. However, you can mitigate this overhead to some extent when you initialize an array; you can specify an initial size that should match the most common requirements for your application.

Object Types in Collections

When you use an array to store several objects, you explicitly specify the type of the data that you want to store in the array. For example, you may have an array of integers or an array of strings. When you store values in the array, or retrieve a value from the array, it is strictly type-safe. If you attempt to store or retrieve data of the wrong type, the compiler detects this problem and fails with an error.

When you use a collection, you do not specify the type of data to store. The collection classes store references to other objects by using the **System.Object** type. This feature enables you to construct collections that store mixed types; for example, you can store **string** objects and **int** values in the same collection. When you retrieve an item from the collection, you must cast the item to the appropriate type.

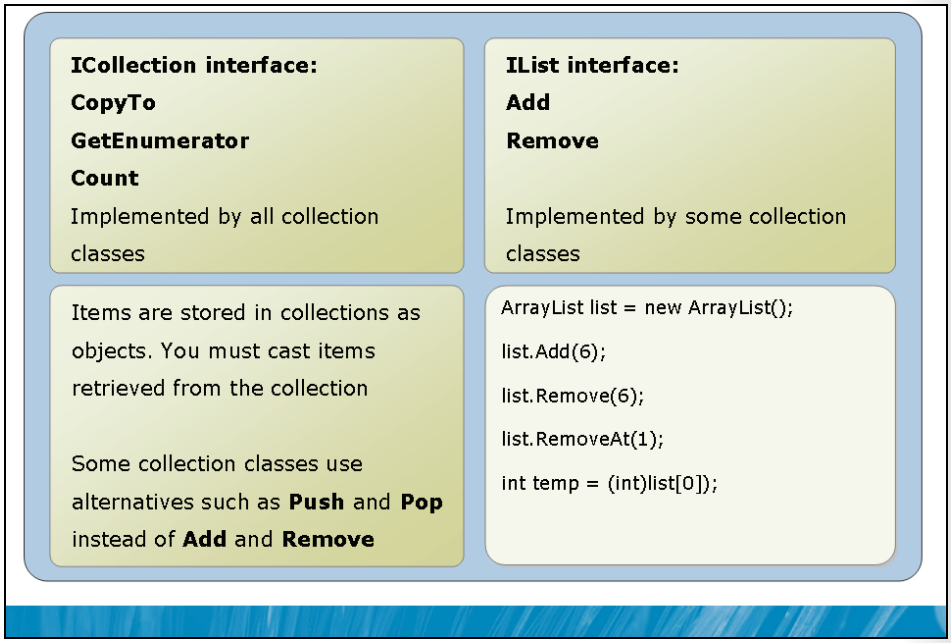
Dimensions in Collections and Arrays

When you define an array, you can specify that it has more than one dimension. Collections do not have dimensions. However, you can imitate a multidimensional collection by storing collections in a collection.

Question: You are developing an application that maintains a rolling buffer of 10 readings taken by a device. Would you use an array or a collection to store the values?

MCT USE ONLY. STUDENT USE PROHIBITED

Using Collection Classes



Key Points

The .NET Framework defines a set of collection classes in the **System.Collections** namespace. Most collections have some similarities; they all implement the **ICollection** interface. This interface defines a small number of methods and properties, including:

- **CopyTo**. This method enables you to copy the contents of a collection to an array.
- **GetEnumerator**. This method returns an object called an **enumerator** that you can use to iterate through the items in the collection.
- **Count**. This is a property that indicates the current number of items in the collection.

Some collections also implement the **IList** interface. This interface defines members that enable you to access the elements in the collection by using array-like notation, in addition to adding and removing members by using methods called **Add** and **Remove**.

MCT USE ONLY. STUDENT USE PROHIBITED

In addition to the members that the **ICollection** and **IList** interfaces require, most collection classes expose specific methods and members that underpin their functionality. The names for these methods are not restricted and are normally closely related to the purpose of the collection. For example, the **Queue** class exposes **Enqueue** and **Dequeue** to add and remove items from the queue in a first-in, first-out (FIFO) manner. The **Stack** class provides **Push** and **Pop** methods to enable you to add items in a first-in, last-out (FILO) manner.

The collection classes in the **System.Collections** namespace store **System.Object** objects, rather than objects of a more specific type. You can add any object to a collection class, but when you retrieve an object from a collection, you must cast the object to its correct type before you can use all of the members that the object exposes. If you attempt to cast an object in a collection to the wrong type, your application will throw an **InvalidCastException** exception.



Warning: You should exercise caution when you retrieve objects from a collection to ensure that you cast the objects correctly. Avoid adding objects of different types to the same collection wherever possible.

The following code example shows how to use the **ArrayList** class (a simple collection class that implements an array that can dynamically resize itself). The **ArrayList** class implements the **IList** interface. It provides the **Add** method that you can use to add an object to the end of the collection, and the **RemoveAt** method that you can use to delete an item from the collection at a specified position. You can also use the **Remove** method to search through the collection and delete the first occurrence of a specified item from the collection. As you add items, the collection can grow automatically.

```
// Create a new ArrayList object.
ArrayList list = new ArrayList();

// Add items to the ArrayList collection.
list.Add(3);
list.Add(4);
list.Add(6);

// Add a string to the ArrayList collection.
// The ArrayList stores objects, not specific
// types so you can add any type to the collection.
list.Add("String Object");

// Remove an object from the ArrayList collection
// by specifying the object to remove.
list.Remove(6);
```

MCT USE ONLY. STUDENT USE PROHIBITED

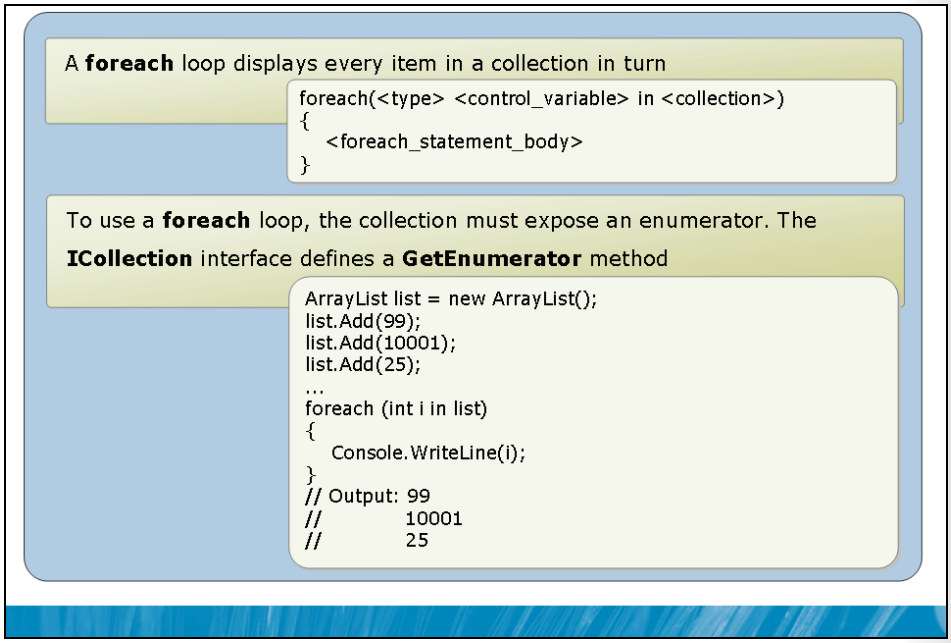
```
// Remove an object from the ArrayList collection
// by specifying the index from which to remove the item.
list.RemoveAt(1);

// Use an indexer to access a specific item in the
// collection. Cast the object to its correct type.
int temp = ((int)list[0]) * 5;
```

Question: Are collections type-safe?

MCT USE ONLY. STUDENT USE PROHIBITED

Iterating Through a Collection



Key Points

All collections implement the **ICollection** interface, and this interface defines a method called **GetEnumerator**. This method returns an object called an enumerator that you can use to quickly iterate through all of the elements in a collection. C# provides the **foreach** statement that you can use for this purpose. The **foreach** statement automatically obtains the enumerator for a collection, and uses this enumerator to fetch each item of the collection in turn, as the following code example shows.

```
ArrayList list = new ArrayList();
list.Add(99);
list.Add(10001);
list.Add(25);
...
foreach (int i in list)
{
    Console.WriteLine(i);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

This example creates and populates an **ArrayList** object that contains a collection of integer values. The **foreach** statement displays each of these values in turn, in the order in which they occur in the **ArrayList** object. The syntax of the **foreach** statement is shown in the following code example.

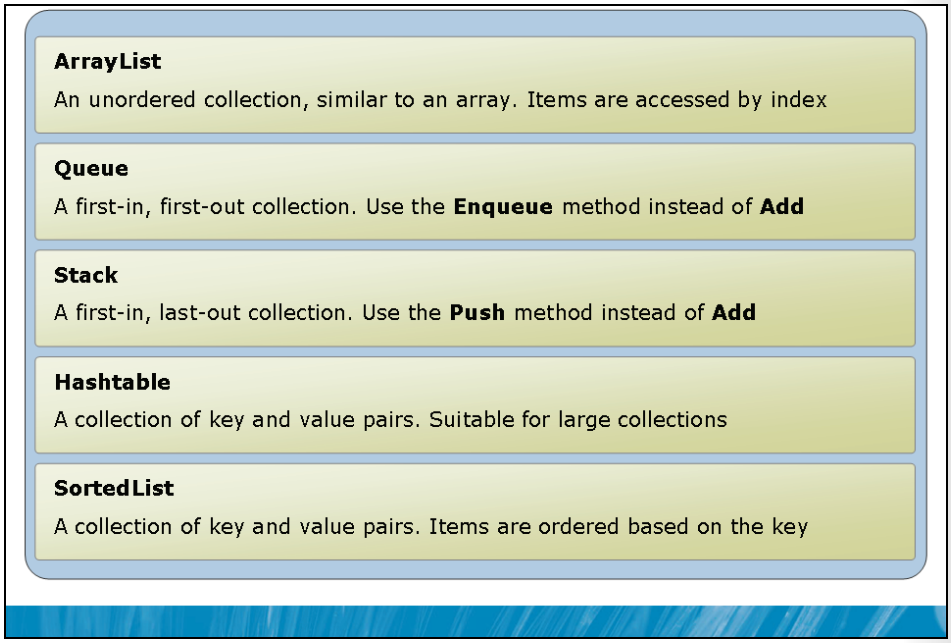
```
foreach(<type> <control_variable> in <collection>)
{
    <foreach_statement_body>
}
```

You define a control variable and specify the type of data in the collection. The control variable is set to each item in the collection in turn, and the statements in the body of the **foreach** loop are performed for each item. The scope of the control variable is the **foreach** statement. Note that it is important to specify the same type as the type of data in the collection; the compiler automatically generates code to cast the data that is retrieved from the collection to this type. If you specify the wrong type, your code will throw an **InvalidCastException** exception at run time.

Question: When you use a **foreach** statement with a collection based on the **SortedList** type, in what order will the **foreach** statement return items from the collection?

MCT USE ONLY. STUDENT USE PROHIBITED

Common Collection Classes



Key Points

The **System.Collections** namespace contains several general-purpose collection classes. Each of these collection classes is optimized to implement a specific mechanism for aggregating and accessing data.

The ArrayList Collection Class

The **ArrayList** collection class is similar to an array. You can add items to the array and retrieve items by using a zero-based index.

The **ArrayList** class dynamically increases in size as you add values to the collection. You can use the **Capacity** property to get or set the current size of the collection. The **ArrayList** class does not automatically shrink when you remove items from the collection. If you remove a significant number of items from the collection, you can use the **TrimToSize** method to reduce the size of the collection, or alternatively, you can set the **Capacity** property to a lower value.

The following code example shows a simple use of the **ArrayList** class.

MCT USE ONLY. STUDENT USE PROHIBITED

```
// Create a new ArrayList object.
ArrayList al = new ArrayList();
// Add values to the ArrayList collection.
al.Add("Value");
al.Add("Value 2");
al.Add("Value 3");
al.Add("Value 4");

// Remove a specific object from the ArrayList collection.
al.Remove("Value 2"); // Removes "Value 2"

// Remove an object from a specified index.
al.RemoveAt(2); // Removes "Value 4"

// Retrieve an object from a specified index.
string valueFromCollection = (string)al[1]; // Returns "Value 3"
```

The Queue Collection Class

The **Queue** class is a FIFO data structure. Rather than expose **Add** and **Remove** methods, the **Queue** class exposes **Enqueue** and **Dequeue** methods. When you use the **Enqueue** method on an object, it is automatically added to the end of the collection; when you use the **Dequeue** method on an object, it is automatically removed from the start of the collection. You can also use the **Peek** method to retrieve the first item in the queue without removing it.

The **Queue** class grows automatically as objects are added to the collection. If you need to recover memory from the **Queue** object, by reducing the size of the collection, you can use the **TrimToSize** method.

The following code example shows a simple use of the **Queue** class.

```
// Create a new Queue object.
Queue queue = new Queue();

// Add values to the Queue collection.
queue.Enqueue("Value");
queue.Enqueue("Value 2");
queue.Enqueue("Value 3");
queue.Enqueue("Value 4");

// Retrieve an object from the Queue collection.
string valueFromCollection = (string)queue.Dequeue();
                             // Returns "Value"
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The Stack Collection Class

The **Stack** class is a FILO data structure. The **Stack** class exposes **Push** and **Pop** methods to add and remove items. When you use the **Push** method to add an object to a **Stack** collection, the object is added to the start of the collection; when you use the **Pop** method on an object, it is automatically removed from the start of the collection and returned. Like the **Queue** class, the **Stack** class provides the **Peek** method to return the item at the start of the **Stack** collection without removing it.

The **Stack** class grows automatically as objects are added to the collection. If you need to recover memory from the **Stack** object, by reducing the size of the collection, you can use the **TrimToSize** method.

The following code example shows a simple use of the **Stack** class.

```
// Create a new Stack object.
Stack stack = new Stack();

// Add values to the Stack collection.
stack.Push("Value");
stack.Push("Value 2");
stack.Push("Value 3");
stack.Push("Value 4");

// Retrieve a value from the Stack  collection without
// removing it from the Stack collection.
string peekValueFromCollection = (string)stack.Peek();
                                 // Returns "Value 4"

// Retrieve an object from the Stack collection.
string valueFromCollection = (string)stack.Pop();
                            // Returns "Value 4"

// Retrieve another object from the Stack collection.
string valueFromCollection2 = (string)stack.Pop();
                             // Returns "Value 3"
```

The Hashtable Collection Class

The **Hashtable** class enables you to store key and value pairs in a rapid access collection. When you add an item to a **Hashtable** class by using the **Add** method, you must provide both a key and a value. The key must be unique in the collection, but the value can be a duplicate. The **Hashtable** class stores objects based on the hash value of the key.

You retrieve a value from a **Hashtable** class by using an indexer and specifying the key of the value that you want to retrieve. The **Hashtable** class hashes the key to

identify the location of the required value. The **Hashtable** class is significantly faster than other collections for retrieving an item from a large collection because it searches fewer items to locate the correct value. However, for smaller collections, the overhead of generating a hash every time you add a value to the collection can actually reduce the performance of your application. Therefore, for smaller collections, you should consider using another collection class.

The **Hashtable** class relies on creating hashes of values that are added to the collection; for this reason, you can only add keys to the collection where the type of the key implements the **GetHashCode** method. Every object includes a default implementation of the **GetHashCode** method, which is inherited from the **System.Object** class. However, you may often need to add a more complex hashing algorithm to any types that you develop. When you develop a hashing algorithm for use with a **Hashtable** collection class, you should use a case-insensitive algorithm.

The following code example shows a simple use of the **Hashtable** class.

```
// Create a new Hashtable object.
Hashtable hashtable = new Hashtable();

// Add values to the Hashtable collection.

hashtable.Add("Key A", "Value");
hashtable.Add("Key B", "Value 2");
hashtable.Add("Key C", "Value 3");
hashtable.Add("Key D", "Value 4");

// Remove an item from the Hashtable collection by specifying the key.

hashtable.Remove("Key C");

// Retrieve an item from the Hashtable collection
// by specifying the key.

string valueFromCollection = (string)hashtable["Key B"];
                             // Returns "Value 2""
```

The SortedList Collection Class

Like a **Hashtable** class, the **SortedList** collection class stores a collection of key/value object pairs. However, values in the collection are sorted by using the key. If you iterate through the data in a **SortedList** collection, the data will be presented in key order.

The following code example shows a simple use of the **SortedList** class.

MCT USE ONLY. STUDENT USE PROHIBITED

```
// Create a new SortedList object.
SortedList sortedList = new SortedList();

// Add values to the SortedList collection.
sortedList.Add("Key A", "Value");
sortedList.Add("Key B", "Value 2");
sortedList.Add("Key C", "Value 3");
sortedList.Add("Key D", "Value 4");

// Remove an item from the SortedList by specifying the key.
sortedList.Remove("Key C");

// Retrieve an item from the SortedList collection
// by specifying the key.
string valueFromCollection = (string)sortedList["Key B"];
                             // Returns "Value 2".

// Retrieve an item from the SortedList collection
// by specifying the index.
string valueFromCollection2 = (string)sortedList.GetByIndex(0);
                              // Returns "Value".
```

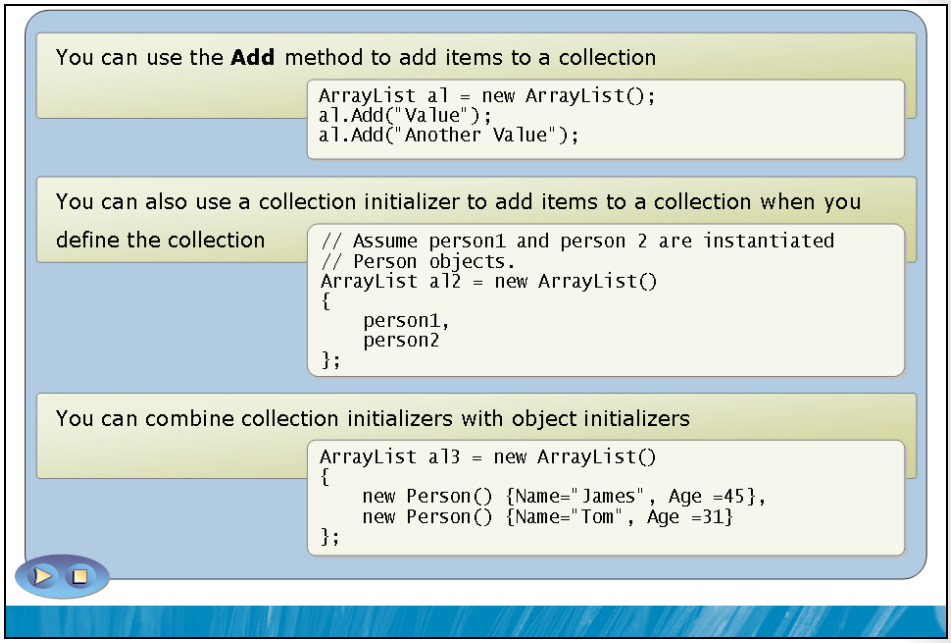
Question: When would you use a **Hashtable** collection?

Additional Reading

For more information about the common collection types, see the Commonly Used Collection Types page at http://go.microsoft.com/fwlink/?LinkId=192963.

MCT USE ONLY. STUDENT USE PROHIBITED

Using Collection Initializers



Key Points

When you create an instance of a collection class, you typically use the **Add** method to add items to the collection. If you need to add items to the collection as soon as you have created it, you may often end up with code that resembles the following code example.

```
ArrayList al = new ArrayList();
al.Add("Value");
al.Add("Another Value");
```

An alternative to writing several statements is to use a collection initializer. A collection initializer has a similar syntax to an object initializer; you define the collection type and then you add values to the collection in braces, separated by commas, before the semicolon. The following code example shows how you can use a collection initializer instead of using the **Add** method.

```
ArrayList al = new ArrayList() {"Value", "Another Value"};
```

MCT USE ONLY. STUDENT USE PROHIBITED

You can combine collection initializers with object initializers to add new objects to a collection. The following code example shows how you can combine object initializers with collection initializers.

```
ArrayList al2 = new ArrayList()
{
    new Person() {Name="James", Age =45},
    new Person() {Name="Tom", Age =31}
};
```



Note: You can only use collection initializers with collection classes that expose an **Add** method. The compiler uses the **Add** method to add objects that are specified in the collection initializer to the collection. Collection classes such as the **Queue** class, which does not expose an **Add** method, do not support collection initializers.

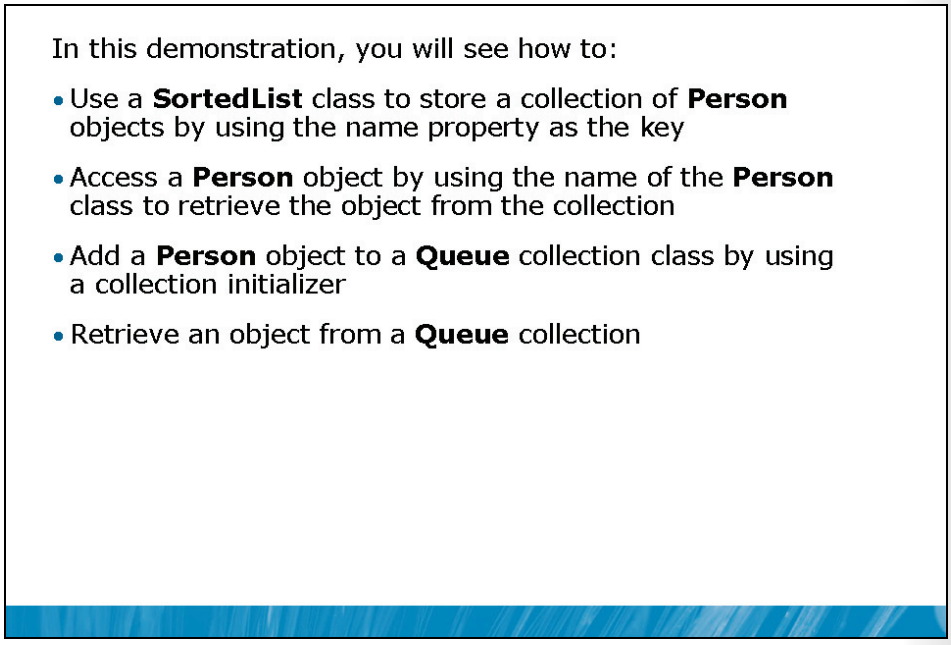
Question: Can you use a collection initializer with the **Stack** collection class?

Additional Reading

For more information about collection initializers, see the Object and Collection Initializers (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192964.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Using Collections



Key Points

In this demonstration, you will see how to:

- Use a **SortedList** class to store a collection of **Person** objects by using the name property as the key.
- Access a **Person** object by using the name of the **Person** class to retrieve the object from the collection.
- Add a **Person** object to a **Queue** collection class by using a collection initializer.
- Retrieve an object from a **Queue** collection.

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.

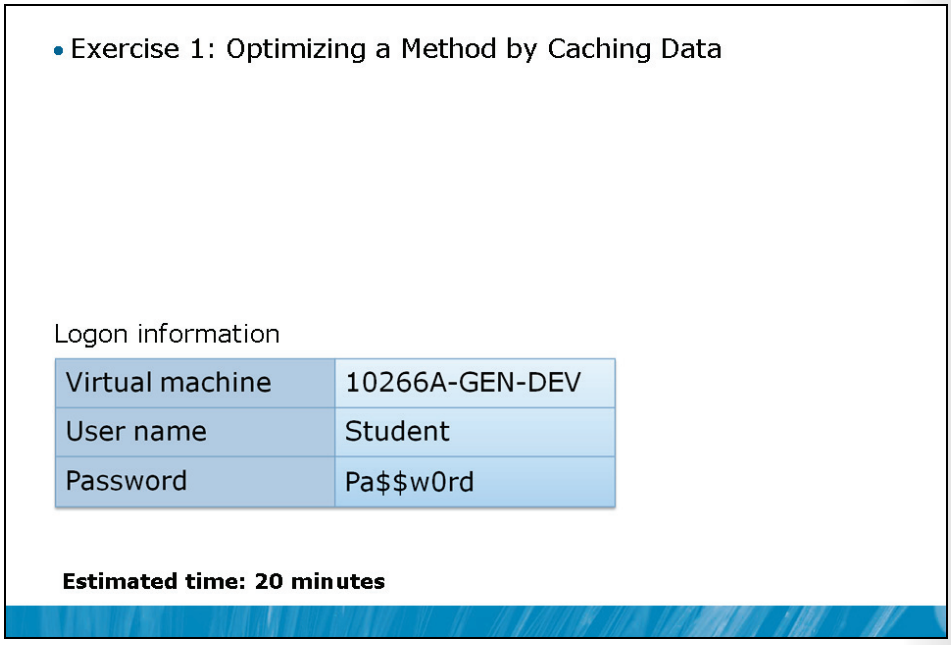
MCT USE ONLY. STUDENT USE PROHIBITED

3. Open the Module 12, Demo1 starter solution.
4. Open the Program.cs file, and then review the **Program** class.
5. Uncomment the code that creates a new **SortedList** collection named **people**.
6. Uncomment the code that adds a **Person** object named **Richard** to the **people** collection. This code uses an object initializer.
7. Uncomment the code that creates a **Person** object named **louisa**.
8. Uncomment the code that adds the **louisa** object to the **people** collection. This uses the **Add** method to add an existing item to an existing collection.
9. Uncomment the code that retrieves a **Person** object from the **people** collection by using the name as an indexer. Note the cast from the **object** type to the **Person** type.
10. Uncomment the code that checks whether the **personFromCollection** field is **null** and, if it is not **null**, writes the information to the screen.
11. Uncomment the code that iterates through every item in the **people** collection. Run the application.
12. When the application pauses, note the data that is returned from the collection and displayed on the screen, and then press ENTER.
13. When the application pauses again, note that the details of two people are displayed on the screen, and then press ENTER.
14. Close the application.
15. Close Visual Studio.

Question: What is the advantage of using the **SortedList** class compared to using a multidimensional array?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab A: Using Collections



Objectives

After completing this lab, you will be able to use collection classes in applications that you develop.

Introduction

In this lab, you will use a collection to cache data and optimize an algorithm that a method implements.

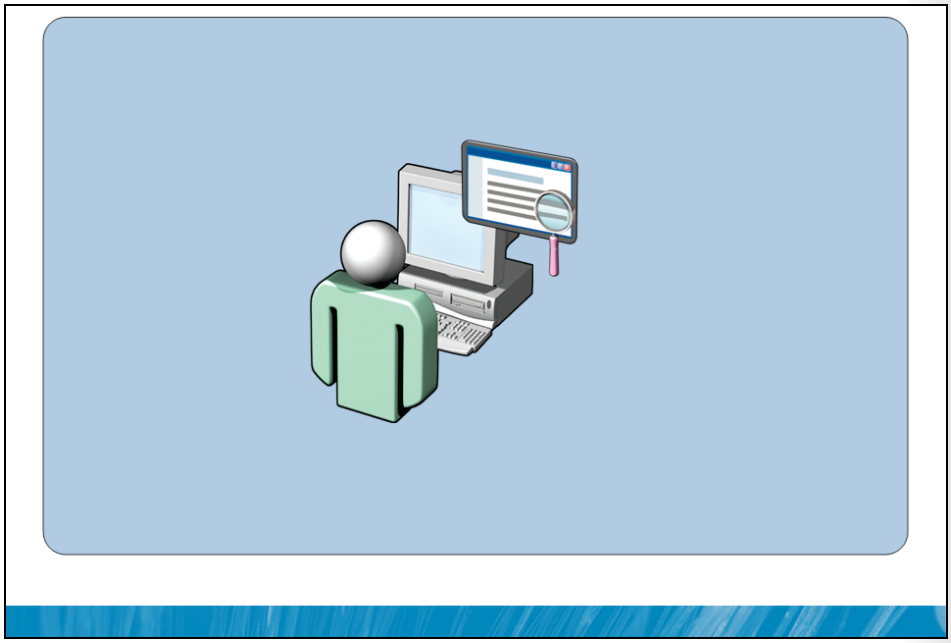
Lab Setup

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



In an earlier project, you worked on the software for an engineering device that solved simultaneous equations by using Gaussian elimination. The functionality is fine, but the performance needs to be improved. Analysis has shown that the device frequently provides the same sets of input variables , so you have decided to add a caching capability.

Exercise 1: Optimizing a Method by Caching Data

In this exercise, you will use a **Hashtable** collection to implement the memorization pattern in the Gaussian elimination method. When the calculation is complete, the result is stored in the **Hashtable** collection together with details of the parameters (the key will be a hash of the parameters, and the data will be a structure that holds the parameter values and the calculated result) before the method returns the result to the caller. If the method is called subsequently, the method first checks the **Hashtable** collection to determine whether the same parameter values have been used before. If they have, the method returns the result from the **Hashtable** collection rather than performing the lengthy, possibly time-consuming calculation.

The main tasks for this exercise are as follows:

MCT USE ONLY. STUDENT USE PROHIBITED

1. Open the Collections solution.
2. Modify the **Gauss** class to implement the memorization pattern.
3. Test the solution.

▶ Task 1: Open the Collections 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 Collections solution in the E:\Labfiles\Lab 12\Lab A\Ex1\Starter folder.

▶ Task 2: Modify the Gauss class to implement the memorization pattern

1. In the TestHarness project, display the MainWindow.xaml window.

   The MainWindow window implements a simple test harness to enable you to test the method that you will use to perform Gaussian elimination. This is a Windows® Presentation Foundation (WPF) application that enables a user to enter the coefficients for four simultaneous equations that consist of four variables (w, x, y, and z). It then uses Gaussian elimination to find a solution for these equations. The results are displayed in the lower part of the screen.
2. Review the task list.
3. In the task list, locate the **TODO - Add a static Hashtable** task, and then double-click this task.

   This task is located in the GaussianElimination project, in the **Gauss** class.
4. At the top of the Gauss.cs file, at the end of the list of **using** statements, add a statement to bring the **System.Collections** namespace into scope.
5. Remove the comment, and then add code to define a static **Hashtable** object named **results**.
6. At the beginning of the **SolveGaussian** method, before the statements that create a deep copy of the parameters, add code to ensure that the **results Hashtable** object is initialized. Create a new instance of this object if it is currently **null**.

MCT USE ONLY. STUDENT USE PROHIBITED

7. Add code to generate a hash key that is based on the method parameters by performing the following tasks:
   a. Define a new **StringBuilder** object named **hashString**.
   b. Iterate through the **coefficients** array, and append each value in the array to the **hashString StringBuilder** object.
   c. Iterate through the **rhs** array, and append each value in the array to the **hashString StringBuilder** object.
   d. Define a new **string** object named **hashValue**, and initialize it to the value that the **hashString.ToString** method returns.



Hint: This procedure generates a hash key by simply concatenating the values that are passed into the method. You can use more advanced hashing algorithms to generate better hashes. The **System.Security.Cryptography** namespace includes many classes that you can use to implement hashing.

8. Add code to check whether the **results** object already contains a key that has the value in the **hashValue** string. If it does, return the value that is stored in the **Hashtable** collection class that corresponds to the **hashValue** key. If the **results** object does not contain the **hashValue** key, the method should use the existing logic in the method to perform the calculation.



Hint: A **Hashtable** object stores and returns values as objects. You must cast the value that is returned from a **Hashtable** object to the appropriate type before you work with it. In this case, cast the returned value to an array of **double** values.

9. In the task list, locate the **TODO - Store the result of the calculation in the Hashtable** task, and then double-click this task.

   This task is located near the end of the **SolveGaussian** method.

10. Remove the comment, and then add code to the method to store the **rhsCopy** array in the **Hashtable** object, specifying the **hashValue** object as the key.

▶ Task 3: Test the solution

1. Build the solution and correct any errors.
2. Run the application.

MCT USE ONLY. STUDENT USE PROHIBITED

3. 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.

   - $2w + x - y + z = 8$
   - $-3w - x + 2y + z = -11$
   - $-2w + x - 2y = -3$
   - $3w - x + 2y - 2z = -5$

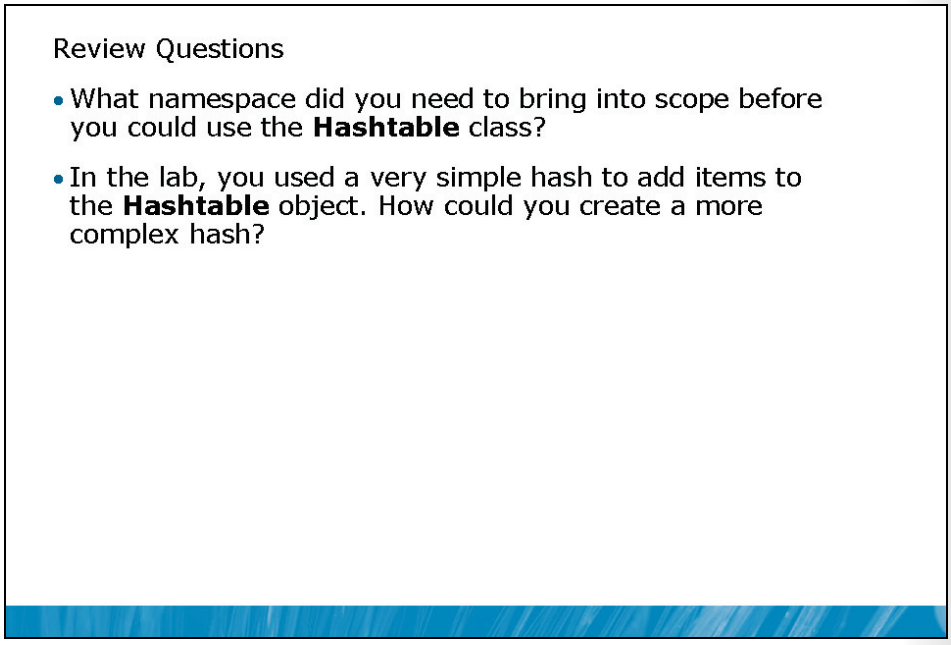
   Observe that the operation takes approximately five seconds to complete.

4. Verify that the following results are displayed:
   - $w = 4$
   - $x = -17$
   - $y = -11$
   - $z = 6$
5. Modify the third equation to match the following equation, and then click **Solve** again:
   - $-2w + x - 2y$ **$+ 3z$** $= -3$

   Observe that this operation also takes approximately five seconds to complete.
6. Verify that the following results are displayed:
   - $w = -2$
   - $x = 25$
   - $y = 7$
   - $z = -6$
7. Undo the change to the third equation so that all of the equations match those in Step 3, and then click **Solve**. Observe that this time, the operation takes much less time to complete because it uses the solution that was generated earlier.
8. Close the application, and then close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

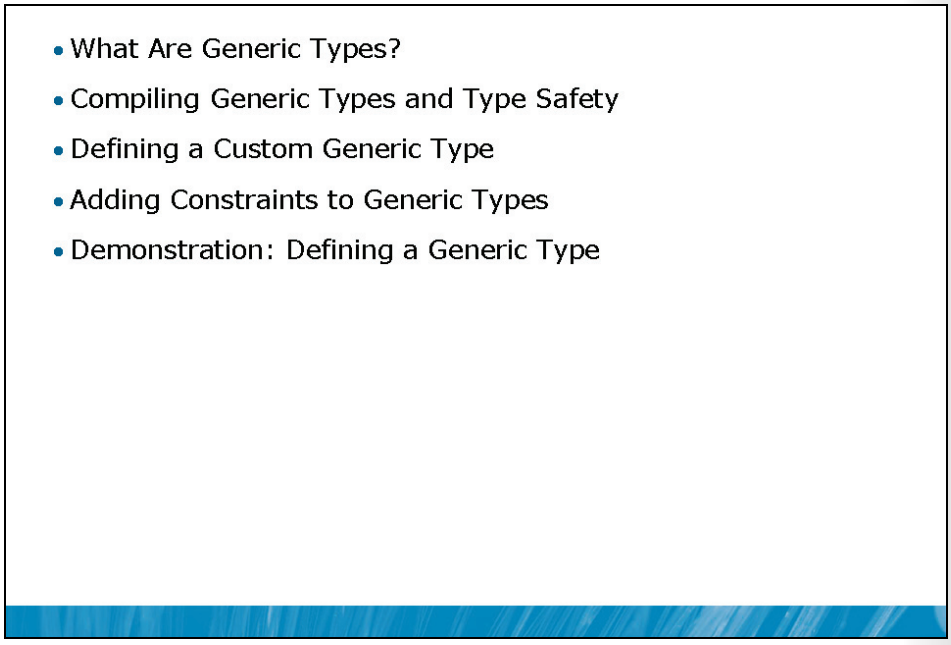


Review Questions

1. What namespace did you need to bring into scope before you could use the **Hashtable** class?
2. In the lab, you used a very simple hash to add items to the **Hashtable** object. How could you create a more complex hash?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Creating and Using Generic Types



The collection classes in the **System.Collections** namespace have one major problem: they are not type-safe. The collection classes store references to **System.Object** objects, rather than instances of a particular type. This risks introducing bugs into your applications because you might attempt to use invalid casts that the compiler cannot detect, and that cause run-time exceptions in your application. C# includes generic types that enable you to specify which type you want to use with a class. Using generic classes, you can ensure type safety while still developing a flexible solution. This lesson introduces generic types and how you can use them in your applications.

Objectives

After completing this lesson, you will be able to:

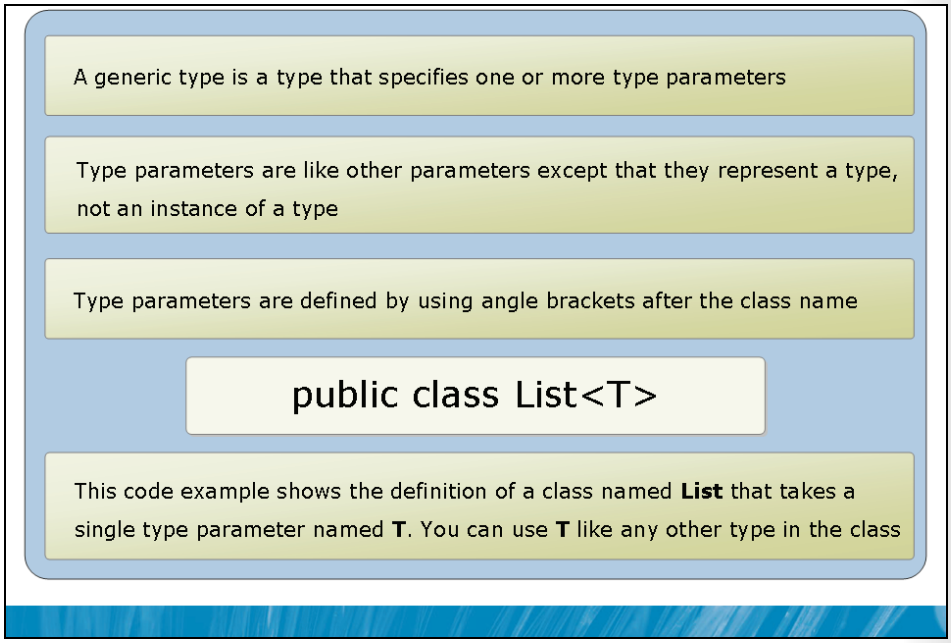
- Describe the purpose of generic types.
- Explain how the compiler uses generic types to implement type safety.

MCT USE ONLY. STUDENT USE PROHIBITED

- Define a generic type.
- Specify constraints for a type parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Generic Types?



Key Points

The basic collection classes do not enable you to specify the type of data that the collection can hold, which can lead to many problems. The code in the following code example will compile, but when you run it, it will throw an **InvalidCastException** exception.

```
ArrayList names = new ArrayList();
names.Add("Alice");
...
int data = (int)names[0];
```

There is also the issue of performance. The built-in collections all hold items as the **System.Object** type. You can use the **Object** type to refer to any reference type with little overhead, but if you store value types in a collection, the compiler generates code to box the data. When you retrieve a value type from the collection, the compiler generates code to unbox the data. Boxing and unboxing can impose a significant overhead.

MCT USE ONLY. STUDENT USE PROHIBITED

To work around these limitations, you need to be able to define collection classes that specify the type of the items that they store. You can do this by using generics.

Generics

A generic type is a type that specifies one or more type parameters. A type parameter is similar to a method parameter except that it specifies a type rather than a value. You specify a type parameter when you define a class by using angle brackets. When you instantiate an object based on a generic type, you specify a type to substitute for the type parameter. The .NET Framework defines generic versions of some of the collection classes in the **System.Collections.Generics** namespace, including the **List** type as shown in the following code example.

```
public class List<T>
```

The next code example shows how you can use the generic **List** collection. The **List** type represents a strongly typed list of objects that you can access by using an index. The type parameter for the **ages** collection is an **int**, and you can only store **int** values in this collection. If you attempt to store a different type, the compiler will spot the problem and fail to build the application. In addition, you can retrieve the data from the collection without using a cast.

```
List<int> ages = new List<int>();
ages.Add(10);
ages.Add(25);
...
int data = ages[0];  // No cast necessary
...
ages.Add("Data");    // Compiler error
```

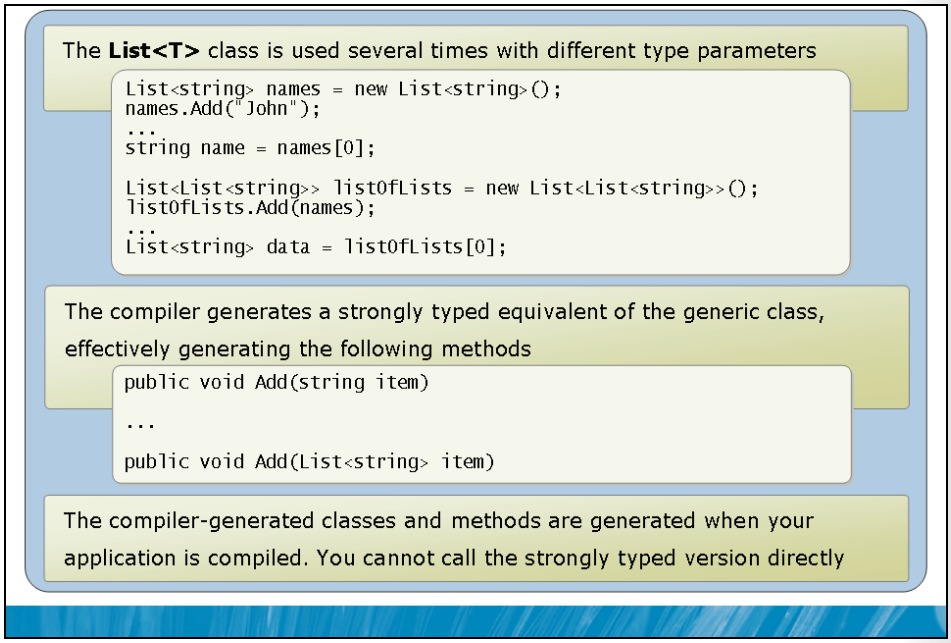
Question: What are the benefits of using a generic class compared to a nongeneric class?

Additional Reading

For more information about generics, see the Generics (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192965.

MCT USE ONLY. STUDENT USE PROHIBITED

Compiling Generic Types and Type Safety



Key Points

The replacement of a type parameter for a specified type is not simply a textual replacement mechanism. Instead, the compiler performs a complete semantic substitution. In the case of the **List<T>** type, you can specify any valid type for the type parameter **T**, as the following code example shows.

```
List<string> names = new List<string>();
names.Add("John");
...
string name = names[0];

List<List<string>> listOfLists = new List<List<string>>();
listOfLists.Add(names);
...
List<string> data = listOfLists[0];
```

The first example creates a **List** collection of string objects, while the second creates a **List** collection that contains **List** collections of **string** objects. For the names variable, the compiler also generates the version of the **Add** method in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public void Add(string item);
```

For the listOfLists variable, the compiler generates a different version of the **Add** method, as the following code example shows.

```
public void Add(List<string> item);
```

This means that you cannot call the **Add** method of either **List** object and pass a parameter that has the wrong type without the compiler generating an error. The same rationale applies to all of the other methods and properties that the generic **List** class implements. Effectively, when you define a variable by using a generic type, you create an entirely new type, and the compiler generates strongly typed methods and properties for this type.

This also affects performance when you use a value type as the type parameter. In the following code example, the compiler generates yet another version of the **Add** method that takes an *int* parameter.

```
List<int> ages = new List<int>();
ages.Add(10);

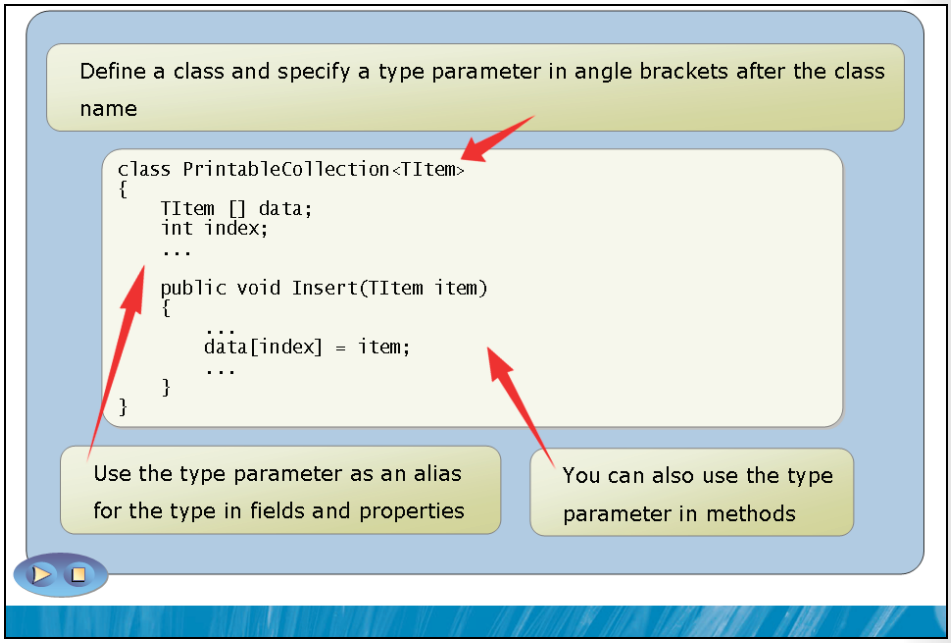
// The compiler generates the following method:
// public void Add(int item);
```

When you call the **Add** method and provide an integer value, there is no need for the compiler to generate code to box or unbox this value.

Question: When the compiler compiles an application that uses a generic type, it generates a concrete version of the generic class. How can you call the concrete version in your application?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining a Custom Generic Type



Key Points

To define a custom generic type, you add one or more type parameters to the type definition. To define type parameters, you add angle brackets immediately after the class name. In the angle brackets, you specify names for each of the type parameters. You can use as many type parameters as necessary in a comma-delimited list. You can then use the names that you specified for your type parameters instead of concrete types in your class. The following code example shows a custom generic type called **PrintableCollection** that uses a type parameter called **TItem**. The intention is that the **PrintableCollection** type defines a collection of items that can be easily formatted for printing and display purposes. The **PrintableCollection** class defines a method called **Insert** that takes a parameter of type **TItem**. Internally, the class uses **TItem** just like any other type. In this case, it creates an array of **TItem** objects, and the **Insert** method inserts the value that is specified by its parameter into this array.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class PrintableCollection<TItem>
{
    TItem [] data;
    int index;

    ...

    public void Insert(TItem item)
    {
        ...
        data[index] = item;
        ...
    }

}
```

You can use your custom generic type by providing a type for the type parameter. The following code example shows how to use the generic **PrintableCollection** class to create and use a printable collection of **Person** structs.

```
struct Person
{
    ...
}

...

PrintableCollection<Person> employeeList =
    new PrintableCollection<Person>();

Person employee = new Person(...);

employeeList.Insert(employee);
```



Note: You can define generic structs in addition to generic classes.

The default Keyword

It is always good practice to initialize members of a type, and the same is true when you define a generic type. You typically achieve this by providing one or more constructors. However, if you want to initialize a member based on a type parameter to a default value, it is not always easy to specify what that default value should be. C# provides the **default** keyword, which you can use to address this problem, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class PrintableCollection<TItem>
{
    TItem [] data;
    int index;
    TItem tempData;
    ...

    public PrintableCollection()
    {
        this.tempData = default(TItem);
        ...
    }
    ...
}
```



Note: Do not include the type parameter in the name of constructors for a managed type.

The **default** keyword generates a meaningful default value for a type that is specified as a type parameter. When the compiler generates the code for a generic type, it replaces the default construct with a default value that depends on the concrete type of the member. The default value will be **null** for reference types and zero for numeric value types. For structs, the **default** keyword will initialize each member of the struct to zero or **null** depending on whether they are value or reference types.

Question: How can you instantiate a variable when you do not know the type and cannot be certain that the type exposes a constructor?

MCT USE ONLY. STUDENT USE PROHIBITED

Adding Constraints to Generic Types

Constraint	Description
where T: struct	The type argument must be a value type
where T : class	The type argument must be a reference type
where T : new()	The type argument must have a public default constructor
where T : <base class name>	The type argument must be, or derive from, the specified base class
where T : <interface name>	The type argument must be, or implement, the specified interface
where T : U	The type argument that is supplied for **T** must be, or derive from, the argument that is supplied for **U**

Key Points

When you define a generic type, you may need to restrict the types that can be used for the type parameters to ensure that they match certain criteria or conform to specific requirements. For example, if you define the **PrintableCollection** class that was shown in the previous topic, you can specify that this class can only be used with types that know how to format and print themselves by implementing an interface called **IPrintable** that defines a method called **Print**.

In C#, you can use constraints to restrict the types that you can use with your generic classes and other types. To add a constraint, you use the **where** keyword after the class definition. The following code example shows how to specify that the type parameter for the **PrintableCollection** type must implement the **IPrintable** interface.

```
class PrintableCollection<TItem> where TItem : IPrintable
{
    TItem [] data;
    ...
    public void PrintItems()
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        foreach (TItem item in data)
        {
            item.Print(); // item implements IPrintable, so it is safe
                          // to call the Print method.
        }
    }
}
```

You can add a **where** clause for each type parameter that you define, and you can use multiple constraints on each type. You define multiple constraints by separating each constraint with a comma, like a class implementing multiple interfaces.

The following table summarizes the clauses that you can use to constrain a generic type.

Constraint	Description
where T: struct	The type argument must be a value type. Any value type except **Nullable** can be specified.
where T : class	The type argument must be a reference type; this also applies to any class, interface, delegate, or array type.
where T : new()	The type argument must have a public default constructor. When it is used together with other constraints, the **new()** constraint must be specified last.
where T : <base class name>	The type argument must be, or derive from, the specified base class.
where T : <interface name>	The type argument must be, or implement, the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic.
where T : U	The type argument that is supplied for **T** must be, or derive from, the argument that is supplied for **U**. This is called a naked type constraint.

The following code example shows how to constrain two type parameters. The **T** type parameter must implement the **IComparable** and **IDisposable** interfaces. The **Y** type parameter must directly or indirectly derive from **T** (or be the same as **T**), and provide a default constructor.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class MyClass<T>
    where T : IComparable, IDisposable
    where Y : T, new()
{
}
```

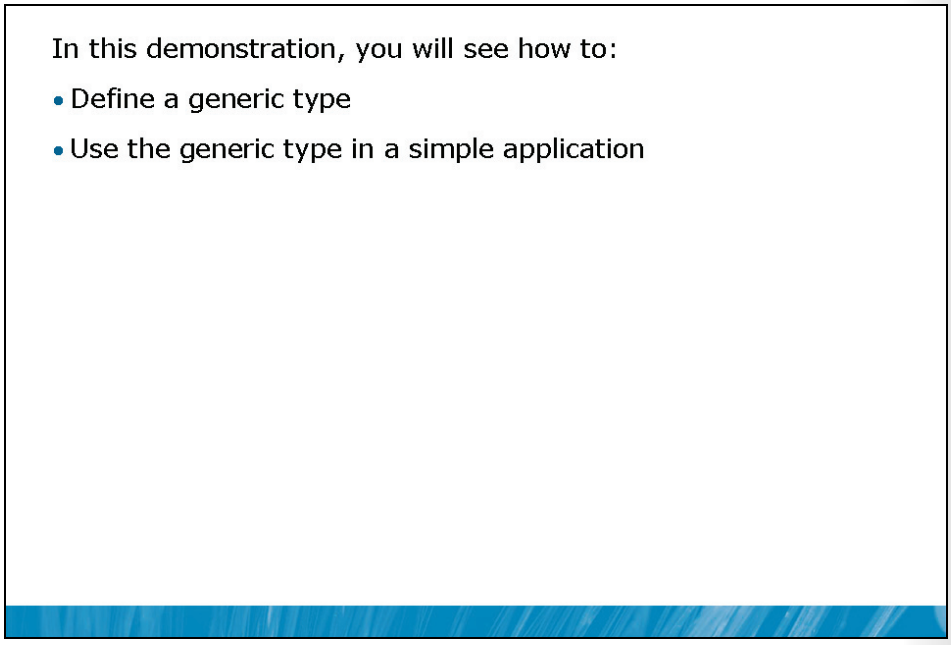
Question: How can you ensure that, when an instance of a generic class is created, a reference type is used for the type parameter?

Additional Reading

For more information about constraints, see the Constraints on Type Parameters (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192966.

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Defining a Generic Type



Key Points

In this demonstration, you will see how to:

- Define a generic type.
- Use the generic type in a simple application.

Demonstration Steps

1. Start Visual Studio.
2. Open the Module 12, Demo2 starter solution.
3. Open the Printer.cs file, and then review the **Printer** class.

   The **Printer** class is a generic class that takes a single type parameter named **DocumentType**. The purpose of the class is to represent a printer that is capable of printing a specific type of document.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Uncomment the code that creates a new **Queue<>** object named **printQueue** by using the **DocumentType** type parameter to specify the type of the **Queue** object. This collection class stores items in a FIFO manner.
5. Uncomment the **AddDocumentToQueue** method. This method uses the generic type parameter to define the types that can be used as a parameter.
6. Uncomment the **PrintDocuments** method. This method removes items from the queue and calls the **Print** method on each item.
7. Open the Program.cs file, and then review the **Program** class.

   The **Program** class currently creates three **Report** objects and three **ReferenceGuide** objects. The **Report** and **ReferenceGuide** classes both implement the **IPrintable** interface, but are not related in any other way.
8. Uncomment the code that creates a new instance of the **Printer** class by specifying the **Report** type and adds three reports to the print queue.
9. Uncomment the code that calls the **PrintDocuments** method on the **reportPrinter** object. This calls the method from the **Printer** class and removes each item from the print queue.
10. Uncomment the code that creates a new instance of the **Printer** class by specifying the **ReferenceGuide** type and adds three reference guides to the print queue.
11. Uncomment the code that calls the **PrintDocuments** method on the **referenceGuidePrinter** object. This calls the method from the **Printer** class and removes each item from the print queue.
12. Run the application.
13. Close the application.
14. Close Visual Studio.

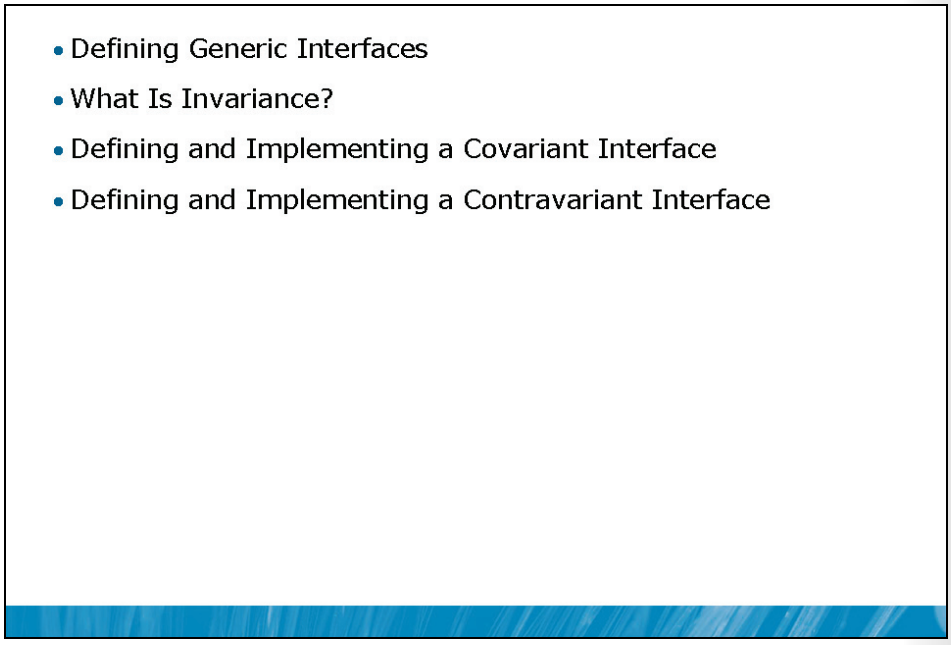
Question: What happens if you attempt to use the wrong type when you call a method that uses a generic type?

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 3

Defining Generic Interfaces and Understanding Variance



You can use generics to define interfaces in addition to classes and structs. However, you must remember that the .NET Framework uses strong typing. When you develop a generic interface, the strong typing can be restrictive, preventing casting between seemingly compatible types that implement these interfaces. This restriction is known as *invariance*. This lesson describes the problems that invariant generic interfaces can cause, and how you can define generic interfaces that can overcome some of these restrictions by using covariance and contravariance.

Objectives

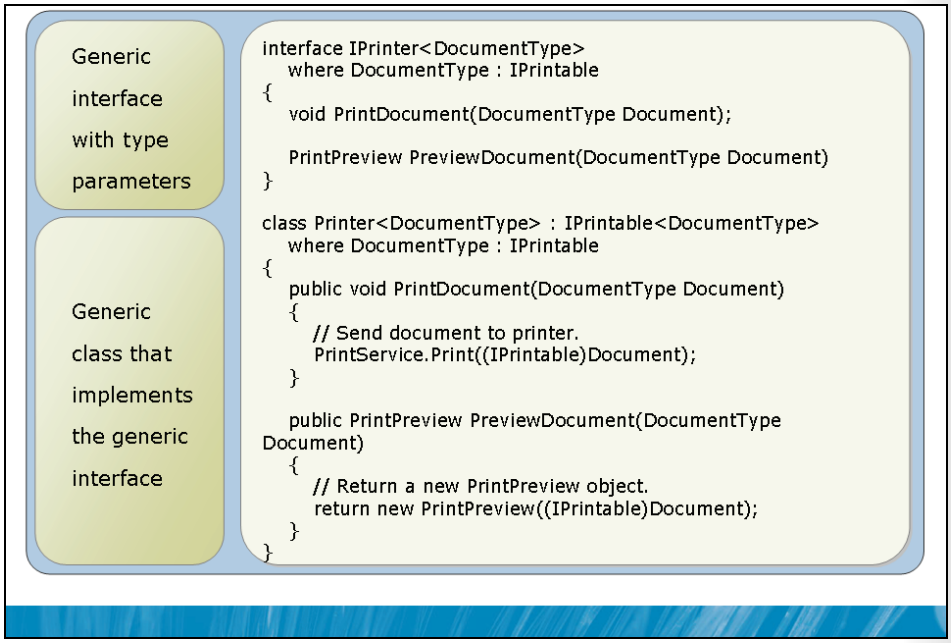
After completing this lesson, you will be able to:

- Define generic interfaces.
- Explain invariance and how it affects the way in which you can assign references to objects.

- Describe how to implement covariance.
- Describe how to implement contravariance.

MCT USE ONLY. STUDENT USE PROHIBITED

Defining Generic Interfaces



Key Points

In addition to defining generic classes and structs, you can also define generic interfaces. A generic interface is much like any other interface except that you can specify type parameters, and use those types in the members that you define in the interface. Like generic classes and structs, generic interfaces can define constraints on type parameters. You can implement a generic interface in a generic class and use a generic interface to reference a class like you would with a normal class and interface.

The following code example shows a generic interface that is implemented in a generic class.

```
interface IPrinter<DocumentType>
    where DocumentType : IPrintable
{
    void PrintDocument(DocumentType Document);

    PrintPreview PreviewDocument(DocumentType Document)
}
class Printer<DocumentType> : IPrintable<DocumentType>
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    where DocumentType : IPrintable
{
    public void PrintDocument(DocumentType Document)
    {
        // Send document to printer.
        IPrintable doc = (IPrintable)Document;
        PrintService.Print(doc);
    }

    public PrintPreview PreviewDocument(DocumentType Document)
    {
        // Return a new PrintPreview object.
        IPrintable doc = (IPrintable)Document;
        return new PrintPreview(doc)
    }
}
```

Question: How can you ensure that types that are used with a generic interface can be compared?

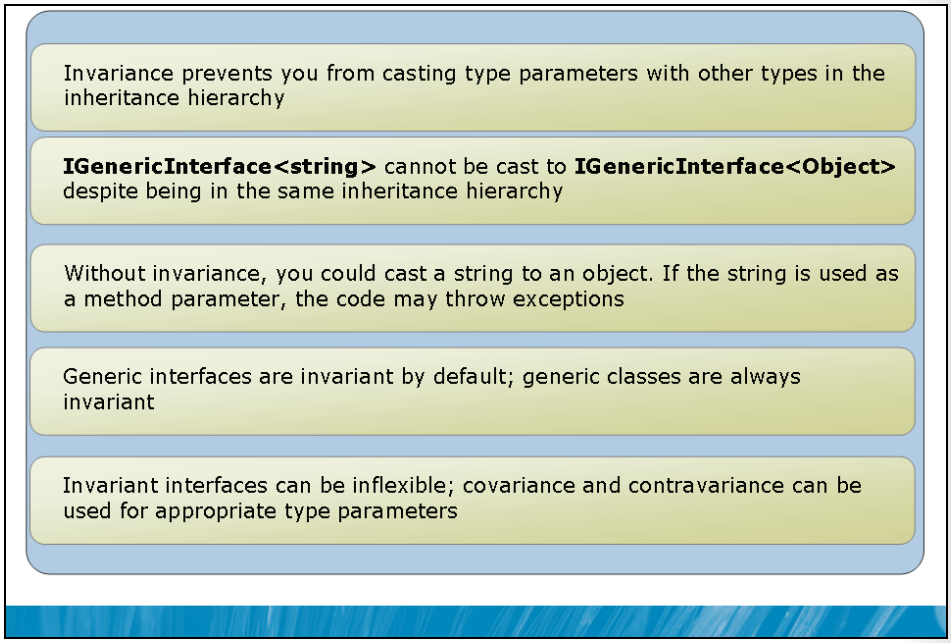
Additional Reading

For more information about generic interfaces, see the Generic Interfaces (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192967.

For more information about variant generic interfaces, see the Creating Variant Generic Interfaces (C# and Visual Basic) page at http://go.microsoft.com/fwlink/?LinkId=192968.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is Invariance?



Key Points

You have previously seen that you can use the **object** type to hold a value or reference of any other type. For example, the code in the following code example is completely legal.

```
string myString = "Hello";
object myObject = myString;
```

Remember that, in inheritance terms, the **String** class is derived from the **Object** class, so all strings are objects.

Now consider the generic interface and class in the following code example.

```
interface IWrapper<T>
{
    void SetData(T data);
    T GetData();
}

class Wrapper<T> : IWrapper<T>
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    private T storedData;

    void IWrapper<T>.SetData(T data)
    {
        this.storedData = data;
    }

    T IWrapper<T>.GetData()
    {
        return this.storedData;
    }

}
```

The **Wrapper<T>** class provides a simple wrapper around a specified type. The **IWrapper** interface defines the **SetData** method that the **Wrapper<T>** class implements to store the data, and the **GetData** method that the **Wrapper<T>** class implements to retrieve the data. You can create an instance of this class and use it to wrap a string as shown in the following code example.

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine("Stored value is {0}",
storedStringWrapper.GetData());
```

The code creates an instance of the **Wrapper<string>** type. It references the object through the **IWrapper<string>** interface to call the **SetData** method. (The **Wrapper<T>** type implements its interfaces explicitly, so you must call the methods through an appropriate interface reference.) The code also calls the **GetData** method through the **IWrapper<string>** interface. If you run this code, it generates the message "Stored value is Hello".

Now look at the line of code in the following code example.

```
IWrapper<object> storedObjectWrapper = stringWrapper;
```

This statement is similar to the one that creates the **IWrapper<string>** reference in the previous code example, the difference being that the type parameter is **object** rather than **string**. Is this code legal? Remember that all strings are objects (you can assign a string value to an object reference, as shown earlier). However, if you try it, the statement will fail to compile with the message "Cannot implicitly convert type 'Wrapper<string>' to 'IWrapper<object>'.".

You can try an explicit cast such as the one in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
IWrapper<object> storedObjectWrapper =
    (IWrapper<object>)stringWrapper;
```

This code compiles, but fails at run time with an **InvalidCastException** exception. The problem is that, although all strings are objects, the converse is not true; not all objects are strings. If this statement was allowed, you could write similar code to that shown in the following code example, which ultimately attempts to store a **Window** object in a **string** field (the **Window** type is the class used to build a window in a WPF application).

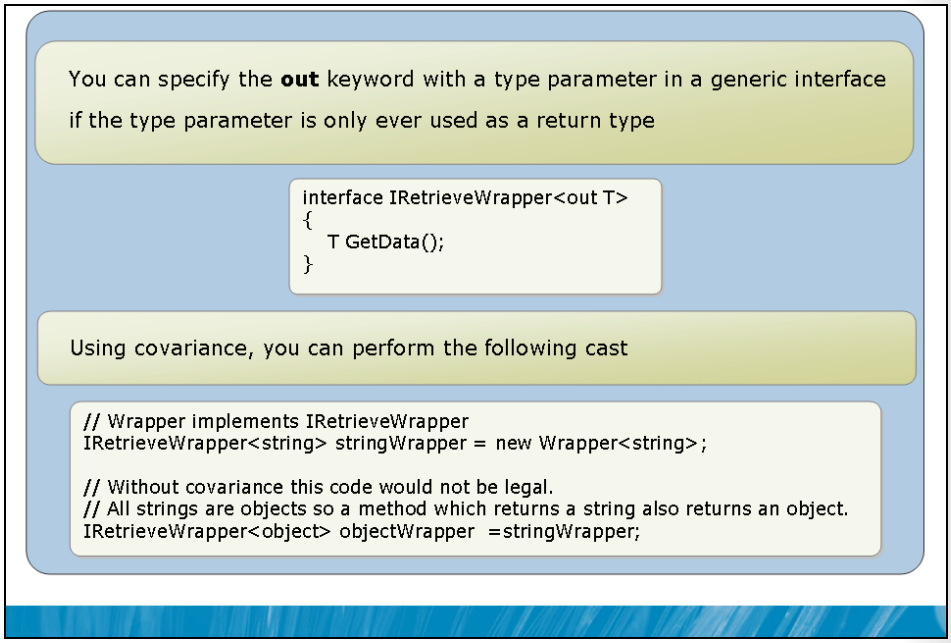
```
IWrapper<object> storedObjectWrapper =
    (IWrapper<object>)stringWrapper;
Window myWindow = new Window();
storedObjectWrapper.SetData(myWindow);
```

The **IWrapper<T>** interface is said to be *invariant*. You cannot assign an **IWrapper<A>** object to a reference of type **IWrapper<B>**, even if type **A** is derived from type **B**. By default, C# implements this restriction to ensure the type safety of your code.

Question: When you define a generic interface, by default, is it invariant, contravariant, or covariant?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining and Implementing a Covariant Interface



Key Points

Suppose that you defined the **IStoreWrapper<T>** and **IRetrieveWrapper<T>** interfaces in place of **IWrapper<T>** and implemented these interfaces in the **Wrapper<T>** class, as shown in the following code example.

```
interface IStoreWrapper<T>
{
    void SetData(T data);
}

interface IRetrieveWrapper<T>
{
    T GetData();
}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>
{
    private T storedData;

    void IStoreWrapper<T>.SetData(T data)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        this.storedData = data;
    }

    T IRetrieveWrapper<T>.GetData()
    {
        return this.storedData;
    }
}
```

Functionally, the **Wrapper<T>** class is the same as before, except that you access the **SetData** and **GetData** methods through different interfaces, as shown in the following code example.

```
Wrapper<string> stringWrapper = new Wrapper<string>();
IStoreWrapper<string> storedStringWrapper = stringWrapper;
storedStringWrapper.SetData("Hello");
IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;
Console.WriteLine("Stored value is {0}",
retrievedStringWrapper.GetData());
```

Now, is the code in the following code example legal?

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

The quick answer is "no," and it fails to compile with the same error as before. However, although the C# compiler has deemed that this statement is not type-safe, the reasons for assuming this are no longer valid. The **IRetrieveWrapper<T>** interface only enables you to read the data that is held in the **IWrapper<T>** object by using the **GetData** method, and it does not provide any way to change the data. In situations such as this, where the type parameter occurs only as the return value of the methods in a generic interface, you can inform the compiler that some implicit conversions are legal and that it does not have to enforce strict type safety. You do this by specifying the **out** keyword when you declare the type parameter, as the following code example shows.

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

This feature is called *covariance*. You can assign an **IRetrieveWrapper<A>** object to an **IRetrieveWrapper<B>** reference as long as there is a valid conversion from type **A** to type **B**, or type **A** derives from type **B**. The code in the following code example now compiles and runs correctly.

MCT USE ONLY. STUDENT USE PROHIBITED

```
// string derives from object, so this is now legal
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

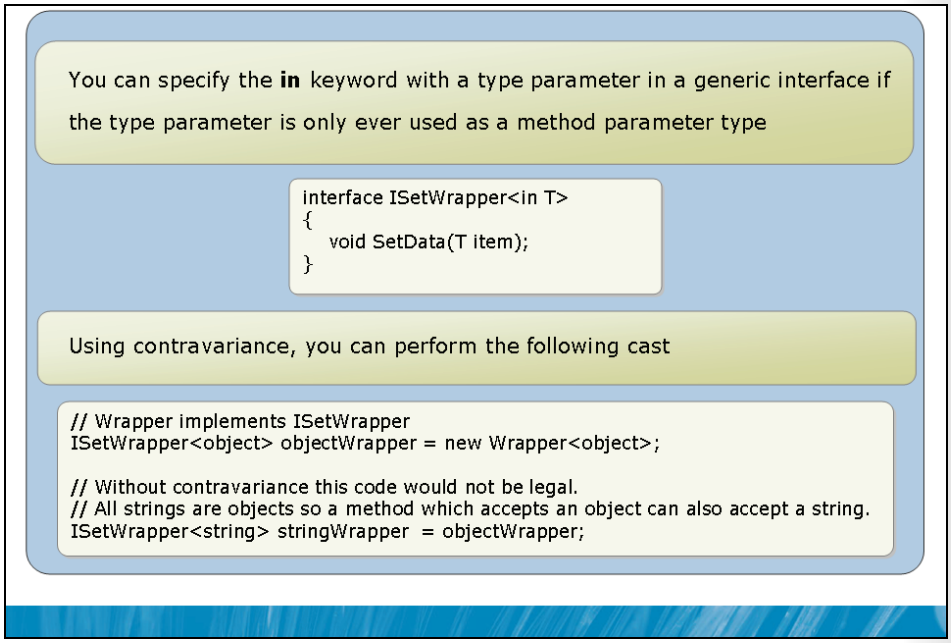
You can specify the **out** qualifier with a type parameter only if the type parameter occurs as the return type of methods. If you use the type parameter to specify the type of any method parameters, the **out** qualifier is illegal and your code will not compile. Also, covariance works only with reference types. This is because value types cannot form inheritance hierarchies. The code in the following code example will not compile because **int** is a value type.

```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper; // this is legal
...
// the following statement is not legal - ints are not objects
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```

Question: Can you use an **out** qualifier with a type parameter that is used as a parameter in a method call?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining and Implementing a Contravariant Interface



Key Points

Contravariance is the corollary of covariance. It enables you to use a generic interface to reference an object of type **B** through a reference to type **A** as long as type **B** derives from type **A**. For example, the **System.Collections.Generic** namespace in the .NET Framework provides an interface called **IComparer**, which looks like the following code example.

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

A class that implements this interface has to define the **Compare** method, which is used to compare two objects of the type that the **T** type parameter specifies. The **Compare** method is expected to return an integer value: zero if the parameters *x* and *y* have the same value, negative if *x* is less than *y*, and positive if *x* is greater than *y*. The following code example shows an example that sorts objects according to their hash code. (The **Object** class implements the **GetHashCode** method. It

MCT USE ONLY. STUDENT USE PROHIBITED

returns an integer value that identifies the object. All reference types inherit this method and can override it with their own implementations.)

```
class ObjectComparer : IComparer<Object>
{
    int Comparer<object>.Compare(Object x, Object y)
    {
        int xHash = x.GetHashCode();
        int yHash = y.GetHashCode();

        if (xHash == yHash)
            return 0;

        if (xHash < yHash)
            return -1;

        return 1;
    }
}
```

You can create an **ObjectComparer** object and call the **Compare** method through the **IComparer<Object>** interface to compare two objects, as shown in the following code example.

```
Object x = ...;
Object y = ...;
ObjectComparer comparer = new ObjectComparer();
IComparer<Object> objectComparator = objectComparer;
int result = objectComparator(x, y);
```

However, you can also reference this same object through a version of the **IComparer** interface that compares strings, as the following code example shows.

```
IComparer<String> stringComparator = objectComparer;
```

Initially, this statement appears to contravene the C# type safety rules. However, the purpose of the **Compare** method is to return a value based on a comparison between the parameters passed in. If you can compare objects, you certainly should be able to compare strings, which are just specialized types of objects; the definition of inheritance specifies that a string should be able to do anything that an object can do.

This still sounds a little presumptive; how does the C# compiler know that you are not going to perform any type-specific operations in the code for the **Compare** method that might fail if you invoke the method through an interface based on a

MCT USE ONLY. STUDENT USE PROHIBITED

different type? If you revisit the definition of the **IComparer** interface, you can see the **in** qualifier before the type parameter, as the following code example shows.

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

The **in** keyword tells the C# compiler that you can either pass the type **T** as the parameter type to methods or pass any type that derives from **T**. You cannot use **T** as the return type from any methods. This feature enables you to reference an object either through a generic interface based on the object type or through a generic interface based on a type that derives from the object type. Basically, if a type **A** exposes some operations, properties, or fields, then, if type **B** derives from type **A**, it must also expose the same operations (which might behave differently if they have been overridden), properties, and fields. Consequently, it should be safe to substitute an object of type **B** for an object of type **A**.



Note: Covariance and contravariance might seem like fringe topics in the world of generics, but they are useful. For example, the **List<T>** generic collection class uses **IComparer<T>** objects to implement the **Sort** and **BinarySearch** methods. A **List<Object>** object can contain a collection of objects of any type, so the **Sort** and **BinarySearch** methods need to be able to sort objects of any type. Without using contravariance, the **Sort** and **BinarySearch** methods would need to include logic that determines the real types of the items being sorted or searched and then implement a type-specific sort or search mechanism.

Summarizing Covariance and Contravariance

You can summarize the concepts of covariance and contravariance based on the examples in this lesson, as follows:

- *Covariance*. If the methods in a generic interface can return strings, they can also return objects. (All strings are objects.)
- *Contravariance*. If the methods in a generic interface can take *object* parameters, they can take *string* parameters. (If you can perform an operation by using an object, you can perform the same operation by using a string because all strings are objects.)

MCT USE ONLY. STUDENT USE PROHIBITED

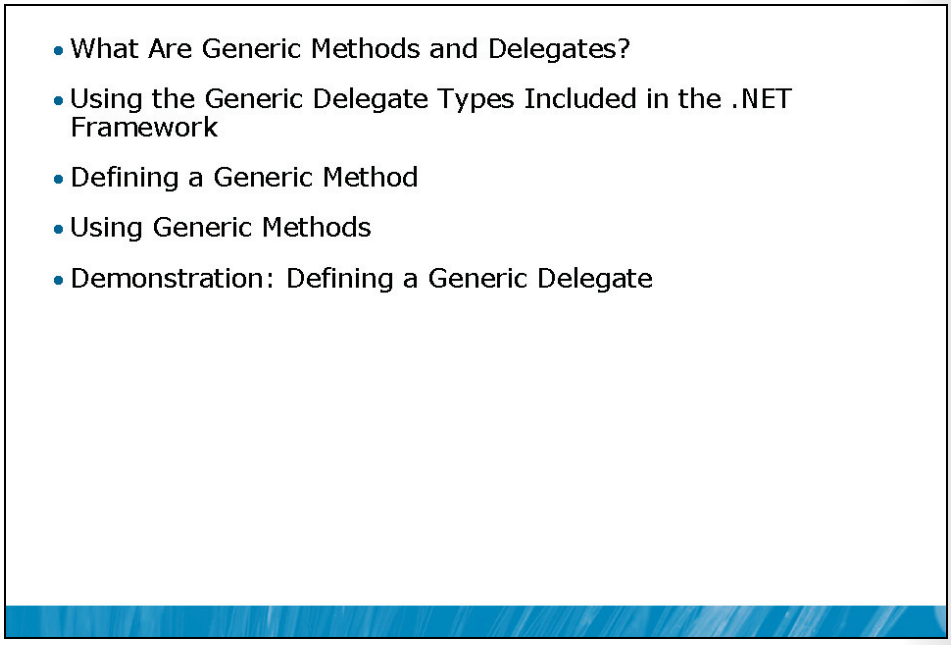
Question: If a type is used for both input and output, can you mark the type with the **in** qualifier?

Additional Reading

For more information about covariance and contravariance in generics, see the Covariance and Contravariance in Generics page at http://go.microsoft.com/fwlink/?LinkId=192969.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 4
Using Generic Methods and Delegates



Generic classes that consume other classes in a type-safe manner are a powerful feature of the .NET Framework. However, you may also need to develop generic methods or delegates. Generic methods and delegates are similar to generic classes in that developers who write consuming applications must specify a type parameter when they use a method or delegate. This lesson introduces you to the generic delegates that the .NET Framework provides, and how to use generic methods and delegates in your applications.

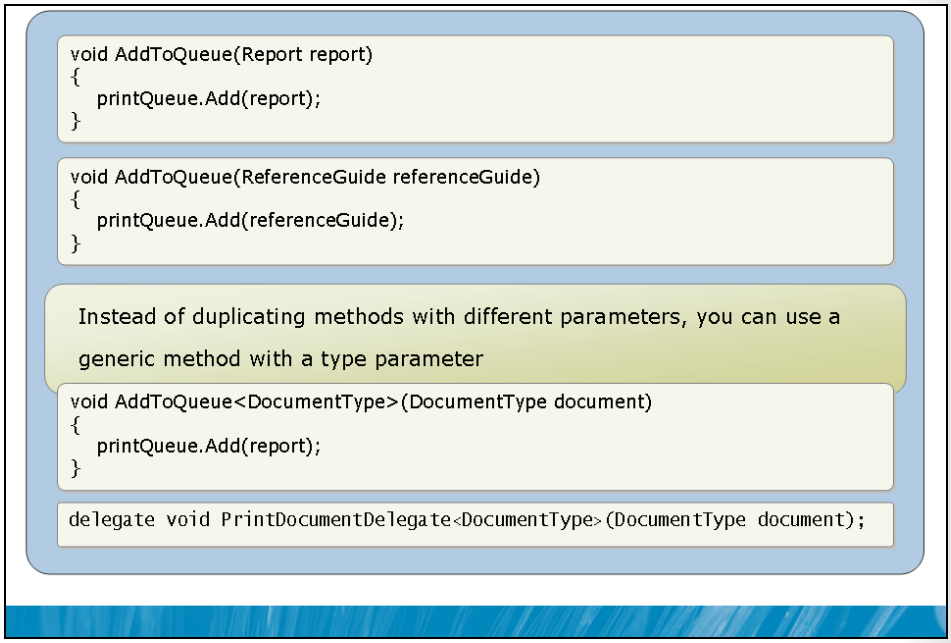
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of generic methods and delegates.
- Use the common generic delegate types that the .NET Framework provides.
- Define generic methods.
- Use generic methods.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Generic Methods and Delegates?



Key Points

Like generic types, generic methods and delegates accept a type parameter, which you can use in the parameter list and return type for the method or delegate.

Generic methods enable you to define methods that perform an action on an object while maintaining type safety. The following code example shows a simple method that adds a report to a queue for printing.

```
void AddToQueue(Report report){

    printQueue.Add(report);
}
```

You also require a method to add a reference guide (a different type of document) to a print queue. You could define the method that is defined in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
void AddToQueue(ReferenceGuide referenceGuide)
{
    printQueue.Add(referenceGuide);
}
```

Although this approach is type-safe, it adds significant extra work for the developer and risks introducing new bugs, or potentially duplicating any bugs that already exist. Using a generic method with a type parameter for the parameter removes this duplication while maintaining type safety; you define the type for the method when you develop the application, and the application will not compile if you attempt to use an incompatible type with the method.

The following code example shows a generic method that provides type-safe functionality while reducing the amount of development by the developer and reducing duplicate code and the associated risks of introducing bugs.

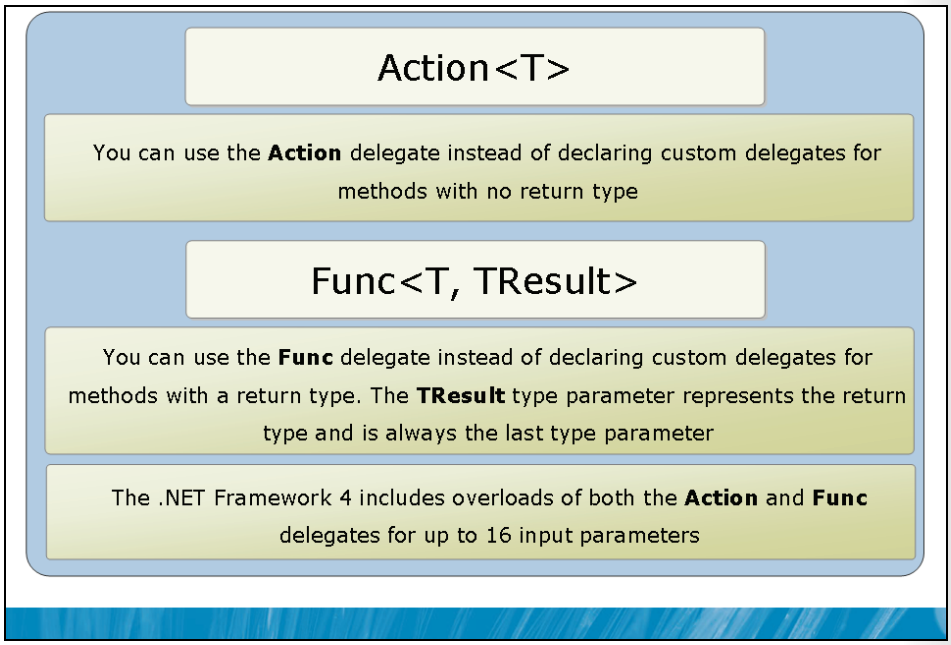
```
void AddToQueue<DocumentType>(DocumentType document)
{
    printQueue.Add(document);
}
```

Although you can use generic methods where you want to provide the method implementation, there may be times when you want to define a delegate based on unknown types. You can define a generic delegate by using type parameters that are similar to using type parameters in a method declaration.

Question: When should you use a generic method?

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Generic Delegate Types Included in the .NET Framework



Key Points

The .NET Framework includes several built-in generic delegates. You should use the built-in delegates wherever appropriate in your code instead of defining custom delegates. The two main generic delegates are the **Action** and **Func** delegates.

The Action Delegate

The **Action** delegate is a delegate that you can use with method calls that do not return a value (effectively, the return type is void). You can use the **Action** delegate instead of declaring custom delegates for methods with no return type. The .NET Framework includes several generic overloads of the **Action** delegate. These overloads enable you to specify the types of parameters that are passed to the delegate; with each overload, there is an increased number of parameters that you can specify. The **Action** delegate has overloads to support between zero and 16 parameters (in previous versions of the .NET Framework, the **Action** delegate only supported up to four parameters).

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows how to use the **Action** delegate with a simple lambda expression.

```
// Define a delegate by using the generic Action delegate.
Action<string, int> myDelegate = null;

// Add a handler for the delegate by using a lambda expression.
myDelegate += ((param1, param2) =>
    {
        Console.WriteLine("{0} : {1}", param1, param2.ToString());
    });

// Invoke the delegate.
if (myDelegate != null)
{
    myDelegate("Value", 5);
}
```

The Func Delegate

The **Func** delegate is very similar to the **Action** delegate, but with one major difference: the **Func** delegate returns a value. When you use the **Func** delegate, you must always specify the type of the return value, in addition to the types of any parameters. Like the **Action** delegate, the **Func** delegate supports between zero and 16 parameters. When you use a **Func** delegate, the last type parameter is always the type of the return value.

The following code example shows how to use the **Func** delegate with a simple lambda expression.

```
// Define a delegate by using the generic Func generic delegate.
Func<string, int, string> myDelegate = null;

// Add a handler for the delegate by using a lambda expression.
myDelegate += ((param1, param2) =>
    {
        return String.Format("{0} : {1}", param1, param2.ToString());
    });

// Invoke the delegate.
if (myDelegate != null)
{
    string returnedValue;
    returnedValue = myDelegate("Value", 5);
    Console.WriteLine(returnedValue);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

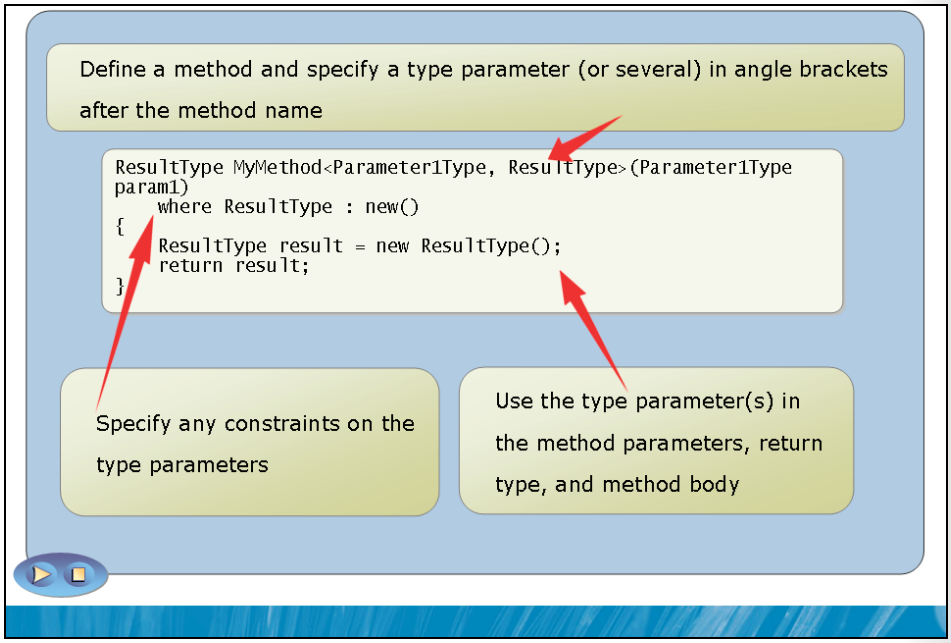


Note: The **Action** and **Func** delegates use contravariance for input type parameters. For the result output type parameter, the **Func** delegate uses covariance. Variance in the **Action** and **Func** delegates is new in the .NET Framework 4.

Question: When might you choose not to use the generic delegates that the .NET Framework provides?

MCT USE ONLY. STUDENT USE PROHIBITED

Defining a Generic Method



Key Points

To define a generic method, you use a type parameter that is enclosed in angle brackets, before you specify the parameter list. When you use type parameters, you specify an identifier for each type parameter that your method requires. You can use these identifiers instead of a concrete type in the parameter list for the method, the return type, and anywhere in the method body that it is required.

You can add constraints to the type parameters by using the same syntax as you would use for constraining the types that you can use for a generic class.

The following code example shows a generic method that has two type parameters, and includes a constraint on the **ResultType** type parameter.

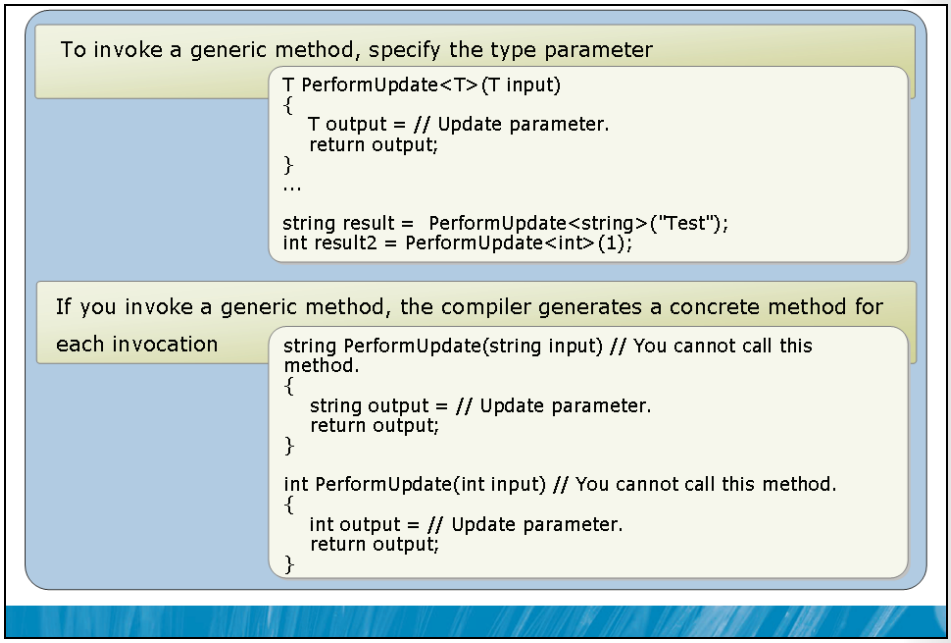
```
ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type param1)
    where ResultType : new()
{
    ResultType result = new ResultType();
    return result;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Question: Can you use variance when you define a generic method?

MCT USE ONLY. STUDENT USE PROHIBITED

Using Generic Methods



Key Points

You use a generic method like any other method, except that you must provide type parameters in addition to other parameters. When you use a generic method, the compiler checks the types that you are using with the method to ensure that it is type-safe. For example, if you specify that you are using a string type and pass an integer to the method, the compiler will return an error as if you had attempted to pass the wrong parameter to a normal method.

The following code example shows how to invoke a generic method by specifying the type parameter.

```
 // Generic method.

T PerformUpdate<T>(T input)
{
    T output = // Update parameter.
    return output;
}

...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
// Generic method usage.

string result =  PerformUpdate<string>("Test");
int result2 = PerformUpdate<int>(1);
```

When you compile an application that uses a generic method, the compiler generates versions of the generic method for each combination of type parameters that your application uses.

The following code example shows how the compiler converts the generic method from the previous code example to nongeneric methods.

```
// Compile time generated methods.
string PerformUpdate(string input)
{
    string output = // Update parameter.
    return output;
}

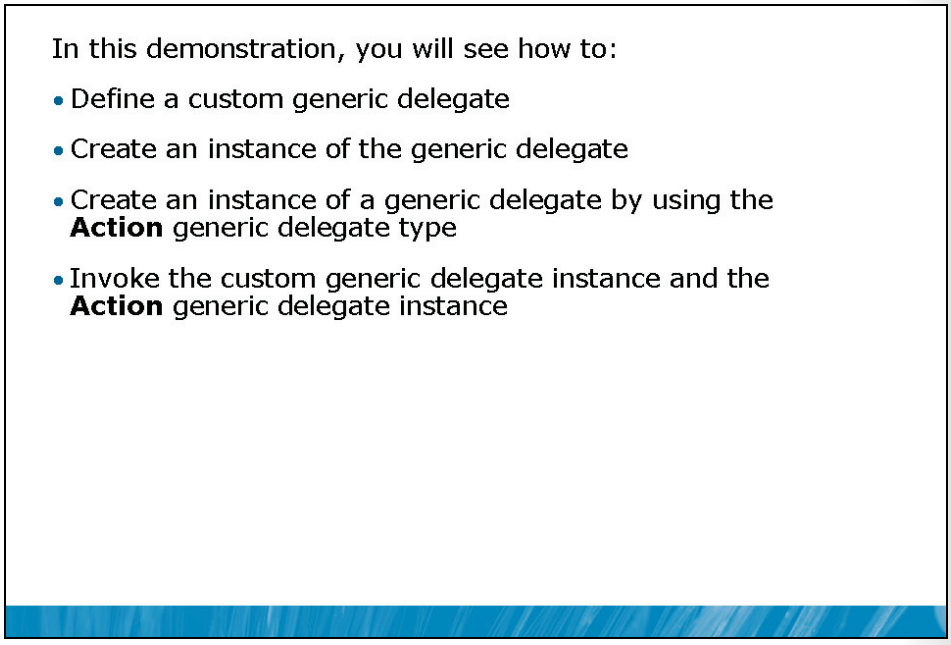
int PerformUpdate(int input)
{
    int output = // Update parameter.
    return output;
}
```

When the compiler compiles your application, it generates the equivalents of these methods to Microsoft intermediate language (MSIL); the compiler converts your generic method calls to calls to the concrete methods. You cannot call these methods directly.

Question: When you define a generic method, how many type parameters can you specify?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Defining a Generic Delegate



Key Points

In this demonstration, you will see how to:

- Define a custom generic delegate.
- Create an instance of the generic delegate.
- Create an instance of a generic delegate by using the **Action** generic delegate type.
- Invoke the custom generic delegate instance and the **Action** generic delegate instance.

Demonstration Steps

1. Start Visual Studio.
2. Open the Module 12, Demo3 starter solution.
3. Open the Printer.cs file, and then review the **Printer** class.

MCT USE ONLY. STUDENT USE PROHIBITED

The **Printer** class is a generic class that takes a single type parameter named **DocumentType**. The purpose of the class is to represent a printer that is capable of printing a specific type of document.

4. Uncomment the code that defines the **DocumentAddingToQueueDelegate** type. Note that the delegate includes a type parameter.
5. Uncomment the code that defines the **DocumentAddingToQueue** event. Note that the event uses the generic delegate that you defined in the previous step.
6. Uncomment the **OnDocumentAddingToQueue** method. This method is used to raise the event.
7. In the **AddDocumentToQueue** method, uncomment the code that raises the **OnDocumentAddingToQueue** event and uses the response to determine whether to add the item to the print queue.
8. Open the Program.cs file, and then review the **Program** class.
9. Uncomment the code that adds a handler for the **OnDocumentAddingToQueue** event, and uncomment the handler method. The handler method displays a message box and returns a value based on the response.
10. Run the application.
11. In the **Reference Guide Printing** dialog box, click **OK**.
12. In the **Reference Guide Printing** dialog box, click **Cancel**.
13. In the **Reference Guide Printing** dialog box, click **OK**.
14. When the application pauses, verify that only two documents have printed, because you canceled the second document.
15. Close the application.
16. Open the Printer.cs file.
17. Uncomment the **DocumentPrintedEventArgs** generic class, and explain how this class inherits from the **EventArgs** class and exposes a **Document** property. This will be used in an event that will notify subscribers that a document has been printed, and provide the document that printed as an argument.
18. Uncomment the code that defines the **DocumentPrinted** event by using the **Action** generic delegate. This event uses an instance of the **Printer** generic class and an instance of the **DocumentPrintedEventArgs** generic class as parameters.

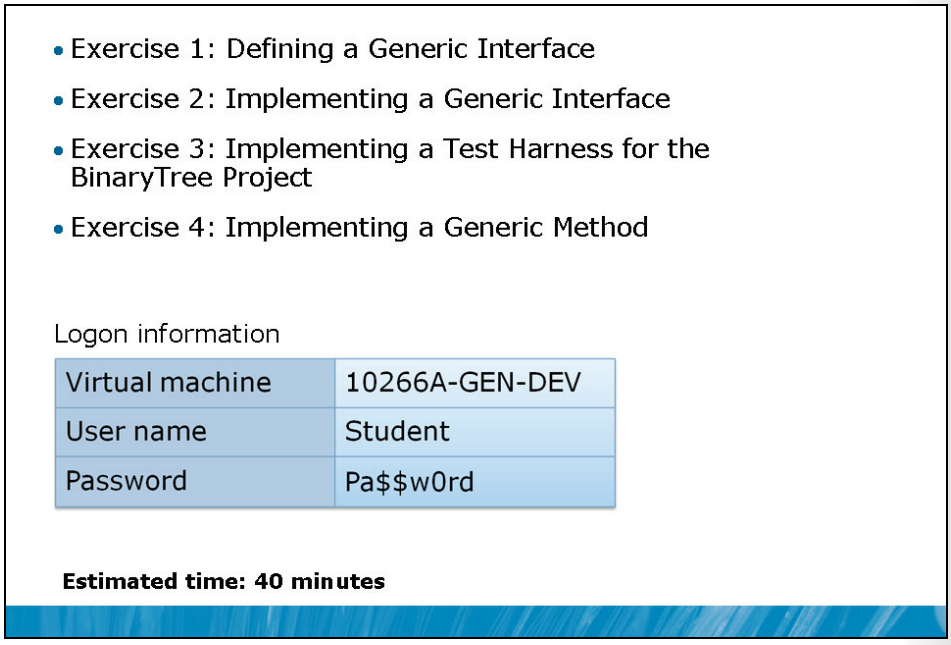
MCT USE ONLY. STUDENT USE PROHIBITED

19. Uncomment the **OnDocumentPrinted** method. This method raises the **DocumentPrinted** event.
20. In the **PrintDocuments** method, uncomment the code that calls the **OnDocumentPrinted** method. This raises the **DocumentPrinted** event whenever a document is printed.
21. Open the Program.cs file, and then review the **Program** class.
22. Uncomment the code that adds a handler for the **DocumentPrinted** event, and uncomment the handler method. The handler method displays a message box.
23. Run the application.
24. In the **Reference Guide Printing** dialog box, click **OK**.
25. In the **Reference Guide Printing** dialog box, click **Cancel**.
26. In the **Reference Guide Printing** dialog box, click **OK**.
27. When the application pauses, verify that only two documents printed, because you canceled the second document, and then press ENTER.
28. In the **Report Printed** dialog box, click **OK** three times.
29. When the application pauses, verify that only three documents printed.
30. Close the application.
31. Close Visual Studio.

Question: How does defining a generic delegate differ from defining a nongeneric delegate?

MCT USE ONLY. STUDENT USE PROHIBITED

Lab B: Building Generic Types



Objectives

After completing this lab, you will be able to:

- Define a generic interface.
- Implement a generic interface.
- Develop a simple application to test a generic interface.
- Implement a generic method.

Introduction

In this lab, you will build a generic collection type and write a generic method that can be used to populate an instance of this type.

MCT USE ONLY. STUDENT USE PROHIBITED

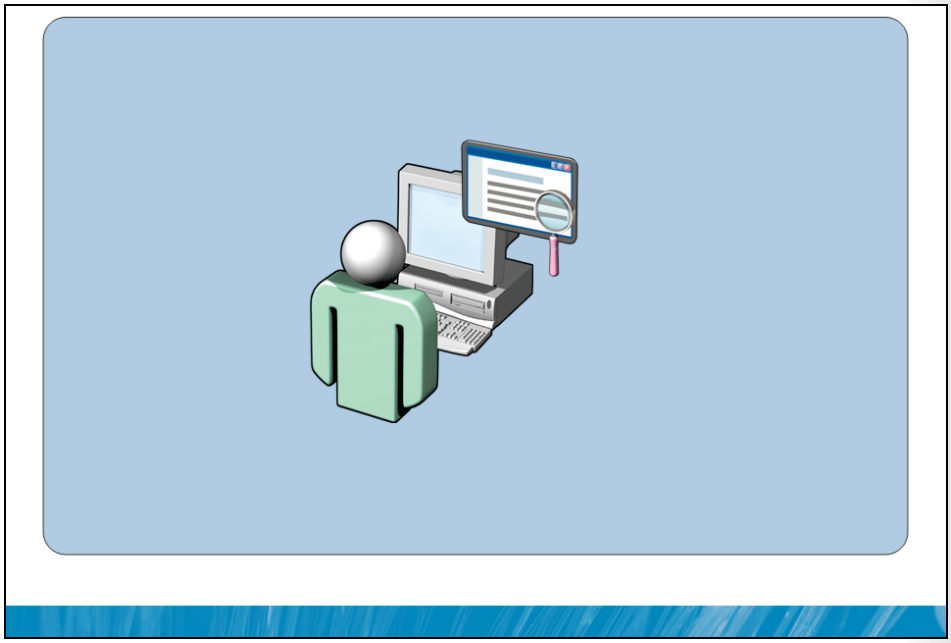
Lab Setup

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

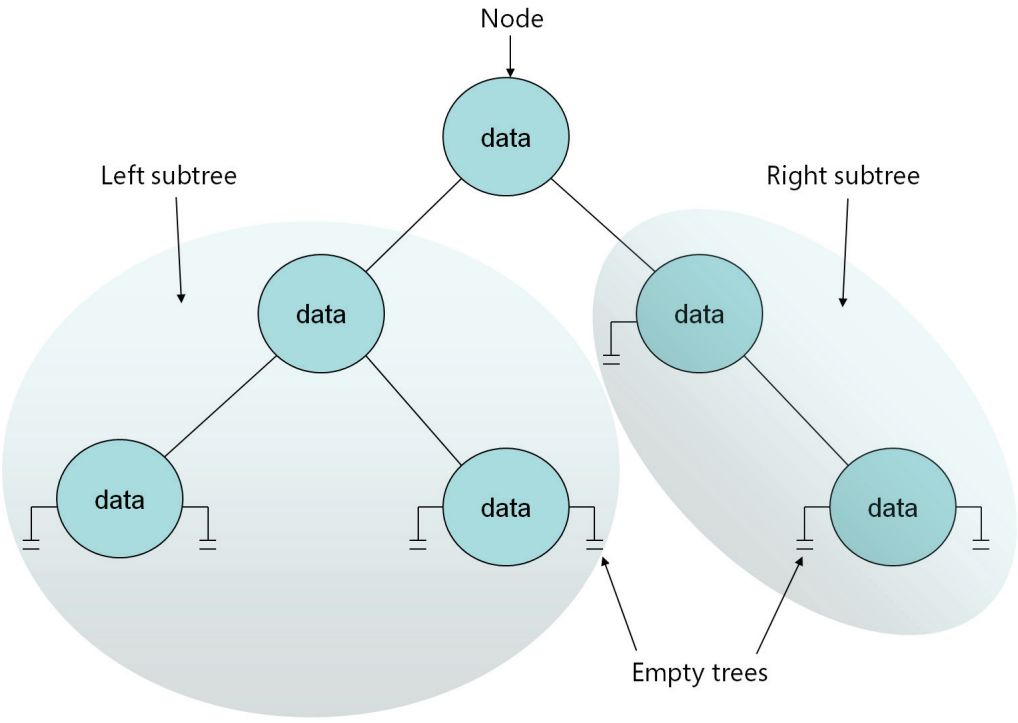


One of the devices that Fabrikam, Inc. produces captures a large amount of data and has to sort this data. You have been asked to design and build the software that can store the data that the device captures, and present it in an ordered manner as quickly as possible. The software must be able to store data of any type that supports valid comparisons.

The software implements an ordered binary tree, with methods that insert data into the appropriate place so that when the tree is traversed, the data is presented in a sorted order.

A binary tree is a recursive (self-referencing) data structure that can either be empty or contain three elements: a datum, which is typically referred to as the *node*, and two subtrees, which are themselves binary trees. The two subtrees are conventionally called the *left subtree* and the *right subtree* because they are typically depicted to the left and right of the node, respectively. Each left subtree or right subtree is either empty or contains a node and other subtrees. In theory, the whole structure can continue endlessly. The following image shows the structure of a small binary tree.

MCT USE ONLY. STUDENT USE PROHIBITED



Binary trees are ideal for sorting data. If you start with an unordered sequence of objects of the same type, you can construct an ordered binary tree and then walk through the tree to visit each node in an ordered sequence. The following code example shows the algorithm for inserting an item *I* into an ordered binary tree *T*.

```
If the tree, T, is empty
Then
  Construct a new tree T with the new item I as the node, and empty
left and
  right subtrees
Else
  Examine the value of the current node, N, of the tree, T
  If the value of N is greater than that of the new item, I
  Then
    If the left subtree of T is empty
    Then
      Construct a new left subtree of T with the item I as the node,
and
      empty left and right subtrees
    Else
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
      Insert I into the left subtree of T
    End If
  Else
    If the right subtree of T is empty
    Then
      Construct a new right subtree of T with the item I as the node,
and
      empty left and right subtrees
    Else
      Insert I into the right subtree of T
    End If
  End If
End If
```

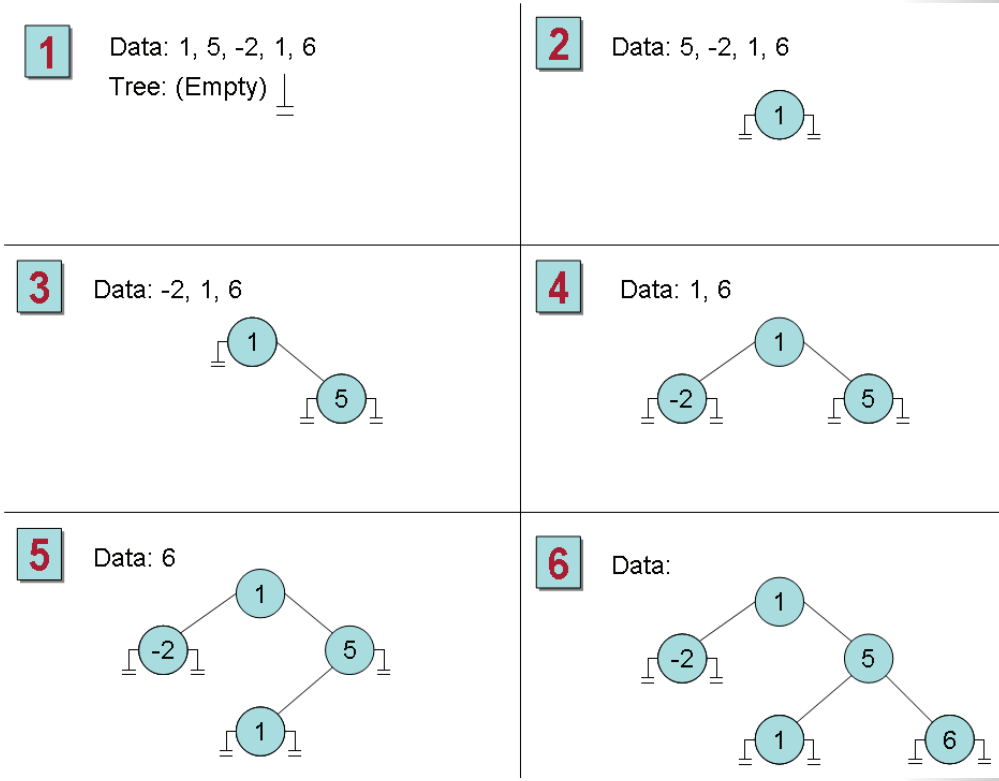
Notice that this algorithm is recursive, calling itself to insert the item into the left or right subtree depending on how the value of the item compares with the current node in the tree.



Note: The definition of the expression *greater than* depends on the type of data in the item and node. For numeric data, greater than can be a simple arithmetic comparison and for text data, it can be a string comparison. However, other forms of data must be given their own means of comparing values. In this exercise, you will use the **CompareTo** method of the **IComparable** interface to compare elements.

If you start with an empty binary tree and an unordered sequence of objects, you can iterate through the unordered sequence, inserting each object into the binary tree by using this algorithm, resulting in an ordered tree. The following image shows the steps in the process for constructing a tree from a set of five integers.

MCT USE ONLY. STUDENT USE PROHIBITED

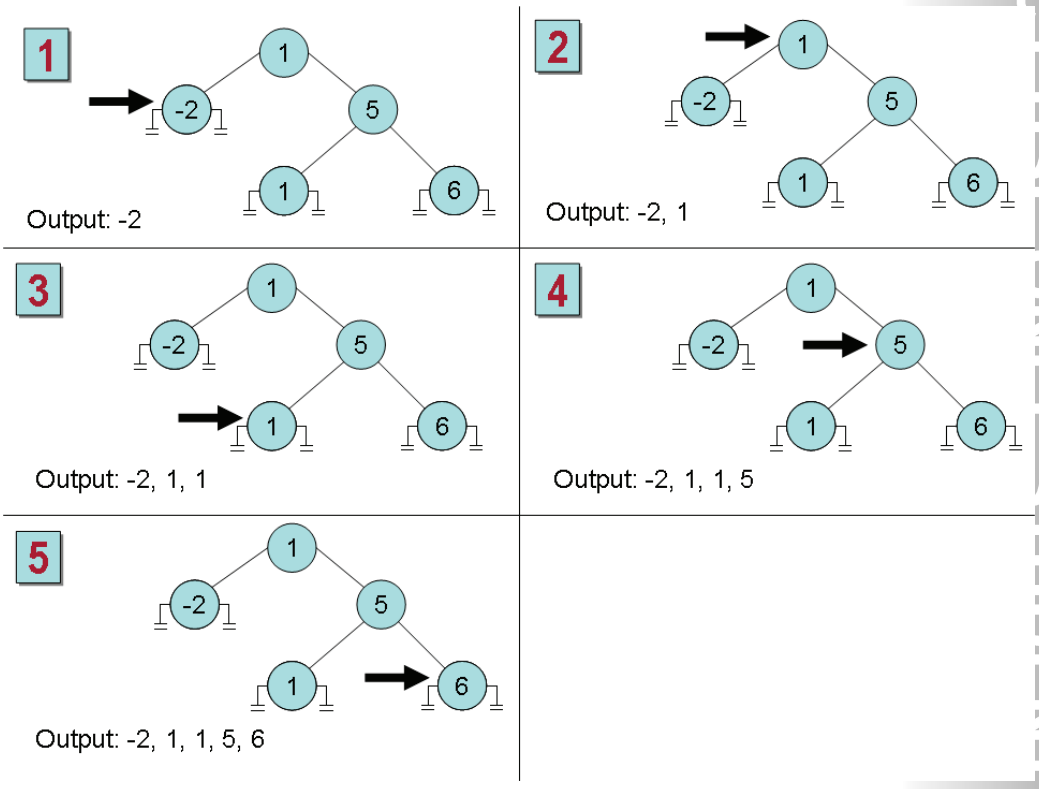


After you have built an ordered binary tree, you can display its contents in sequence by visiting each node in turn and printing the value found. The algorithm for achieving this task is also recursive, as the following code example shows.

```
If the left subtree is not empty
Then
  Display the contents of the left subtree
End If
Display the value of the node
If the right subtree is not empty
Then
  Display the contents of the right subtree
End If
```

The following image shows the steps in the process for displaying the tree. Notice that the integers are now displayed in ascending order.

MCT USE ONLY. STUDENT USE PROHIBITED



Exercise 1: Defining a Generic Interface

In this exercise, you will define an interface for a generic binary tree called **IBinaryTree**. The interface will specify the following methods:

- **Add**. This method will add an item to the tree.
- **Remove**. This method will remove the first item with the specified value from the tree.
- **WalkTree**. This method will display the contents of the tree, in sorted order.

The main tasks for this exercise are as follows:

1. Open the GenericTypes solution.
2. Define the generic **IBinaryTree** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the GenericTypes solution

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 12\Lab B\Snippets folder.
4. Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex1\Starter folder.

▶ Task 2: Define the generic IBinaryTree interface

1. Review the task list.
2. In the task list, locate the **TODO – Define the IBinaryTree interface** task, and then double-click this task. This task is located in the IBinaryTree.cs file.
3. In the **BinaryTree** namespace, define a new generic public interface named **IBinaryTree**. This interface should take a single type parameter named **TItem**. Specify that the type parameter must implement the generic **IComparable** interface.
4. In the **IBinaryTree** interface, define the following public methods:
   a. An **Add** method, which takes a **TItem** object named **newItem** as a parameter and does not return a value.
   b. A **Remove** method, which takes a **TItem** object named **itemToRemove** as a parameter and does not return a value.
   c. A **WalkTree** method, which takes no parameters and does not return a value.
5. Build the solution and correct any errors.

Exercise 2: Implementing a Generic Interface

In this exercise, you will create the generic binary tree type called **BinaryTree** that implements the **IBinaryTree** interface.

The main tasks for this exercise are as follows:

1. Open the GenericTypes solution.
2. Create the **Tree** class.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 1: Open the GenericTypes solution



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **IBinaryTree** interface successfully, proceed directly to **Task 2: Create the Tree class**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex2\Starter folder.

▶ Task 2: Create the Tree class

1. In the BinaryTree project, add a new class named **Tree**.
2. Modify the **Tree** class definition. This class should be a public generic class that takes a single type parameter called **TItem** and implements the **IBinaryTree** interface. The **TItem** type parameter must implement the generic **IComparable** interface.
3. Add the following automatic properties to the **Tree** class:
   a. A **TItem** property named **NodeData**.
   b. A generic **Tree<TItem>** property named **LeftTree**.
   c. A generic **Tree<TItem>** property named **RightTree**.
4. Add a public constructor to the **Tree** class. The constructor should take a single **TItem** parameter called *nodeValue*. The constructor should initialize the **NodeData** member by using the *nodeValue* parameter, and then set the **LeftTree** and **RightTree** members to **null**.
5. After the constructor, define a method called **Add**. This method should take a **TItem** object as a parameter, but not return a value.
6. In the **Add** method, add code to insert the **newItem** object into the tree in the appropriate place by performing the following tasks:
   a. Compare the value of the **newItem** object with the value of the **NodeData** property. Both items implement the **IComparable** interface, so use the **CompareTo** method of the **NodeData** property. The **CompareTo** method returns zero if both items have the same value, a positive value if the value of the **NodeData** property is greater than the value of the **newItem** object, and a negative value if the value of the **NodeData** property is less than the value of the **newItem** object.

b. If the value of the **newItem** object is less than the value of the **NodeData** property, perform the following actions to insert a **newItem** object into the left subtree:

   i. If the **LeftTree** property is **null**, initialize it and pass the **newItem** object to the constructor.

   ii. If the **LeftTree** property is not **null**, recursively call the **Add** method of the **LeftTree** property and pass the **newItem** object as the parameter.

c. If the value of the **newItem** object is greater than or equal to the value of the **NodeData** property, perform the following actions to insert the **newItem** object into the right subtree:

   i. If the **RightTree** property is **null**, initialize it and pass the value of the **newItem** object to the constructor.

   ii. If the **RightTree** property is not **null**, recursively call the **Add** method of the **RightTree** property and pass the **newItem** object as the parameter.

7. After the **Add** method, add another public method called **WalkTree** that does not take any parameters and does not return a value.

8. In the **WalkTree** method, add code that visits each node in the tree in order and displays the value that each node holds by performing the following tasks:

   a. If the value of the **LeftTree** property is not **null**, recursively call the **WalkTree** method on the **LeftTree** property.

   b. Display the value of the **NodeData** property to the console by using a **Console.WriteLine** statement.

   c. If the value of the **RightTree** property is not **null**, recursively call the **WalkTree** method on the **RightTree** property.

9. After the **WalkTree** method, add the **Remove** method to delete a value from the tree, as the following code example shows. It is not necessary for you to fully understand how this method works, so you can either type this code manually or use the Mod12Remove code snippet.

```
public void Remove(TItem itemToRemove)
{

    // Cannot remove null.
    if (itemToRemove == null)
    {
        return;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }

    // Check if the item could be in the left tree.
    if (this.NodeData.CompareTo(itemToRemove) > 0
        && this.LeftTree != null)
    {
        // Check the left tree.
        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.LeftTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The LeftTree property has no children - set the
            // LeftTree property to null.
            if (this.LeftTree.LeftTree == null
                && this.LeftTree.RightTree == null)
            {
                this.LeftTree = null;
            }
            else // Remove LeftTree.
            {
                RemoveNodeWithChildren(this.LeftTree);
            }
        }
        else
        {
            // Keep looking - call the Remove method recursively.
            this.LeftTree.Remove(itemToRemove);
        }
    }

    // Check if the item could be in the right tree.?
    if (this.NodeData.CompareTo(itemToRemove) < 0
        && this.RightTree != null)
    {
        // Check the right tree.

        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.RightTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The RightTree property has no children - set the
            // RightTree property to null.
            if (this.RightTree.LeftTree == null
                && this.RightTree.RightTree == null)
            {
                this.RightTree = null;
            }
            else // Remove the RightTree.
            {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
                RemoveNodeWithChildren(this.RightTree);
            }
        }
        else
        {
            // Keep looking - call the Remove method recursively.
            this.RightTree.Remove(itemToRemove);
        }
    }

    // This will only apply at the root node.
    if (this.NodeData.CompareTo(itemToRemove) == 0)
    {
        // No children - do nothing, a tree must have at least
        // one node.
        if (this.LeftTree == null && this.RightTree == null)
        {
            return;
        }
        else // The root node has children.
        {
            RemoveNodeWithChildren(this);
        }
    }
}
```

10. After the **Remove** method, add the **RemoveNodeWithChildren** method to remove a node that contains children from the tree, as the following code example shows. This method is called by the **Remove** method. Again, it is not necessary for you to understand how this code works, so you can either type this code manually or use the Mod12RemoveNodeWithChildren code snippet.

```
private void RemoveNodeWithChildren(Tree<TItem> node)
{
    // Check whether the node has children.
    if (node.LeftTree == null && node.RightTree == null)
    {
        throw new ArgumentException("Node has no children");
    }
    // The tree node has only one child - replace the
    // tree node with its child node.
    if (node.LeftTree == null ^ node.RightTree == null)
    {
        if (node.LeftTree == null)
        {
            node.copyNodeToThis(node.RightTree);
        }
        else
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        {
            node.copyNodeToThis(node.LeftTree);
        }
    }
    else
    // The tree node has two children - replace the tree node's value
    // with its "in order successor" node value and then remove the
    // in order successor node.
    {
        // Find the in order successor - the leftmost descendant of
        // its RightTree node.
        Tree<TItem> successor = getLeftMostDescendant(node.RightTree);

        // Copy the node value from the in order successor.
        node.NodeData = successor.NodeData;

        // Remove the in order successor node.
        if (node.RightTree.RightTree == null &&
            node.RightTree.LeftTree == null)
        {
            node.RightTree = null; // The successor node had no
                                   // children.
        }
        else
        {
            node.RightTree.Remove(successor.NodeData);
        }
    }
}
```

11. After the **RemoveNodeWithChildren** method, add the **CopyNodeToThis** method, as the following code example shows. The **RemoveNodeWithChildren** method calls this method to copy another node's property values into the current node. You can either type this code manually or use the Mod12CopyNodeToThis code snippet.

```
private void CopyNodeToThis(Tree<TItem> node)
{
    this.NodeData = node.NodeData;
    this.LeftTree = node.LeftTree;
    this.RightTree = node.RightTree;
}
```

12. After the **CopyNodeToThis** method, add the **GetLeftMostDescendant** method, as the following code example shows. The **RemoveNodeWithChildren** method also calls this method to retrieve the

MCT USE ONLY. STUDENT USE PROHIBITED

leftmost descendant of a tree node. You can either type this code manually or use the Mod12GetLeftMostDescendant code snippet.

```
private Tree<TItem> GetLeftMostDescendant(Tree<TItem> node)
{
    while (node.LeftTree != null)
    {
        node = node.LeftTree;
    }
    return node;
}
```

13. Build the solution and correct any errors.

Exercise 3: Implementing a Test Harness for the BinaryTree Project

In this exercise, you will modify the test harness to use the BinaryTree project.

The main tasks for this exercise are as follows:

1. Open the GenericTypes solution.
2. Import the TestHarness project.
3. Complete the test harness.
4. Test the BinaryTree project.

▶ Task 1: Open the GenericTypes solution



Note: Perform this task only if you have not been able to complete Exercise 2. If you have defined the **IBinaryTree** interface and built the **Tree** class successfully, proceed directly to **Task 3: Complete the test harness**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter folder.

▶ Task 2: Import the TestHarness project



Note: Perform this task only if you have completed Exercise 2 successfully.

MCT USE ONLY. STUDENT USE PROHIBITED

1. Import the TestHarness project in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter \TestHarness folder into the GenericTypes solution.
2. In the TestHarness project, update the reference to the BinaryTree project.
3. Set the TestHarness project as the startup project.

▶ Task 3: Complete the test harness

1. Open the Program.cs file.
2. In the **Main** method, add code to instantiate a new **IBinaryTree** object named **tree**, using **int** as the type parameter. Pass the value **5** to the constructor. This code creates a new binary tree of integers and adds an initial node that contains the value **5**.
3. Add code to the **Main** method to add the following values to the tree, in the following order:
   a. **1**
   b. **4**
   c. **7**
   d. **3**
   e. **4**
4. Add code to the **Main** method to perform the following actions:
   a. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.
   b. Print the message "Add 15" to the console, and then add the value **15** to the tree.
   c. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.
   d. Print the message "Remove 5" to the console, and then remove the value **5** from the tree.
   e. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.
   f. Pause at the end of the method until ENTER is pressed.
5. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Test the BinaryTree project

1. Run the application.
2. Verify that the output in the console window resembles the following code example. Note that the data in the binary tree is sorted and is displayed in ascending order.

```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
3
4
4
7
15
```

3. Press ENTER to close the console window, and then return to Visual Studio.

Exercise 4: Implementing a Generic Method

In this exercise, you will define a generic method called **BuildTree** that creates an instance of the generic binary tree. The method will take a **params** array of a type that a type parameter specifies, and construct the binary tree by using the data in this array. The binary tree will be returned from the method.

The main tasks for this exercise are as follows:

1. Open the GenericTypes solution.
2. Create the **BuildTree** method.

3. Modify the test harness to use the **BuildTree** method.

▶ Task 1: Open the GenericTypes solution

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex4\Starter folder.



Note: The GenericTypes solution in the Ex4 folder is functionally the same as the code that you completed in Exercise 3. However, it includes an updated task list and a new test project to enable you to complete this exercise.

▶ Task 2: Create the BuildTree method

1. Review the task list.
2. In the task list, locate the **TODO - Add the BuildTree generic method** task, and then double-click this task. This task is located at the end of the **Tree** class.
3. Remove the **TODO - Add the BuildTree generic method** comment, and then add a public static generic method named **BuildTree** to the **Tree** class. The type parameter for the method should be called **TreeItem**, and the method should return a generic **Tree<TreeItem>** object. The **TreeItem** type parameter must represent a type that implements the generic **IComparable** interface.

   The method should take two parameters: a **TreeItem** object called **nodeValue** and a **params** array of **TreeItem** objects called **values**.
4. In the **BuildTree** method, add code to construct a new **Tree** object that uses the data that is passed in as the parameters by performing the following actions:
   a. Define a new **Tree** object named **tree** that uses the **TreeItem** type parameter, and initialize the new **Tree** object by using the **nodeValue** parameter.
   b. Iterate through the **values** array, and add each value in the array to the **tree** object.
   c. Return the **tree** object at the end of the method.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Modify the test harness to use the BuildTree method

1. In the task list, locate the **TODO - Modify the test harness to use the BuildTree method** task, and then double-click this task.

   This task is located in the **Main** method of the Program.cs class file in the TestHarness project.

2. In the **Main** method, remove the existing code that instantiates the **tree** object and adds the first five values to the tree. Replace this code with a statement that calls the **BuildTree** method to create a new **Tree** object named **tree**, based on the integer type, with the following integer values:

   a. **1**

   b. **4**

   c. **7**

   d. **3**

   e. **4**

   f. **5**

3. Build the solution and correct any errors.

4. Run the application.

5. Verify that the output in the console window resembles the following code example.

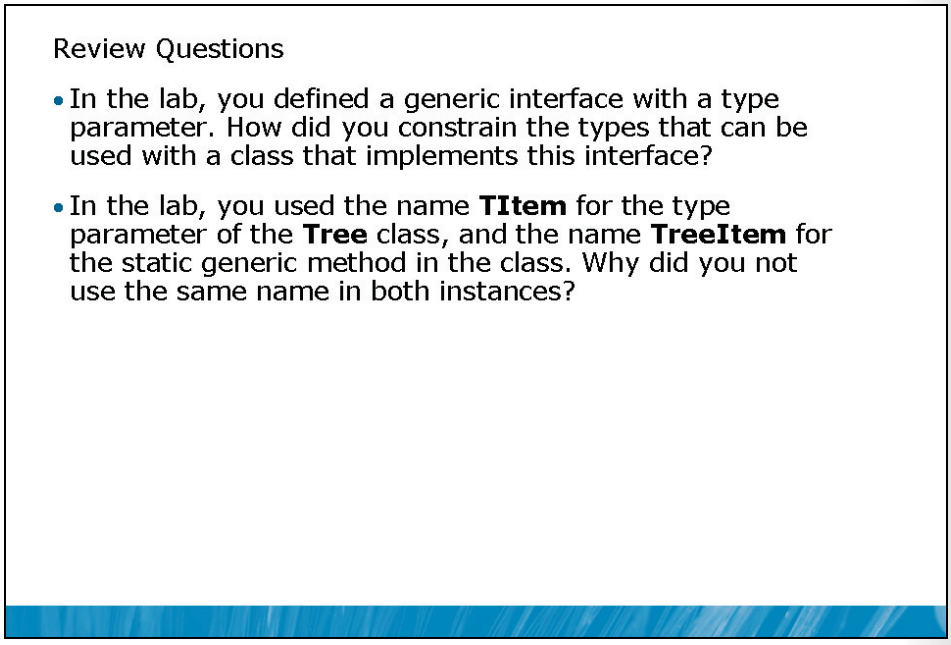
```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
```

```
3
4
4
7
15
```

6. Press ENTER to close the console window.
7. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

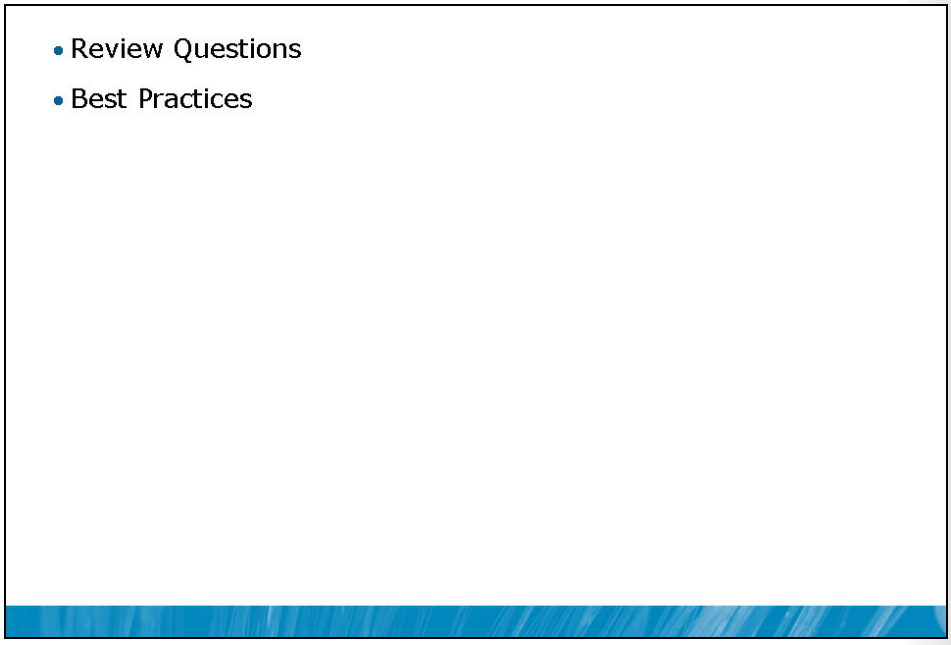


Review Questions

1. In the lab, you defined a generic interface with a type parameter. How did you constrain the types that can be used with a class that implements this interface?
2. In the lab, you used the name **TItem** for the type parameter of the **Tree** class, and the name **TreeItem** for the static generic method in the class. Why did you not use the same name in both instances?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What are the main advantages of using a collection class instead of an array?
2. How do generic collection classes differ from nongeneric collection classes?
3. When would you use a generic type instead of a nongeneric type?

Best Practices Related to Collections

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

- Use collections instead of arrays where you do not know the size of the collection in advance.
- Use **Hashtable** objects for large key-value pair collections, but avoid them for smaller collections.

MCT USE ONLY. STUDENT USE PROHIBITED

Best Practices Related to Generic Types

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

- Use generic types wherever possible to improve type safety.
- Use constraints on generic types to provide control over types that are used with your generic classes.

Best Practices Related to Generic Methods and Delegates

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

- Use the **Action** and **Func** generic delegates instead of custom delegates wherever possible.

Best Practices Related to Variant Interfaces

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

- Use the **in** and **out** keywords to specify covariant and contravariant type parameters wherever possible. In this way, you can make your generic interfaces as flexible as possible.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 13

Building and Enumerating Custom Collection Classes

Contents:

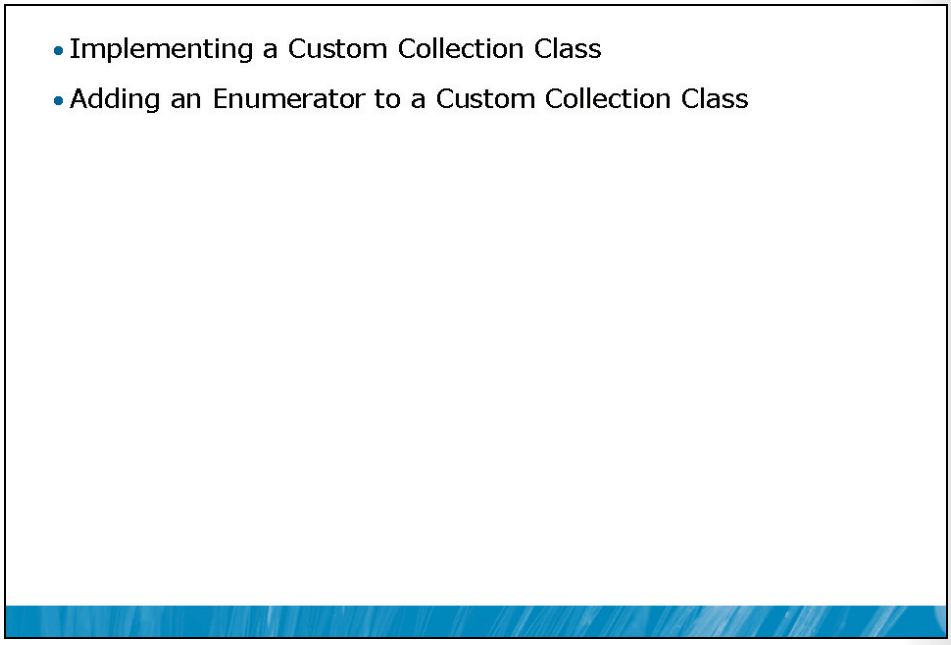
Lesson 1: Implementing a Custom Collection Class 13-3

Lesson 2: Adding an Enumerator to a Custom Collection Class 13-21

Lab: Building and Enumerating Custom Collection Classes 13-37

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



When you develop applications, you often need to store collections of objects. In many circumstances, you can use the collection classes that the Microsoft® .NET Framework includes; however, sometimes these collection classes do not provide the functionality that you require. For example, you may need to store objects in a sorted order that is based on a custom sorting algorithm.

This module introduces you to custom collection classes. It also explains how you can develop collection classes that support the language constructs that C# provides, such as enumeration and collection initialization.

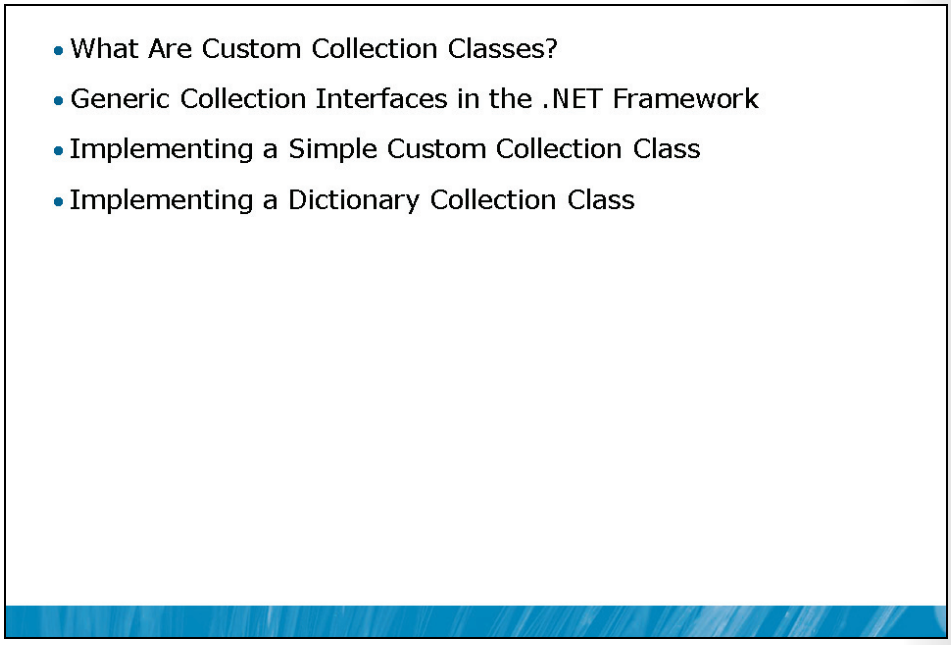
Objectives

After completing this module, you will be able to:

- Implement a custom collection class.
- Define an enumerator in a custom collection class.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Implementing a Custom Collection Class



The .NET Framework provides a range of collection classes that enable you to store and retrieve data by using a variety of semantics. For example, the **Queue** class enables you to store and retrieve data in a first-in, first-out (FIFO) manner, and the **Stack** class implements a last-in, first-out (LIFO) mechanism. Other collection classes, such as the **Hashtable** class, enable you to store data and access it by using a key in the form of a dictionary. If none of the collection classes that the .NET Framework class library provides meet the specific requirements of an application, you can implement your own custom collection class.

The C# language provides several features that are intended to be used with collections. When you develop a custom collection class, you must ensure that you implement the necessary functionality to support these features. For example, you may need to ensure that collection initializers can be used with your collection and you can iterate over the items in a collection. This lesson introduces you to custom collection classes. It also introduces you to the interfaces that the .NET Framework includes, which you can implement to provide this support.

MCT USE ONLY. STUDENT USE PROHIBITED

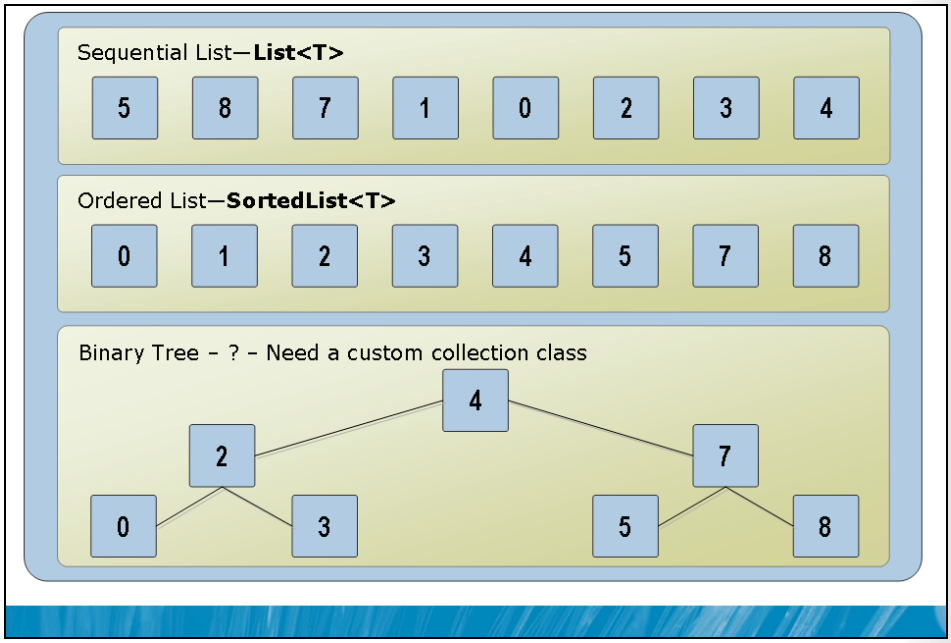
Objectives

After completing this lesson, you will be able to:

- Describe the purpose of a custom collection class.
- Describe the generic interfaces that the .NET Framework provides, which custom collection classes should implement.
- Implement a simple collection class.
- Implement a dictionary collection class.

MCT USE ONLY. STUDENT USE PROHIBITED

What Are Custom Collection Classes?



Key Points

A custom collection class is a class that enables you to store and retrieve a collection of objects and that implement the various interfaces that the .NET Framework defines. By implementing the interfaces that the .NET Framework defines, custom collection classes can provide support for language constructs. For example, to use a collection class with the **foreach** construct, the collection class must provide an enumerator. The **IEnumerable** interface defines the **GetEnumerator** method, and you should implement this interface. In addition, an enumerator must provide methods that the **IEnumerator** interface defines, which an application can use to move through the items in a collection in an orderly manner.

If you must support Language-Integrated Query (LINQ) queries, you should implement the **ICollection** interface. This interface extends the **IEnumerable** interface with methods to convert the collection into a queryable object.



Note: LINQ is described in more detail in a later module.

MCT USE ONLY. STUDENT USE PROHIBITED

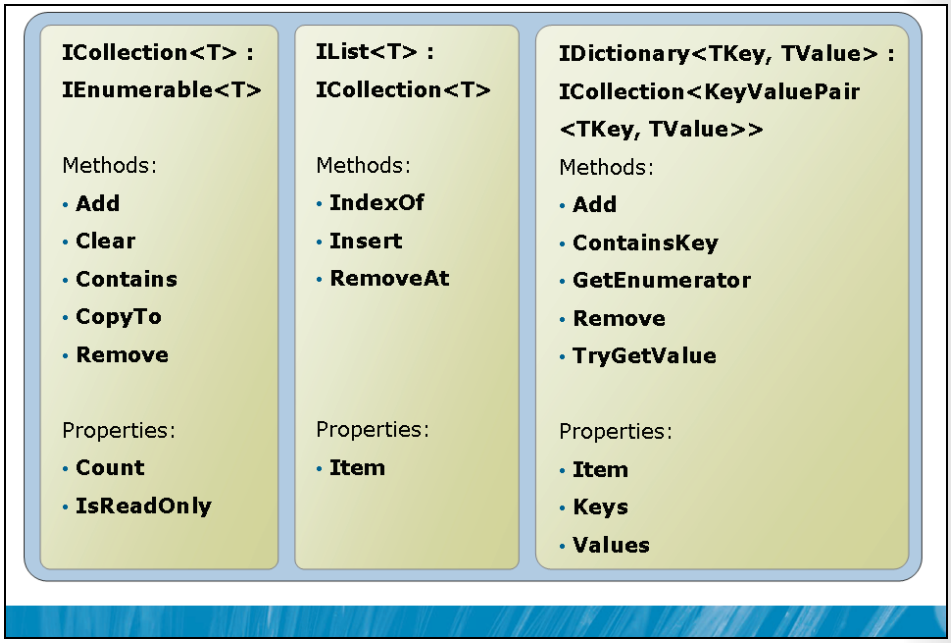
Depending on the features that you want to provide in a custom collection class, you can also implement other interfaces. For example, if you must provide access to elements in a collection by using an index, you should implement the **IList** interface.

The .NET Framework includes both generic and nongeneric versions of the collection's interfaces. Depending on your requirements, you can choose to develop either a generic or a nongeneric custom collection class. Nongeneric custom collection classes are typically designed to work with a specific type, which may limit their use in other applications, or you can design them to store and retrieve **System.Object** items, which may have an impact on performance and type safety. Unless you have a specific reason to implement a nongeneric collection class, you should normally use a generic collection class.

Question: When might you develop a custom collection class?

MCT USE ONLY. STUDENT USE PROHIBITED

Generic Collection Interfaces in the .NET Framework



Key Points

The .NET Framework includes several interfaces that you should implement when you develop custom collection classes. The .NET Framework provides both generic and nongeneric versions of these interfaces. The principal interfaces that the .NET Framework provides are:

- The **ICollection<T>** interface.
- The **IEnumerable<T>** interface.
- The **IList<T>** interface.
- The **IDictionary<TKey, TValue>** interface.

The ICollection<T> and IEnumerable<T> Interfaces

The **ICollection<T>** interface is the base interface for all generic collection classes. You should implement the **ICollection<T>** interface in every collection class that you develop. The **ICollection<T>** interface inherits from the **IEnumerable<T>** interface, which is covered in the next lesson; when you implement the

MCT USE ONLY. STUDENT USE PROHIBITED

ICollection<T> interface, you must also implement the **IEnumerable<T>** interface. The **ICollection<T>** interface defines the following methods:

- **Add**. This method should add a new item to the collection. You must expose a public **Add** method if you need to use collection initializers with your type.



Note: To support collection initializers, your type must also implement the **IEnumerable** interface. The **IEnumerable** interface is covered in the next lesson.

- **Clear**. This method should remove all items from the collection.
- **Contains**. This method should return a value to indicate whether a particular item exists in the collection.
- **CopyTo**. This method copies the collection to a specified array.
- **Remove**. This method removes a single item from the collection.

In addition, the **ICollection<T>** interface also defines the following properties:

- **Count**. This property indicates how many items are in the collection.
- **IsReadOnly**. This property indicates whether the collection is read-only.

The IList<T> Interface

The **IList<T>** interface is the base interface for linear collections that enable you to access elements by using an index. The **IList<T>** interface inherits from the **ICollection<T>** interface. The **IList<T>** interface defines the following methods:

- **IndexOf**. This method returns a value that indicates the index in the collection of the first occurrence of a particular value.
- **Insert**. This method inserts a new item at the specified index in the collection.
- **RemoveAt**. This method removes a single item from the collection at the specified index.

In addition, the **IList<T>** interface also defines the **Item** property. This property is an indexer that enables you to get or set the value at a specified index in the collection.

MCT USE ONLY. STUDENT USE PROHIBITED

The IDictionary<TKey, TValue> Interface

The **IDictionary<TKey, TValue>** interface is the base interface for dictionary-based collections. Dictionary-based collections store key and value pair combinations; when you add an item to a dictionary collection, you must specify a key and the value. The key must be unique in the collection. To retrieve an item from a dictionary collection, you specify the key to use. The **IDictionary<TKey, TValue>** interface inherits from the **ICollection<T>** interface. However, remember that the **ICollection<T>** interface takes a single type parameter, but the **IDictionary <TKey, TValue>** interface takes two type parameters. The .NET Framework provides the **KeyValuePair<TKey, TValue>** generic type to help address this mismatch. When you develop a generic collection class, each item in the collection can be represented as a **KeyValuePair<TKey, TValue>** object. You use this type as the type parameter when you implement the **ICollection<T>** interface in a dictionary class.

The **IDictionary<TKey, TValue>** interface defines the following methods:

- **Add**. This overloaded method adds a new item to the collection with the specified key and value.
- **ContainsKey**. This method returns a Boolean value to indicate whether the collection contains an item with a particular key.
- **GetEnumerator**. This overloaded method returns an enumerator of **KeyValuePair<TKey, TValue>** objects. The next lesson discusses how to implement enumerators in more detail.
- **Remove**. This overloaded method removes an item with a specified key from the collection.
- **TryGetValue**. This method returns a Boolean value to indicate whether the method succeeds. If the method is successful, it sets the value of an *output* parameter to the value that is associated with the specified key.

In addition, the **IDictionary<T>** interface also defines the following properties:

- **Item**. This property is an indexer that enables you to get or set the value in the collection, based on the specified key.
- **Keys**. This property returns a collection, referenced by using the **ICollection<T>** interface, of the keys in the collection.
- **Values**. This property returns a collection, referenced by using the **ICollection<T>** interface, of the values in the collection.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: When you develop a custom collection class, which interface must you always implement?

Additional Reading

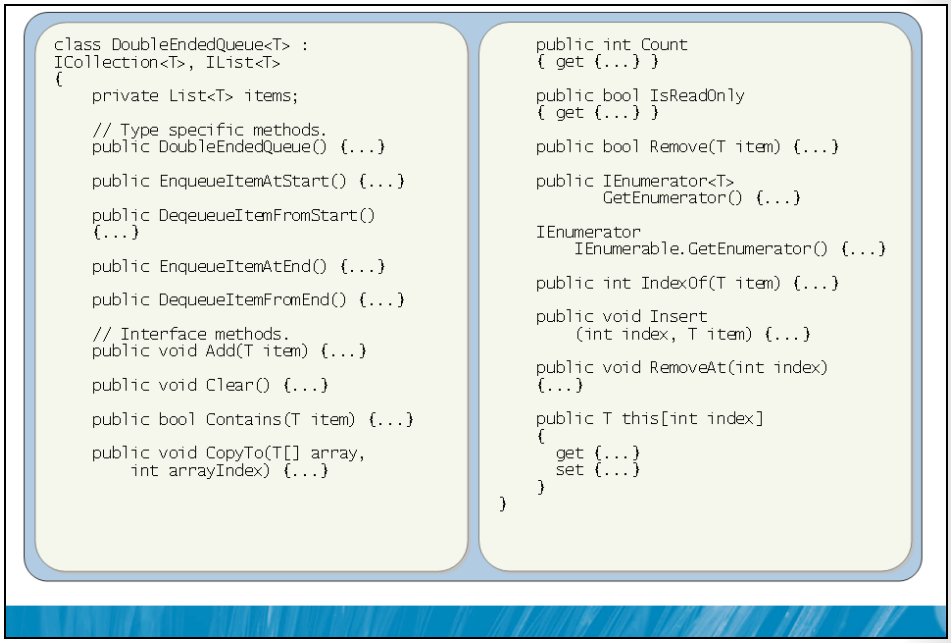
For more information about the **ICollection<T>** interface, see the ICollection(OfT) Interface page at http://go.microsoft.com/fwlink/?LinkId=192970.

For more information about the **IList<T>** interface, see the IList(OfT) Interface page at http://go.microsoft.com/fwlink/?LinkId=192971.

For more information about the **IDictionary<TKey, TValue>** interface, see the IDictionary(OfTKey, TValue) Interface page at http://go.microsoft.com/fwlink/?LinkId=192972.

MCT USE ONLY. STUDENT USE PROHIBITED

Implementing a Simple Custom Collection Class



Key Points

A simple collection class stores single values instead of key/value pairs. To implement a simple custom collection class, you define a generic class that should implement an appropriate structure to store items. In some situations, you may use the collection types that the .NET Framework includes from within your type; in other scenarios, you may define custom types to store the items that are added to the collection. You must implement the **ICollection<T>** interface in your type. If you build a linear collection, you should also implement the **IList<T>** interface.

The following code example shows a simple collection class that implements a double-ended queue. It provides methods that enable you to enqueue and dequeue items from either end. It uses a **List<T>** object internally to store the items in the collection.

```
class DoubleEndedQueue<T> : ICollection<T>, IList<T>
{
    // Define a List<T> object to store items that are added
    // to the collection.
    private List<T> items;
    // Add a constructor to the class.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
public DoubleEndedQueue()
{
    items = new List<T>();
}

// Methods for enqueuing and dequeuing items.
public void EnqueueItemAtStart(T item)
{
    // The implementation details are not shown.
    ...
}

public T DeQueueItemFromStart()
{
    // The implementation details are not shown.
    ...
}

public void EnqueueItemAtEnd(T item)
{
    // The implementation details are not shown.
    ...
}

public T DeQueueItemFromEnd()
{
    // The implementation details are not shown.
    ...
}

#region ICollection<T> Members

// Define an Add method that adds an item to the collection.
public void Add(T item)
{
    items.Add(item);
}

// Define a Clear method that removes all items from
// the collection.
public void Clear()
{
    items.Clear();
}

// Define a Contains method that returns a bool object
// according to whether the collection contains a particular item.
public bool Contains(T item)
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        return items.Contains(item);
    }

    // Define a CopyTo method that copies the entire contents of the
    // collection to an array. The array is provided as a parameter,
    // and the first item from the collection should be stored in the
    // index that is passed as a parameter to the method.

    public void CopyTo(T[] array, int arrayIndex)
    {
        items.CopyTo(array, arrayIndex);
    }

    // Define a read-only Count property that returns the number of
    // items in the collection.

    public int Count
    {
        get { return items.Count; }
    }

    // Define a read-only IsReadOnly property that returns whether
    // the collection is read-only.

    public bool IsReadOnly
    {
        get { return false; }
    }

    // Define a Remove method that removes the specified item from
    // the collection and returns a bool object to indicate whether
    // the item was removed successfully.

    public bool Remove(T item)
    {
        return items.Remove(item);
    }

    #endregion

    #region IEnumerable<T> Members

    // Define a GetEnumerator method, which returns an IEnumerator<T>
    // object.

    public IEnumerator<T> GetEnumerator()
    {
        // This is not implemented. Enumerators are covered in the
        // next lesson.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        throw new NotImplementedException();
    }
    #endregion

    #region IEnumerable Members

    // Add an IEnumerable.GetEnumerator method. This is a nongeneric
    // version of the GetEnumerator method.

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Always return the result of calling the GetEnumerator
        // method.
        return GetEnumerator();
    }

    #endregion

    #region IList<T> Members

    // Add an IndexOf method that returns the index of the first
    // occurrence of an item in the collection.

    public int IndexOf(T item)
    {
        return items.IndexOf(item);
    }

    // Add an Insert method that adds an item to the collection at a
    // specified index.

    public void Insert(int index, T item)
    {
        items.Insert(index, item);
    }

    // Add a RemoveAt method that removes an item from the collection
    // at the specified index.

    public void RemoveAt(int index)
    {
        items.RemoveAt(index);
    }

    // Add an indexer that enables read/write access to items in the
    // collection, based on the specified index.

    public T this[int index]
    {
```

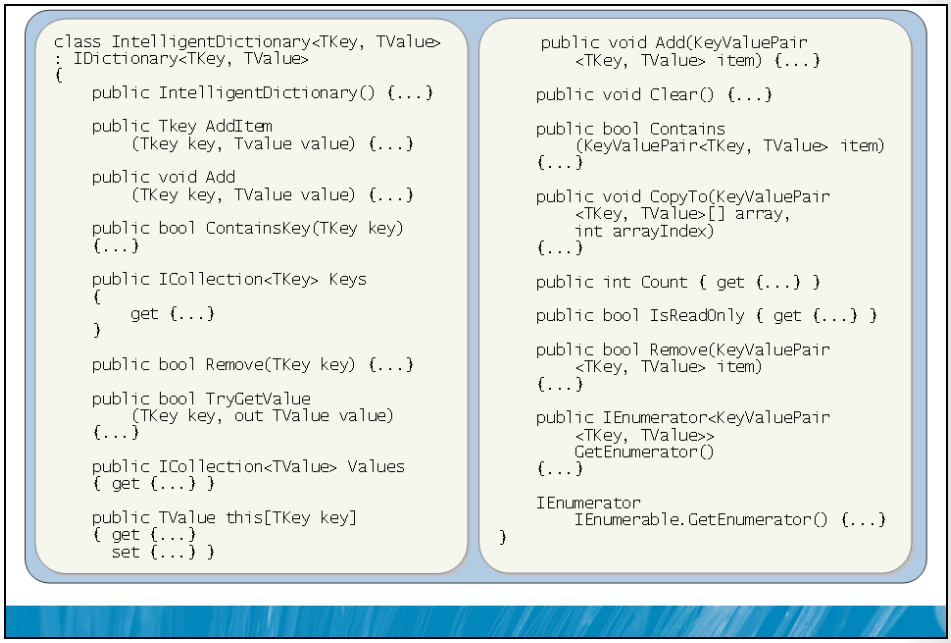
MCT USE ONLY. STUDENT USE PROHIBITED

```
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
    #endregion
}
```

Question: You develop an application that stores data in a file in a custom format. The file structure is predefined and uses a structure that is similar to, but not the same as, XML. Data that is stored in the file is stored in a specified order, and the application needs to both read and write to the file. How can you use a custom collection class to make it easier to use the data file in the application?

MCT USE ONLY. STUDENT USE PROHIBITED

Implementing a Dictionary Collection Class



Key Points

You implement a dictionary collection class in almost the same way that you implement a simple collection class—the difference is that you should implement the **IDictionary<TKey, TValue>** interface instead of the **IList<T>** interface. The **IDictionary<TKey, TValue>** interface inherits from the **ICollection<T>** interface, so you must also implement that interface.

The following code example shows a dictionary collection class that uses a **Dictionary** object as an internal data store. An ordinary **Dictionary** collection requires that the items that are added to the collection have a unique key value, and the **Add** method of the **IDictionary** interface throws an **ArgumentException** exception if a duplicate key is detected. The **IntelligentDictionary** collection class in the following code example implements an additional method called **AddItem**. This method takes the same parameters as the **Add** method of the **IDictionary** interface, but if it detects a duplicate key, it generates a new random key and tries again. The key that is used is passed back as the return value.

MCT USE ONLY. STUDENT USE PROHIBITED

```
class IntelligentDictionary<TKey, TValue> : IDictionary<TKey, TValue>
{
    // Define a Dictionary<TKey, TValue> object, or some other
    // data store to store items that are added to the collection.
    Dictionary<TKey, TValue> items;

    // Define a constructor for the class.
    public IntelligentDictionary()
    {
        items = new Dictionary<TKey, TValue>();
    }

    // Add a key/value pair to the dictionary.
    // Detect a key clash, and generate a new random key if necessary.
    // Return the key that is used.
    public TKey AddItem(TKey key, TValue value)
    {
        // The implementation details are not shown.
        ...
    }

    #region IDictionary<TKey,TValue> Members

    // Define an Add method that adds a new item with the specified
    // key and value to the collection.
    public void Add(TKey key, TValue value)
    {
        items.Add(key, value);
    }

    // Define a ContainsKey method that returns a bool object
    // if the collection contains a particular key.
    public bool ContainsKey(TKey key)
    {
        return items.ContainsKey(key);
    }

    // Define a Keys property that returns a collection of keys
    // that are stored in the collection.
    public ICollection<TKey> Keys
    {
        get { return items.Keys; }
    }

    // Define a Remove method that removes an item with the specified
    // key from the collection and returns a bool value that indicates
    // whether the method succeeded.
    public bool Remove(TKey key)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        return items.Remove(key);
    }

    // Define a TryGetValue method that returns a bool value to
    // indicate whether the method was successful, and if it was, sets
    // the value of the output parameter to the value that corresponds
    // to the specified key.
    public bool TryGetValue(TKey key, out TValue value)
    {
        return items.TryGetValue(key, out value);
    }

    // Define a Values property that returns an ICollection object
    // of the values that are stored in the collection.
    public ICollection<TValue> Values
    {
        get { return items.Values; }
    }

    // Define an indexer that reads/writes the value for an item in
    // the collection, based on the key specified.
    public TValue this[TKey key]
    {
        get
        {
            return items[key];
        }
        set
        {
            items[key] = value;
        }
    }

    #endregion

    #region ICollection<KeyValuePair<TKey,TValue>> Members

    // Define an overload of the Add method to implement the
    // ICollection<T> interface. Use the KeyValuePair<TKey, TValue>
    // generic type as the type parameter.
    public void Add(KeyValuePair<TKey, TValue> item)
    {
        // Call the Add method that was already defined by using the
        // properties of the item parameter as the parameters for the
        // Add method.
        Add(item.Key, item.Value);
    }
    // Define a Clear method that removes all of the items from the
    // collection.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    public void Clear()
    {
        items.Clear();
    }

    // Define an overload of the Contains method to implement the
    // ICollection<T> interface. Use the KeyValuePair<TKey, TValue>
    // generic type as the type parameter.
    public bool Contains(KeyValuePair<TKey, TValue> item)
    {
        return ((ICollection<KeyValuePair
                    <TKey, TValue>>)items).Contains(item);
    }

    // Define a CopyTo method that copies the items in the
    // collection to an array that is passed as a parameter. The array
    // has one dimension and stores instances of the
    // KeyValuePair<TKey, TValue> type, not instances of the TKey or
    // TValue types.
    public void CopyTo(KeyValuePair<TKey, TValue>[] array,
                       int arrayIndex)
    {
        ((ICollection<KeyValuePair<TKey, TValue>>)items).CopyTo
                                                    (array, arrayIndex);
    }

    // Define a Count property that returns the number of items in
    // the collection.
    public int Count
    {
        get { return items.Count; }
    }

    // Define an IsReadOnly property that returns a bool value
    // to indicate whether the collection is read-only.
    public bool IsReadOnly
    {
        get { return false; }
    }

    // Add a Remove method that removes an item from the collection,
    // based on the KeyValuePair<TKey, TValue> object that is
    // passed as a parameter.
    public bool Remove(KeyValuePair<TKey, TValue> item)
    {
        return items.Remove(item.Key);
    }

    #endregion
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    #region IEnumerable<KeyValuePair<TKey,TValue>> Members

    // Define a GetEnumerator method that returns an
    // IEnumerator<KeyValuePair<TKey, TValue>> object.
    public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()
    {
        // Enumerators are discussed in the next lesson.
        throw new NotImplementedException();
    }

    #endregion

    #region IEnumerable Members

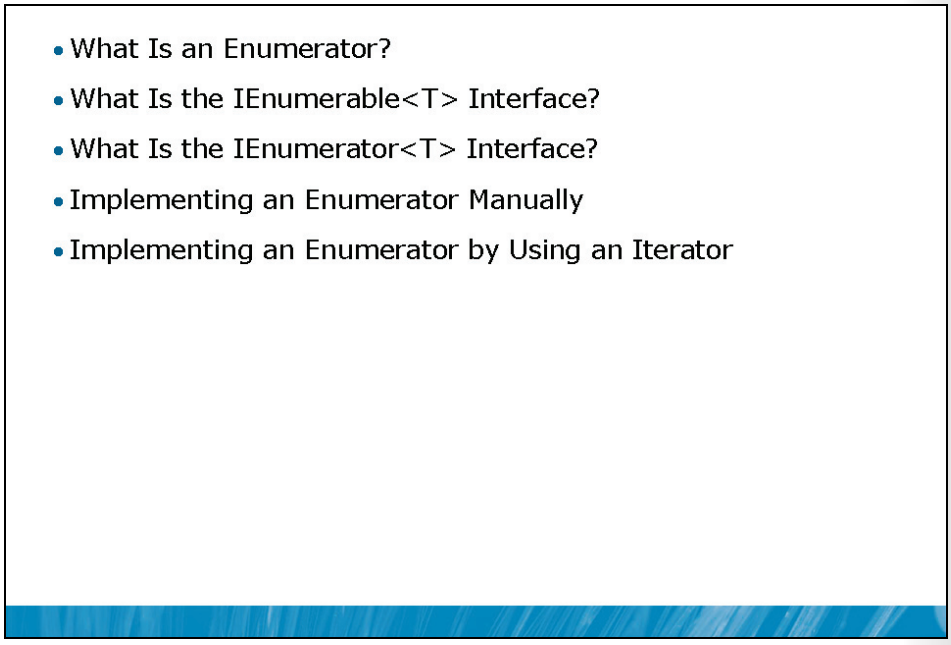
    // Add a IEnumerable.GetEnumerator method to implement the
    // nongeneric IEnumerable interface.
    IEnumerator IEnumerable.GetEnumerator()
    {
        // This should always return the result of calling the generic
        // GetEnumerator method.
        return GetEnumerator();
    }

    #endregion
}
```

Question: You develop a dictionary collection class and implement the **IDictionary<TKey, TValue>** interface. You must use the underlying **Dictionary<TKey, TValue>** class as an internal data store. When you implement the **CopyTo** method that the interface requires, you discover that the generic dictionary class does not expose a public **CopyTo** method. The generic dictionary class implements the **CopyTo** method that the interface defines explicitly. Implementing the interface explicitly avoids exposing the **CopyTo** method when referencing a collection by using the **Dictionary<TKey, TValue>** type. When might you use this approach in collections that you develop?

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Adding an Enumerator to a Custom Collection Class



C# provides the **foreach** statement to enable you to iterate through the contents of a collection. The **foreach** statement generates an enumerator object for a collection that provides the means to obtain each item in turn from the collection. When you define a custom collection class, you must provide a mechanism that the **foreach** statement can use to generate this enumerator.

This lesson introduces you to enumerators and explains how to implement enumerators in your custom collection classes.

Objectives

After completing this lesson, you will be able to:

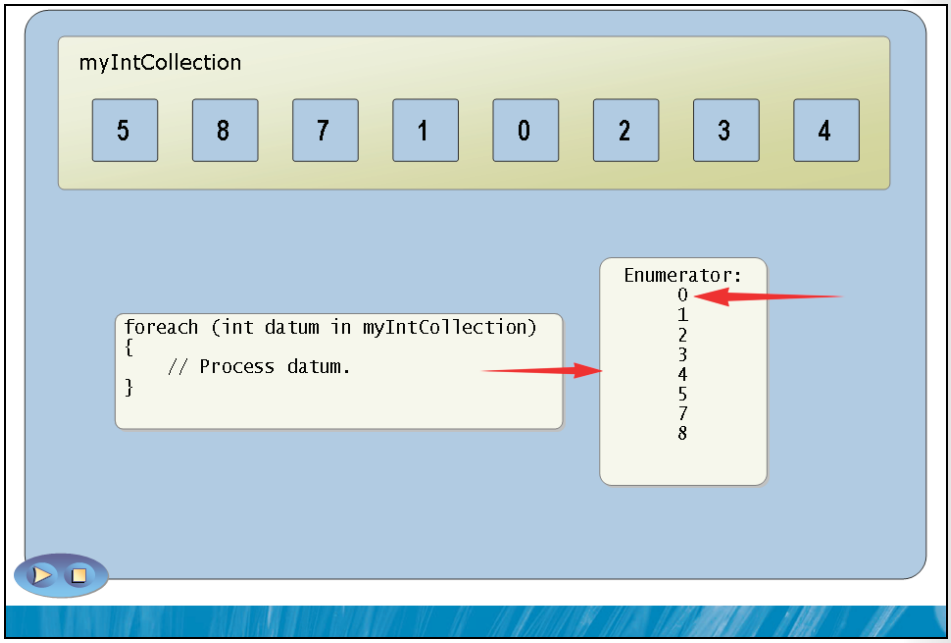
- Explain the purpose of an enumerator.
- Describe the **IEnumerable<T>** interface.
- Describe the **IEnumerator<T>** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

- Implement an enumerator manually.
- Implement an enumerator by using an iterator.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Enumerator?



Key Points

An enumerator is a type that provides a method that returns each item in a collection in an order that is appropriate to the collection. With some collections, the order is not important. For example, with a list class, the sequence in which items are added to the list governs the order. In other collection classes, the order is more significant. For example, in the **SortedList** class, the items are returned in ascending order according to the key that was specified when the items were added to the collection.

You can think of an enumerator as a pointer to items in a collection. When you instantiate an enumerator, you can initialize it to refer to the first item in the collection. You can retrieve the value that is found in the collection through this reference, and then you can move the pointer to reference the next item, in sequence.

When you use a **foreach** statement in your application, the **foreach** statement uses an enumerator that your class generates to retrieve each of the items from the collection. You can implement the enumerator to return items from your collection class in the order that is most appropriate to your collection class.

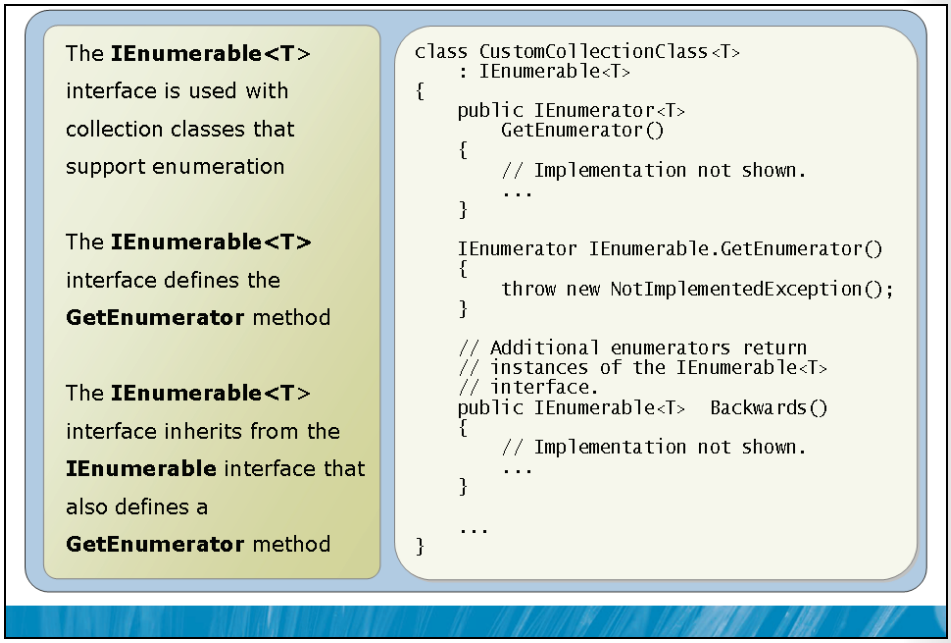
MCT USE ONLY. STUDENT USE PROHIBITED

An enumerator provides read-only access to a collection in a single direction. You may want to enable applications to iterate through items in your collection class in more than one order. For example, you may want to support forward and reverse iteration. You can implement multiple enumerators in a collection class. When an application uses the collection class by using a **foreach** class, it automatically uses the default enumerator for your class, which by convention performs forward iteration. However, you can also manually invoke a named enumerator to enable your application to iterate through the collection in a different sequence.

Question: An enumerator provides read-only access to items in a collection. Why would it be inappropriate to permit write access to items that are accessed by using an enumerator?

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the IEnumerable<T> Interface?



Key Points

A collection class that supports enumeration must implement the **IEnumerable<T>** interface. The **IEnumerable<T>** interface defines a single method called **GetEnumerator**. The **GetEnumerator** method returns an **IEnumerator<T>** object that provides the logic that the **foreach** statement requires.

When you implement the **IEnumerable<T>** interface, the **GetEnumerator** method should expose the default enumerator for your collection. You can define additional enumerators in your type; however, you must provide additional methods or properties in your collection class to enable consuming applications to access these other enumerators.

The **IEnumerable<T>** interface inherits from the **IEnumerable** interface. The **IEnumerable** interface is a nongeneric version of the **IEnumerable<T>** interface with its own nongeneric **GetEnumerator** method. This interface and this method are provided for backward compatibility with older applications that do not use generics. However, if you do not need to provide this degree of support, it is common practice to throw a **NotImplementedException** exception in the

MCT USE ONLY. STUDENT USE PROHIBITED

GetEnumerator method of the **IEnumerable** interface to ensure that applications use the generic, type-safe version of the **GetEnumerator** method.

In addition to exposing an enumerator from your type, you must also implement the **IEnumerable** interface if your type must support collection initializers. To support collection initializers, you must implement the **IEnumerable** interface and define an **Add** method in your type.

The following code example shows an implementation of the **IEnumerable<T>** interface in a collection class. The collection class also implements a method called **Backwards** that generates an enumerator that you can use to iterate through the collection in reverse order. Note that to implement an additional enumerator, you return an **IEnumerable<T>** object. This object can reference the same data as the collection, but it must implement the **GetEnumerator** method to generate an enumerator that returns the data in the appropriate sequence.

```
class CustomCollectionClass<T> : IEnumerable<T>
{
    public IEnumerable<T> Backwards()
    {
        // The implementation details are not shown.
        ...
    }

    #region IEnumerable<T> Members

    public IEnumerator<T> GetEnumerator()
    {
        // The implementation details are not shown.
        ...
    }

    #endregion

    #region IEnumerable Members

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }

    #endregion

    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows how you can use a default enumerator and the **Backwards** enumerator.

```
// The CustomCollectionClass class implements the IEnumerable<T>
// interface.
// The CustomCollectionClass class also exposes a Backwards method.
// The Backwards method returns an IEnumerable<T> instance.
CustomCollectionClass<int> intCollection = new
CustomCollectionClass<int>();
intCollection.Add(3);
intCollection.Add(5);
intCollection.Add(8);
intCollection.Add(2);
intCollection.Add(9);
intCollection.Add(1);
intCollection.Add(0);

// This foreach statement uses the default enumerator.
foreach (int temp in intCollection)
{
    ...
}

foreach (int temp in intCollection.Backwards())
{
    ...
}
```

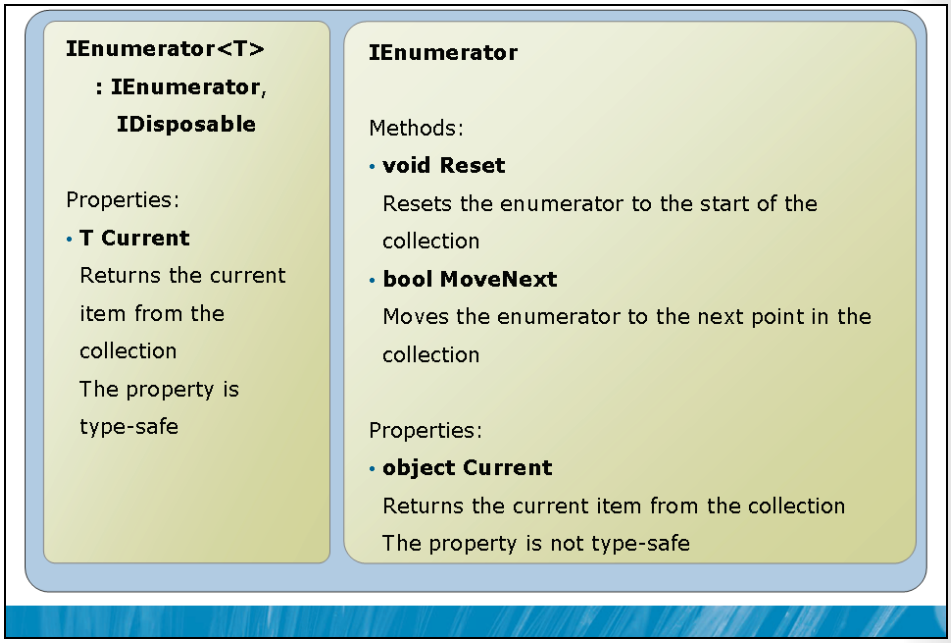
Question: How does the **foreach** statement use the **IEnumerable<T>** interface?

Additional Reading

For more information about the **IEnumerable<T>** interface, see the IEnumerable(OfT) Interface page at http://go.microsoft.com/fwlink/?LinkId=192973.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the IEnumerator<T> Interface?



Key Points

The enumerator object that the **GetEnumerator** method of the **IEnumerator<T>** interface returns must implement the **IEnumerator<T>** interface. This interface defines the following members:

- **bool MoveNext();**
- **T Current { get; }**
- **void Reset();**

When you instantiate an enumerator, it does not initially reference any items in the collection. You use the **MoveNext** method to move to the first item in the collection. This method returns a Boolean value: it returns **true** if it successfully found an item to reference; otherwise, it returns **false**. You can then retrieve the item by using the **Current** property. This property is read-only; you cannot use it to update data. You can then move to the next item by calling the **MoveNext** method and use the **Current** property to access this item. You can access the **Current** property repeatedly without calling the **MoveNext** method; however, you will

MCT USE ONLY. STUDENT USE PROHIBITED

obtain multiple references to the same item. The **Reset** method resets the internal state of the enumerator so that a subsequent call to the **MoveNext** method will place the enumerator over the first item in the collection again.

The fact that the **Current** property is read-only has an effect on the control variable in a **foreach** loop. In the following code example, the datum variable is populated by using the **Current** property. This variable is also read-only, and your code will fail to compile if you attempt to modify it explicitly.

```
foreach (int datum in integerCollectionObject)
{
    ...
    datum = datum / 2;  // Compilation error
    ...
}
```

Note that it is also an error to access the **Current** property before you call the **MoveNext** method at least once after you have created the enumerator or called the **Reset** method. In this situation, the enumerator should throw an **InvalidOperationException** exception. Using the **Current** property should also throw an exception if the last call to the **MoveNext** method returns **false**, because this indicates that you have moved past the end of the collection. The **foreach** statement prevents you from doing this accidentally, but you can also access the methods of an enumerator directly; in this case, you should be prepared to catch and handle these exceptions.

An enumerator remains valid as long as the collection remains unchanged. If the collection changes—for example, elements are added, modified, or deleted—the enumerator is irrecoverably invalidated, and the next call to the **MoveNext** or **Reset** methods should throw an **InvalidOperationException** exception. If the collection is modified between calls to the **MoveNext** method and accessing the **Current** property, the **Current** property returns the element that it is set to, even if the enumerator is already invalidated.

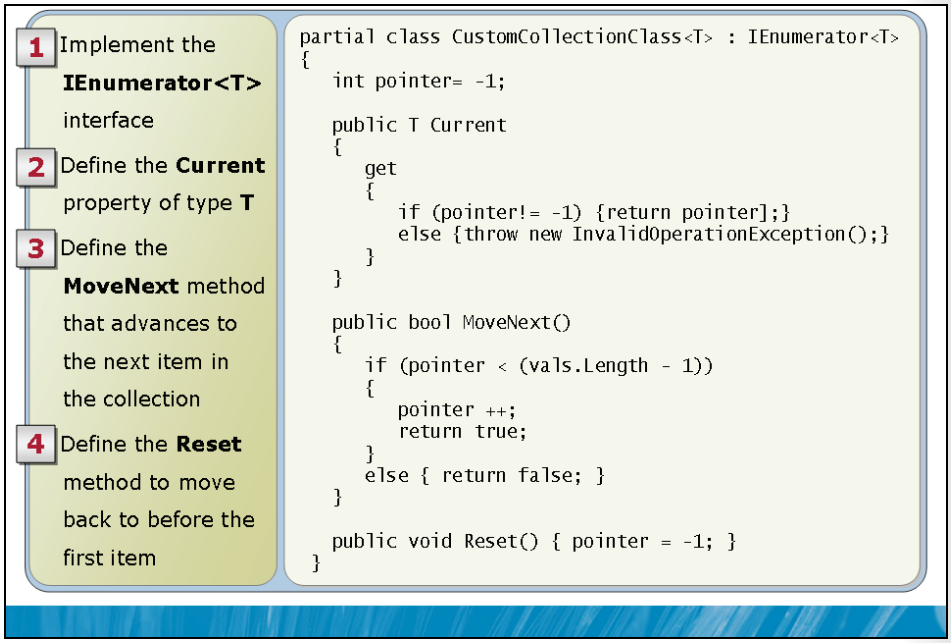
Question: How does the **MoveNext** method indicate to the caller that advancing any further would result in an invalid state because the enumerator has reached the end of the collection?

Additional Reading

For more information about the **IEnumerator<T>** interface, see the IEnumerator(OfT) Interface page at http://go.microsoft.com/fwlink/?LinkId=192974.

MCT USE ONLY. STUDENT USE PROHIBITED

Implementing an Enumerator Manually



Key Points

When you manually implement the **IEnumerator<T>** interface, you must provide implementations for the **Current** property, the **MoveNext** method, and the **Reset** method.

The following code example shows how you can implement the **IEnumerator<T>** interface manually in a custom collection class. The collection class stores data internally in an array. The **MoveNext** method maintains an index into the array, and the **Current** property returns the item that is located at this index.

```
class CustomCollectionClass<T> : IEnumerator<T>
{
    // An array that stores data internally.
    T[] vals = new T[20];

    ...

    // An int object to hold the current position of the enumerator.
    int pointer = -1;
    #region IEnumerator<T> Members
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
// Define a Current property to return the current item.
// You should include validation to ensure that the pointer is
// currently valid.
public T Current
{
    get
    {
        // Check that the state is valid.
        if (pointer != -1)
        {
            return vals[pointer];
        }
        // Throw an exception if the current state is not valid.
        else
        {
            throw new InvalidOperationException();
        }
    }
}

#endregion

#region IDisposable Members

// Implement the IDisposable interface and associated methods.
public void Dispose()
{
    GC.SuppressFinalize(this);
    finalized = true;
}

#endregion

#region IEnumerator Members

// Define a Current property for the IEnumerable interface.
// Return the Current property from the type-safe generic
// IEnumerator<T> interface.
object IEnumerator.Current
{
    get { return (object)Current; }
}

// Define a MoveNext method that advances the enumerator to the
// next item in the collection.
public bool MoveNext()
{
    // You should check that the enumerator will not pass the end
    //  of the collection if you advance it.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        if (pointer < (vals.Length - 1))
        {
            pointer++;
            // Return true if the enumerator successfully advances.
            return true;
        }
        else
        {
            // Return false if the enumerator will pass the end of the
            // collection if you advance it.
            return false;
        }
    }

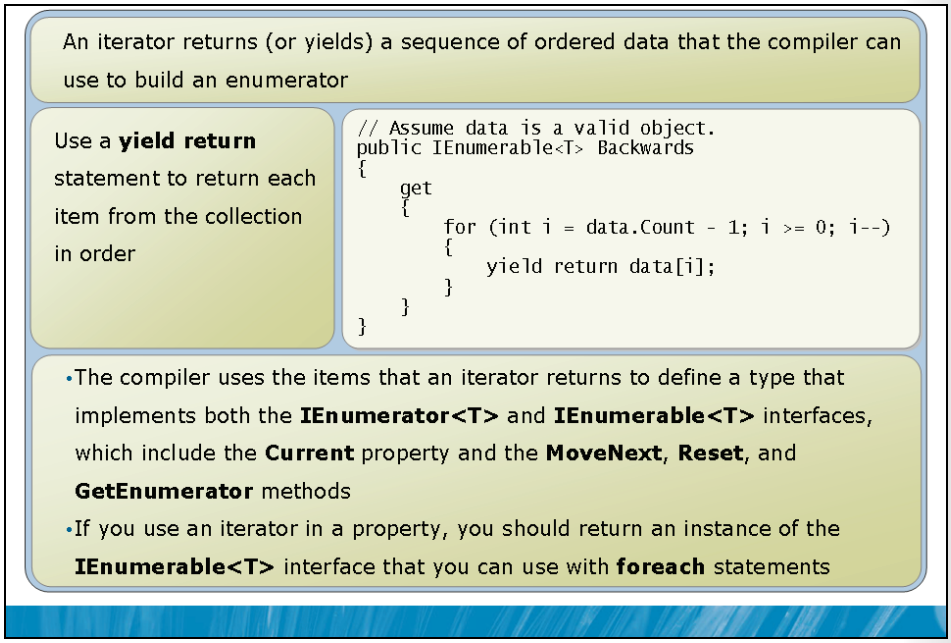
    // Define a Reset method that restores the enumerator to the
    // initial state, before the first item.
    public void Reset()
    {
        pointer = -1;
    }

    #endregion
}
```

Question: In the code example on the slide that is associated with this topic, the **get** accessor checks that the value of the pointer variable is not −1 before it returns a value. −1 is the initial value for the pointer. Why would −1 be the initial value, and why does the **Reset** method set the pointer back to −1?

MCT USE ONLY. STUDENT USE PROHIBITED

Implementing an Enumerator by Using an Iterator



Key Points

As an alternative to implementing an enumerator manually, you can implement an enumerator by using an iterator. An iterator is a block of code that yields an ordered sequence of values. In addition, an iterator is not a member of an enumerable class; instead, it specifies the sequence that an enumerator should use to return its values. In other words, an iterator is just a description of the enumeration sequence that the C# compiler can use to create its own enumerator.

The **BasicCollection<T>** class in the following code example illustrates how to implement an iterator. The class uses a **List<T>** object to hold data and provides the **FillList** method to populate this list. Also note that the **BasicCollection<T>** class implements the **IEnumerable<T>** interface. The **GetEnumerator** method is implemented by using an iterator.

```
using System;
using System.Collections.Generic;
using System.Collections;

class BasicCollection<T> : IEnumerable<T>
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    private List<T> data = new List<T>();

    public void FillList(params T [] items)
    {
        foreach (var datum in items)
        data.Add(datum);
    }

    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException
    }
}
```

If you examine the **GetEnumerator** method, you should notice that it does not appear to return an **IEnumerator<T>** type. Instead, it loops through the items in the **data** array and returns each item in turn. The key point is the use of the **yield** keyword. The **yield** keyword indicates the value that each iteration should return. You can think of the **yield** statement as calling a temporary halt to the method, passing back a value to the caller. When the caller requires the next value, the **GetEnumerator** method continues from the point at which it left off; it loops around and then yields the next value. Eventually, the data is exhausted, the loop finishes, and the **GetEnumerator** method terminates. At this point, the iteration is complete.

This is not a normal method in the usual sense. The code in the **GetEnumerator** method defines an iterator. The compiler uses this code to generate an implementation of the **IEnumerator<T>** interface that contains a **Current** property and a **MoveNext** method. This implementation exactly matches the functionality that the **GetEnumerator** method specifies. You do not see this generated code, but you can use the **yield** statement to reduce the amount of code that you must write and decrease the possibility of errors. You can invoke the enumerator that the iterator generates in the usual manner, as the following code example shows.

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
{
    Console.WriteLine(word);
}
```

This code simply displays the contents of the **bc** object in the order that the following code example shows.

```
Twas, brillig, and, the, slithy, toves
```

If you want to provide alternative iteration mechanisms to present the data in a different sequence, you can implement additional properties that implement the **IEnumerable** interface and use an iterator to return data. For example, the following code example shows the **Backwards** property of the **BasicCollection<T>** class, which emits the data in the list in reverse order.

```
class BasicCollection<T> : IEnumerable<T>
{
    ...
    public IEnumerable<T> Backwards
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }
    }
}
```

The following code example shows how to invoke this property.

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

This code displays the contents of the **bc** object in reverse order, as the following code example shows.

```
toves, slithy, the, and, brillig, Twas
```

MCT USE ONLY. STUDENT USE PROHIBITED

Question: How can you use an iterator in conjunction with a property to expose an enumerator?

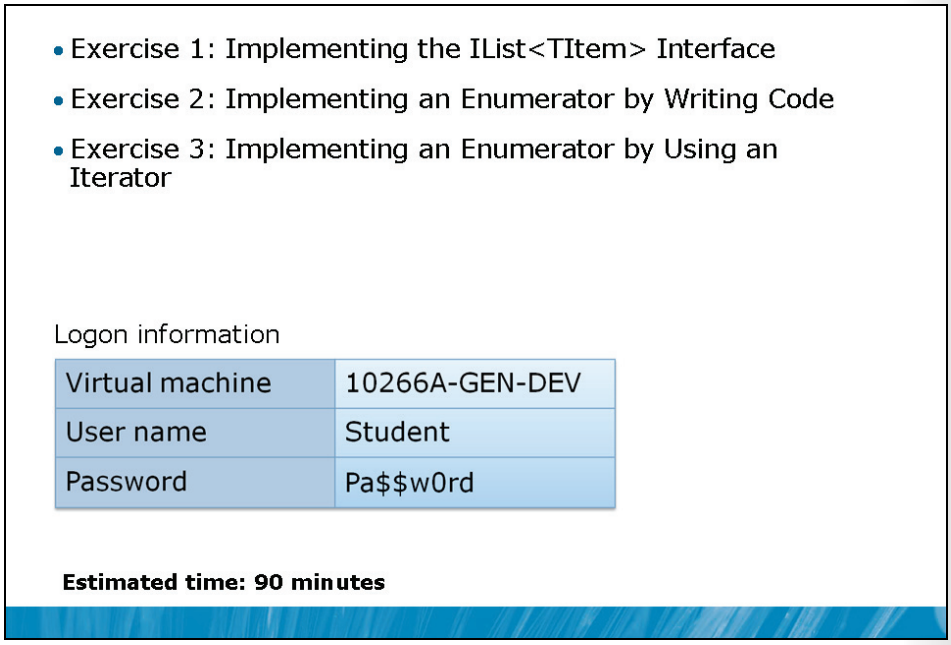
Additional Reading

For more information about iterators, see the Iterators (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192975.

For more information about the **yield** keyword, see the yield (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192976.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Building and Enumerating Custom Collection Classes



Objectives

After completing this lab, you will be able to:

- Implement the **IList<T>** interface in a generic collection class.
- Create an enumerator that implements the **IEnumerator<T>** interface.
- Implement an enumerator by using an iterator.

Introduction

In this lab, you will implement the **IList<T>** interface in a custom collection class. You will then add enumerators to this collection class.

MCT USE ONLY. STUDENT USE PROHIBITED

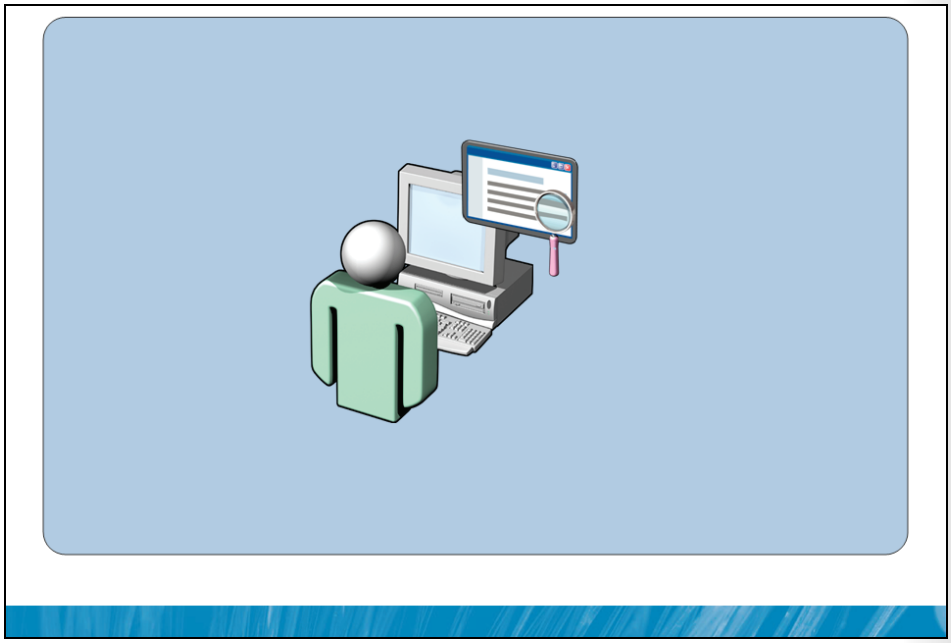
Lab Setup

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



The **Tree** class that is used to store and present sorted data works well, but it does not currently support the complete functionality that is expected of collection classes in the .NET Framework. Many of the engineering applications that Fabrikam, Inc. builds require this functionality. You have been asked to extend the **BinaryTree** class and add the necessary features.

Exercise 1: Implementing the IList<TItem> Interface

Collection classes that enable objects to be accessed by index should implement the **IList<T>** interface. The **Tree** collection class already provides some of this functionality, but you must implement the additional methods that this interface specifies.

In this exercise, you will implement the generic **IList<T>** interface in the **BinaryTree** class. You will add the following methods and properties to the class:

- **Add**. This method will add an item to the tree. The item will be inserted into the appropriate place, depending on its value.
- **Clear**. This method will return an empty **BinaryTree** object.

MCT USE ONLY. STUDENT USE PROHIBITED

- **Contains**. This method will return a Boolean value to indicate whether the binary tree contains the specified item.
- **CopyTo**. This method will throw a **NotSupportedException** exception.
- **GetEnumerator**. This method will throw a **NotImplementedException** exception in this exercise, but it will be implemented in Exercise 2.
- **IndexOf**. This method will search through the binary tree for a specified item and return its index, or −1 if the item is not found.
- **Insert**. This method will throw a **NotSupportedException** exception, because it is not appropriate for sorted data.
- **Remove**. This method will remove the specified item from the binary tree.
- **RemoveAt**. This method will remove the item at the specified index from the binary tree.
- **Count**. This method will return the number of items in the binary tree.
- **IsReadOnly**. This property will always return the value **false**.
- **Item**. This property will return the item at the specified index. The **set** accessor for this property will throw a **NotSupportedException** exception.

The main tasks for this exercise are as follows:

1. Open the CustomCollections solution.
2. Modify the **Tree** class to implement the **IList<TItem>** interface.
3. Add support for indexing items in the **Tree** class.
4. Implement the **IList<T>** interface methods and properties.
5. Use the BinaryTreeTestHarness application to test the solution.

▶ Task 1: Open the CustomCollections 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 CustomCollections solution in the E:\Labfiles\Lab 13\Ex1\Starter folder.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 2: Modify the Tree class to implement the IList<TItem> interface

1. Review the task list.
2. In the task list, locate the **TODO - Implement the generic IList<TItem> interface** task, and then double-click this task.

   This task is located in the **Tree** class.
3. Remove the **TODO - Implement the generic IList<TItem> interface** comment, and then modify the class definition to implement the generic **IList** interface. Specify the value **TItem** as the type parameter (this is the type parameter that the **Tree** class references).
4. Add method and property stubs that implement the **IList** interface.

   Visual Studio generates method stubs for each method that is defined in the interface, and adds them to the end of the class file. You will add code to complete some of these methods later in this exercise.

▶ Task 3: Add support for indexing items in the Tree class

1. In the task list, locate the **TODO - Add a member to define node position** task, and then double-click this task.
2. Remove the **TODO - Add a member to define node position** comment, and then add code to define a private integer member named **position**.
3. In the class constructor, add code to initialize the **position** member to −1.



Note: The **position** member is the index for items in the tree. When you add or remove items from the tree, you will invalidate the **position** member of any following elements in the tree. By setting the **position** member to –1, you indicate to the tree that the index has become invalid. When the application attempts to use the index to perform an action, and encounters a negative value, the application can rebuild the index by invoking the **IndexTree** method that you will add later.

4. At the beginning of the **Add** method, add code to set the **position** member to −1.
5. In the task list, locate the **TODO - Set the position member to -1** task, and then double-click this task.

   This task is located in the **Remove** method.

6. Remove the **TODO - Set the position member to -1** comment, and then add code to set the **position** member to −1.
7. In the task list, locate the **TODO - Add methods to enable indexing the tree** task, and then double-click this task.

   This task is located at the end of the **Tree** class, in the **Utility methods** code region.
8. Delete the **TODO - Add methods to enable indexing the tree** comment, and then add a method named **IndexTree**. This method should accept an integer parameter named *index*, and return an integer value. Add code to the method to perform the following actions:
   a. If the **LeftTree** property is not **null**, call the **IndexTree** method of the **LeftTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.
   b. Update the local **position** member with the value of the *index* parameter.
   c. Increment the *index* parameter.
   d. If the **RightTree** property is not **null**, call the **IndexTree** method of the **RightTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.
   e. At the end of the method, return the value of the *index* parameter.
9. After the **IndexTree** method, add a private method named **GetItemAtIndex**. This method should accept an integer parameter named *index*, and return a **Tree<TItem>** object. In the method, add code to perform the following actions:
   a. If the value of the **position** member is **−1**, call the local **IndexTree** method. Pass **0** as the parameter to the **IndexTree** method.
   b. If the value of the **position** member is greater than the value of the index variable, call the **GetItemAtIndex** method of the **LeftTree** property and return the value that is generated . Pass the value of the *index* parameter to the **GetItemAtIndex** method.
   c. If the value of the **position** member is less than the value of the index variable, call the **GetItemAtIndex** method of the **RightTree** property and return the value that is generated. Pass the value of the *index* parameter to the **GetItemAtIndex** method.
   d. At the end of the method, return a reference to the current object.

MCT USE ONLY. STUDENT USE PROHIBITED

10. After the **GetItemAtIndex** method, add a private method named **GetCount**. This method should accept an integer parameter named *accumulator*, and return an integer value. Add code to the method to perform the following actions:
    a. If the **LeftTree** property is not **null**, call the **GetCount** method of the **LeftTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.
    b. Increment the value in the accumulator variable.
    c. If the **RightTree** property is not **null**, call the **GetCount** method of the **RightTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.
    d. At the end of the method, return the value of the accumulator variable.

▶ Task 4: Implement the IList<T> interface methods and properties

1. Locate the **IndexOf** method. This method accepts a **TItem** object named **item**, and returns an integer value.

   This method should iterate through the tree and return a value that indicates the index of the **TItem** object in the tree. The method currently throws a **NotImplementedException** exception.

2. Replace the code in the **IndexOf** method with code to perform the following actions:
    a. If the *item* parameter is **null**, return the value **−1**.
    b. If the value of the **position** member is **−1**, call the **IndexTree** method and pass the value **0** as a parameter to the **IndexTree** method.
    c. Compare the value of the *item* parameter to the local **NodeData** property value:
        i. If the value of the *item* parameter is less than the value in the **NodeData** property, and if the *LeftTree* parameter is **null**, return **−1**. Otherwise, return the result of a recursive call to the **LeftTree**.**IndexOf** method, passing the **item** value to the **IndexOf** method.
        ii. If the value of the *item* parameter is greater than the value in the **NodeData** property, and if the *RightTree* parameter is **null**, return **−1**. Otherwise, return the result of a recursive call to the

MCT USE ONLY. STUDENT USE PROHIBITED

RightTree.IndexOf method, passing the **item** value to the **IndexOf** method.



Hint: Use the **CompareTo** method to compare the value in the *item* parameter and the value in the **NodeData** property.

d. At the end of the method, return the value of the local **position** member.

3. Locate the **this** indexer.

   The **this** indexer should return the **TItem** object at the index that the *index* parameter specifies. Currently, both **get** and **set** accessors throw a **NotImplementedException** exception.

4. Replace the code in the **get** accessor with code to perform the following actions:

   a. If the value of the *index* parameter is less than zero, or greater than the value of the **Count** property, throw an **ArgumentOutOfRangeException** exception with the following parameters:

      i. A string value, "index".

      ii. The *index* parameter value.

      iii. A string value, "Indexer out of range".

   b. At the end of the **get** accessor, call the **GetItemAtIndex** method. Pass the value of the index variable to the **GetItemAtIndex** method. Return the value of the **NodeData** property from the item that is retrieved by calling the **GetItemAtIndex** method.

5. Locate the **Clear** method. This method accepts no parameters, and does not return a value.

   This method should clear the contents of the tree and return it to a default state. Currently, the method throws a **NotImplementedException** exception.

6. Replace the code in the **Clear** method with code to perform the following actions:

   a. Set the **LeftTree** property to **null**.

   b. Set the **RightTree** property to **null**.

   c. Set the **NodeData** property to the default value for a **TItem** object.

MCT USE ONLY. STUDENT USE PROHIBITED

7. Locate the **Contains** method. This method accepts a *TItem* parameter, *item*, and returns a Boolean value.

   This method should iterate through the tree and return a Boolean value that indicates whether a node that matches the value of the *item* parameter exists in the tree. Currently, the method throws a **NotImplementedException** exception.

8. Replace the code in the **Contains** method with code to perform the following actions:

   a. If the value of the **NodeData** property is the same as the value of the *item* parameter, return **true**.

   b. If the value of the **NodeData** property is greater than the value of the *item* parameter, and if the **LeftTree** property is not **null**, return the result of a recursive call to the **LeftTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   c. If the value of the **NodeData** property is less than the value of the *item* parameter, and if the **RightTree** property is not **null**, return the result of a recursive call to the **RightTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   d. At the end of the method, return **false**.

9. Locate the **Count** property.

   This property is read-only, and should return an integer that represents the total number of items in the tree. Currently, the **get** accessor throws a **NotImplementedException** exception.

10. Replace the code in the **get** accessor with code to invoke the **GetCount** method, by passing **0** to the method call. Return the value that the **GetCount** method calculates.

11. Locate the **IsReadOnly** property.

    This property should return a Boolean value that signifies whether the tree is read-only.

12. Replace the code in the **get** accessor with a statement that returns the Boolean value **false**.

13. Locate the **ICollection<TItem>.Remove** method. This method accepts a *TItem* parameter named *item*, and returns a Boolean value.

    This method should check whether a node with a value that matches the *item* parameter exists in the tree, and if so, remove the item from the tree. If an item

MCT USE ONLY. STUDENT USE PROHIBITED

is removed, the method should return **true**; otherwise, the method should return **false**.



Note: This version of the **Remove** method is fully qualified with the name of the interface. This is to disambiguate it from the local **Remove** method that is defined elsewhere in the **Tree** class.

14. In the **ICollection<TItem>.Remove** method, replace the existing code with statements that perform the following actions:
    a. If the tree contains a node that matches the value in the *item* parameter, call the local **Remove** method, and then return **true**.
    b. At the end of the method, return **false**.
15. Build the solution and correct any errors.

▶ Task 5: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file and examine the **Main** method.

   The BinaryTreeTestHarness project contains code that you will use to test the completed **BinaryTree** class. You will continue to extend the **BinaryTree** class in the following exercises, so the **BinaryTree** class is not currently complete. For this reason, this exercise does not use some methods in the test harness.

   The **Main** method contains method calls to each of the test methods that you are about to examine.

2. Examine the **TestIntegerTree** method.

   The **TestIntegerTree** method tests the **Remove** and **Contains** methods, and the indexer functionality of the **BinaryTree** class. First, the method invokes the **CreateATreeOfIntegers** method to build a sample tree that contains 10 values. Then, the method invokes the **WalkTree** method, which prints each node value to the console in numerical order.



Note: The **CreateATreeOfIntegers** method creates a **Tree** object that contains the values 10, 5, 11, 5, –12, 15, 0, 14, –8, and 10 in the order that the method adds them.

The method then invokes the **Count** method and prints the result to the console. The method casts the tree to an **ICollection** object, and then calls the

MCT USE ONLY. STUDENT USE PROHIBITED

Remove method to remove the value **11** from the tree. The method again prints the result of the **Count** method to the console to prove that an item has been removed.



Note: The **BinaryTree** method contains two **Remove** methods, and in this case, the test method should invoke the interface-defined **ICollection.Remove** method. To enable the test method to do this, it must cast the **Tree** object to an **ICollection** object.

The method then tests the **Contains** method by invoking the **Contains** method with the value **11** (which has just been removed) and then −12 (which is known to exist in the list).

Finally, the method tests the tree indexer by first retrieving the index of the value **5** in the tree and printing the index to the console, and then using the same index to retrieve the value **5** from that position in the tree.

3. Examine the **TestDeleteRootNodeInteger** method.

   The **TestDeleteRootNodeInteger** method tests the functionality of the **Remove** method when it attempts to remove the tree root node. When the root node value is removed from the tree, the next available node should be copied into its place to enable the tree to continue to function.

   In this test, the root node has the value **10**. There is a second node with the value **10**, so the **Remove** method must be invoked twice to remove both values.

   The method first invokes the **CreateATreeOfIntegers** method to build a sample tree, and then prints the tree to the console by invoking the **WalkTree** method. The method then casts the **Tree** object to an **ICollection** object, and then invokes the **Remove** method twice to remove both values of **10**. Finally, the method again invokes the **WalkTree** method to verify that the tree still functions correctly.

4. Examine the **TestStringTree** method.

   This method uses similar logic to the **TestIntegerTree** method to test the **Count**, **Remove**, **Contains**, and indexer method functionality. This method uses a **BinaryTree** object that contains the string values "k203", "h624", "p936", "h624", "a279", "z837", "e762", "r483", "d776", and "k203". In this test, the **Remove** method is tested by using the "p936" string value, and the indexer is tested by using the "h624" string value.

5. Examine the **TestDeleteRootNodeString** method.

MCT USE ONLY. STUDENT USE PROHIBITED

This method uses similar logic to the **TestDeleteRootNodeInteger** method to test the **Remove** method functionality, using the same string-based tree as the **TestStringTree** method. In this test, the "k203" string value is removed twice to test root node removal.

6. Examine the **TestTestResultTree** method.

   This method uses similar logic to the **TestIntegerTree** and **TestStringTree** methods to test the **Count**, **Remove**, **Contains**, and indexer method functionality, but it uses a **BinaryTree** object based on the **TestResult** type.



Note: The **TestResult** class implements the **IComparable** interface, and uses the **Deflection** property to compare instances of the **TestResult** object. Therefore, items in this tree are indexed by their **Deflection** property value.

   In this case, the **Remove** method is tested with the **TestResult** object that has a **Deflection** value of **226**. The indexer is tested with the **TestResult** object that has a **Deflection** value of **114**.

7. Examine the **TestDeleteRootNodeTestResult** method.

   This method uses similar logic to the **TestDeleteRootNodeInteger and TestDeleteRootNodeString** methods to test the **Remove** method functionality, using the same **TestResult**-based tree as the **TestTestResultTree** method. In this test, the **TestResult** object that has a **Deflection** value of **190** is removed twice to test root node removal.

8. Run the BinaryTreeTestHarness application.

9. Verify that the output in the console window resembles the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
5
10
10
11
14
15
Count: 10
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

Note that:

a. The console shows the output of the **TestIntegerTree** method.

b. The tree is displayed in numerical order by the **WalkTree** method.

c. Initially, the list contains 10 items, and then after the **Remove** method is called, the tree contains nine items.

d. The **Remove** method removes the value **11**, so the result of the **Contains** method is **false**. Note also that the **Contains** method verifies the presence of the value **–12**.

e. The **IndexOf** method reports that the value **5** is in position 3 in the list. This is confirmed by retrieving the value in position 3, which is shown to be **5**.

10. Press ENTER, and then verify that the output in the console window resembles the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15

Remove 10 twice
After
-12
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
-8
0
5
5
11
14
15
```

Note that the tree shows two instances of the value **10** in the first list. Then, after those values are removed, the list no longer contains them. Also note that, after removing the root node value, the tree retains the remaining values and continues to function as expected.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This is the same test as the one you performed in step 9, but it is performed by using string data. Items in the list are displayed in alphabetical order.

MCT USE ONLY. STUDENT USE PROHIBITED

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
p936
r483
z837
```

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
Count: 10
Remove(def266)

Count: 9
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Contains(def266): False
Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This test is the same as the one you performed in steps 9 and 11, but this test is based on **TestResult** objects. Items are displayed in numerical order based on the value of the **Deflection** property.

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

15. Press ENTER twice to return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Implementing an Enumerator by Writing Code

The **BinaryTree** class must support the C# **foreach** statement to iterate through the contents of a **BinaryTree** object. You will implement the **GetEnumerator** method of the **IEnumerable<T>** interface in the **BinaryTree** class to return an enumerator that the **foreach** statement can use.

You will implement the **Dispose**, **MoveNext**, and **Reset** methods and the **Current** property. You will then modify the generic version of the **GetEnumerator** method of the **BinaryTree** class to return an instance of this enumerator, and you will test the enumerator by using a **foreach** loop.

The main tasks for this exercise are as follows:

1. Open the CustomCollections solution.
2. Create the **TreeEnumerator** class.
3. Add class-level variables and a constructor.
4. Add a method to populate the queue.
5. Implement the **IEnumerator<T>** and **IEnumerator** methods.
6. Implement the **IDisposable** interface.
7. Modify the **Tree** class to return a **TreeEnumerator** object.
8. Use the BinaryTreeTestHarness application to test the solution.

▶ Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex2\Starter folder.



Note: The CustomCollections solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

▶ Task 2: Create the TreeEnumerator class

- In the BinaryTree project, add a new class named **TreeEnumerator**. This class should implement the **IEnumerator** interface, and should take a type parameter, **TItem**, where the **TItem** type implements the **IComparable** interface.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Add class-level variables and a constructor

1. In the **TreeEnumerator** class, add the following members:
   a. A **Tree<TItem>** object named **currentData**, initialized to a **null** value.

      This member will store the initial **Tree** object data that is passed to the class when it is constructed, and will be used to populate the internal queue with data. The data is also stored to enable the internal queue to reset.

   b. A **TItem** object named **currentItem**, initialized to a default **TItem** object.

      This member will store the last item that is removed from the queue.

   c. A private **Queue<TItem>** object named **enumData**, initialized to a **null** value.

      This member holds an internal queue of items that the enumerator will iterate over. You will populate this queue with the items in the **Tree** object.
2. Add a constructor. The constructor should accept a *Tree<TItem>* parameter named *data*, and should initialize the **currentData** member with the value of this parameter.

▶ Task 4: Add a method to populate the queue

- Below the constructor, add a new private method named **Populate**. The method should accept a *Queue<TItem>* parameter named *enumQueue*, and a *Tree<TItem>* parameter named *tree*. It should not return a value. Add code to the method to perform the following actions:
  a. If the **LeftTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**LeftTree** property as parameters to the method.

  b. Add the **tree**.**NodeData** property value of the *tree* parameter to the **enumQueue** queue.

  c. If the **RightTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**RightTree** property as parameters to the method.

  This code walks the tree and fills the queue with each item that is found, in order.

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Implement the IEnumerator<T> and IEnumerator methods

1. In the class definition, right-click **IEnumerator**, point to **Implement Interface**, and then click **Implement Interface Explicitly**.

   Visual Studio will generate stubs for the methods and properties that the **IEnumerator<>**, **IEnumerator**, and **IDisposable** interfaces expose.

2. Locate the **Current** property.

   This property should return the last **TItem** object that was removed from the queue.

3. In the **get** accessor of the **Current** property, replace the existing code with code to perform the following actions:

   a. If the **enumData** member is **null**, throw a new **InvalidOperationException** exception with the message "Use MoveNext before calling Current".

   b. Return the value of the **currentItem** member.

4. Locate the **MoveNext** method. The method accepts no parameters and returns a Boolean value.

   The **MoveNext** method should ensure that the internal queue is initialized, retrieve the next item from the internal queue, and then store it in the **currentItem** property. If the operation succeeds, the method returns **true**, otherwise, it returns **false**.

5. In the **MoveNext** method, replace the existing code with code to perform the following actions:

   a. If the **enumData** object is **null**, create a new queue object, and then invoke the **Populate** method, passing the new **queue** object and the **currentData** member as parameters to the method call.

   b. If the **enumData** object contains any values, retrieve the first item in the queue, store it in the **currentItem** member, and then return the Boolean value **true**.

   c. At the end of the method, return the Boolean value **false**.

6. Locate the **Reset** method. This method accepts no parameters, and does not return a value.

   This method should reset the enumerator to its initial state. You do this by repopulating the internal queue with the data from the **Tree** object.

7. In the **Reset** method, replace the existing code with code that invokes the **Populate** method, passing the **enumData** and **currentData** members as parameters to the method.
8. Build the solution and correct any errors.

▶ Task 6: Implement the IDisposable interface

1. In the **TreeEnumerator** class, locate the **Dispose** method. This method accepts no parameters and does not return a value.

   The method should dispose of the class, relinquishing any resources that may not be reclaimed if they are not disposed of explicitly, such as file streams and database connections.



Note: The **Queue** object does not implement the **IDisposable** interface, so you will use the **Dispose** method of the **TreeEnumerator** class to clear the queue of any data.

2. In the **Dispose** method, replace the existing code with code that clears the **enumQueue** queue object.



Hint: Use the **Clear** method of the **Queue** class to empty a **Queue** object.

3. Build the solution and correct any errors.

▶ Task 7: Modify the Tree class to return a TreeEnumerator object

1. In the task list, locate the **TODO - Update the Tree class to return the TreeEnumerator class** task, and then double-click this task.

   This task is located in the **Tree** class.
2. Remove the comment. In the **GetEnumerator** method, replace the existing code with code that creates and initializes a new **TreeEnumerator** object. Specify the **TItem** type as the type parameter, and pass the current object as the parameter to the **TreeEnumerator** constructor. Return the **TreeEnumerator** object that is created.
3. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 8: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

   This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 1. However, it has been updated to test the enumerator functionality that you just added.

2. Examine the **TestIteratorsIntegers** method.

   This method tests the iterator functionality that you just implemented, by using the same integer tree as in Exercise 1. The method builds the tree by invoking the **CreateATreeOfIntegers** method, and then uses a **foreach** statement to iterate through the list and print each value to the console. The method then attempts to iterate through the tree in reverse order, and print each item to the console.



Note: You will add the functionality to enable reverse iteration of the tree in the next exercise. It is expected that attempting to reverse the tree will throw a **NotImplementedException** exception. The **TestIteratorsIntegers** method will catch this exception when it occurs, and print a message to the console.

3. Examine the **TestIteratorsStrings** method.

   This method uses similar logic to the **TestIteratorsIntegers** method to test the iterator functionality of the **BinaryTree** object, but it uses the same string-based tree as the one you used in Exercise 1. The method uses the **CreateATreeOfStrings** method to build the tree, iterates through the tree, and then prints all items to the console. This method also attempts to display the data in the tree in reverse order, and will encounter a **NotImplementedException** exception (you will implement this feature in the next exercise).

4. Examine the **TestIteratorsTestResults** method.

   This method uses similar logic to the **TestIteratorsIntegers** and **TestIteratorsStrings** methods to test the iterator functionality of the **BinaryTree** object. It uses a **TestResult**-based tree by invoking the **CreateATreeOfTestResults** method as in Exercise 1.

5. Run the BinaryTreeTestHarness application.

6. Verify that the output in the console window matches the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10

Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercise 1, and confirms that you have not compromised existing functionality by adding the iterator functionality.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15
Remove 10 twice
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
After
-12
-8
0
5
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercise 1, and again confirms that existing functionality works as expected.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that the items in the list are displayed in numerical order, and note that the **Reverse** method displays a message that indicates that the **Reverse** functionality is not yet implemented.

9. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
k203
k203
p936
r483
z837

Count: 10
Remove("p936")

Count: 9
Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercise 1.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
p936
r483
z837
```

MCT USE ONLY. STUDENT USE PROHIBITED

This output matches the **TestDeleteRootNodeString** method output from Exercise 1.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in step 8. It uses string data to verify the iterator functionality, and all items are displayed in alphabetical order.

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10
Remove(def266)

Count: 9
Contains(def266): False
```

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

```
Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercise 1.

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercise 1.

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in steps 8 and 11. It uses **TestResult** object data to verify the iterator functionality, and all items are displayed in numerical order based on the value of the **Deflection** property.

15. Press ENTER twice to return to Visual Studio.

Exercise 3: Implementing an Enumerator by Using an Iterator

The existing enumerator enables an application to iterate through the contents of a **BinaryTree** object in ascending order. However, some applications need to be able to iterate through a **BinaryTree** object in descending order. You have been asked to add a second enumerator to the **BinaryTree** class that can help to perform this task.

You will add an enumerator that enables a program to iterate through a **BinaryTree** object in reverse order. You will implement this enumerator by using an iterator.

The main tasks for this exercise are as follows:

1. Open the CustomCollections solution.
2. Add an enumerator to return an enumerator that iterates through data in reverse order.
3. Use the BinaryTreeTestHarness application to test the solution.

▶ Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex3\Starter folder.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: The CustomCollections solution in the Ex3 folder is functionally the same as the code that you completed in Exercise 2. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

▶ Task 2: Add an enumerator to return an enumerator that iterates through data in reverse order

1. Review the task list.
2. In the task list, locate the **TODO - Add a method to return the list in reverse order** task, and then double-click this task.

   This task is located at the end of the **Tree** class.
3. Remove the task comment, and then add a new public method named **Reverse**. The method should accept no parameters, and return an **IEnumerable** collection based on the **TItem** type parameter.
4. Add code to the method to perform the following actions:
   a. If the **RightTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **RightTree** property, and then yield each item that is found.



Hint: The **yield** statement is used in an **iterator** block to return a value to the enumerator object, or to signal the end of an iteration.

   b. Yield the value in the **NodeData** property of the current item.
   c. If the **LeftTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **LeftTree** property, and then yield each item that is found.
5. Build the solution and correct any errors.

▶ Task 3: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

   This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 2. Now that you have implemented the **Reverse** method in the **BinaryTree** object, the test application should not

MCT USE ONLY. STUDENT USE PROHIBITED

encounter the **NotImplementedException** exception in the **TestIteratorsIntegers**, **TestIteratorsStrings**, and **TestIteratorsTestResults** methods.

2. Run the BinaryTreeTestHarness application.
3. Verify that the output in the console window matches the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10
Remove(11)
Count: 9
Contains(11): False
Contains(-12): True
IndexOf(5): 3
tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercises 1 and 2, and confirms that you have not compromised existing functionality by adding the reverse iterator functionality.

4. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
14
15

Remove 10 twice

After
-12
-8
0
5
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercises 1 and 2, and again confirms that the existing functionality works as expected.

5. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
15
14
11
10
10
5
5
0
-8
-12
```

MCT USE ONLY. STUDENT USE PROHIBITED

This output is similar to the **TestIteratorsIntegers** method in Exercise 2, but the **Reverse** method is now implemented, so the tree is also displayed in descending numerical order.

6. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10
Remove("p936")

Count: 9
Contains("p936"): False
Contains("a279"): True

IndexOf("h624"): 3
tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercises 1 and 2.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
p936
r483
z837
```

This output matches the **TestDeleteRootNodeString** method output from Exercises 1 and 2.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
z837
r483
p936
k203
k203
h624
h624
e762
d776
a279
```

This test uses string data to verify the iterator functionality, and all items are displayed in alphabetical order, and then reverse alphabetical order.

MCT USE ONLY. STUDENT USE PROHIBITED

9. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10

Remove(def266)
Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercises 1 and 2.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercises 1 and 2.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
```

This test uses **TestResult** object data to verify iterator functionality. Therefore, all items are displayed in numerical order based on the value of the **Deflection**

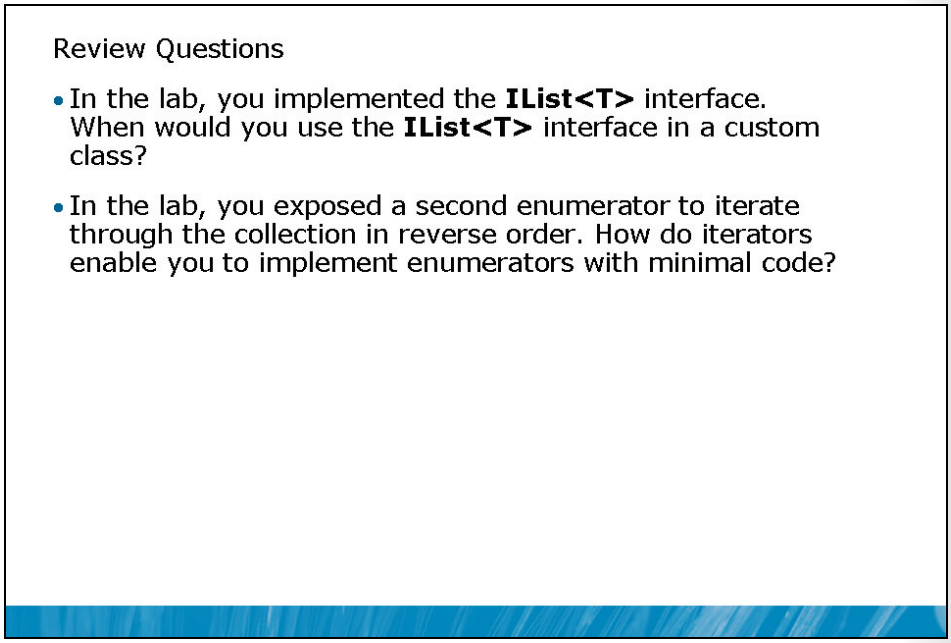
MCT USE ONLY. STUDENT USE PROHIBITED

property, and then the list is reversed to display data in descending numerical order based on the value of the **Deflection** property.

12. Press ENTER twice to return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

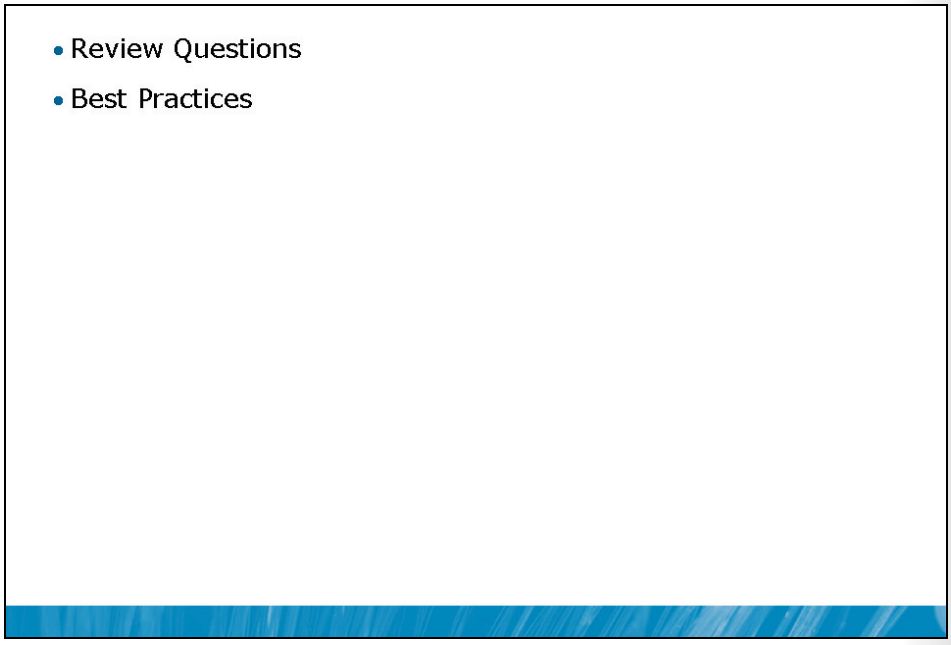


Review Questions

1. In the lab, you implemented the **IList<T>** interface. When would you use the **IList<T>** interface in a custom class?
2. In the lab, you exposed a second enumerator to iterate through the collection in reverse order. How do iterators enable you to implement enumerators with minimal code?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What are the advantages of developing a custom collection class?
2. You want to ensure that your custom collection class can be enumerated by using a **foreach** statement, and you need to support collection initializers with your type. What steps would you take to achieve this?
3. You develop a custom collection class. Applications that consume your class need to iterate over the data that is stored in your collection class in several different ways and obtain data in different orders. How would you implement this functionality in a custom collection class, and how would you provide a default order for iterating over the data in your collection class?

Best Practices Related to Developing Custom Collection Classes

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

- Implement the appropriate interfaces to ensure that your class is compatible with the standard collection-handling constructs in C#.

MCT USE ONLY. STUDENT USE PROHIBITED

- Always use the generic interfaces in preference to nongeneric interfaces when you develop a custom collection class.

Best Practices Related to Implementing Enumerators

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

- Implement the **IEnumerable<T>** interface to define a default enumerator for your type.
- Expose additional enumerators by using a property or a method in your custom collection class.
- Implement enumerators by using iterators to minimize the possibility of errors in your code.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 14

Using LINQ to Query Data

Contents:

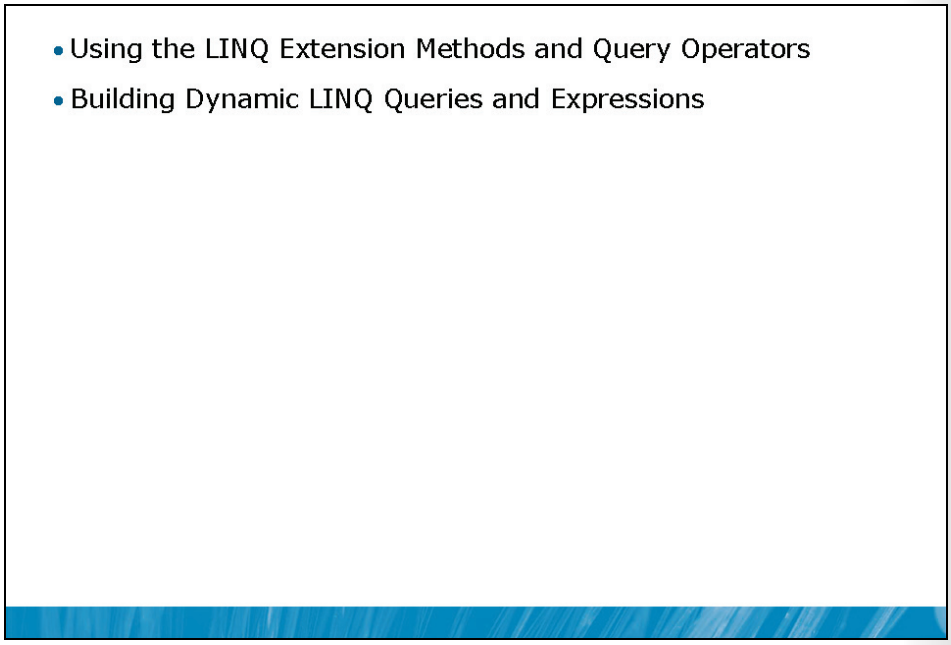
Lesson 1: Using the LINQ Extension Methods and Query Operators 14-3

Lesson 2: Building Dynamic LINQ Queries and Expressions 14-28

Lab: Using LINQ to Query Data 14-47

MCT USE ONLY. STUDENT USE PROHIBITED

Module Overview



This module introduces you to Language-Integrated Query (LINQ) queries and explains how you can use them to process data in your Microsoft® .NET Framework applications. This module also explains the difference between static and dynamic LINQ queries and describes how you can use dynamic LINQ to create highly flexible queries that you build at run time.

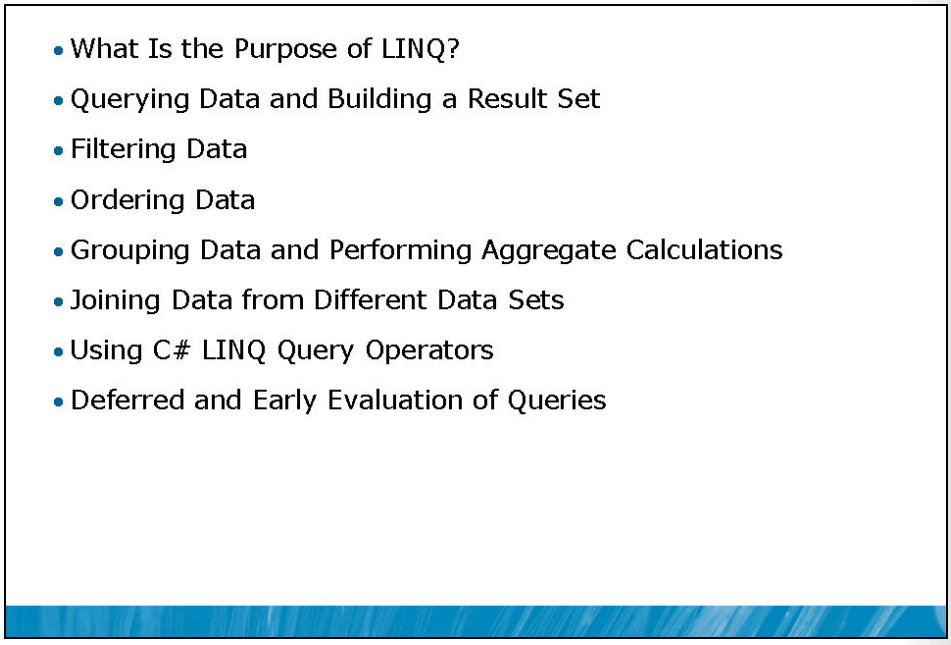
Objectives

After completing this module, you will be able to:

- Describe how to use the LINQ extension methods and query operators.
- Describe how to build dynamic LINQ queries and expressions.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1
Using the LINQ Extension Methods and Query Operators



This lesson introduces you to the LINQ feature of the .NET Framework. By using LINQ, you can abstract the mechanism that an application uses to query data from the application code. This lesson introduces you to some of the fundamental concepts and features that will enable you to use LINQ in your applications.

Objectives

After completing this lesson, you will be able to:

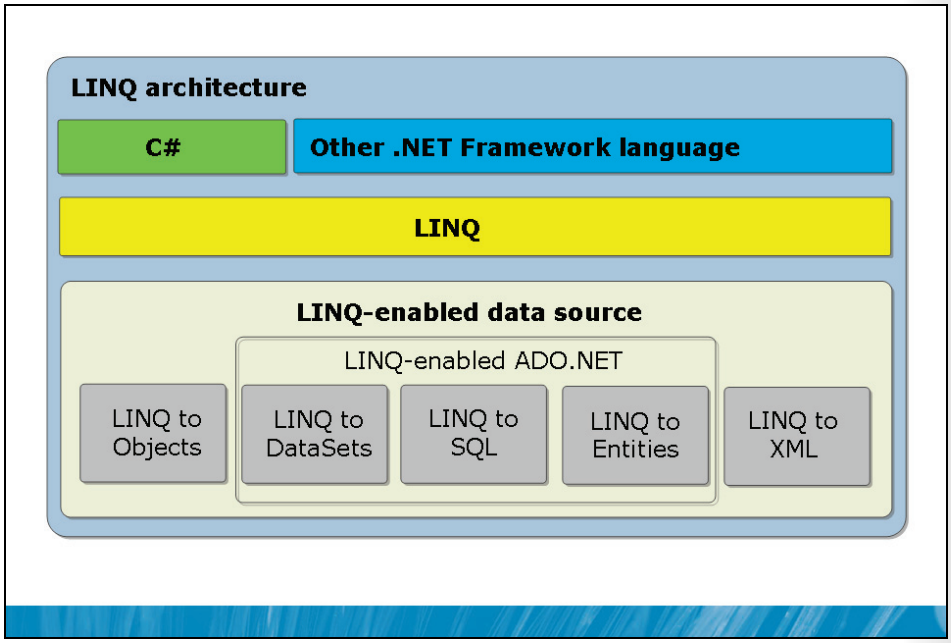
- Describe the purpose of LINQ.
- Describe how to use the **Select** extension method to query data and build an enumerable result.
- Describe how to use the **Where** extension method to filter data.
- Describe how to order data.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe how to group data and perform aggregate calculations.
- Describe how to join data from different data sets.
- Describe how the LINQ operators map to the LINQ extension methods.
- Describe how LINQ uses deferred evaluation, and describe how you can force early evaluation of a LINQ query.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the Purpose of LINQ?



Key Points

Most applications perform some kind of data processing. This processing may be trivial; for example, an application may retrieve a list of application configuration settings from a file. However, it can also be complex; for example, an application may perform a bulk update to 5 million records in a relational database. Historically, the logic to perform such operations has been tightly coupled to the application's architecture and to the structure of the data. This means that if the structure of the data changes, you may also need to make considerable changes to the data-processing logic that handles this data.

However, when you build .NET Framework applications, you can take advantage of LINQ. LINQ simplifies the development of data-processing logic by providing features that abstract the mechanisms that are required to query data from the code in applications.

LINQ uses a syntax that is similar to the Structured Query Language (SQL) statements that are used to query relational databases. By using LINQ, you provide a high-level description of the data that the application should retrieve, but you do

MCT USE ONLY. STUDENT USE PROHIBITED

not need to indicate exactly how to retrieve the data. A LINQ provider takes this description and generates the appropriate code for you.

A LINQ provider is a component that implements the necessary interfaces to expose querying capabilities to your code. Each LINQ provider is customized to work with a particular type of data source. The following list highlights some of the available LINQ providers:

- LINQ to XML
- LINQ to Objects
- LINQ to Entities
- LINQ to SQL
- LINQ to DataSets
- LINQ to SystemSearch



Note: You can also extend LINQ by creating your own providers. Creating custom providers is beyond the scope of this course.

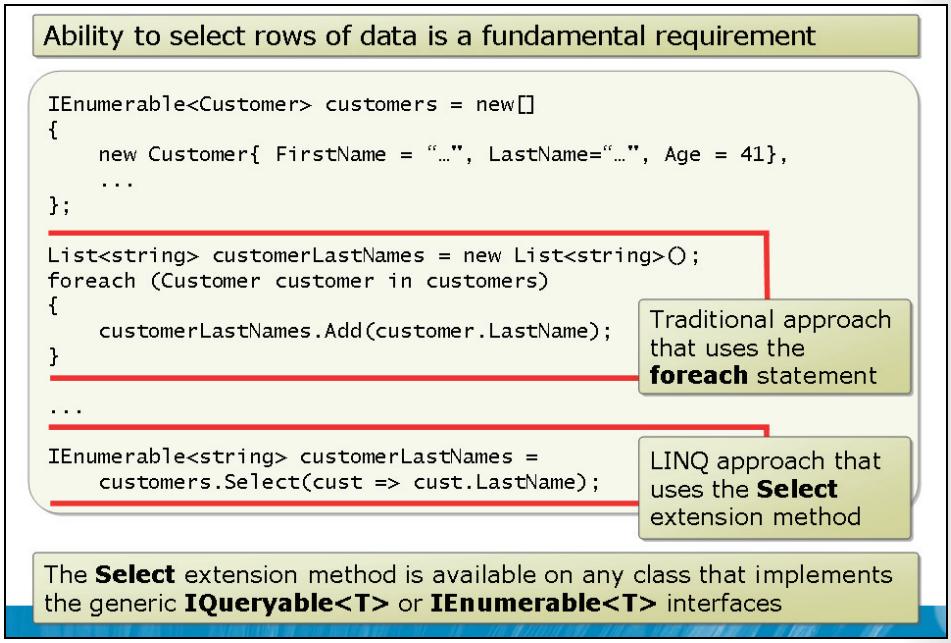
Question: Briefly summarize the purpose of LINQ.

Additional Reading

For more information about LINQ, see the Language-Integrated Query (LINQ) page at http://go.microsoft.com/fwlink/?LinkId=192977.

MCT USE ONLY. STUDENT USE PROHIBITED

Querying Data and Building a Result Set



Key Points

LINQ provides various functions that enable you to query data and build an in-memory data set. LINQ exposes these functions through a series of extension methods that are found in the **System.Linq** namespace.

Note: This topic focuses on how to use LINQ functionality through extension methods that types in the **System.Linq** namespace define. The alternative approach is to use LINQ operators, which are covered later in this lesson.

One of the most fundamental functions in LINQ is the ability to project data from a collection. For example, you may have a list of **Customer** objects, and you may want to derive a second list that contains only each customer's last name. You can use a **foreach** statement to iterate through the entire collection and extract the required data, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
IEnumerable<Customer> customers = new[]
{

    new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},
    new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},
    new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},
    new Customer{ FirstName = "David", LastName="Pelton", Age = 25},
    new Customer{ FirstName = "John", LastName="Peoples", Age = 37},
    new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},
    new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}

};

List<string> customerLastNames = new List<string>();

foreach (Customer customer in customers)
{
    customerLastNames.Add(customer.LastName);
}
```

A much simpler and more intuitive solution is to use the **Select** extension method that LINQ provides.

Using the Select Extension Method

The **Select** extension method is available on any enumerable collection class that implements the generic **IQueryable<T>** or **IEnumerable<T>** interfaces. The simplest form of the **Select** method takes a generic delegate that identifies the data to project. The simplest way to implement this delegate is to use a lambda expression. The following code example shows how you can use the **Select** extension method to derive a list of last names from an array of **Customer** objects.

```
IEnumerable<string> customerLastNames =
    customers.Select(cust => cust.LastName);
```

The value that the **Select** method returns is a reference to an enumerable collection. You can iterate through this collection to fetch and process the data. The following code example iterates through the collection that the previous **Select** method call returned and then displays the results.

```
foreach (string name in customerLastNames)
{
    Console.WriteLine(name);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

Using the Select Extension Method with Anonymous Types

If you must return data from more than one property, you can generate an anonymous type. An anonymous type is a type without a name that the compiler automatically generates. The compiler implicitly creates the type, and the type does not require an explicit declaration that you would expect with a standard type, such as a class. As with any other type, you still use the **new** keyword to instantiate an instance, but then you provide a pair of braces that define the names of the fields and values that you want the type to contain. The compiler infers the types of those fields from the values that you provide.

The following code example shows how to use the **Select** extension method to extract both the **FirstName** and **LastName** properties into an anonymous type and return an enumerable collection of this type. The code then iterates through the collection and displays the results.



Note: In this example, the name of each field in the anonymous type matches the name of the field that is retrieved from the **Customer** type. However, you can give the fields in an anonymous type any valid identifier; they do not have to be the same as the field names in the underlying type.

```
var customerNames =
    customers.Select(cust =>
        new { FirsName = cust.FirstName, LastName = cust.LastName });

foreach (var customer in customerNames)
{
    Console.WriteLine("{0} {1}",
                      customer.FirstName, customer.LastName);
}
```

Note that in this example, you do not know the name of the type that the **Select** method returns, so the customerNames and customer variables are defined as **var**.

Question: In the following code example, what does the **employeeDetails** array contain?

```
var employeeDetails = employees.Select(empl => empl.ID);
```

MCT USE ONLY. STUDENT USE PROHIBITED

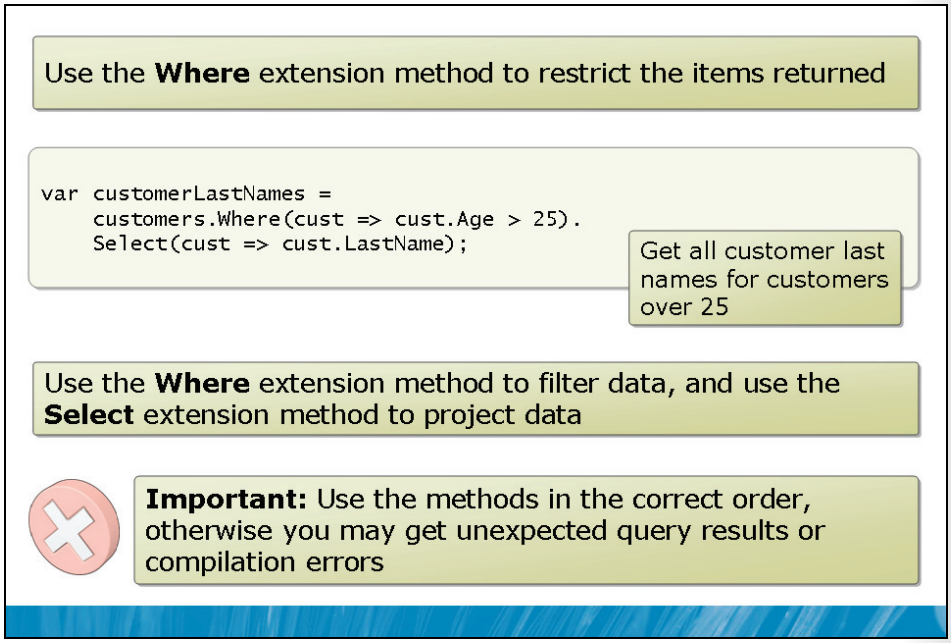
Additional Reading

For more information about LINQ expressions, see the LINQ Query Expressions (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192978.

For more information about anonymous types, see the Anonymous Types (C# Programming Guide) page at http://go.microsoft.com/fwlink/?LinkId=192979.

MCT USE ONLY. STUDENT USE PROHIBITED

Filtering Data



Key Points

The **Select** extension method enables you to specify the fields that you want to return from an enumerable collection. However, you may want to restrict the items that are returned. LINQ enables you to filter data by using the **Where** extension method.

Using the Where Extension Method

Syntactically, the **Where** extension method is similar to the **Select** extension method in that the method expects a delegate and returns an **IEnumerable** object. The delegate should take an instance of the data that is evaluated, and it should return a Boolean value to indicate whether this data should be included in the enumerable result set. As with the **Select** method, the simplest way to provide this delegate is to use a lambda expression.

The **Where** extension method returns an enumerable result; therefore, you can apply the **Select** extension method to this result and apply a projection. The following code example shows how to use the **Where** extension method to return a collection of last names for customers who are over 25 years old.

MCT USE ONLY. STUDENT USE PROHIBITED

```
var customerLastNames =
    customers.Where(cust => cust.Age > 25).
    Select(cust => cust.LastName);
```

In the preceding example, the extension methods are applied as follows:

1. The **Where** extension method is applied to the **customers** array to perform filtering; it returns an **IEnumerable** object that contains only records that match the condition. The records that are returned include every field in the **Customer** object.
2. The **Select** extension method is then applied, which specifies that only the **LastName** property should be returned.

It is important that you order the extension method calls correctly; otherwise, you may produce a query that does not compile or returns unexpected results. In the previous example, if you called the **Where** extension method after the **Select** extension method, the query would not compile. This is because the lambda expression in the **Where** extension method refers to the **Age** field in the **Customer** object, but the **Select** extension method only projects the **LastName** field. If the **Select** extension method projected the entire **Customer** object, the query would compile and run as normal.

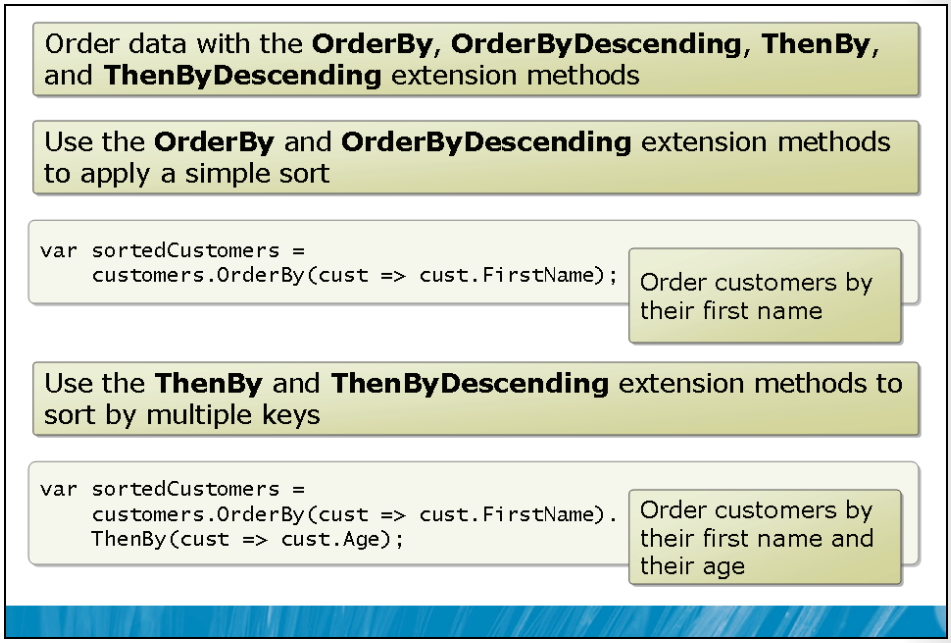
Question: What does the **result** object represent in the following code example?

```
IEnumerable<Customer> customers = new[]
{
    new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},
    new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},
    new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},
    new Customer{ FirstName = "David", LastName="Pelton", Age = 25},
    new Customer{ FirstName = "John", LastName="Peoples", Age = 37},
    new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},
    new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}
};

var result = customers.Where(cust => cust.LastName == "Poe");
```

MCT USE ONLY. STUDENT USE PROHIBITED

Ordering Data



Key Points

As you have seen, you can filter rows of data and select specific fields from types; you can also return data in a specific order. If you are familiar with SQL, you may have used the **Order By** clause. LINQ exposes similar functionality with the **OrderBy**, **OrderByDescending**, **ThenBy**, and **ThenByDescending** extension methods.

Note: Most extension methods, such as **Select**, **Where**, and **OrderBy**, are generic methods. This means that the compiler works out what types to use based on the context, but occasionally you might need to specify the appropriate type parameters if there is a potential ambiguity.

Using the OrderBy and OrderByDescending Extension Methods

The **OrderBy** and **OrderByDescending** extension methods enable you to sort data by a specific field in either ascending or descending order. Similar to other extension methods, these two methods expect a delegate that identifies the field or an expression to sort the data.

MCT USE ONLY. STUDENT USE PROHIBITED

The following code example shows how to use the **OrderBy** extension method to sort an array of **Customer** objects by the **FirstName** field in ascending order.

```
var sortedCustomers =
    customers.OrderBy(cust => cust.FirstName);
```

The following code example shows how to use the **OrderByDescending** extension method to sort an array of **Customer** objects by the **FirstName** field into descending order.

```
var sortedCustomers =
    customers.OrderByDescending(cust => cust.FirstName);
```

Using the ThenBy and ThenByDescending Extension Methods

Although the **OrderBy** and **OrderByDescending** extension methods enable you to perform basic sorting, sometimes you may want to perform additional sorting in the same statement. The **OrderBy** and **OrderByDescending** extension methods return an **IOrderedEnumerable** object, which exposes two additional extension methods: the **ThenBy** and **ThenByDescending** extension methods. The **ThenBy** and **ThenByDescending** extension methods enable you to specify additional sort keys for data that has the same value for the initial sort key.

The following code example shows how to use the **OrderBy** and **ThenBy** extension methods. The example uses the **OrderBy** method to sort the results by the **FirstName** field and then uses the **ThenBy** method to sort the records by the **Age** field.

```
var sortedCustomers =
    customers.OrderBy(cust => cust.FirstName).
    ThenBy(cust => cust.Age);
```

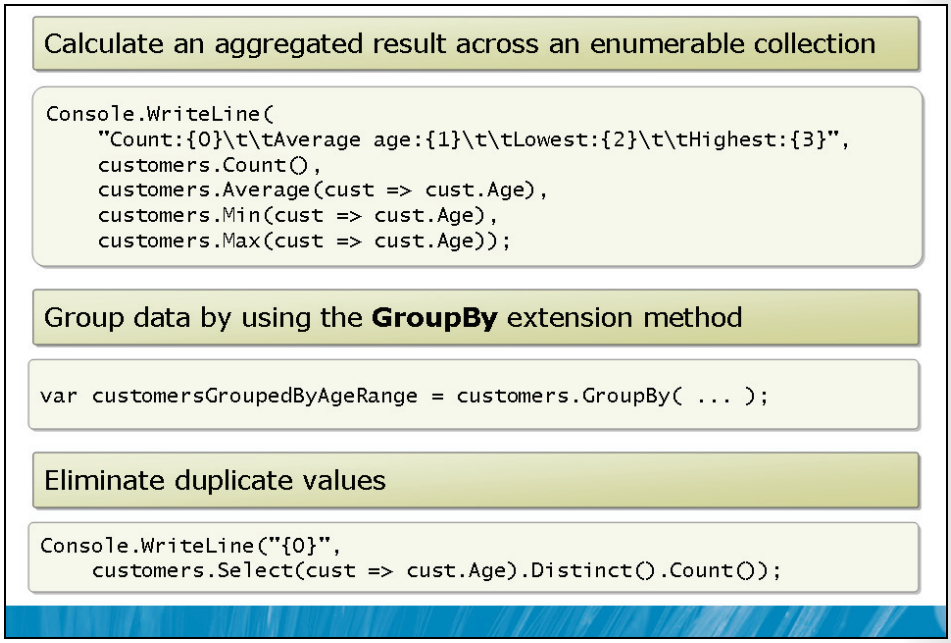
The **ThenByDescending** extension method enables you to apply an additional descending sort sequence, as the following code example shows.

```
var sortedCustomers =
    customers.OrderByDescending(cust => cust.FirstName).
    ThenByDescending(cust => cust.Age);
```

Question: Which extension method would you use to sort an array of strings into descending order?

MCT USE ONLY. STUDENT USE PROHIBITED

Grouping Data and Performing Aggregate Calculations



Key Points

LINQ provides several methods that enable you to calculate an aggregated result across an enumerable collection. These methods include **Average**, **Count**, **Max**, and **Min**. The **Average**, **Max**, and **Min** methods can take a delegate that specifies the field over which to calculate the aggregate value. The **Count** method can take a delegate that specifies a predicate to evaluate and only includes an item if the predicate returns the value **true**. However, it is common to use the **Count** method simply to determine the total number of rows in a collection, and this delegate is frequently omitted (all of the aggregate methods are overloaded).

The following code example shows how to use each of these methods over an array of **Customer** objects. The example displays the total number of customers together with the average, minimum, and maximum ages.

```
IEnumerable<Customer> customers = new[]
{
    new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},
    new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},
    new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},
    new Customer{ FirstName = "David", LastName="Pelton", Age = 29},
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    new Customer{ FirstName = "John", LastName="Peoples", Age = 37},
    new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},
    new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}
};

Console.WriteLine(
    "Count:{0}\t\tAverage age:{1}\t\tLowest:{2}\t\tHighest:{3}",
    customers.Count(), customers.Average(cust => cust.Age),
    customers.Min(cust => cust.Age), customers.Max(cust => cust.Age));
```

Grouping Data

You can perform aggregate calculations directly against an enumerable collection, but you may often want to calculate an aggregate value for different groups of data; for example, you may want to determine how many customers are inside specific age ranges. You can divide an enumerable collection into groups by using the **GroupBy** extension method. The **GroupBy** method expects a delegate that indicates how to group the data; the delegated method returns a selector, and all of the items that have the same value for this selector are placed in the same group. You can then perform the aggregate methods over each group.

The value that the **GroupBy** method returns is an enumerable collection of objects that implement the **IGrouping** interface. The **IGrouping** interface represents a collection of items that have a common key value, and you can access this key by using the **Key** property.

The following code example uses the **GroupBy** method to split customers into groups that are determined by their age. The example then displays the total number of customers in each group and the selector that identifies the group. Note that the value of the selector is available through the **Key** property of the group. The code example orders the groups by using this key and also displays the key as part of the output.

```
var customersGroupedByAgeRange = customers.GroupBy(cust =>
{
    if (cust.Age < 20)
        return "age < 20";

    if (cust.Age >= 20 && cust.Age < 40)
        return "age >= 20 and < 40";

    if (cust.Age >= 40 && cust.Age < 60)
        return "age >= 40 and < 60";
    if (cust.Age >= 60)
        return "age >= 60";
    return "Error";
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
});
foreach (var cust in
    customersGroupedByAgeRange.OrderBy(cust => cust.Key))
{
    Console.WriteLine("{0}\t\t{1}", cust.Key, cust.Count());
}
```

The results that this code generates resemble the following code example.

```
age >= 20 and < 40              4
age >= 40 and < 60              2
age >= 60               1
```

Eliminating Duplicates in Aggregate Calculations

Aggregate methods such as **Count** or **Average** include every matching item in their calculations. However, sometimes you may want to discard any duplicate values from these calculations. The query in the following code example counts the number of ages that are found in the **customers** array.

```
Console.WriteLine("{0}", customers.Select(cust => cust.Age).Count());
```

If two or more customers have the same age, this age is counted twice. You can eliminate duplicate values by using the **Distinct** extension method before you perform the aggregation, as the following code example shows.

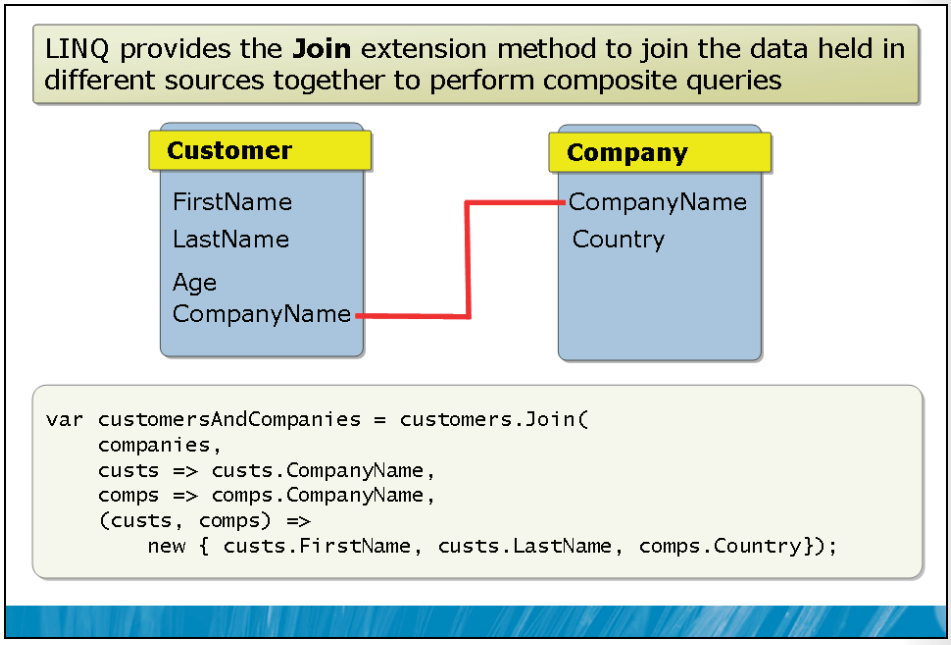
```
Console.WriteLine("{0}",
    customers.Select(cust => cust.Age).Distinct().Count());
```

Question: In the following code example, what does the **result** object contain?

```
IEnumerable<Customer> customers = new[]
{
    new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},
    new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},
    new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},
    new Customer{ FirstName = "David", LastName="Pelton", Age = 41},
    new Customer{ FirstName = "John", LastName="Peoples", Age = 23},
    new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},
    new Customer{ FirstName = "Jeff", LastName="Price", Age = 23}
};
var result = customers.Count(cust => cust.LastName.StartsWith("P"));
```

MCT USE ONLY. STUDENT USE PROHIBITED

Joining Data from Different Data Sets



Key Points

You can join together data that is held in different collections to perform composite queries as long as there is a logical key field that is common to both collections. LINQ provides the **Join** extension method for this purpose.

For example, the following code example declares two arrays. The first array contains several customers and includes their first name, last name, age, and the name of the company that employs them. The second array contains several companies and includes the company name and the country where the company is based.

```
IEnumerable<Customer> customers = new[]
{
    new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41,
        CompanyName = "Contoso"},
    new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23,
        CompanyName = "Fabrikam"},
    new Customer{ FirstName = "Keith", LastName="Harris", Age = 59,
        CompanyName = "Contoso"},
    new Customer{ FirstName = "David", LastName="Pelton", Age = 41,
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        CompanyName = "Contoso"},
    new Customer{ FirstName = "John", LastName="Peoples", Age = 23,
        CompanyName = "Contoso"},
    new Customer{ FirstName = "Toni", LastName="Poe", Age = 29,
        CompanyName = "Fabrikam"},
    new Customer{ FirstName = "Jeff", LastName="Price", Age = 23,
        CompanyName = "Fabrikam"}
};

IEnumerable<Company> companies = new[]
{
    new Company{CompanyName = "Contoso", Country = "United Kingdom"},
    new Company{CompanyName = "Fabrikam", Country = "United States"}
};
```

With the **Join** extension method, you can derive an enumerable collection that contains data from both the **customers** and **companies** arrays, joined across the **CompanyName** field.

Using the Join Extension Method to Join Multiple Data Sets

To use the **Join** extension method, you provide the following parameters:

- The enumerable collection with which you want to join.
- A method that identifies the common fields that the **Select** extension method identifies.
- A method that identifies the common key fields on which to join the different data sets.
- A method that identifies the fields that you require from the enumerable result set that the **Join** extension method returns.

The following code example shows how you can use the **Join** extension method to join the **customers** and **companies** arrays over the **CompanyName** field and display a customer's first and last name and the country in which the customer resides.

```
var customersAndCompanies = customers.Join(
    companies, custs => custs.CompanyName, comps => comps.CompanyName,
    (custs, comps) =>
        new { custs.FirstName, custs.LastName, comps.Country});
foreach (var item in customersAndCompanies)
{
    Console.WriteLine(item);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

The **Join** method returns an **IQueryable** collection that you can use in conjunction with other LINQ extension methods, such as **Select**, **Where**, and **OrderBy**.

Question: In the following code example, what does the **result** object contain?

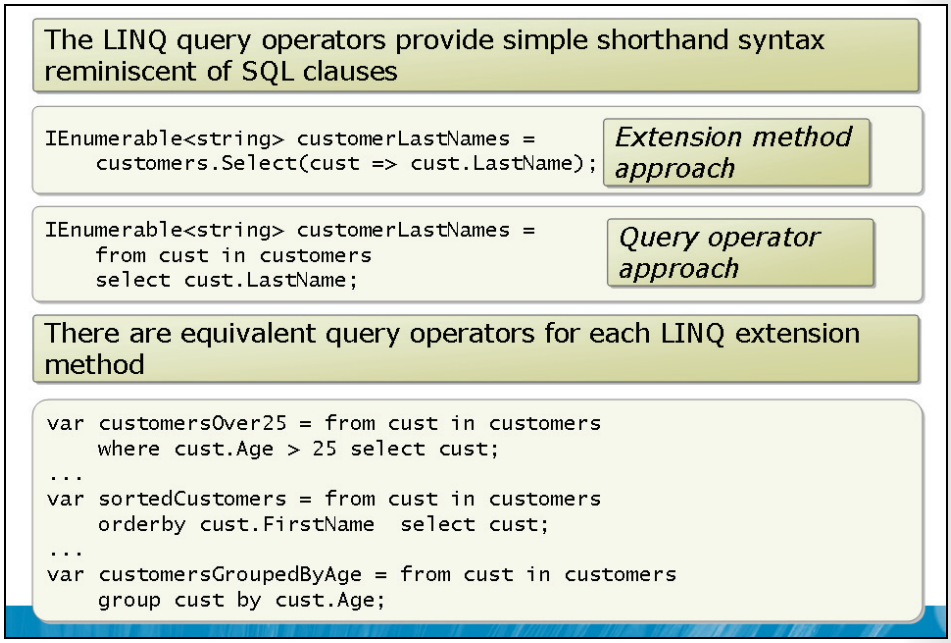
```
IEnumerable<CellPhone> phones = new[] {
    new CellPhone { ID =1, Make="...", Model="...", NetworkID=1},
    new CellPhone { ID =2, Make="...", Model="...", NetworkID=1},
    new CellPhone { ID =3, Make="...", Model="...", NetworkID=2},
};

IEnumerable<CellPhoneNetwork> networks = new[] {
    new CellPhoneNetwork { ID =1, Name="..."},
    new CellPhoneNetwork { ID =2, Name="..."}
};

var result = phones.
    Select(p => p).
    Join(
        networks,
        p => p.NetworkID,
        n => n.ID,
        (p, n) => new { p.Make, p.Model, n.Name });
```

MCT USE ONLY. STUDENT USE PROHIBITED

Using C# LINQ Query Operators



Key Points

The LINQ extension methods and types that are defined in the **System.Linq** namespace are very powerful, but the syntax can be quite cumbersome, and it is easy to introduce subtle errors that are difficult to spot and correct.

The designers of C# recognized this fact and have provided an alternative approach to the use of LINQ through a series of LINQ query operators that are part of the C# language. The LINQ query operators provide simple shorthand syntax that is reminiscent of SQL clauses; these operators expose the same functionality as the equivalent extension methods. At compile time, the shorthand syntax is compiled into the equivalent code that uses the LINQ extension methods.

For example, you have seen how to use the **Select** extension method to project a list of last names from an array of **Customer** objects, as the following code example shows.

```
IEnumerable<string> customerLastNames =
    customers.Select(cust => cust.LastName);
```

MCT USE ONLY. STUDENT USE PROHIBITED

However, you can use the **from**, **in**, and **select** query operators to achieve the same result, as the following code example shows.

```
IEnumerable<string> customerLastNames =
    from cust in customers
    select cust.LastName;
```

The query operator approach is much more intuitive and easier to follow. You can read this statement as "*from* a customer record *in* the **customer** collection, *select* the **LastName** field."



Note: If you are familiar with SQL, notice that the order of the clauses in a LINQ query are different; the **from** clause always precedes the **select** clause.

As with extension methods, you can take advantage of anonymous types and derive new entities to store a subset of fields from the original data set. The following code example shows how to use query operators and anonymous types.

```
var custs =
    from cust in customers
    select new { cust.FirstName, cust.LastName };
```

LINQ Extension Methods vs. C# LINQ Query Operators

There is an equivalent query operator for each LINQ extension method. The following list compares the syntax of the **Where**, **OrderBy**, **GroupBy**, and **Join** extension methods and shows the equivalent query operator:

- The following code examples illustrate the **Where** extension method and the **where** query operator.

```
[Where extension method example]

var customersOver25 =
    customers.Where(cust => cust.Age > 25);
```

```
[where query operator example]

var customersOver25 =
    from cust in customers
    where cust.Age > 25
    select cust;
```

MCT USE ONLY. STUDENT USE PROHIBITED



Note: When you use the query operators, the **select** clause is mandatory. The preceding example uses a **select** clause that retrieves the entire set of fields in each matching **Customer** object.

- The following code examples illustrate the **OrderBy** extension method and the **orderby** query operator.

```
[OrderBy extension method example]

var sortedCustomers =
    customers.OrderBy(cust => cust.FirstName);
```

```
[orderby query operator example]

var sortedCustomers =
    from cust in customers
    orderby cust.FirstName
    select cust;
```

- The following code examples illustrate the **GroupBy** extension method and the **groupby** query operator.

```
[GroupBy extension method example]

var customersGroupedByAge = customers.GroupBy(cust => cust.Age);
```

```
[groupby query operator example]

var customersGroupedByAge =
    from cust in customers
    group cust by cust.Age;
```

- The following code examples illustrate the **Join** extension method and the **join** query operator.

```
[Join extension method example]

var customersAndCountries =
    customers.Join(
        companies,
        cust => cust.CompanyName,
        comp => comp.CompanyName,
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        (cust, comp) => new
        {
            cust.FirstName,
            cust.LastName,
            comp.Country

        });
```

```
[join query operator example]

var customersAndCountries1 =
    from cust in customers
    join comp in companies
    on cust.CompanyName equals comp.CompanyName
        select new {cust.FirstName, cust.LastName, comp.Country};
```

You can also perform aggregate calculations on the results that LINQ queries that use query operators return. A query that uses the LINQ query operators returns an enumerable result set. To use the aggregate methods, such as **Count**, **Max**, **Min**, and **Distinct**, you must wrap the LINQ query in parentheses and then apply the appropriate method. The following code example shows how to use the **Count** extension method with a simple query that uses query operators.

```
var customerCount =
    (from cust in customers
     select cust).Count();
```

You can adopt the same approach with the other summary extension methods. The following list provides examples of how to use the **Max**, **Min**, and **Distinct** extension methods with query operators:

- The following code example illustrates how to use the **Max** extension method to get the age of the oldest customer.

```
var maxAge =
    (from cust in customers
     select cust.Age).Max();
```

- The following code example illustrates how to use the **Min** extension method to get the age of the youngest customer.

MCT USE ONLY. STUDENT USE PROHIBITED

```
var minAge =
    (from cust in customers
     select cust.Age).Min();
```

- The following code example illustrates how to use the **Distinct** extension method to get a list of all of the possible ages with duplicate values removed.

```
var possibleAges =
    (from cust in customers
     select cust.Age).Distinct();
```

Question: In the following code examples, are both queries equivalent?

```
IEnumerable<double> payments =
    new[] {232.12, 8378.53, 66.01, 4312.11, 156.00};

var queryOperatorResult =
    (from payment in payments
    where payment > 100 && payment < 500
    select payment).Max();

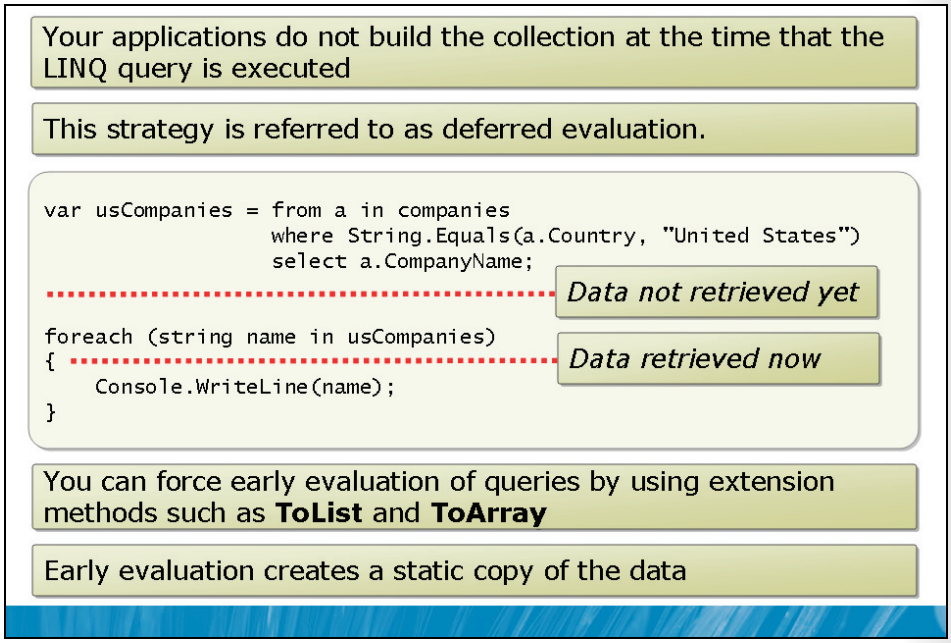
var extensionMethodResult =
    payments.Where(payment => payment > 100 && payment < 500).Max();
```

Additional Reading

For more information about LINQ query operators, see the LINQ Query Syntax versus Method Syntax (C#) page at http://go.microsoft.com/fwlink/?LinkId=192980.

MCT USE ONLY. STUDENT USE PROHIBITED

Deferred and Early Evaluation of Queries



Key Points

When you use LINQ to define an enumerable collection either by using the LINQ extension methods or by using query operators, the application does not build the collection at the time that the LINQ extension methods or query operators are executed; the data is retrieved only when you iterate over the collection. This means that the data in the original collection can change after you execute a LINQ query, but before you retrieve the data that the query identifies; you will always fetch the most up-to-date data. For example, the query in the following code example defines an enumerable collection of U.S. companies.

```
var usCompanies = from a in companies
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

The data in the **companies** array is not retrieved, and any conditions that are specified in the **where** clause are not evaluated, until you iterate through the **usCompanies** collection, as the following code example shows.

MCT USE ONLY. STUDENT USE PROHIBITED

```
foreach (string name in usCompanies)
{
    Console.WriteLine(name);
}
```

If you modify the data in the **companies** array after you define the **usCompanies** collection, but before you iterate through the collection (for example, if you add a new company that is based in the United States), you will see this new data. This strategy is referred to as deferred evaluation.

You can also force evaluation of a LINQ query early and generate a static, cached collection. This collection is a copy of the original data and will not change if the data in the collection changes. LINQ provides the **ToList** method to build a static **List** object that contains a cached copy of the data, as the following code example shows.

```
var usCompanies = from a in companies.ToList()
                  where String.Equals(a.Country, "United States")
                  select a.CompanyName;
```

This time, the list of companies is fixed when you define the query. If you add more U.S. companies to the **companies** array, you will not see them when you iterate through the **usCompanies** collection. LINQ also provides the **ToArray** method that stores the cached collection as an array.

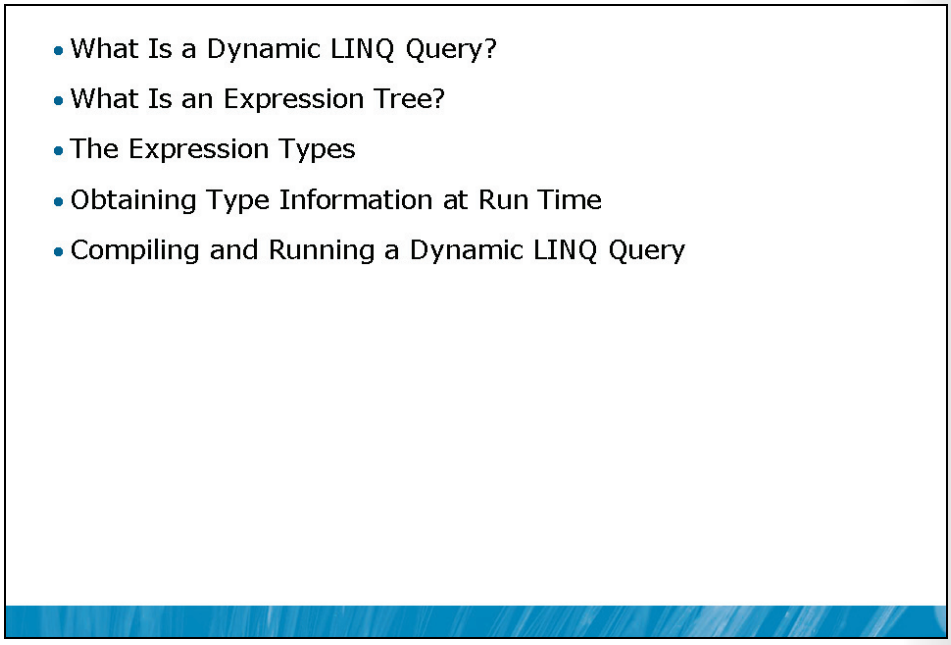
Question: In your application, you construct a LINQ query to retrieve all of the employee records from a database. In your code, after the LINQ query, you add a **foreach** statement to iterate through each record. You start to debug and step over the LINQ query and then pause before you enter the **foreach** statement. You then make a change to one of the employee records in the database. Finally, you return to the Microsoft Visual Studio® debugger and continue to step into the **foreach** statement. Will the change that you made to the employee record be visible when you iterate through the results?

Additional Reading

For more information about LINQ queries and deferred evaluation, see the Introduction to LINQ Queries (C#) page at http://go.microsoft.com/fwlink/?LinkId=192981.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Building Dynamic LINQ Queries and Expressions



By using LINQ, you can query data from application code without worrying about the implementation of the underlying data source. However, with static LINQ queries, you must provide the details of the query that you need to perform when you compile the application. You can also construct LINQ queries dynamically. This approach enables you to optimize queries by shaping them to the exact requirements that the user of the application specifies.

This lesson describes how to build dynamic LINQ queries by using the types in the **System.Linq.Expressions** namespace in the .NET Framework.

Objectives

After completing this lesson, you will be able to:

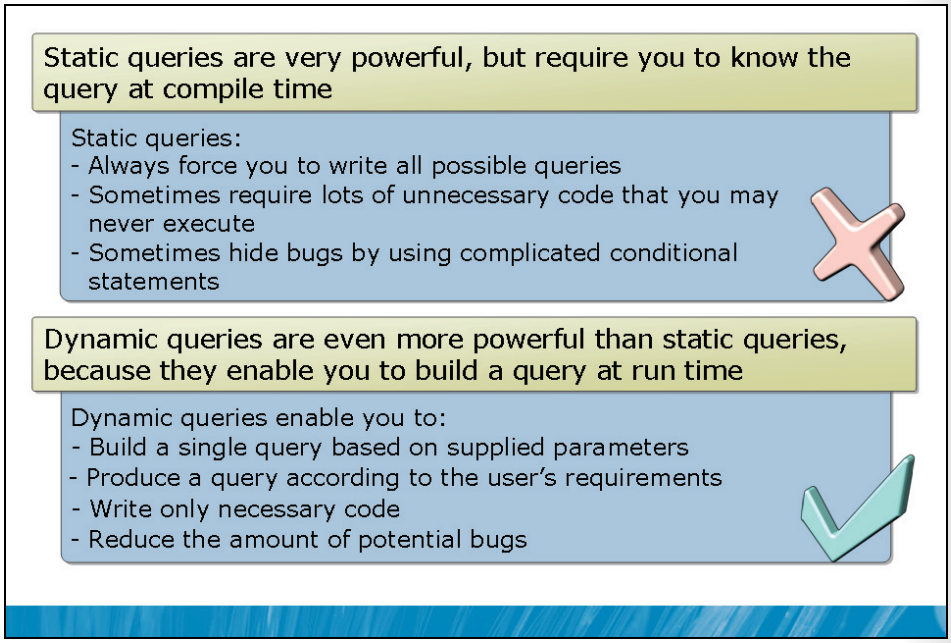
- Explain the purpose of a dynamic LINQ query.

MCT USE ONLY. STUDENT USE PROHIBITED

- Describe expression trees and explain how you can use them to construct lambda expressions to build dynamic LINQ queries.
- Explain how to use reflection to obtain metadata for a type to use in a dynamic lambda expression.
- Describe the expression types.
- Build and compile a dynamic lambda expression and use it in a LINQ query.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is a Dynamic LINQ Query?



Key Points

LINQ provides a very powerful mechanism that enables you to decouple the business logic of an application from the logic that is required to retrieve the data that the application uses. You can use static LINQ queries in most situations, but there might be occasions when the various query criteria, the order in which data is required, or even the data to be retrieved is not known until run time. For example, the form of a query may depend on user input or the results of some other processing.

For example, you may need to query a database table that stores the results of tests that students take. This table might contain fields that hold the student's name, the name of the test, and the test score for that student. Using this simple scheme, you can develop an application to query the database and apply filters to the data. You can perform simple queries by using static LINQ.

However, suppose that you need to make the application more flexible. For example, you may want to filter the results based on students who achieved a score in a particular range for one or more tests and sort them by a selection of keys (ascending/descending by test name, score, student name, or any combination).

MCT USE ONLY. STUDENT USE PROHIBITED

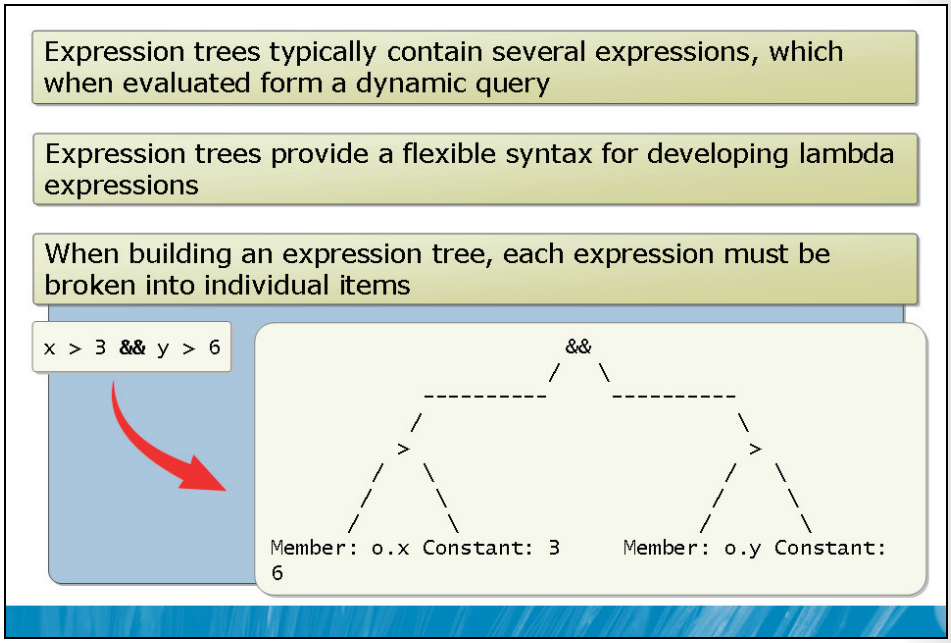
On some occasions, you might want to retrieve all three fields from the database, but on others, you may be interested only in a subset of the fields. One option is to write several queries and then use **if** statements or **switch** statements to select the query to use; however, you must write a lot of code, and this approach increases the risk that you will introduce bugs. Another option is to retrieve all of the data into a collection in memory and then programmatically fetch the appropriate data from this collection, but this approach is also potentially error-prone and obviates the rationale for employing LINQ.

Using a dynamic LINQ query helps to solve this problem. When you develop a dynamic LINQ query, you build a representation of the query in the form of an expression tree, compile the expression tree at run time, and then run it. The query then exactly represents the user's needs, without the need to develop several queries or manually process the data in memory.

Question: When might you use a dynamic LINQ query instead of a static LINQ query?

MCT USE ONLY. STUDENT USE PROHIBITED

What Is an Expression Tree?



Key Points

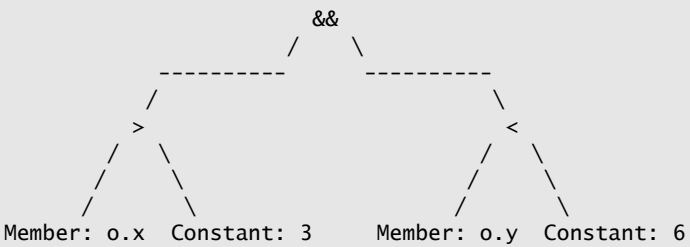
An expression tree is a data structure that represents an expression that a query uses. A simple expression might be a simple constant or a reference to a variable or a property of an object. You can combine trees that represent simple expressions into more complex trees that involve operators. For example, consider the expression **x > 3**. This expression consists of two smaller expressions (the member variable x of object **o** and the constant 3) that are combined into a bigger expression by using the greater than operator. The following diagram shows the way in which you can visualize the expression.

```
          >
        /   \
       /     \
      /       \
Member: o.x   Constant: 3
```

The corresponding expression tree for this expression consists of a **MemberExpression** object that references the **x** field in the **o** object and a **ConstantExpression** object that represents the constant value 3; these objects are

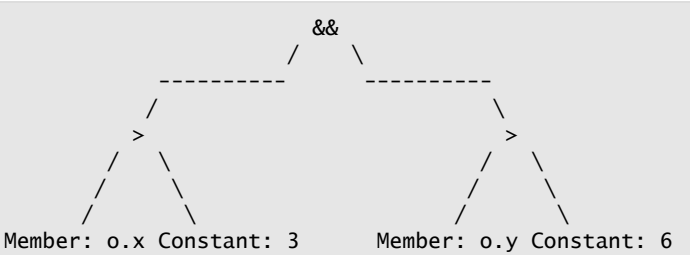
MCT USE ONLY. STUDENT USE PROHIBITED

combined into a **BinaryExpression** object that compares them by using the greater than operator. You can see from this structure that expression trees are naturally recursive. An expression tree can represent a constant or a member, or it can combine expression trees together by using operators. As an example, the expression tree in the following diagram represents the expression **o.x > 3 && o.y < 6**.



An expression tree provides a very flexible mechanism for developing an expression that you can use as part of a LINQ query. You can construct an expression tree that represents a lambda expression and then reference this lambda expression in a LINQ query. You can compile the expression tree that represents a lambda expression at run time and then invoke this expression in the same way as an ordinary lambda expression.

Question: How can you extend the tree in the following diagram to satisfy the expression **(o.x > 3 && o.y > 6) ||o. y > 20**?

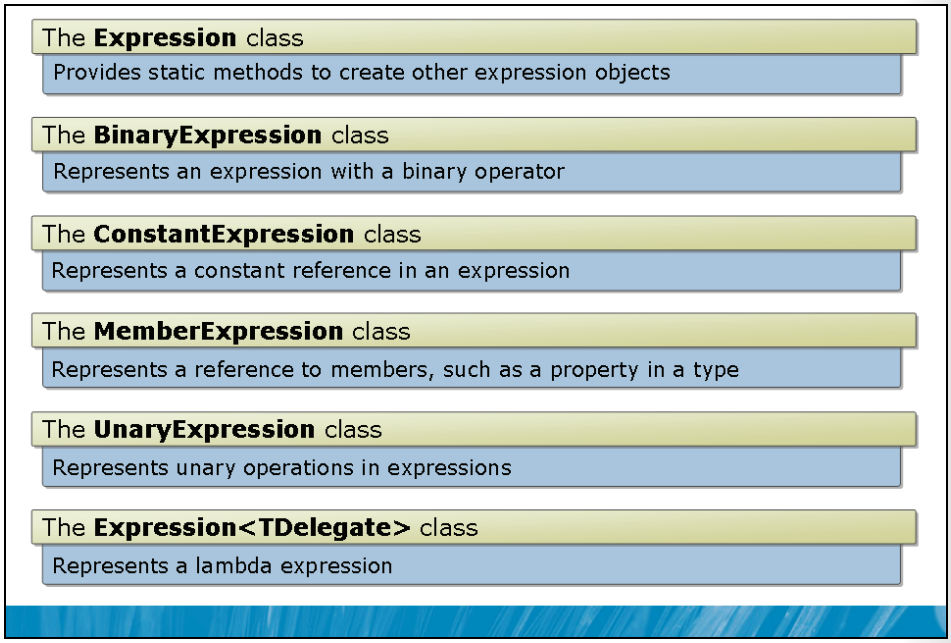


Additional Reading

For more information about expression trees, see the Expression Trees (C# and Visual Basic) page at http://go.microsoft.com/fwlink/?LinkId=192982.

MCT USE ONLY. STUDENT USE PROHIBITED

The Expression Types



Key Points

Every node in an expression tree is an expression object. The **System.Linq.Expressions** namespace in the .NET Framework class library defines expression types that you can use to represent any valid C# expression. All of these types inherit from the **Expression** class in the **System.Linq.Expressions** namespace.

Note: If you develop a Windows® Presentation Foundation (WPF) application, you should be aware that WPF defines an **Expression** type. You will need to use the **Expression** type from the **System.Linq.Expressions** namespace, so it is useful to create an alias for the namespace and use the alias to avoid ambiguity.

The following code example shows how to add an alias for the **System.Linq.Expressions** namespace named **Expressions**.

```
using Expressions = System.Linq.Expressions;
```

MCT USE ONLY. STUDENT USE PROHIBITED

Combining Expressions by Using the Expression Class

You build simple expressions by using the appropriate expression type. You can then combine these simple expressions into more complex expressions by using the **Expression** class. The **Expression** class is an abstract type that acts as the parent of all of the various expression types. The **Expression** class also provides a large number of static factory methods for combining expressions together into new expression trees by using any of the operators that are available in the C# language. For example, the **Expression.Add** factory method takes two **Expression** objects and generates a new expression tree that combines them by using the addition operator (**+**).



Note: The following sections describe some of the commonly used expression types and the static factory methods that you use to construct them. There are many other expression types available in the **System.Linq.Expressions** namespace.

The BinaryExpression Class

The **BinaryExpression** class represents an expression that uses a binary operator, such as **+**, **-**, **>**, or **<**, to combine child expressions. Methods such as **Expression.Add**, **Expression.Subtract**, **Expression.GreaterThan**, and **Expression.LessThan** return a **BinaryExpression** object. The following code example shows how to create a **BinaryExpression** object that adds two values.

```
// Assume that expression1 and expression2 are expression tree
// objects.
BinaryExpression myExp = Expression.Add(expression1, expression2);
```

The ConstantExpression Class

You represent constant values by using the **ConstantExpression** class. To create a **ConstantExpression** object, you use the **Expression.Constant** static method. You must provide the value—and optionally the type as a **System.Type** object—as arguments to the **Constant** method. Note that you can use the **typeof** operator to return the type of a type.

The following code example shows how to define a **ConstantExpression** expression for the integer value **5**.

```
ConstantExpression constant = Expression.Constant(5, typeof(int));
```

MCT USE ONLY. STUDENT USE PROHIBITED

The MemberExpression Class

The **MemberExpression** class enables you to reference a property or field of an object. For example, you can use a **MemberExpression** object to represent the **x.member** expression in the binary expression **x.member > 20**.

To create an instance of the **MemberExpression** class, you use the static **MakeMemberAccess** method of the **Expression** class. The **MakeMemberAccess** method takes two arguments: a reference to the object that contains the member in the form of an **Expression** object and a reference to the member itself in the form of a **System.Reflection.MemberInfo** object. The **Expression** object that represents the object can be a reference to a parameter that is passed to a lambda expression (this is described later in this topic) or it can be a **Constant** expression if you need to refer to an object that was created outside a lambda expression.



Note: You can use reflection to create the **MemberInfo** object that represents a member of an object. Obtaining member references by using reflection is covered in the next topic.

The following code example shows how to create a **MemberExpression** object to access a property of an object called **myData** of type **MyType**.

```
// Assume that the propertyInfo object is a valid instance of the
// MemberInfo class that references a field that the MyType type
// exposes.

MyType myData = ...;

MemberExpression member = Expression.MakeMemberAccess(
    Expression.Constant(myData), propertyInfo);
```

Occasionally, you may need to access a static member of a type. In this case, you can specify null instead of a reference to a specific instance of an object as the first parameter to the **MakeMemberAccess** method. The following code example shows how to create a **MemberExpression** object to access a static property.

```
// Assume that the propertyInfo object is a valid MemberInfo object
// that references a static property of a type.
MemberExpression staticProperty =
    Expression.MakeMemberAccess(null, propertyInfo);
```

MCT USE ONLY. STUDENT USE PROHIBITED

The UnaryExpression Class

The **UnaryExpression** class represents expressions that are based on unary operations; that is, operations that act only on a single argument. An example of a unary operation is negating a value. To create a unary expression that negates a value, you use the static **Negate** method of the **Expression** class, as the following code example shows.

```
// Assume that the parameter is a valid ParameterExpression object.
Expressions.UnaryExpression negation =
    Expressions.Expression.Negate(parameter);
```

You can also use the following methods to create a **UnaryExpression** object:

- **ArrayLength**. This method creates an expression that returns the length of a one-dimensional array that is provided as the parameter.
- **Convert**. This method creates an expression that converts the object that the first parameter specifies to the type that is specified by using a **System.Type** object as the second parameter.
- **ConvertChecked**. This method creates an expression that converts the object that the first parameter specifies to the type that is specified by using a **System.Type** object as the second parameter, and it performs overflow checking on the conversion.
- **Negate**. This method creates an expression that negates the value of an expression.
- **NegateChecked**. This method creates an expression that negates the value of an expression and checks for numeric overflow.
- **Not**. This method creates an expression that performs a bitwise **NOT** operation on the parameter.
- **Quote**. This method creates an expression that returns a constant value of the type of the parameter.
- **TypeAs**. This method creates an expression that performs an explicit reference or boxing conversion and returns null if the conversion fails.
- **UnaryPlus**. This method creates an expression that performs a unary plus operation.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Building Lambda Expressions

Lambda expressions contain two main elements: a list of parameters and a body that can return a value that is based on a calculation that involves these parameters. For example, the following code example shows a lambda expression that takes a single parameter called *x*. The body of the lambda expression is a binary expression that is based on this parameter.

```
x => x > 2
```

You use the generic **Expression<TDelegate>** type to construct an expression tree that represents a lambda expression. The *TDelegate* type parameter should reference a delegate that matches the signature of the lambda expression. In this example, you can use the type **Func<int, bool>**, which represents a delegate that takes a single integer parameter and returns a Boolean value.

You use the **Expression.Parameter** method to build an expression tree that represents a parameter to a lambda expression. This method expects the type of the parameter and a name as arguments. You can then construct a **BinaryExpression** object that references this parameter and performs the specified calculation. Finally, you can combine these two expression trees into a lambda expression by using the generic **Expression.Lambda** method, specifying the type of the delegate that the lambda expression references as the type parameter. The following code example shows how to build an expression tree for the lambda expression in the preceding code example.

```
Expression<Func<int, bool>> lambda = null;
ParameterExpression param = Expression.Parameter(typeof(int), "x");
ConstantExpression two = Expression.Constant(2, typeof(int));
BinaryExpression body = Expression.GreaterThan(param, two);
lambda = Expression.Lambda<Func<int, bool>>(body, param);
Console.WriteLine(lambda.ToString());
```

The following code example shows the output that this code generates.

```
x => (x > 2)
```

Question: How would you define an expression that checks whether a member named **x** is equal to the constant value **24**?

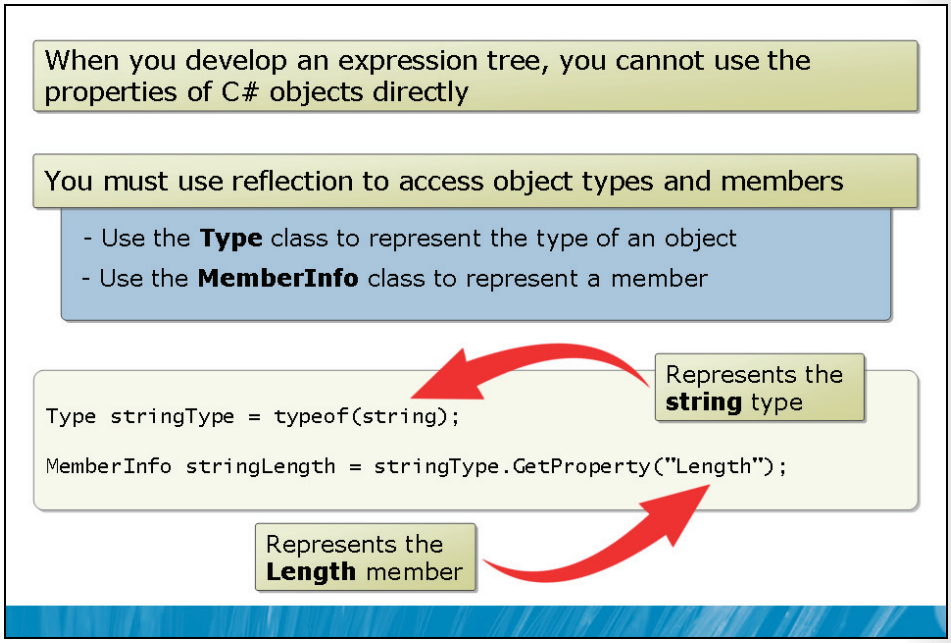
Additional Reading

For more information about the expression types in the **System.Linq.Expressions** namespace, see the System.Linq.Expressions Namespace page at http://go.microsoft.com/fwlink/?LinkId=192983.

For more information about the **Expression** class, see the Expression Class page at http://go.microsoft.com/fwlink/?LinkId=192984.

MCT USE ONLY. STUDENT USE PROHIBITED

Obtaining Type Information at Run Time



Key Points

The C# compiler enforces type safety and generates compiler errors if your code is not type safe. When you use expression trees, type safety is checked at run time after an expression has been constructed, but before it is evaluated. Some **Expression** methods, such as **MakeMemberAccess**, expect you to provide type information about the member that is referenced. Other methods, such as **Expression.Constant** and **Expression.Parameter**, expect you to provide information about the type of the constant or parameter that is involved in an expression. You can use reflection to obtain this type information dynamically.

The System.Type Type

The simplest way to obtain information about a type is to use the **typeof** operator. This operator expects a type as its argument and returns a **System.Type** object that contains detailed metadata that describes this type. The common language runtime (CLR) can use this metadata to perform type checking.

Using reflection to obtain type information is a computationally expensive operation. If you only need to reference a type once, you can use the **typeof**

MCT USE ONLY. STUDENT USE PROHIBITED

keyword inline; however, if you need to reference the same type multiple times, you should create an instance of the **Type** type and use that instance instead.

The following code example shows how to create an instance of the **Type** type to represent the **string** type.

```
Type stringType = typeof(string);
```

You can then reference the **stringType** object to specify the type of a **ParameterExpression** object, as the following code example shows.

```
ParameterExpression param = Expression.Parameter(stringType, "data");
```

The MemberInfo Type

The **Type** class enables you to reference a type; however, you may often need to access a member in a object rather than the object. To reference a member of a type, you can use the **MemberInfo** class. The **MemberInfo** class is defined in the **System.Reflection** namespace. To create a **MemberInfo** object, you use the instance methods that the **Type** class exposes. There are a large number of these methods, including **GetField** and **GetProperty**, which you can use to obtain information about the fields and properties that a type exposes. Many of these methods require you to specify the name of the member as a string. However, be warned that if you provide an invalid member name, the method will return null.

The following code example shows how to create a **MemberInfo** object to represent the **Length** property of a **string** object. You can use the **stringLength** object as the **MemberInfo** argument to the **MakeMemberAccess** method and construct an expression tree that compares the length of a string to a specific value.

```
Type stringType = typeof(string);
MemberInfo stringLength = stringType.GetProperty("Length");
string data = "Hello, World!";
MemberExpression length = Expression.MakeMemberAccess(
    Expression.Constant(data), stringLength);
ConstantExpression maxLength = Expression.Constant(25);
BinaryExpression compareLength = Expression.GreaterThanOrEqual(
    length, maxLength);
Console.WriteLine(compareLength.ToString());
```

The following code example shows the output that this code generates.

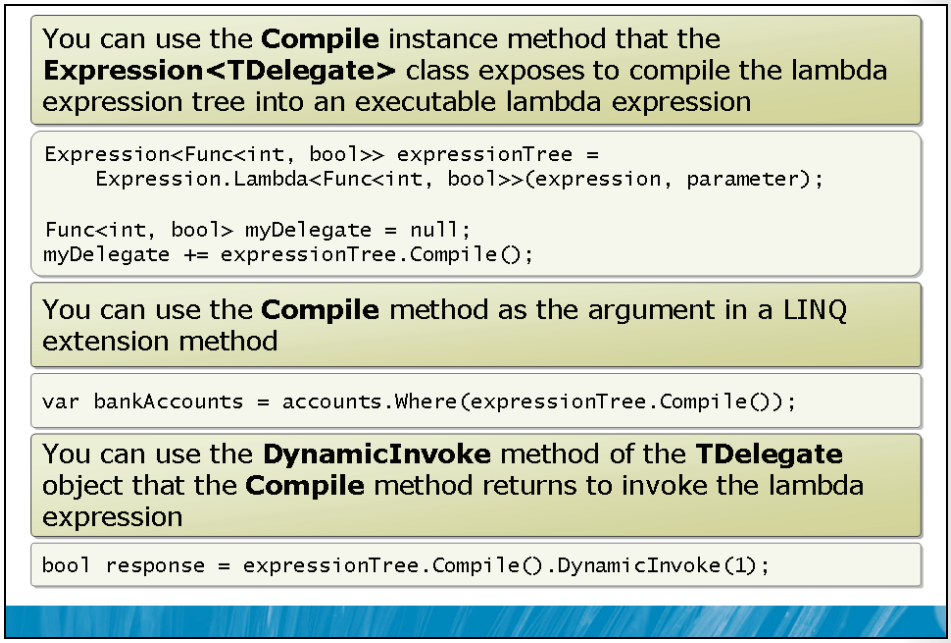
```
("Hello, World!".Length >= 25)
```

MCT USE ONLY. STUDENT USE PROHIBITED

Question: What is the result of calling the **GetProperty** method on an instance of the **Type** class that represents the **string** type and providing the string "ToString" as the argument?

MCT USE ONLY. STUDENT USE PROHIBITED

Compiling and Running a Dynamic LINQ Query



Key Points

An expression tree is simply a data structure. To use an expression tree at run time, you must compile it. You can achieve this by using the **Compile** method of the **Expression<TDelegate>** type.

The **Compile** method takes an expression tree that represents a lambda expression and returns a **TDelegate** object that you can invoke. If the lambda expression defines an element of a LINQ query, you simply iterate over the results that the query returns to run the expression.



Note: You are not confined to consuming dynamic lambda expressions in LINQ queries; you can use lambda expressions that you have constructed dynamically anywhere that you can use an ordinary, static lambda expression. To run a dynamic lambda expression in these situations, you can use the **DynamicInvoke** method of the **TDelegate** object that the **Compile** method returns.

The following code example shows a complete example of how to build a LINQ query dynamically and the results that you will see when you run it. The example

MCT USE ONLY. STUDENT USE PROHIBITED

iterates over **TestScore** objects that are held in a **List<TestScore>** collection. It retrieves all test scores with a mark of more than 50 and displays the names of the candidates with these scores in alphabetical order.

```
class TestScore
{
    public int Score { get; set; }
    public string Name { get; set; }
}

...
List<TestScore> scores = new List<TestScore>()
{
    new TestScore{Score=90, Name="Mike"},
    new TestScore{Score=60, Name="Louisa"},
    new TestScore{Score=85, Name="Antony"},
    new TestScore{Score=100, Name="Richard"},
    new TestScore{Score=45, Name="Jason"},
    new TestScore{Score=35, Name="Tom"},
    new TestScore{Score=96, Name="Chris"},
    new TestScore{Score=26, Name="Adam"},
    new TestScore{Score=71, Name="Charles"},
    new TestScore{Score=91, Name="Alison"},
    new TestScore{Score=34, Name="John"},
};

/* The following code generates a LINQ query that is equivalent
   to the following:

  var passes = scores.Where(testResult => testResult.Score > 50).
                      OrderBy(testResult => testResult.Name).
                      Select(testResult => testResult.Name);

  Note that this query involves three lambda expressions:

  1. The lambda expression for the Where method takes a TestScore
     object and returns a Boolean value.
  2. The lambda expression for the OrderBy method takes a TestScore
     object and returns a string (the data to order by).
  3. The lambda expression for the Select method takes a TestScore
     object and returns a string (the candidate names). */

// Build the lambda expression for the Where method:
// testResult => testResult.Score > 50
Type testScoreType = typeof(TestScore);

ParameterExpression testResultParam =
    Expression.Parameter(testScoreType, "testResult");
MemberInfo scoreProperty = testScoreType.GetProperty("Score");
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
MemberExpression valueInScoreProperty =
    Expression.MakeMemberAccess(testResultParam, scoreProperty);
ConstantExpression fifty = Expression.Constant(50, typeof(int));
BinaryExpression scoreGreaterThanFifty =
    Expression.GreaterThan(valueInScoreProperty, fifty);

Expression<Func<TestScore, bool>> whereExpression =
    Expression<Func<TestScore, bool>>.Lambda<Func<TestScore, bool>>
    (scoreGreaterThanFifty, testResultParam);

// Build the lambda expression for the OrderBy method:
// testResult => testResult.Name

MemberInfo nameProperty = testScoreType.GetProperty("Name");
MemberExpression valueInNameProperty =
    Expression.MakeMemberAccess(testResultParam, nameProperty);

Expression<Func<TestScore, string>> orderByExpression =
    Expression<Func<TestScore, string>>.Lambda
    <Func<TestScore, string>>
    (valueInNameProperty, testResultParam);

// Build the lambda expression for the Select method:
// testResult => testResult.Name

Expression<Func<TestScore, string>> selectExpression =
    Expression<Func<TestScore, string>>.Lambda
    <Func<TestScore, string>>
    (valueInNameProperty, testResultParam);

// Compile the lambda expressions, starting with the Where expression.
IEnumerable<TestScore> passingScores =
    scores.Where(whereExpression.Compile());

// Now append the OrderBy expression.
passingScores = passingScores.OrderBy(orderByExpression.Compile());

// Finally, add the Select expression.
IEnumerable<string> passes = passingScores.Select
    (selectExpression.Compile());

// Run the query and display the results.
foreach (var pass in passes)
{
    Console.WriteLine(pass);
}
```

The following code example shows the output that this code generates.

MCT USE ONLY. STUDENT USE PROHIBITED

```
Alison
Antony
Charles
Chris
Louisa
Mike
Richard
```

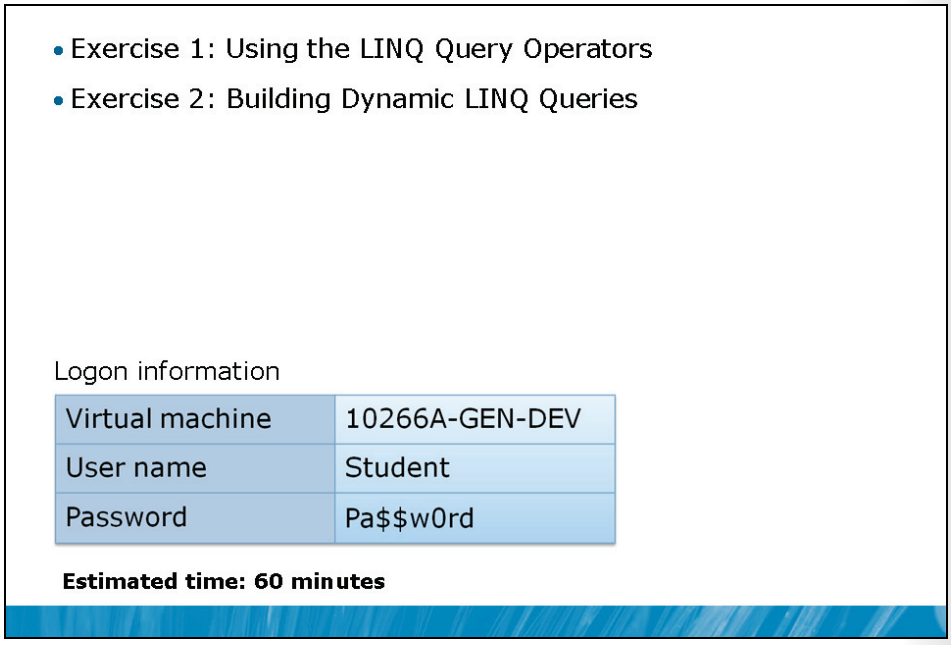
Question: What is the purpose of the **Compile** method?

Additional Reading

For more information about using expression trees to build dynamic queries, see the How to: Use Expression Trees to Build Dynamic Queries (C# and Visual Basic) page at http://go.microsoft.com/fwlink/?LinkId=192985.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab: Using LINQ to Query Data



Objectives

After completing this lab, you will be able to:

- Use the LINQ query operators to retrieve data from an enumerable collection.
- Use expression trees and the LINQ extension methods to build dynamic LINQ queries.

Introduction

In this lab, you will use the LINQ query operators to retrieve data from a collection. You will then examine how to construct a LINQ query dynamically and optimize it for better performance.

MCT USE ONLY. STUDENT USE PROHIBITED

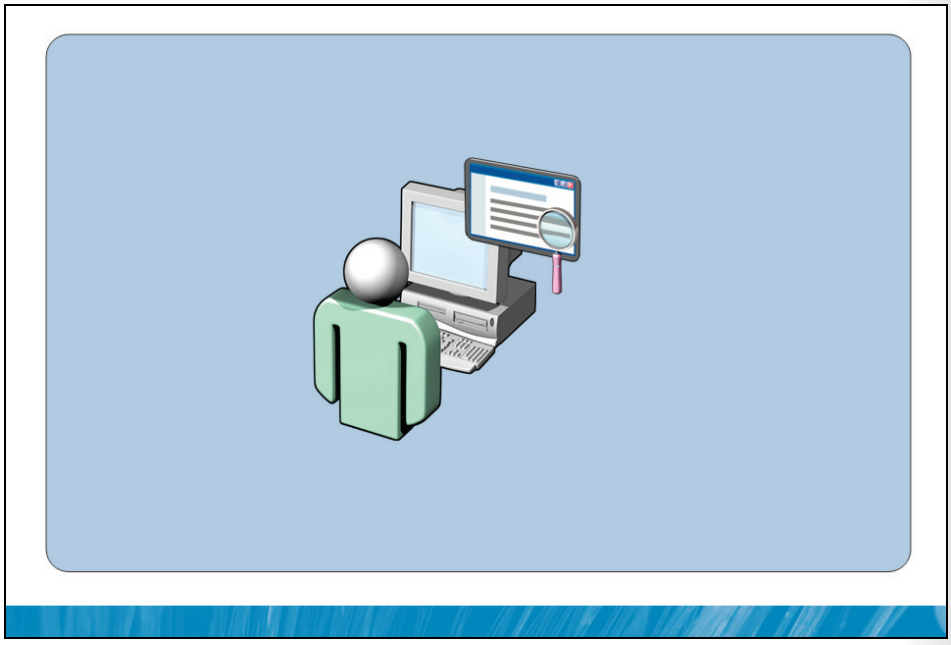
Lab Setup

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.

Fabrikam, Inc. has a large number of analytical applications that analyze data. This data is held in files that various measuring devices have generated. However, the logic in many of these applications is convoluted, and the applications themselves are difficult to use. You have been asked to build a more user-friendly application to analyze the results of one specific set of data: the results of girder stress tests. This data consists of the following fields:

- The date of the test.
- The temperature at which the test was recorded.
- The stress that was applied to the girder.
- The deflection of the girder that this stress caused.

This application must enable users to filter the data that they want to view according to the criteria that they specify.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 1: Using the LINQ Query Operators

In this exercise, you will write a program that uses the LINQ query operators to retrieve and display data. The data is provided in a binary file. The application will read this data into a **BinaryTree** object and present a WPF window that enables the user to specify criteria for viewing the data. The window will then fetch and display all of the matching data from the **BinaryTree** object in date order.

The main tasks for this exercise are as follows:

1. Open the starter solution.
2. Declare variables to specify the stress data file name and the **Tree** object.
3. Add a method to read the test data.
4. Read the test data by using a **BackgroundWorker** object.
5. Define the LINQ query.
6. Execute the query.
7. Run the query by using a **BackgroundWorker** object.
8. Display the results.
9. Test the solution.

▶ Task 1: Open the starter solution

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 14\Snippets folder.
4. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex1\Starter folder.
5. Examine the user interface (UI) for the StressDataAnalyzer application. Note the following features of the application:
   - The stress test data is generated by a stress test device. The data is stored in a binary data file, and this application reads the data from this file when the application starts to run. The application holds the data in memory by using a **Tree** object.
   - The UI contains two main areas. The upper area enables the user to specify criteria to match stress data. The lower area displays the data.

- The stress test data criteria are:
  i. The date that the test was performed.
  ii. The temperature at which the test was performed.
  iii. The stress that was applied during the test.
  iv. The deflection that resulted from applying the stress.

  Each criterion is specified as a range by using the slider controls.
- After selecting the criteria to match, the user clicks **Display** to generate a LINQ query that fetches the matching data from the **Tree** object in memory and shows the results.

▶ Task 2: Declare variables to specify the stress data file name and the Tree object

1. Review the task list.
2. In the task list, locate the **TODO - Declare filename and tree variables** task, and then double-click this task. This task is located in the **DataAnalyzer.xaml.cs** class.
3. Delete the **TODO - Declare filename and tree variables** comment, and then add code to declare the following variables:
   a. A private constant **string** object named stressDataFilename. Initialize the object with the string "E:\Labfiles\Lab 14\StressData.dat". This is the name of the data file that holds the stress data.
   b. A private **Tree** object named stressData that is based on the **TestResult** type. This **Tree** object will hold the data that is read from the stress data file. Initialize this object to null.

   The **TestResult** type is a struct that contains the following four fields, corresponding to the data for each stress test record:
   - **TestDate**. This is a **DateTime** field that contains the date on which the stress test was performed.
   - **Temperature**. This is a **short** field that contains the temperature, in Kelvin, at which the test was performed.
   - **AppliedStress**. This is another **short** field that specifies the stress, in kiloNewtons (kN), that was applied during the test.

MCT USE ONLY. STUDENT USE PROHIBITED

- **Deflection**. This is another **short** field that specifies the deflection of the girder, in millimeters (mm), when the stress was applied.

The **TestResult** type implements the **IComparable** interface. The comparison of test data is based on the value of the **Deflection** field.

▶ Task 3: Add a method to read the test data

1. In the task list, locate the **TODO - Add a method to read the contents of the StressData file** task, and then double-click this task.
2. Delete the **TODO - Add a method to read the contents of the StressData file** comment, and then add the method in the following code example, which is named **ReadTestData**. This method reads the stress data from the file and populates the **Tree** object. It is not necessary for you to fully understand how this method works, so you can either type this code manually, or you can use the Mod14ReadTestData code snippet.

```
private void ReadTestData()
{
    // Open a stream over the file that holds the test data.

    using (FileStream readStream =
        File.Open(stressDataFilename, FileMode.Open))
    {
        // The data is serialized as TestResult instances.
        // Use a BinaryFormatter object to read the stream and
        // deserialize the data.

        BinaryFormatter formatter = new BinaryFormatter();
        TestResult initialNode =
            (TestResult)formatter.Deserialize(readStream);

        // Create the binary tree and use the first item retrieved
        // as the root node. (Note: The tree will likely be
        // unbalanced, because it is probable that most nodes will
        // have a value that is greater than or equal to the value in
        // this root node - this is because of the way in which the
        // test results are generated and the fact that the TestResult
        // class uses the deflection as the discriminator when it
        // compares instances.)

        stressData = new Tree<TestResult>(initialNode);

        // Read the TestResult instances from the rest of the file
        // and add them into the binary tree.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        while (readStream.Position < readStream.Length)
        {
            TestResult data =
                (TestResult)formatter.Deserialize(readStream);
            stressData.Insert(data);
        }
    }
}
```

▶ Task 4: Read the test data by using a BackgroundWorker object

1. In the **Window_Loaded** method, add code to perform the following tasks:
   a. Create a **BackgroundWorker** object named **workerThread**.
   b. Configure the **workerThread** object; the object should not report progress or support cancellation.
2. In the **Window_Loaded** method, add an event handler for the **workerThread.DoWork** event. When the event is raised, the event handler should invoke the **ReadTestData** method.
3. Add an event handler for the **workerThread**.**RunWorkerComplete** event. When the event is raised, the event handler should perform the following tasks:
   a. Enable the **displayResults** button.
   b. Display the message 'Ready' in the **statusMessage** StatusBarItem in the status bar at the bottom of the WPF window.



Hint: Set the **Content** property of a status bar item to display a message in that item.

4. At the end of the **Window_Loaded** method, add code to perform the following tasks:
   a. Start the **workerThread BackgroundWorker** object running asynchronously.
   b. Display the message "Reading Test Data..." in the **statusMessage** item in the status bar at the bottom of the WPF window.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 5: Define the LINQ query

1. In the task list, locate the **TODO - Define the LINQ query** task, and then double-click this task. This task is located in the **CreateQuery** method.
2. Replace the existing code in the method with code that defines an **IEnumerable<TestResult>** object called **query**. Initialize the query variable with a LINQ query that retrieves all of the **TestResult** objects in the **stressData** tree that meet the following criteria. The query should order returned values by the **TestDate** property. The query should evaluate each object by using the following criteria:
   a. The value of the **TestDate** property is greater than or equal to the *dateStart* parameter value.
   b. The value of the **TestDate** property is less than or equal to the *dateEnd* parameter value.
   c. The value of the **Temperature** property is greater than or equal to the **temperatureStart** parameter value.
   d. The value of the **Temperature** property is less than or equal to the *temperatureEnd* parameter value.
   e. The value of the **AppliedStress** property is greater than or equal to the *appliedStressStart* parameter value.
   f. The value of the **AppliedStress** property is less than or equal to the *appliedStressEnd* parameter value.
   g. The value of the **Deflection** property is greater than or equal to the *deflectionStart* parameter value.
   h. The value of the **Deflection** property is less than or equal to the *deflectionEnd* parameter value.
3. At the end of the method, return the **query** object.
4. Build the solution and correct any errors.

▶ Task 6: Execute the query

1. In the task list, locate the **TODO - Execute the LINQ query** task, and then double-click this task. This task is located in the **FormatResults** method. This method takes an enumerable collection of **TestResult** objects as a parameter and generates a string that contains a formatted list of **TestResult** objects. The

parameter is the item that the **CreateQuery** method returns. Iterating through this list runs the LINQ query.

2. Delete the **TODO - Execute the LINQ query** comment, and then add code to the **FormatResults** method to perform the following task:
   - For each item that the query returns, format and append the details of each item to the **builder** StringBuilder object. Each item should be formatted to display the following properties in a double-tab delimited format:
     i. **TestDate**
     ii. **Temperature**
     iii. **AppliedStress**
     iv. **Deflection**
3. Build the solution and correct any errors.

▶ Task 7: Run the query by using a BackgroundWorker object

1. In the task list, locate the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** task, and then double-click this task. This task is located in the **DisplayResults_Click** method. This method calls the **CreateQuery** method to generate the LINQ query that matches the criteria that the user specifies, and it then runs the query to generate and format the results by using a **BackgroundWorker** object called **workerThread**.
2. Delete the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** comment, and then define an event handler for the **workerThread.DoWork** event. Add code to the event handler to invoke the **FormatResults** method, passing the **query** object as the parameter to the method. Store the value that the method returns in the *Result* parameter of the **DoWork** event handler.
3. Build the solution and correct any errors.

▶ Task 8: Display the results

1. Below the event handler for the **DoWork** event, add an event handler for the **workerThread**.**RunWorkerComplete** event. Add code to the event handler to perform the following tasks:

MCT USE ONLY. STUDENT USE PROHIBITED

a. Update the **results**.**Text** property with the value of the *Result* parameter of the **RunWorkerComplete** event handler.

b. Enable the **displayResults** button.

c. Update the **statusMessage** status bar item to "Ready".

2. Build the solution and correct any errors.

▶ Task 9: Test the solution

1. Run the application.
2. Click **Display**, and make a note of the **Time (ms)** value that is displayed next to the **Display** button.
3. Click **Display** two more times. The times for these operations will probably be lower than the time that the initial query took because the various internal data structures have already been initialized. Make a note of these times.



Note: The time that is displayed is the time that is required to fetch the data by using the LINQ query, but not the time that is taken to format and display this data. This is why the "Fetching results" message appears for several seconds after the data has been retrieved.

4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 40,641 values.
5. Use the **DatePicker** and slider controls to modify the search criteria to the values in the following table, and then click **Display** again.

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

6. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values. Note the time that it took to complete the search—the time should be less than the times that you recorded in Step 3. Keep a note of these values for comparison in Exercise 2.

MCT USE ONLY. STUDENT USE PROHIBITED

7. Close the Stress Data Analyzer window, and then return to Visual Studio.

   Currently, any search through the data uses all four criteria—date, temperature, applied stress, and deflection—regardless of the values that are specified in the UI. If the user does not change the default values for any criteria, the LINQ query that the application generates still contains criteria for each field. This is rather inefficient. However, you can construct dynamic LINQ queries to enable you to generate a custom query that is based only on the criteria that are specified at run time. You will implement this functionality in the next exercise.

Exercise 2: Building Dynamic LINQ Queries

In this exercise, you will extend the WPF application to enable users to specify the sort sequence and limit the number of rows that are retrieved. You will modify the application to build a dynamic LINQ query that matches the users' specifications, run it, and display the results.

The main tasks for this exercise are as follows:

1. Open the StressDataAnalyzer solution.
2. Dynamically build a lambda expression for the query criteria.
3. Dynamically build the date expression tree.
4. Dynamically build numeric expression trees.
5. Combine the expression trees.
6. Build a lambda expression for the **OrderBy** statement.
7. Examine the **CreateQuery** method.
8. Test the solution.

▶ Task 1: Open the StressDataAnalyzer solution

1. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex2\Starter folder.
2. Review the task list.
3. Examine the modified UI for the StressDataAnalyzer application. Note the following features of the application:

MCT USE ONLY. STUDENT USE PROHIBITED

- The UI is an extended version of that used in Exercise 1. The user can specify which criteria to apply by using check boxes. Any criteria that are not selected are not included in the LINQ query.
- The user can change the order in which the data is displayed by selecting the appropriate option button in the **Order By** section of the window.
- The user can limit the number of items that a query returns by selecting the **Limit** check box and by using the slider control to specify the number of items.

▶ Task 2: Dynamically build a lambda expression for the query criteria

1. In the task list, locate the **TODO - Complete the BuildLambdaExpressionForQueryCriteria method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionForQueryCriteria** method.

   The **BuildLambdaExpressionForQueryCriteria** method dynamically constructs a lambda expression from the values that are passed in as parameters. There are 12 parameters, which are divided into four groups. The *dateRangeSpecified* parameter is a Boolean value that indicates whether the user has selected the date criteria in the window, and the *startDate* and *endDate* parameters contain the start date and end date values that the user specifies. If the *dateRangeSpecified* parameter is **false**, the date is not included in the criteria for matching stress data. The same logic applies to the remaining parameters.

   The value that the **BuildLambdaExpressionForQueryCriteria** method returns is an **Expression** object. The **Expression** type represents a strongly typed lambda expression as a data structure in the form of an expression tree. The type parameter is a delegate that indicates the form of the lambda expression. In the **BuildLambdaExpressionForQueryCriteria** method, the lambda expression takes a **TestResult** object and returns a Boolean value that indicates whether this object should be included in the results that are generated by running the lambda expression.

   The existing code in this method creates a reference to an **Expression** object named **lambda**. You will add code to populate this object with an expression tree that represents a lambda expression that matches the query criteria that the 12 parameters specify. If the user does not specify any query criteria, this method returns a null value.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: The **Expression** type is located in the **System.Linq.Expressions** namespace. The application creates an alias for this namespace called **Expressions**. You cannot refer to the **Expression** type without the qualifying namespace in a WPF application because the WPF assemblies also contain a type called **Expression**.

2. Delete the **TODO - Complete the BuildLambdaExpressionForQueryCriteria method** comment, and then add code to perform the following tasks:
   a. Create a **Type** reference for the **TestResult** type named **testResultType**



Hint: Creating a type reference in this way enables you to repeatedly refer to an object type without repeatedly calling the **typeof** method. The **typeof** method is a relatively costly method compared to retrieving an object reference.

   b. Create an **Expressions.ParameterExpression** object named **itemBeingQueried** by using the **Expressions.Expression.Parameter** static method. Specify the **testResultType** type reference as the type of the parameter, and use the string **"item"** as the name of the parameter.



Hint: The string that is passed as the second parameter to the method call defines how your lambda expression will refer to the object that is being queried. In this example, one part of the resultant expression will resemble **"item.TestDate >= startDate"**.

3. Add code to the method to create the following **Expressions.BinaryExpression** objects; each object should have an initial value of null**:**
   a. **dateCondition**
   b. **temperatureCondition**
   c. **appliedStressCondition**
   d. **deflectionCondition**

   You will populate these expression objects with query criteria that match the parameters that are passed in to the method. You will then combine these expression objects together to form the complete lambda expression tree.
4. Add code to the method to invoke the **BuildDateExpressionBody** method, and store the result in the **dateCondition** object. Pass the following values as parameters to the method call:

MCT USE ONLY. STUDENT USE PROHIBITED

a. **dateRangeSpecified**

b. **startDate**

c. **endDate**

d. **testResultType**

e. **itemBeingQueried**



Note: The **BuildDateExpressionBody** method returns a **BinaryExpression** object that checks the stress test data against the **startDate** and **endDate** values. You will update the **BuildDateExpressionBody** method in the following task.

5. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **temperatureCondition** object. Pass the following values as parameters to the method call:

   a. **temperatureRangeSpecified**

   b. **fromTemperature**

   c. **toTemperature**

   d. **testResultType**

   e. A string that contains the value "**Temperature**"

   f. **itemBeingQueried**



Note: The **BuildNumericExpressionBody** method also returns a **BinaryExpression** object that will form part of the dynamic LINQ query. In this case, the data that this part of the query checks will contain numeric data rather than a **DateTime** value, and the name of the field that is being checked is **Temperature**. You will update the **BuildNumericExpressionBody** method later in the lab.

6. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **appliedStressCondition** object. Pass the following values as parameters to the method call:

   a. **appliedStressRangeSpecified**

   b. **fromStressRange**

   c. **toStressRange**

   d. **testResultType**

MCT USE ONLY. STUDENT USE PROHIBITED

e. A string that contains the value "**AppliedStress**"

f. **itemBeingQueried**

7. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **deflectionCondition** object. Pass the following values as parameters to the method call:

   a. **deflectionRangeSpecified**

   b. **fromDeflection**

   c. **toDeflection**

   d. **testResultType**

   e. A string that contains the value "**Deflection**"

   f. **itemBeingQueried**

8. Add code to the method to invoke the **BuildLambdaExpressionBody** method, and store the result in a new **Expressions**.**Expression** object named **body**. Pass the **dateCondition**, **temperatureCondition**, **appliedStressCondition**, and **deflectionCondition** objects as parameters to the method.



Note: The **BuildLambdaExpressionBody** method takes the four expression objects, each of which evaluate a single property in a **TestResult** object, and combines them into a complete lambda expression that evaluates all of the properties that the user specifies criteria for. You will complete the **BuildLambdaExpressionBody** method later in the lab.

9. Add code to the method to invoke the **Expression.Lambda** generic method, and store the response in the **lambda** object. The **Expression.Lambda** method should construct a lambda expression from the body of the lambda expressions in the **body Expression** object and the **itemBeingQueried ParameterExpression** object. Specify the delegate type **Func<TestResult, bool>** as the type parameter of the method.



Hint: The static **Expression.Lambda** method constructs an expression tree that represents a completed lambda expression, including the data that is being queried by the expression.

10. Build the project and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 3: Dynamically build the date expression tree

1. In the task list, locate the **TODO - Complete the BuildDateExpressionBody method** task, and then double-click this task. This task is located in the **BuildDateExpressionBody** method.

   The existing code in this method defines a **BinaryExpression** object named **dateCondition**. This object will be used to return the expression tree that evaluates date values. The method then checks that the *dateRangeSpecified* parameter is **true**. You will add code to this conditional statement to build an expression tree that is equivalent to the condition in the following code example.

```
item.TestDate >= startDate && item.TestDate <= endDate
```

   If the user did not specify any date criteria, this method returns a null expression tree.

2. Delete the **TODO - Complete the BuildDateExpressionBody method** comment, and then add code to create a new **MemberInfo** object named **testDateProperty**. Call the **GetProperty** method to generate code that retrieves the **TestDate** property from the *testResultType* type parameter. Pass the string "TestDate" as the parameter to the **GetProperty** method.

3. Add code to the method to create a **MemberExpression** object named **testDateMember**. Populate the object with the value that is returned by calling the **Expression**.**MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testDateProperty** value as parameters to the method.



Note: A **MemberExpression** object is an expression that represents access to a property of the object that is being queried. In this case, the object represents the ***item*.TestDate** property.

4. Add code to create an **Expressions.ConstantExpression** object named **lowerDate**, and populate the object with the result of calling the **Expression.Expressions.Constant** method. Pass the *startDate* parameter as a parameter to the method call.



Note: A **ConstantExpression** object is an expression that represents the results of evaluating a constant value. In this case, the object represents the value in the startDate variable.

5. Add code to create an **Expressions.BinaryExpression** object named **lowerDateCondition**, and populate the object with the result of calling the **Expressions.Expression.GreaterThanOrEqual** method. Pass the **testDateMember** and **lowerDate** objects as parameters to the method call.



Note: The **GreaterThanOrEqual** method generates a binary expression that combines the **testDateMember** object (representing the "this.startDate" portion of the expression) and the **lowerDate** object (representing the "startDate" portion of the expression) to generate a tree for the expression **"this.startDate >= startDate"**.

6. By using the same principles that you saw in Steps 4 and 5, add code to perform the following tasks:
   a. Create a **ConstantExpression** object named **upperDate** by passing the *endDate* parameter as a parameter to the method call.
   b. Create a **BinaryExpression** object named **upperDateCondition** by invoking the **Expression**.**LessThanOrEqual** method. Pass the **testDateMember** and **upperDate** objects as parameters to the method call.



Note: This code should build the second part of the date evaluation expression, which represents **"endDate <= testDateMember"**.

7. Add code to combine the expressions in the **lowerDate** and **upperDate ExpressionTree** objects into a single Boolean expression tree that returns **true** if both conditions are true or **false** otherwise; call the **Expressions**.**Expression.AndAlso** static method to combine the expressions together, and store the result in the **dateCondition** object.



Note: The **Expressions.Expression.AndAlso** method combines the two discrete expressions that you just created, **"item.TestDate >= startDate"** and **"Item.TestDate <= endDate"** into a **BinaryExpression** object that represents the expression **"item.TestDate >= startDate && Item.TestDate <= endDate"**.

8. Build the project and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 4: Dynamically build numeric expression trees

1. In the task list, locate the **TODO - Complete the BuildNumericExpressionBody method** task, and then double-click this task. This task is located in the **BuildNumericExpressionBody** method.

   The existing code in this method defines a **BinaryExpression** object named **booleanCondition**. This object will be used to return the expression tree that evaluates conditions based on **short** integer values. You will add code to this conditional statement to build an expression tree that is equivalent to the expression in the following code example, where *propertyName* represents the value of the *propertyName* parameter.

```
item.PropertyName >= lowerRange && item.PropertyName <= upperRange
```

2. Delete the **TODO - Complete the BuildNumericExpressionBody method** comment, and then add code to generate the first half of the expression by performing the following tasks:

   a. Create a new **MemberInfo** object named **testProperty**. Call the **GetProperty** method to generate code that retrieves the property that the *propertyName* parameter specifies from the *testResultType* type parameter.

   b. Create a **MemberExpression** object named **testMember** that represents access to the property that the **testProperty** object specifies; call the static **Expression.MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testProperty** object as parameters.

   c. Create a **ConstantExpression** object named **lowerValue** by invoking the **Expression.Constant** method. Pass the *lowerRange* parameter as a parameter to the method call.

   d. Create a **BinaryExpression** object named **lowerValueCondition**, which combines the **testMember** and **lowerValue** expression objects into a **GreaterThanOrEqual** binary expression.



Hint: Your code should build the first half of the target expression, which represents **"item.***PropertyName* **>= lowerRange"**, where *PropertyName* represents the value of the *propertyName* parameter. Your code should use similar syntax to that used to generate the expression in Task 3

3. Add code to generate the second half of the target expression by performing the following tasks:

a. Create a **ConstantExpression** object named **upperValue** by invoking the static **Expression**.**Constant** method. Pass the *upperRange* parameter as a parameter to the method call.

b. Create a **BinaryExpression** object named **upperValueCondition**, which combines the **testMember** and **upperValue** expression objects into a **LessThanOrEqual** binary expression.



Hint: Your code should build the second half of the target expression, which represents **"item**.*PropertyName* **<= upperRange"**, where *PropertyName* represents the value of the *propertyName* parameter. Your code should again use similar syntax to that used to generate the expression in Task 3.

4. At the end of the method, add code to set the **booleanCondition** object to an expression that combines the **lowerValueCondition** and **upperValueCondition** expressions by using the static **Expressions.Expression.AndAlso** method.
5. Build the project and correct any errors.

▶ Task 5: Combine the expression trees

1. In the task list, locate the **TODO - Complete the BuildLambdaExpressionBody method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionBody** method.

   This method takes four parameters that define the expression trees for each of the possible criteria that the user can enter. If any criteria are missing, the corresponding expression tree is null. The purpose of this method is to combine these expression trees together into an overall expression tree that includes all of the criteria that the user enters.

   The existing code in this method creates an **Expression** object named **body**. This object will contain the combined binary expressions that form the body of the LINQ query. If the user did not specify any criteria, the **body** object is assigned an expression tree that contains the Boolean constant **true**. This expression tree causes all items to be retrieved.

2. Delete the **TODO - Complete the BuildLambdaExpressionBody method** comment, and then add code to check whether the *dateCondition* parameter is null. If not, set the **body** object to the **dateCondition** expression tree.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Add code to check whether the *temperatureCondition* parameter is null, and if not, perform the following tasks:
   a. If the **body** object is null, set the **body** object to the **temperatureCondition** expression tree.
   b. If the **body** object is not null, combine the expression tree in the body object with the expression tree in the *temperatureCondition* parameter by using the static **Expressions.Expression**.**AndAlso** method. Assign the result to the **body** object.
4. Repeat the logic in Step 3 and add the expression tree that the *appliedStressCondition* parameter defines to the **body** expression tree.
5. Repeat the logic in Step 3 and add the expression tree that the *deflectionCondition* parameter defines to the **body** expression tree.
6. Build the project and correct any errors.

▶ Task 6: Build a lambda expression for the OrderBy statement

1. In the task list, locate the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionForOrderBy** method.

   The purpose of this method is to construct an expression that specifies the order in which the data should be retrieved. The parameter to this method is a value from the **OrderByKey** enumeration. This enumeration is defined as part of the application and contains the following values: **ByDate**, **ByTemperature**, **ByAppliedStress**, **ByDeflection**, and **None**. The following code example shows the form of the lambda expression that this method generates.

```
item => item.Property
```

   In this example, *Property* references the property from the **TestResult** type that corresponds to the parameter that is passed into the method. If the user does not specify a sort key, this method returns a null value.

2. Delete the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** comment, and then add code to the method to create the **ParameterExpression** object that defines the parameter for the lambda expression by performing the following tasks:

MCT USE ONLY. STUDENT USE PROHIBITED



Note: You will need to create a **Type** reference to the **TestResult** object, and the lambda expression should refer to the object **item**.

a. Create a **Type** reference named **testResultType** by using the **typeOf** operator and passing a **TestResult** object as a parameter.

b. Create a **ParameterExpression** object named **itemBeingQueried** by using the static **Expressions.Expression.Parameter** method. Specify the **testResultType** object and a string that contains the text **"item"** as parameters to the method.

3. In the **BuildLambdaExpressionForOrderBy** method, replace the **TODO - Create a MemberExpression and MemberInfo object** comment with code to perform the following tasks:

   a. Create a **MemberExpression** object named **sortKey**, and initialize this object to null.

   b. Create a **MemberInfo** object named **property**, and initialize this object to null.

4. Replace the **TODO - Evaluate the orderByKey parameter to determine the property to sort by** comment with code to evaluate the *orderByKey* parameter. Use the **GetProperty** method of the testResultType variable to generate code that retrieves the corresponding property value from the item that is specified as the parameter to the lambda expression. Store the result in the property variable. The following table lists the name of each property to use, depending on the value of the *orderByKey* parameter.

orderByKey value	testResultType property to use
ByDate	**"TestDate"**
ByTemperature	**"Temperature"**
ByAppliedStress	**"AppliedStress"**
ByDeflection	**"Deflection"**



Note: Near the beginning of the **BuildLambdaExpressionForOrderBy** method, a conditional statement prevents the method from performing this code if the *orderByKey* parameter has the value **OrderByKey.None**; therefore, you do not need to check for this value.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Replace the **TODO - Construct the expression that specifies the OrderBy field** comment with code that retrieves the value that the property variable specifies from the item that the itemBeingQueried variable specifies. To do this, call the static **Expressions.Expression.MakeMemberAccess** method, and pass the **itemBeingQueried** expression tree and the **property** object as parameters to this method.
6. Replace the **TODO - Create a UnaryExpression object to convert the sortKey object to a ValueType** comment with code to create a new **UnaryExpression** object named **convert** by invoking the static **Expressions.Expression.Convert** method. Pass the **sortKey** object and the type of the **ValueType** type as parameters to the method call. This step is necessary because the possible sort keys are all value types, and they must be converted to **ValueType** objects for the ordering to function correctly.
7. Replace the **TODO - Create the OrderBy lambda expression** comment with code to combine the converted unary expression that contains the sort key and the itemBeingQueried variable into a lambda expression by using the static **Expression.Lambda** generic method. Specify the type **Func<TestResult, ValueType>** as the type parameter to the **Lambda** method; the resulting lambda expression takes a **TestResult** object as the parameter and returns a **ValueType** object.
8. Build the project and correct any errors.

▶ Task 7: Examine the CreateQuery method

- In the task list, locate the **TODO - Examine the CreateQuery method** task, and then double-click this task. This task is located in the **CreateQuery** method.

  This method is the starting point for the lambda expression generation. The method accepts parameters that indicate which query criteria the lambda expression should include and the upper and lower ranges for each of these criteria.

  The method first calls the **BuildLambdaExpressionForQueryCriteria** method to construct a lambda expression that incorporates the query criteria. It then calls the **BuildLambdaExpressionForOrderBy** method to construct the lambda expression that defines the sort order for retrieving the data. Note that, at this point, it is possible that either of these expressions may still be null if the user either did not specify any criteria or did not specify a sort key.

MCT USE ONLY. STUDENT USE PROHIBITED

After the method creates the expression objects, it creates an **IEnumerable** generic collection named **query** that is based on the **TestResult** type, and it initializes the object with the data in the **stressData** parameter.

If the lambda expression that specifies the query criteria is not null, the method then filters the data in the **IEnumerable** collection by invoking the **Where** LINQ extension method on the collection. The parameter to the **Where** method is the lambda expression that contains the query criteria. Note that the **Compile** method of an **Expression<TDelegate>** object converts the expression tree into a compiled lambda expression that the CLR can execute.

If the lambda expression that defines the sort order is not null, this method then applies this lambda expression to the **IEnumerable** collection by using the **OrderBy** LINQ extension method. As before, the **Compile** method converts the expression tree that defines the sort key into code that can be executed by using the CLR.

If the user specifies that the query should return a limited number of rows, the **Take** LINQ extension method is applied to the **IEnumerable** collection with the limit that the user specifies.

Finally, the **IEnumerable** collection is returned to the caller. Note that this method does not run the LINQ query. This action occurs in the **DisplayResults_Click** method, when the code calls the **Count** method of the **IEnumerable** collection.

▶ Task 8: Test the solution

1. Run the application.
2. In the Stress Data Analyzer window, click **Display** to display all results with no query criteria, sort key, or limit to the number of items that are returned. Note the time that it takes to execute the query.



Note: This test is different from the test that you performed at the end of the first exercise. In the original application, the LINQ query used a lambda expression that contained criteria for all properties, whereas this test does not use any criteria. Therefore, the operation should be faster.

3. Select the **Test Date** and **Temperature** check boxes, modify the search criteria to the values in the following table, and then click **Display** again.

MCT USE ONLY. STUDENT USE PROHIBITED

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

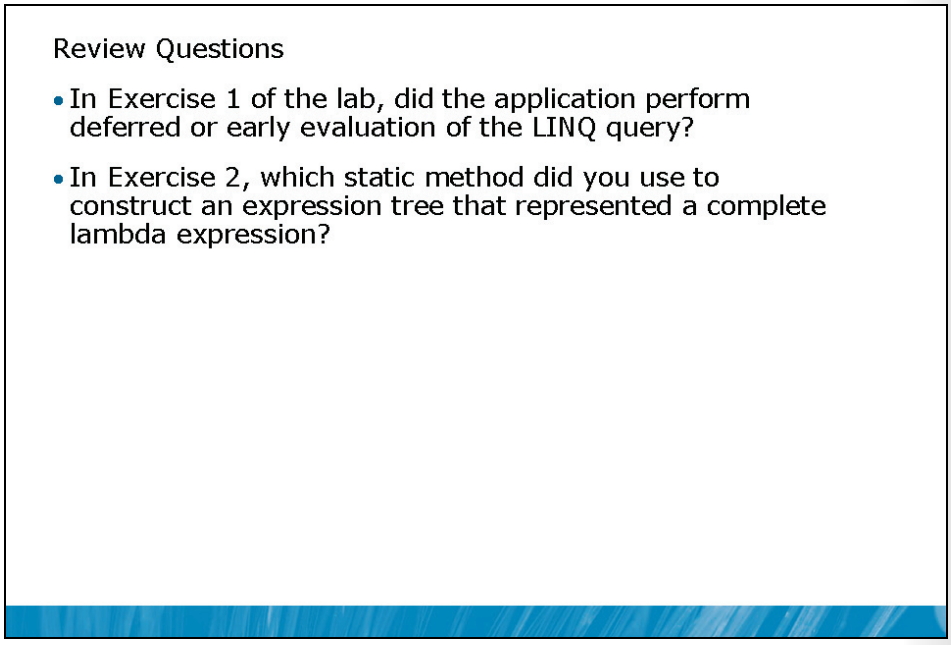
4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values, as in the test in Exercise 1. However, the time it takes to execute the query should again be less than the time that you recorded in Exercise 1.
5. Clear the **Test Date** and **Temperature** check boxes, and then select the **Limit?** check box. Set the limit value to 2,000, and then click **Display**.

   Note that when the number of rows is reduced, the time it takes to execute the query is substantially reduced.
6. In the **Order By** section, select **Temperature**, and then click **Display** again.

   Note that the expression takes substantially longer to execute when a sort key is included in the expression.
7. Close the Stress Data Analyzer window, and then return to Visual Studio.
8. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lab Review

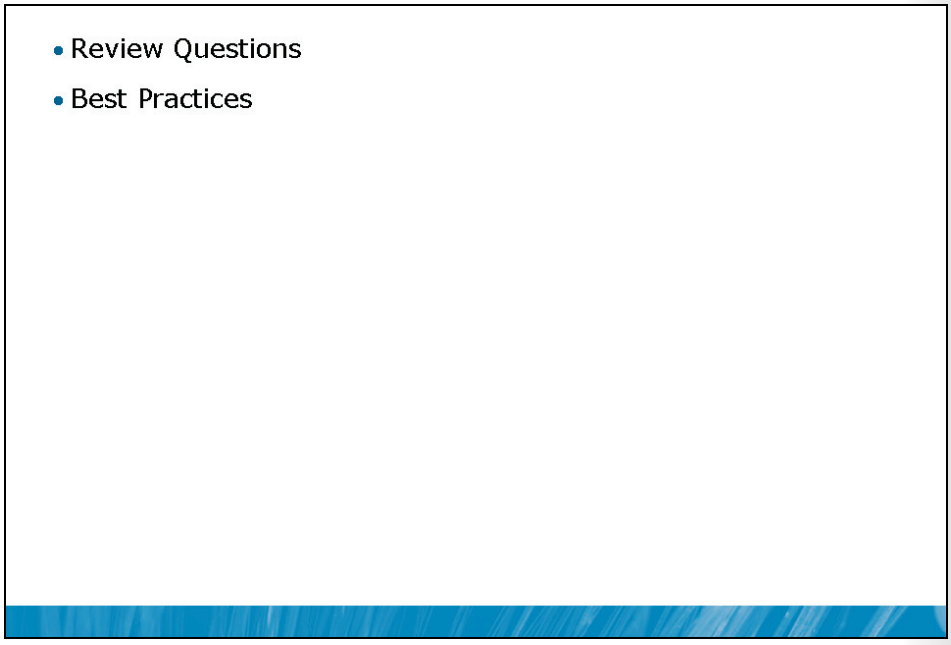


Review Questions

1. In Exercise 1 of the lab, did the application perform deferred or early evaluation of the LINQ query?
2. In Exercise 2, which static method did you use to construct an expression tree that represented a complete lambda expression?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. Which query operator would you use to implement filtering functionality on an enumerable collection?
2. Which extension method would you use to remove duplicate values from an enumerable collection?
3. How can you force early evaluation of a LINQ query?
4. When would you use an expression tree in an application?

Best Practices Related to Using LINQ

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

- Use LINQ queries rather than manually writing your own code to retrieve data and to help reduce dependencies that your applications have on the structure that data sources use.

MCT USE ONLY. STUDENT USE PROHIBITED

- Use anonymous types to model the data that queries return instead of creating new types.

Best Practices Related to Using Dynamic LINQ

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

- Use dynamic LINQ queries where you require flexibility rather than developing several variants of a query, which risks introducing errors and duplicate code.
- Use the **Type** and **MemberInfo** classes to reference types and type members. Use the **typeof** keyword and the **Get** methods (such as **GetProperty**) minimally to avoid excessive performance issues.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Module 15

Integrating Visual C# Code with Dynamic Languages and COM Components

Contents:

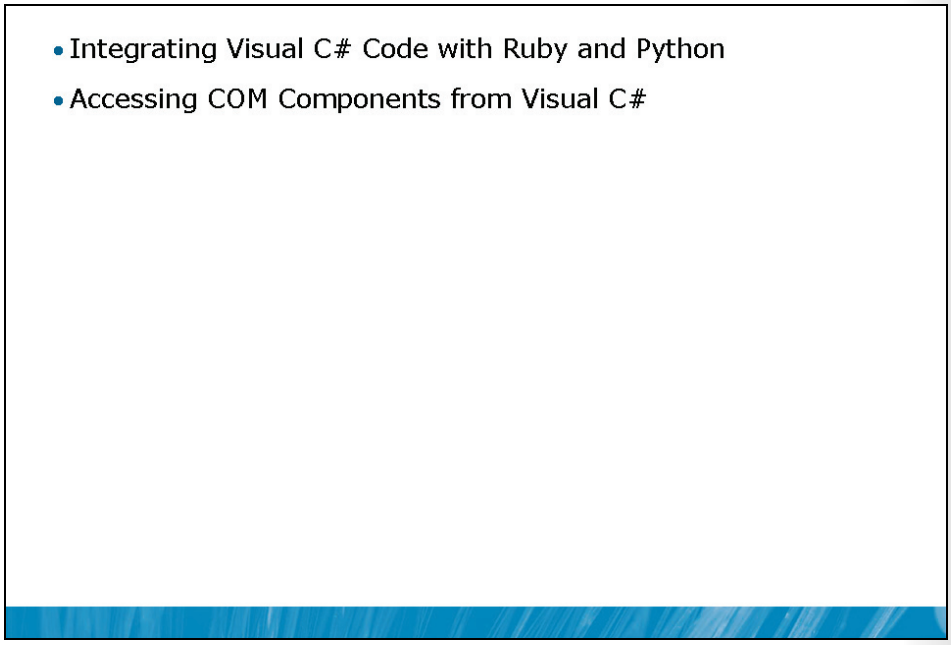
Lesson 1: Integrating Visual C# Code with Ruby and Python 15-4

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

Module Overview



Integration with other technologies is a key feature of the Microsoft® .NET Framework. Previous versions of the .NET Framework enabled you to combine components that were developed by using different languages that have compilers that the .NET Framework supports. The .NET Framework 4 now supports integration of components built by using dynamic languages. This enables you to reuse items built by using a wide range of scripting languages that are not easily accessible from Microsoft Visual C#® code.

In addition, previous versions of the .NET Framework have always enabled you to integrate Component Object Model (COM) services and components into your managed applications, although it required a good understanding of the differences between the way in which the common language runtime (CLR) and the COM environment operated. The new features of C# 4.0 have simplified the way in which you can invoke COM components, so it is much easier for you to reuse these items in a Visual C# application.

This module describes how to integrate code written by using a dynamic language such as Ruby and Python, or technologies such as COM, into a C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

Objectives

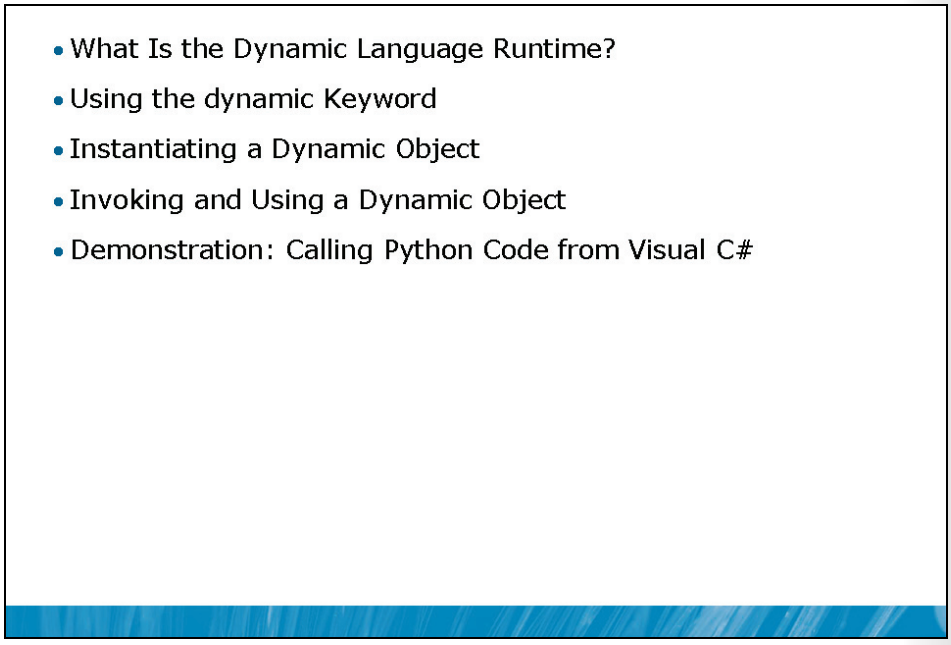
After completing this module, you will be able to:

- Integrate Ruby and Python code into a C# application.
- Invoke COM components and services from a C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 1

Integrating Visual C# Code with Ruby and Python



The .NET Framework enables you to build solutions that can combine components and services together that have been developed by using different languages and technologies. If you build Visual C# applications, this feature enables you to reuse existing code—you do not have to rewrite this code as C# components and services. Previous versions of the .NET Framework provided support, through the CLR, for simple integration between Visual C# and the other managed languages that the .NET Framework supports. For example, you can write a class in Microsoft Visual Basic®, compile it into an assembly, and then reference this assembly and use the class in a Visual C# application. The Visual C# application is not aware of the language used to build the class. The compilers for each of the managed languages (Visual C#, Visual Basic, Microsoft Visual C++®, Microsoft Visual F#, and so on) all convert code written by using these languages into common intermediate language (CIL). When you run an application, the .NET Framework runtime converts the CIL code into machine instructions and then runs them.

MCT USE ONLY. STUDENT USE PROHIBITED

However, not all modern computer languages are compiled. There are a large number of scripting languages currently in use that are interpreted at run time. In previous versions of the .NET Framework, it was not easy to integrate code written by using these languages into a Visual C# application. The .NET Framework 4 has addressed this issue with the dynamic language runtime (DLR).

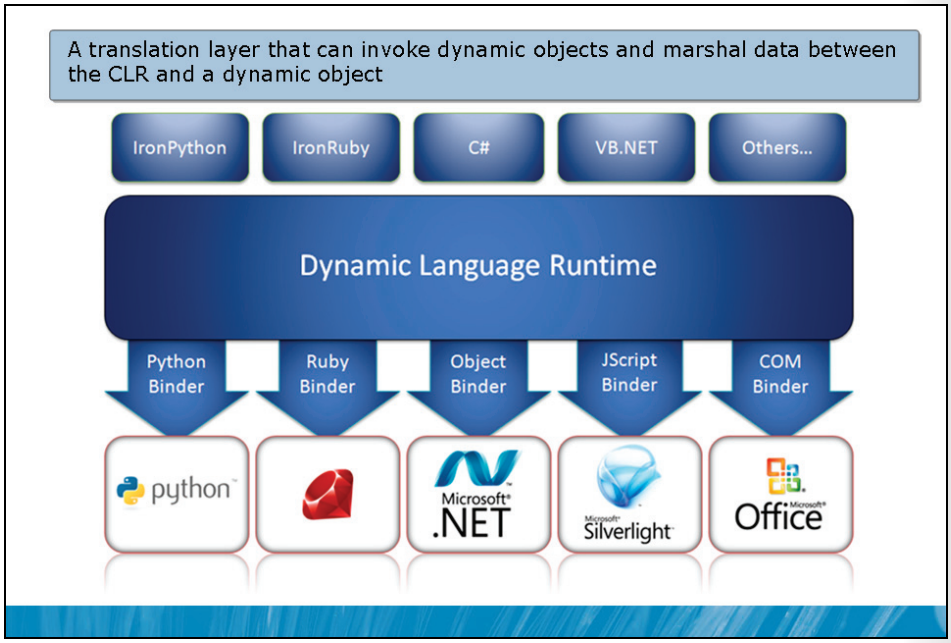
Objectives

After completing this lesson, you will be able to:

- Explain the purpose of the DLR.
- Use the **dynamic** type to specify that the DLR should perform late binding on an object.
- Instantiate a **dynamic** object.
- Invoke methods and access properties of a **dynamic** object, and pass a **dynamic** object as a method parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

What Is the Dynamic Language Runtime?



Key Points

C# is a strongly typed language. When you create a variable, you specify the type of that variable, and you can only invoke methods and access members that this type defines. If you try to call a method that the type does not implement, your code will not compile. This is good because it catches a large number of possible errors early, before you even run your code.

However, strong typing becomes a problem if you want to incorporate objects that languages such as Ruby and Python define, and which are interpreted rather than compiled into your C# code. It is very difficult, if not impossible, for the Visual C# compiler to verify that any members that you access in your Visual C# code exist in these objects. In addition, if you call a method on a Ruby or Python object, the Visual C# compiler cannot check that you have passed the correct number of parameters and that each parameter has the appropriate type.

There is another issue. The types that C# and the .NET Framework define mostly have a different internal representation from those that Ruby and Python use. For example, if you call a Ruby method that returns an integer, this integer must be converted from the representation that Ruby uses to the one that C# expects. A

MCT USE ONLY. STUDENT USE PROHIBITED

similar problem arises if you pass an integer as a parameter from a C# application into a Ruby method; the integer must be converted from the C# representation to that of Ruby.

The process of converting data between formats is known as marshaling, and it is a problem familiar to any developer who has built applications that invoke COM components. The solution is to use an intermediary layer. In the .NET Framework 4, this intermediary layer is called the DLR.

In addition to marshaling data between languages, the DLR also provides many of the services that the compiler provides with a strongly typed language. For example, when you invoke a method on a Ruby or Python object, the DLR checks at run time that this method call is valid.

The DLR is not tied to a specific set of languages; it implements an architecture that is based on language binders. A language binder is a component that slots into the DLR and understands how to invoke methods in a specified language and how to marshal and unmarshal data between the different formats that the language and the .NET Framework expect. The binder also performs a certain amount of checking, such as verifying that an object exposes a method that is being invoked and the parameters and return types are valid.

The .NET Framework 4 provides binders for IronPython, IronRuby, COM (which you can use to access COM components, such as those in the 2007 Microsoft Office system), Microsoft JScript®, and the .NET Framework itself.

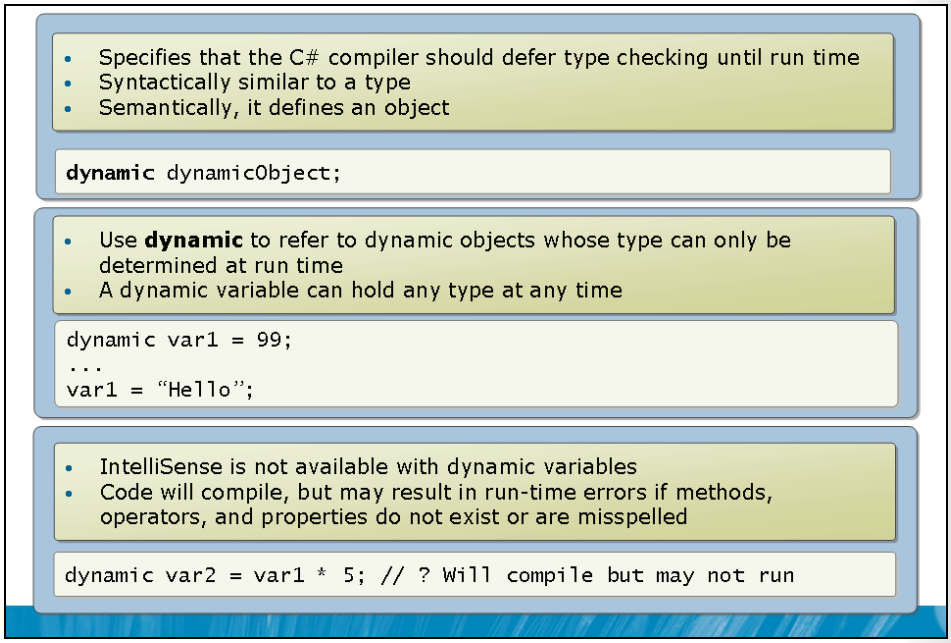
Question: What is the purpose of the DLR?

Additional Reading

For more information about the DLR, see the Dynamic Language Runtime Overview page at http://go.microsoft.com/fwlink/?LinkId=192986.

MCT USE ONLY. STUDENT USE PROHIBITED

Using the dynamic Keyword



Key Points

The DLR performs its work at run time. This means that any type checking for objects referenced through the DLR is deferred until your application executes.

How do you indicate in a C# application that type checking for an object should be deferred in this way? The answer lies in the **dynamic** keyword.

You use the **dynamic** keyword in exactly the same way that you use a type. For example, the following code example creates a variable called **dynamicObject** by using the **dynamic** type.

```
dynamic dynamicObject;
```

You should be aware that although the syntax you use for the **dynamic** keyword is similar to the syntax you use to specify a type in Visual C#, there is no **dynamic** type. All this keyword does is create a variable of type **object**, but with type checking deferred until run time. You can assign any value to this variable and attempt to call methods by using a variable that is declared as **dynamic**. At

MCT USE ONLY. STUDENT USE PROHIBITED

runtime, the DLR uses the appropriate binder to validate your code, instantiate objects, and invoke methods and marshal and unmarshal data.

Note that because the type checking is deferred until run time, the Microsoft IntelliSense® feature in Microsoft Visual Studio® cannot help you by providing the names of members exposed through a **dynamic** object reference. If you attempt to call an invalid method or reference a nonexistent field in a **dynamic** object, you will not know about it until run time, when it will throw a **RuntimeBinderException** exception.

The same consideration applies when you use a variable that is declared as **dynamic** in an expression. The following code example will compile, but may not run if the DLR detects that the object that is referenced does not support the * operator.

```
dynamic dynamicObject;
...
dynamicObject *= 2;
```

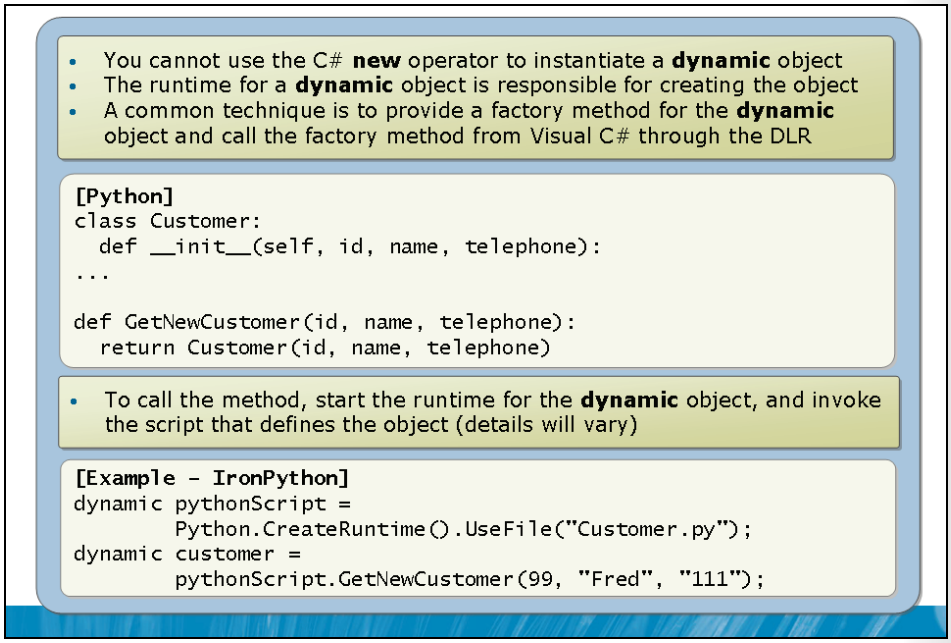
Question: What is the difference between the **var** keyword and the **dynamic** keyword?

Additional Reading

For more information about the **dynamic** keyword, see the dynamic (C# Reference) page at http://go.microsoft.com/fwlink/?LinkId=192987.

MCT USE ONLY. STUDENT USE PROHIBITED

Instantiating a Dynamic Object



Key Points

The appropriate language runtime creates and manages **dynamic** objects. The DLR acts as a bridge between the CLR that C# applications use and the runtime that hosts the dynamic types that you use.

When you instantiate an object based on a .NET Framework type, you typically use the **new** operator. The purpose of the **new** operator is to allocate memory and invoke a constructor to initialize the object. However, the **new** operator only works with managed types. Objects that are implemented by using a dynamic language run in unmanaged space and are controlled by the runtime that is specific to the language. For example, the Python runtime is responsible for resolving references to Python objects in a Python script and instantiating them. Consequently, in most cases, you must provide a mechanism that a Visual C# application can use to instantiate a **dynamic** object.

A common technique is to provide a factory method that hides the details of object creation from the calling environment. The following code example shows a Python script that defines a simple Python class called **Customer**. In common with many dynamic scripting languages, Python supports global functions, which are

MCT USE ONLY. STUDENT USE PROHIBITED

functions that are not members of a specific class. The following script includes a global function called **GetNewCustomer** that creates a new instance of the **Customer** class and returns it. The **GetNewCustomer** function takes three parameters that are passed to the constructor (the **__init__** function) in the **Customer** class. The **__init__** function uses these parameters to populate three fields called **custID**, **custName**, and **custTelephone**. The **Customer** class also defines a method called **__str__** that returns a string representation of the object



Note: The **__str__** method is similar to the **ToString** method in the .NET Framework and is called whenever a Python method attempts to print or display an object as a string.

```
class Customer:
  def __init__(self, id, name, telephone):
    self.custID = id
    self.custName = name
    self.custTelephone = telephone


  def __str__(self):
    return str.format("ID: {0}\tName: {1}\tTelephone: {2}",
                      self.custID, self.custName, self.custTelephone)

def GetNewCustomer(id, name, telephone):
  return Customer(id, name, telephone)
```

A C# application can call the **GetNewCustomer** function through the DLR and specify values for the customer ID, name, and telephone number. The DLR will marshal these arguments to the Python runtime and call the **GetNewCustomer** method. The Python runtime executes the **GetNewCustomer** method and creates the **Customer** object. When the **GetNewCustomer** function returns the reference to the **Customer** object, the DLR unmarshals this reference back to the C# application.

Accessing the Runtime for a Dynamic Language

To invoke the **GetNewCustomer** method (or any other factory method) through the DLR, you must provide the DLR with information about the dynamic language that is invoked and the runtime binder to use. The techniques for achieving this depend on the implementation of the dynamic language and the binder.

If you use IronPython (an implementation of Python that is integrated into the .NET Framework and that is available on the CodePlex Web site), you can use the types in the **IronPython.dll** assembly to create an instance of the IronPython runtime by using the static **CreateEngine** method of the **Python** class. The

MCT USE ONLY. STUDENT USE PROHIBITED

reference to the IronPython runtime is returned as a **ScriptEngine** object (the .NET Framework provides the **ScriptEngine** type to enable access to dynamic scripting languages). The following code example illustrates this.

```
ScriptEngine pythonEngine = Python.CreateEngine();
```



Note: You must add references to the **IronPython** and **Microsoft.Scripting** assemblies to use the **Python** and **ScriptEngine** types. These assemblies are located in the IronPython installation folder.

You can then use the **ExecuteFile** method of the **ScriptEngine** object to load a script that contains Python code. For example, if the **Customer** class shown earlier is implemented in a Python script called Customer.py, you can use the following code example to load this script.

```
dynamic pythonScript = pythonEngine.ExecuteFile("Customer.py");
```

The **ExecuteFile** method returns a **ScriptScope** object, but this code stores the result in a **dynamic** object. The principal purpose of the **ScriptScope** type is to provide a context to create and invoke **dynamic** objects and act as the entry point into the DLR for managed code. You can use this object to invoke the functions defined in the Python script, as the following code example shows,

```
dynamic customer = pythonScript.GetNewCustomer(99, "Fred", "111");
```



Note: Remember when you use the **dynamic** keyword, the compiler cannot perform any type checking when you build the application and instead defers resolution until run time. If the pythonScript variable was declared as a **ScriptScope** object, the application would not build, because the **ScriptScope** type does not contain a method called **GetNewCustomer**.

The preceding code example illustrates one way to start the Python runtime. As an alternative, you can use the static **CreateRuntime** method of the **Python** class and invoke the **UseFile** method to specify the name of a Python script to run, as the following code example shows.

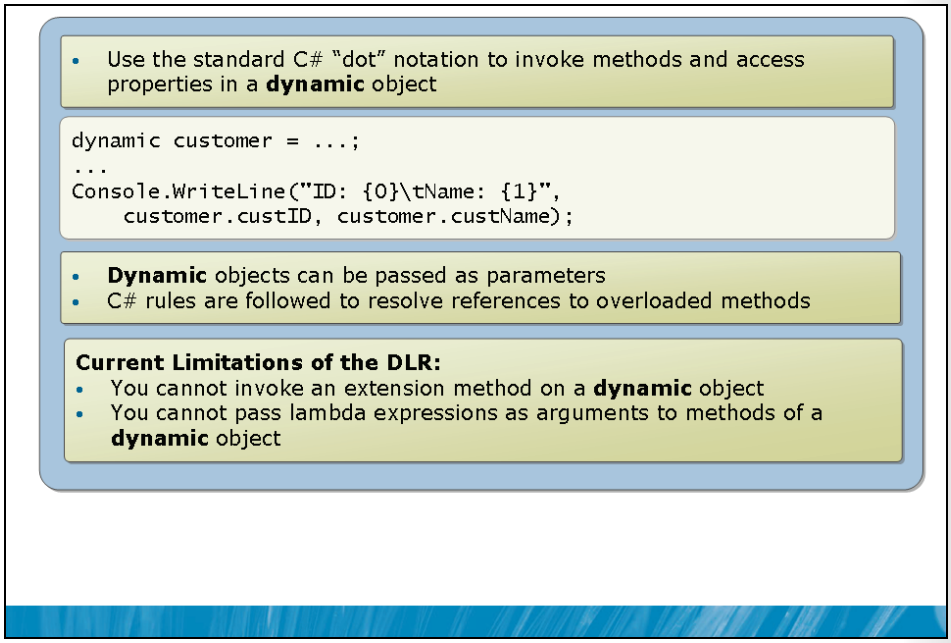
```
dynamic pythonScript = Python.CreateRuntime().UseFile("Customer.py");
```

MCT USE ONLY. STUDENT USE PROHIBITED

Question: Why does the .NET Framework not define a constructor for the **dynamic** type?

MCT USE ONLY. STUDENT USE PROHIBITED

Invoking and Using a Dynamic Object



Key Points

In the code example shown in the previous topic, the value returned by the **GetNewCustomer** method is a reference to a Python **Customer** object. Notice that this reference is stored in a variable that is declared as **dynamic**. You can invoke methods and access properties of this object through the DLR by using ordinary C# "dot" notation. The following code example displays the values in the **custID** and the **custName** properties of a **Customer** object.

```
dynamic customer = pythonScript.GetNewCustomer(99, "Fred", "111");
...
Console.WriteLine("ID: {0}\tName: {1}",
                  customer.custID, customer.custName);
```

Passing Dynamic Objects as Parameters

Under many circumstances, you can also pass a **dynamic** object as an argument to a C# method. However, be aware that when your code runs, the DLR may use the dynamically determined type of the argument to ascertain which version of a method to run if the method is overloaded. In the following code example, the

MCT USE ONLY. STUDENT USE PROHIBITED

TestClass class provides an overloaded version of a method called **TestMethod**; one version takes an integer parameter and the other takes a **dynamic** object.

The code shows two calls to the **TestMethod** method; they both specify a **dynamic** object as the argument. However, in the first case, the DLR determines that the argument is really an integer and not a reference to an object that is defined elsewhere, so it follows the Visual C# overload-resolution rules and invokes the version of the **TestMethod** method that takes an integer parameter. In the second case, the argument is not an integer, so the version of the **TestMethod** method that takes a **dynamic** object as the parameter runs.



Note: When you reference a **dynamic** object in this way, the DLR follows a well-defined process to determine the type. It first examines the object to see whether it is a COM component; if so, it passes the **dynamic** reference to the COM object. Failing that, the DLR determines whether the object is a reference to some other dynamic type that is accessed by using a binder (such as a Python or Ruby object). Again, if this is the case, the object is passed as a **dynamic** reference. If the object is not a COM component or a reference to a dynamic type, the DLR determines that it must be a reference to a .NET Framework type and uses a technique called reflection to determine its real type. The DLR then uses the determined type to choose which overload to call.

```
class TestClass
{
    public void TestMethod(int x)
    {
        Console.WriteLine("Parameter is an integer");
    }

    public void TestMethod(dynamic x)
    {
        Console.WriteLine("Parameter is dynamic");
    }

}

...
TestClass testObject = new TestClass();

dynamic data = 99;
testObject.TestMethod(data);  // calls TestMethod(int x)

dynamic data2 = "Hello";
testObject.TestMethod(data);  // calls TestMethod(dynamic x)
```

MCT USE ONLY. STUDENT USE PROHIBITED

It is important to realize that any method that takes a *dynamic* parameter cannot make any assumptions about the real type of that parameter or the data that it contains; you should handle a *dynamic* parameter in the same way that you would handle an *object* parameter.

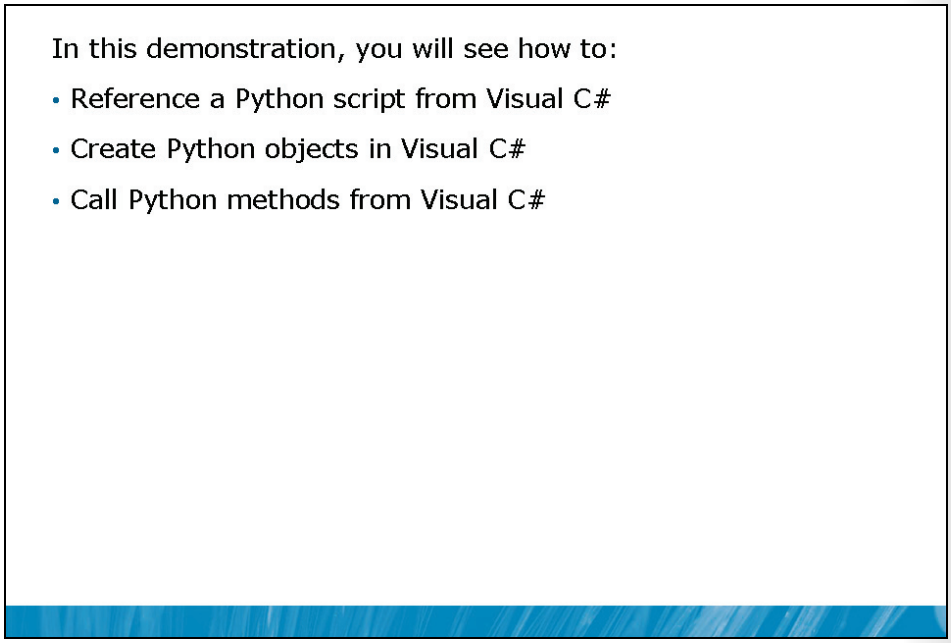
Limitations of Dynamic Objects

The current implementation of the DLR has some limitations. Specifically, you cannot call an extension method through a **dynamic** reference, and you cannot pass anonymous methods or lambda expressions as arguments to a method of a **dynamic** object.

Question: Why do you think that you cannot pass a lambda expression as an argument to a method of a **dynamic** object?

MCT USE ONLY. STUDENT USE PROHIBITED

Demonstration: Calling Python Code from Visual C#



Objectives

In this demonstration, you will see how to:

- Reference a Python script from Visual C#.
- Create Python objects in Visual C#.
- Call Python methods from Visual C#.

Demonstration Steps

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open the Python file **CustomerDB.py** in the **E:\Demofiles\Mod15** folder.
3. Walk through the code. Point out the following items in the Python file:
   - The **Customer** class.
   - The **CustomerDB** class.

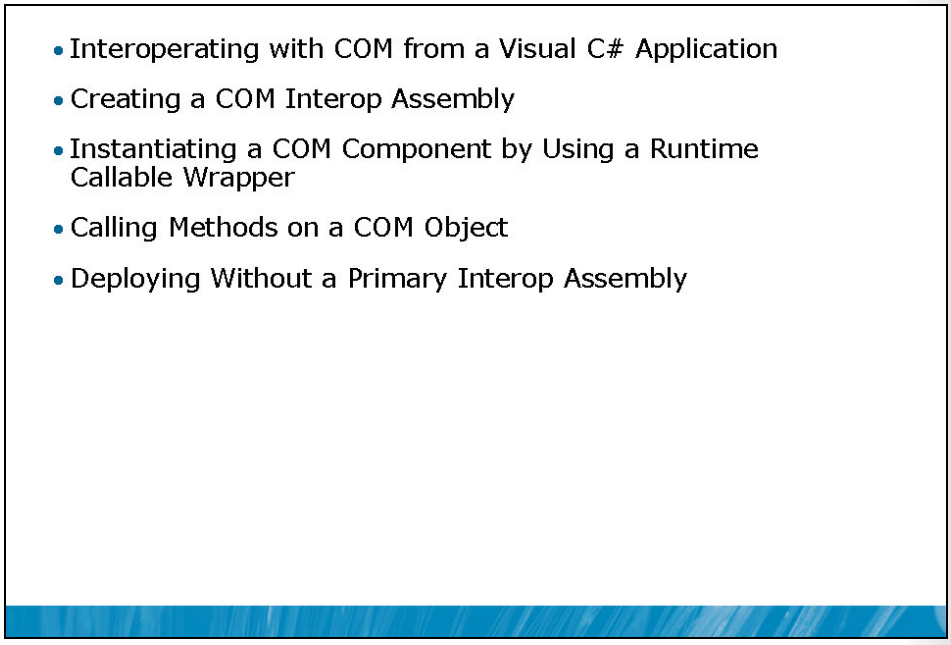
MCT USE ONLY. STUDENT USE PROHIBITED

- The **GetNewCustomer** method.
- The **GetCustomerDB** method.

4. Close the Python file.
5. Start Visual Studio:
6. Open the **PythonInteroperability** solution in the **E:\Demofiles\Mod15\PythonInteroperability** folder.
7. Display the code for the project:
8. Walk through the code in the **Main** method.
9. Show the references that the solution uses.
10. Build and run the application.
11. Close the console window.
12. Close Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

Lesson 2
Accessing COM Components from Visual C#



COM is a well-established technology, and a large base of useful COM components is now available. Microsoft makes extensive use of COM throughout the Windows® operating system. Likewise, many other organizations have made a considerable investment in COM. It may prove costly and time-consuming for most organizations to reimplement existing functionality based on COM components by using the .NET Framework. COM components execute in unmanaged space, but you can integrate these components into your .NET Framework applications by using COM Interop.

Objectives

After completing this lesson, you will be able to:

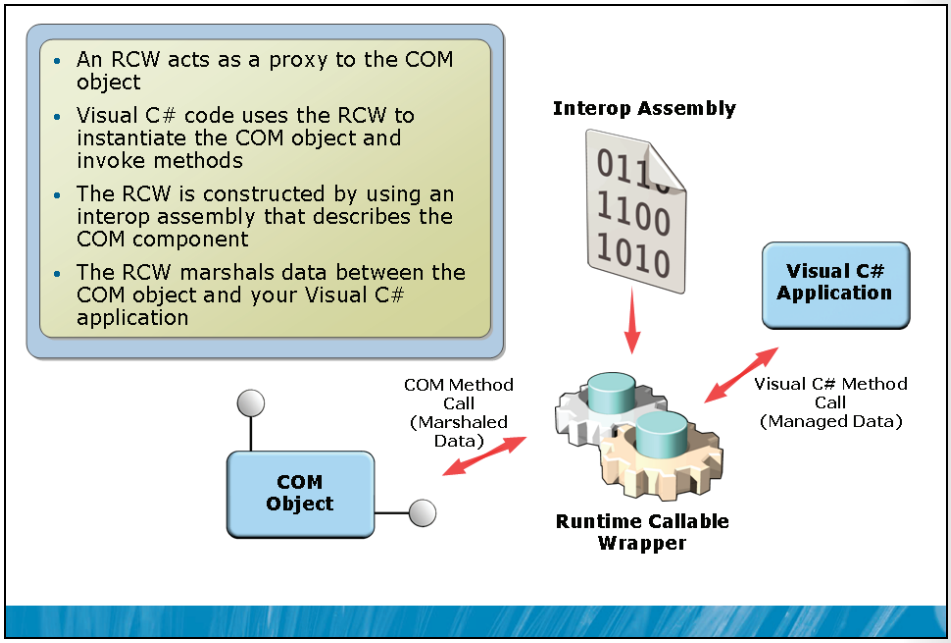
- Describe the issues that may arise when interoperating with COM from Visual C#.
- Add a COM interop assembly to a Visual C# application.
- Instantiate a COM object from a Visual C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

- Pass parameters from Visual C# into a method exposed by a COM object and handle values returned by a COM method.
- Explain how to configure an application to enable deployment that does not require a primary interop assembly (PIA).

MCT USE ONLY. STUDENT USE PROHIBITED

Interoperating with COM from a Visual C# Application



Key Points

Visual C# code runs by using the CLR. COM components execute in the unmanaged environment by using the facilities that the operating system provides. To incorporate a COM component into a Visual C# application, you must perform the following tasks:

- Instantiate the COM component. To do this, you must request that the operating system creates an unmanaged object.
- Call methods on the COM component. COM components are object-oriented items, and they expose methods that you can call. However, COM uses a different set of data types from Visual C#, so when you call a method on a COM object, you must convert the arguments that you provide from the format that Visual C# uses to the format that COM requires. Similarly, when a COM method may return a value, you must be prepared to convert this value from the COM format back into the format that Visual C# expects.
- Destroy the COM component when you have finished with it. When you create a managed object by using Visual C#, the CLR destroys it and reclaims

MCT USE ONLY. STUDENT USE PROHIBITED

the resources that it uses. COM objects run outside the auspices of the CLR, so you must explicitly destroy them.

Fortunately, the .NET Framework provides tools that can simplify these tasks. You can create, manage, and communicate with a COM component from a Visual C# application by using a runtime callable wrapper (RCW). The RCW acts as a proxy to the COM object. It hides the differences between the COM and .NET Framework runtimes. The overall aim of the RCW is to make the COM object appear the same as any other ordinary managed object to a Visual C# application.

How the CLR Interacts with a COM Object

The CLR creates an RCW from an assembly that contains metadata that describes the COM component. This assembly is referred to as an *interop assembly*. The interop assembly describes the custom COM interfaces and classes that the component implements.

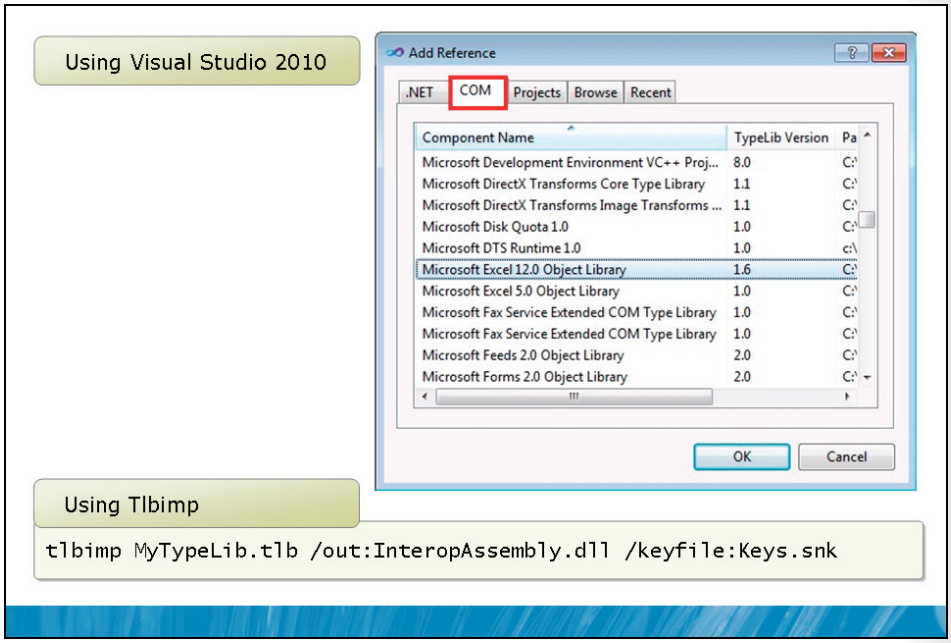
A Visual C# application uses the interop assembly to determine the types and methods that the COM component exposes. When a Visual C# application attempts to instantiate an object that a class in the interop assembly defines, the CLR locates and loads the library for the COM component and creates an instance of this component in unmanaged space. The CLR then creates the RCW to communicate with the COM object on behalf of the Visual C# application. The COM object sends any responses to the Visual C# application through the RCW (the COM object views the RCW as the client and is not aware that it is simply a proxy for a managed application).

The RCW exposes a managed interface to the methods in the COM component. A managed application calls these methods. The RCW converts arguments that a Visual C# application provides into the format that the COM component expects and then invokes the corresponding method in the COM component. Data that a COM component returns is marshaled back through the RCW to convert it back into the format that Visual C# expects before it is passed to your application.

Question: Why is it necessary to create an RCW to invoke a COM component from a Visual C# application?

MCT USE ONLY. STUDENT USE PROHIBITED

Creating a COM Interop Assembly



Key Points

To use an RCW to call a COM object, you must first create an interop assembly that contains the metadata that describes the COM classes and interface in the component. The .NET Framework and Visual Studio 2010 provide several tools that you can use to create an interop assembly.

Note: To create an interop assembly by using the tools that Visual Studio 2010 or the .NET Framework provide, you must use a COM component that provides a type library. A type library is typically either provided as a type library file that has a .tlb suffix or is embedded in the dynamic-link library (DLL) that implements the COM component. If the COM component does not provide a type library, you must either define the interop assembly manually or use late binding and reflection to discover the methods and interfaces that the COM component exposes at run time.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating an Interop Assembly by Using Visual Studio 2010

You can create an interop assembly in Visual Studio 2010 by using the **Add Reference** command on the **Project** menu. In the **Add Reference** dialog box, click the **COM** tab, and then select the appropriate COM component.

The **COM** tab lists the COM components and type libraries that are recorded in the Windows registry of the local computer. You can select any of these items, or you can click the **Browse** tab and search for the DLL that implements the COM component on the hard disk. When you select a component, Visual Studio discovers the interfaces, methods, and classes that the component implements and then creates the interop assembly. The interop assembly is added as a reference to your project.

Creating an Interop Assembly at a Command Prompt

As an alternative to using Visual Studio, the .NET Framework software development kit (SDK) provides the Tlbimp utility, which you can use at a command prompt to create an interop assembly. The Tlbimp utility imports type information that describes the COM component, either from a type library file that is provided with the COM component or from the DLL that implements the COM component.



Note: If the COM component does not provide a type library, but provides a description in a COM interface definition language (IDL) file, you can use the Microsoft IDL compiler (midl.exe) that is provided with the platform SDK to generate a type library.

To create an interop assembly at a command prompt, open a Visual Studio 2010 Command Prompt window, move to the folder that contains the DLL or type library file for the COM component, and then run the Tlbimp utility by providing the name of the DLL or type library as an argument. By default, the Tlbimp utility creates an assembly that is based on the name of the COM component (which may not be the same as the name of the DLL or type library file). However, you can modify this behavior by using the /**out** flag and specifying a different file name. You can also provide flags that indicate other options, such as the /**keyfile** flag. This flag specifies the name of a key file for signing the assembly if you want to deploy it to the global assembly cache (GAC), as the following code example shows.

```
tlbimp MyTypeLib.tlb /out:InteropAssembly.dll /keyfile:Keys.snk
```

After you have created the interop assembly, you can manually add it as a reference to your Visual C# project.

MCT USE ONLY. STUDENT USE PROHIBITED

Creating a Primary Interop Assembly

Vendors who produce COM components may also provide the necessary interop assemblies that are required to use these components from a Visual C# application. For example, Microsoft supplies interop assemblies that enable you to invoke Microsoft Office components from Visual C#.

However, it is important that you verify that these interop assemblies are correct and have been obtained from a reputable source. Manufacturers should sign their assemblies to enable them to be identified and to prevent them from being tampered with. A signed interop assembly for a set of COM components is referred to as a PIA.

You specify that an interop assembly is a PIA by using the /**primary** switch to the Tlbimp utility, as the following code example shows.

```
tlbimp MyTypeLib.tlb /primary /out:MyPIA.dll /keyfile:Keys.snk
```



Note: A COM type library that is imported as an assembly and signed by someone other than the publisher of the original type library cannot be a PIA. Only the publisher of a type library can produce a true PIA, which becomes the unit of official type definitions for interoperating with the underlying COM types.

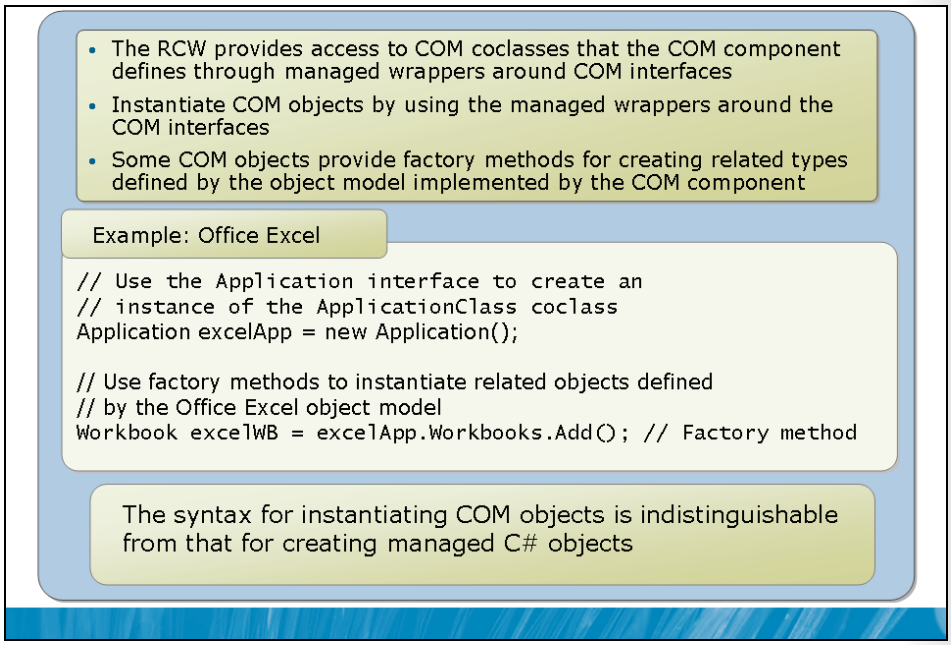
Question: Why are PIAs important?

Additional Reading

For more information about using the Tlbimp utility, see the Tlbimp.exe (Type Library Importer) page at http://go.microsoft.com/fwlink/?LinkId=192988.

MCT USE ONLY. STUDENT USE PROHIBITED

Instantiating a COM Component by Using a Runtime Callable Wrapper



Key Points

You use the interop assembly to create an RCW that instantiates and controls the lifetime of a COM object. You can invoke the methods of a COM object through an RCW in the same way that you invoke methods in an ordinary managed object. The RCW hides the fact that the object is a COM object and marshals method parameters and return values.

Creating a COM Object

An interop assembly contains a managed version of each COM interface that is defined in the type library for the COM component. A COM type library also defines one or more classes (called COM *coclasses*) that an application can instantiate and that implement one or more of the COM interfaces. The interop assembly defines a managed wrapper around each coclass and gives the wrapper the same name as the coclass, but adds the **Class** suffix. The interop assembly also contains a managed version of each COM interface. You create a COM object by creating an instance of the managed coclass, but you reference the class through the appropriate managed interface.

MCT USE ONLY. STUDENT USE PROHIBITED

In a typical COM component, a single coclass may act as the entry point into the object model that the component implements. In this situation, the coclass typically provides factory methods to create the other types of object that implement other COM interfaces that the type library provides. You use these factory methods through the RCW to instantiate managed wrappers around these types.

For example, the COM library that Microsoft provides for Microsoft Office Excel® defines a coclass called **ApplicationClass** that acts as the entry point into the Office Excel object model. It also defines a managed wrapper around the corresponding COM interface that this coclass implements, called **Application**. You instantiate a COM type through the RCW by using the managed wrapper that is associated with the COM interface. For example, to start Office Excel running, you create a new instance of the **Application** object.



Note: To instantiate a COM object, always use the managed wrapper for the interface that a coclass implements rather than the managed version of the coclass itself. For example, when you use Office Excel, create an instance of the **Application** type rather than the **ApplicationClass** type.

The COM library for Office Excel also defines coclasses and interfaces for other objects that the Office Excel object model implements, and it provides factory methods for instantiating these objects. In the COM Office Excel object model, the **Application** object contains a collection of **Workbook** objects, and a **Workbook** object can contain **Worksheet** and **Chart** objects. You can create a new **Workbook** object by calling the **Add** method of the **Workbook** property that the **Application** COM object exposes through the RCW, and you can create a new **Worksheet** object and a new **Chart** object in a workbook by calling the **Add** method of the **Worksheets** or **Charts** properties respectively. The following code example shows some instances.

```
// Namespace generated by the Excel PIA for the Excel COM library
using Excel = Microsoft.Office.Interop.Excel;
...
Excel.Application excelApp = new Excel.Application();
Excel.Workbook excelWB = excelApp.Workbooks.Add();
Excel.Worksheet excelWS = excelWB.Worksheets.Add();
Excel.Chart excelChart = excelWB.Charts.Add();
...
```

This code shows the COM object model that Office Excel exposes. Other applications, components, and services will expose their own models that may be

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

similar or quite different from the model that Office Excel uses. The key point to note is that as far as your code is concerned, all of the statements that create COM objects look like ordinary Visual C# method calls. The code that interacts with the COM subsystem is hidden by the RCW.

Question: What is the difference between instantiating a C# object and a COM object in a C# application?

Additional Reading

Managing the Lifetime of a COM Object

COM performs its own memory management, separate from that of the CLR. When you create a COM object, the COM subsystem allocates memory for that object. When an unmanaged COM client application creates or adds a reference to a COM object, it must indicate this fact to the COM object by calling its **AddRef** method. This method increments a reference counter in the COM object. When an unmanaged COM client application has finished using an object, it must call the **Release** method. This method decrements the reference counter. If the reference counter reaches zero, the COM object is destroyed and the memory that it used is released. When you use COM Interop from a managed application, the RCW automatically handles the lifetime management of a COM object by calling **AddRef** when you instantiate a COM object and by calling **Release** when you finish using the RCW. The CLR finalizes the COM object before the CLR collects garbage.

If you must release a COM object early, you can call the **ReleaseComObject** method of the **Marshal** class to decrement the reference count of the COM object, or you can call the **FinalReleaseComObject** method to set the reference count to zero.



Note: The **Marshal** class contains a selection of other methods that you can use to allocate and free memory for unmanaged COM objects and obtain information about COM objects, classes, and interfaces.

Managing COM Apartments from Managed Code

COM objects execute in apartments. An apartment specifies whether the COM component uses a single-threaded or multithreaded model when it runs. When a managed application creates a COM object, the CLR creates an apartment for the COM object by examining the **ApartmentState** property of the managed thread that is calling the COM object. If the **ApartmentState** property of the managed

thread matches the apartment requirements of the COM component, the CLR instantiates the COM object in this apartment. Every method call to the COM object from managed code passes through the interop marshaler to the COM object in unmanaged space. If the **ApartmentState** property of the managed thread does not match the apartment requirements of the COM component, the CLR instantiates the COM object in a new apartment with the appropriate threading model. Every method call that the managed application makes is marshaled by the interop marshaler to unmanaged space and then by the COM marshaler to the new apartment. The COM marshaler ensures that method calls are synchronized correctly between apartments that implement different threading models. This is known as cross-apartment marshaling.

The CLR runs managed threads by using the multithreaded apartment (MTA) model by default. However, you can modify this behavior by calling the **SetApartmentState** method of a thread to set the apartment state to **ApartmentState.STA** to specify the single-threaded apartment (STA) model. You must set the apartment state before the thread starts to run. If you must modify the apartment state of the initial thread that runs an application, you can apply the **MTAThread** or **STAThread** attributes to the **Main** method of the application.

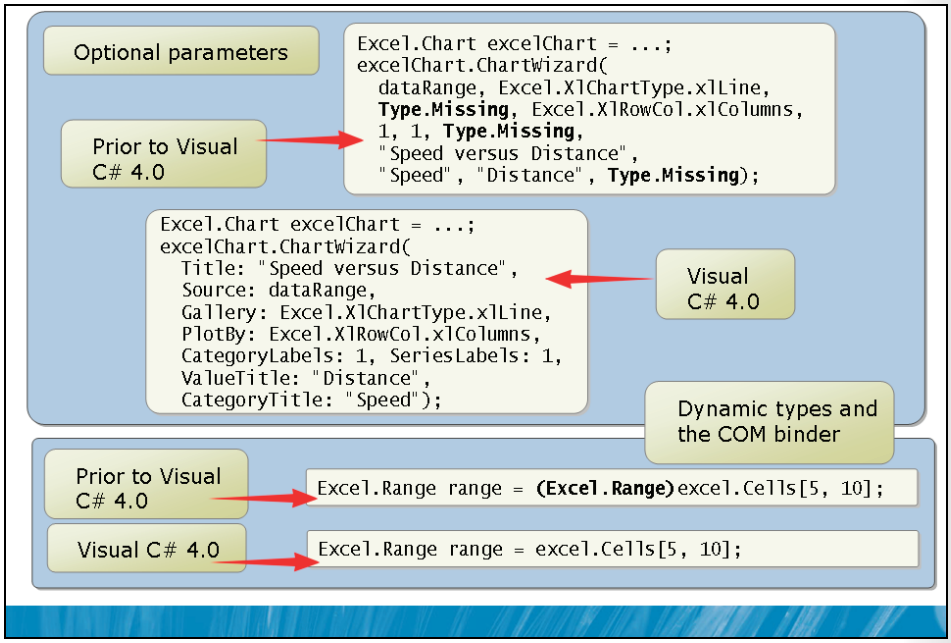


Note: If you create an application by using the Windows Application template in Visual Studio, the template tags the **Main** method with the **STAThread** attribute. You must not change this attribute, because a Windows graphical user interface (GUI) expects all of the user interaction to occur in a single thread. If you must call COM components that use the MTA model from a Windows application by using a GUI, you should consider creating an additional thread for calling these COM components and then set the **ApartmentState** property of this thread to **ApartmentState.MTA**.

Cross-apartment marshaling can be expensive and time consuming, so you should try to ensure that managed code calls COM components by using the same apartment. However, this may not always be feasible. If the same managed thread makes calls to STA COM components and MTA COM components, some cross-apartment COM marshaling is unavoidable. As a workaround, you have at least two choices. You could create a new thread for each COM component that you want to use and set its **ApartmentState** property appropriately (this might prove to be more expensive than performing cross-apartment marshaling if you use it to excess). Alternatively, you could try to ensure that the COM components that you use support the Both COM threading model so that they can run in single-threaded and multithreaded apartments.

MCT USE ONLY. STUDENT USE PROHIBITED

Calling Methods on a COM Object



Key Points

After you create a COM object, you can interact with it through the RCW to call its methods and set and query its properties. The methods and properties that are available depend on the COM object. You use the same syntax as you use to access a Visual C# object. The RCW marshals data between your Visual C# code and the COM object.

Handling Optional Parameters

If you examine many older Visual C# applications that perform COM interop, you may see method calls that pass the value **Type.Missing** as one or more arguments. Many COM components are written by using languages such as C++ that support optional parameters. Prior to Visual C# 4.0, you had to provide an argument for every parameter that a COM method specified, even if you wanted to omit that parameter and use the default that the COM component defined. The workaround was to specify the value **Type.Missing**. This caused the RCW to omit the argument when it called the COM component, which would then use the default value.

C# 4.0 circumvents this issue by supporting named arguments and optional parameters. The RCW now defines all optional parameters to COM methods as

MCT USE ONLY. STUDENT USE PROHIBITED

optional C# parameters with a default value of **Type.Missing**. When you call a method of a COM component, you can now specify parameters by name and omit any arguments that you want to leave at their default values. The syntax is exactly the same as calling a C# method that implements default parameter values.

For example, the **ChartWizard** method of the **Chart** type in the Office Excel object model takes a significant number of parameters, all of which have default values. Prior to Visual C# 4.0, to use any of these default values, you had to use code that resembled the following code example.

```
Excel.Chart excelChart = ...;
excelChart.ChartWizard(dataRange, Excel.XlChartType.xlLine,
                       Type.Missing, Excel.XlRowCol.xlColumns,
                       1, 1, Type.Missing,
                       "Speed versus Distance",
                       "Speed", "Distance", Type.Missing);
```

With C# 4.0, you can use named parameters. The RCW implements default parameter values for the **ChartWizard** method (it sets them to **Type.Missing**). Any parameters that you omit are propagated as missing values to the Office Excel COM library and are given their default values, as the following code example shows.

```
Excel.Chart excelChart = ...;
excelChart.ChartWizard(Title: "Speed versus Distance",
                       Source: dataRange,
                       Gallery: Excel.XlChartType.xlLine,
                       PlotBy: Excel.XlRowCol.xlColumns,
                       CategoryLabels: 1, SeriesLabels: 1,
                       ValueTitle: "Distance",
                       CategoryTitle: "Speed");
```

Named parameters have the additional advantage of making your code easier to understand and maintain.

Avoiding Casts

Another feature that you may notice in older C# applications is the use of casts to convert the types that many COM methods return. Prior to C# 4.0, all COM object references were returned as the **object** type. The RCW generated for C# 4.0 returns COM object references as **dynamic**. This enables the runtime to determine the type dynamically when your application runs. For example, in the Office Excel object model, a **Worksheet** object exposes the cells of a worksheet through the **Cells** property. You can create an Office Excel **Range** object that consists of a set of contiguous cells. Prior to C# 4.0, you had to use code similar to the following code

MCT USE ONLY. STUDENT USE PROHIBITED

example. The collection returned by querying the **Cells** property was an **object**, and you had to cast it to a **Range** object to store it in a **Range** reference.

```
Excel.Range range = (Excel.Range)excel.Cells[5, 10];
```

With Visual C# 4.0, the value returned by querying the **Cells** property is now **dynamic**, and the runtime can determine the correct type without requiring a cast, as the following code example shows.

```
Excel.Range range = excel.Cells[5, 10];  // No cast required
```

Question: If you omit an argument to a COM method, what value does the RCW use as the default?

Additional Reading

Handling Exceptions in COM Method Calls

When you call a method in a COM object, it can report error conditions. A COM object usually returns an **HRESULT** value from the method to report an error. The **S_OK** value indicates that the method call was successful. Any other value indicates an error, and the value itself specifies the reason for the error. An RCW does not pass **HRESULT** values back to managed client applications as return values. Instead, the RCW either returns no value at all or returns the value of any parameter that is marked with the **retval** COM attribute in the code that defines the COM method. (**retval** is an attribute that the Microsoft IDL (MIDL) compiler uses to indicate that a parameter should be marshaled as the return value.)

If a COM method attempts to pass back an **HRESULT** value other than **S_OK**, the CLR traps it and raises a managed exception. The managed exception that it raises depends on the value of **HRESULT**. The CLR automatically maps several well-known COM **HRESULT** values to the equivalent managed exception. For example, the **HRESULT** value **E_NULLREFERENCE** causes the CLR to raise a **NullReferenceException** exception. You can find a full list of **HRESULT** mappings in the Visual Studio 2005 documentation.

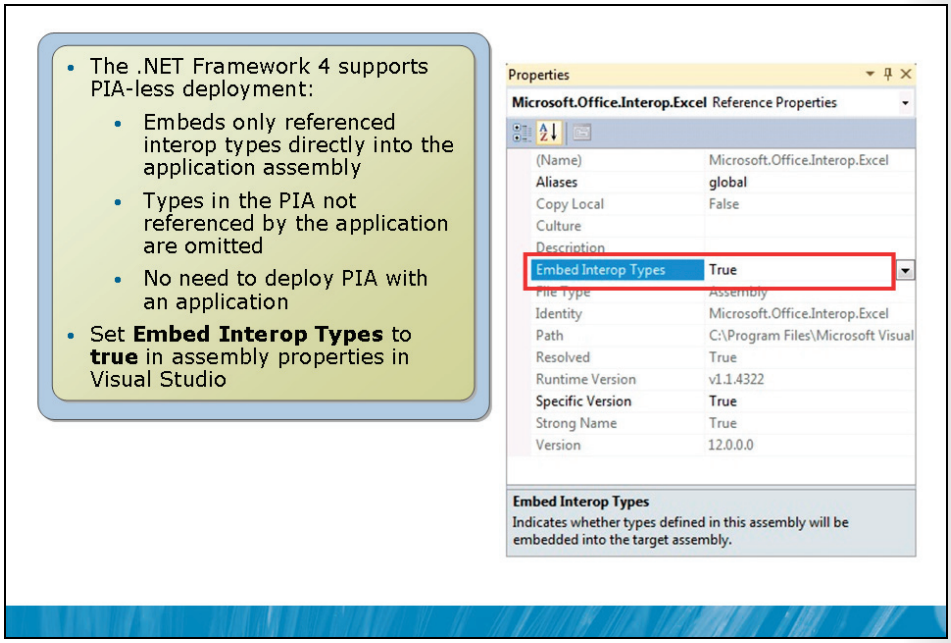
If a COM method returns an unrecognized **HRESULT** value, the CLR raises a **COMException** exception. The **HRESULT** value that the COM method returns is available in the **ErrorCode** property of the **COMException** object. In addition, if the COM object supports the COM **IErrorInfo** interface, the **Message** property of the **COMException** object is populated with the string that the **IErrorInfo.GetDescription** method returns, the **Source** property is filled in with

MCT USE ONLY. STUDENT USE PROHIBITED

the return value from the **IErrorInfo.GetSource** method, and the **HelpLink** property contains information that the **IErrorInfo.GetHelpFile** and **IErrorInfo.GetHelpContext** methods return.

MCT USE ONLY. STUDENT USE PROHIBITED

Deploying Without a Primary Interop Assembly



Key Points

Some PIAs for large COM libraries and components may define a large number of types and methods. Consequently, the PIAs may also be quite large. In addition, prior to the .NET Framework 4, you had to ensure that each PIA that an application used was deployed correctly on each computer that you installed the application on. Deploying a PIA requires administrator privileges.

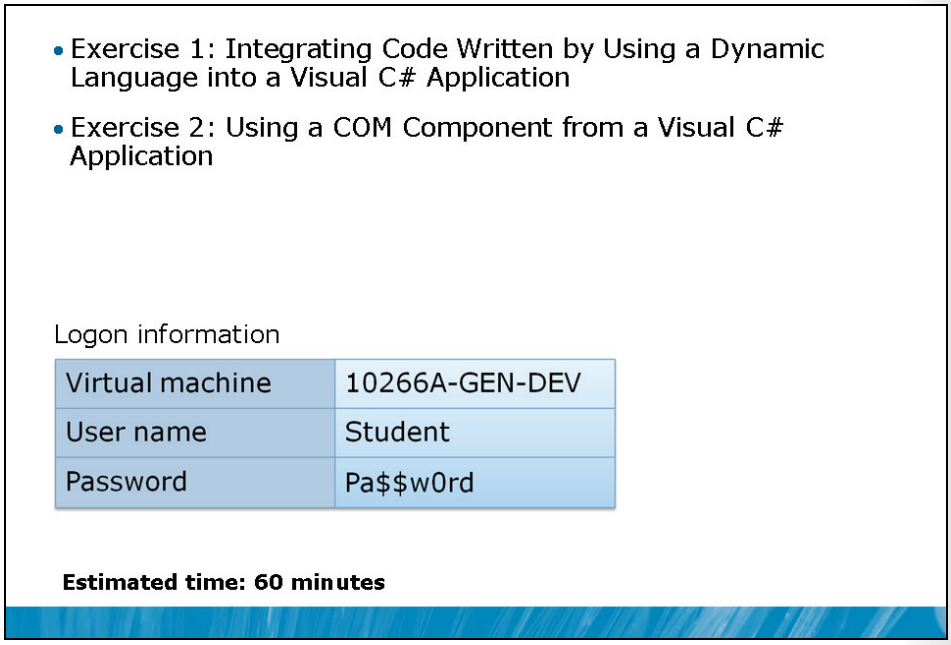
The .NET Framework 4 now supports intelligent, PIA-less deployment of applications. When you add a PIA to an application at design time, the PIA exposes a property called **Embed Interop Types**. If you set this property to **true**, when you build the application, the interop types from that PIA that you use in your application will be compiled into the assembly for your application. Consequently, you do not need to deploy the PIA with your application. In addition, the assembly for your application will contain only the types from the PIA that your application uses; any unreferenced types are omitted. This ensures that the size of the application assembly is kept as small as possible.

MCT USE ONLY. STUDENT USE PROHIBITED

Question: If you specify PIA-less deployment, when you deploy an application that uses a COM component, is it still necessary to deploy the COM component, too?

MCT USE ONLY. STUDENT USE PROHIBITED

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



Objectives

After completing this lab, you will be able to:

- Instantiate an object defined by using a dynamic language and invoke its methods from a Visual C# application.
- Instantiate a COM component and invoke its methods from a Visual C# application.

Introduction

In this lab, you use the DLR to access objects defined in an IronRuby or IronPython script from a Visual C# application. You will also use COM interop to instantiate and use a COM component from a Visual C# application.

MCT USE ONLY. STUDENT USE PROHIBITED

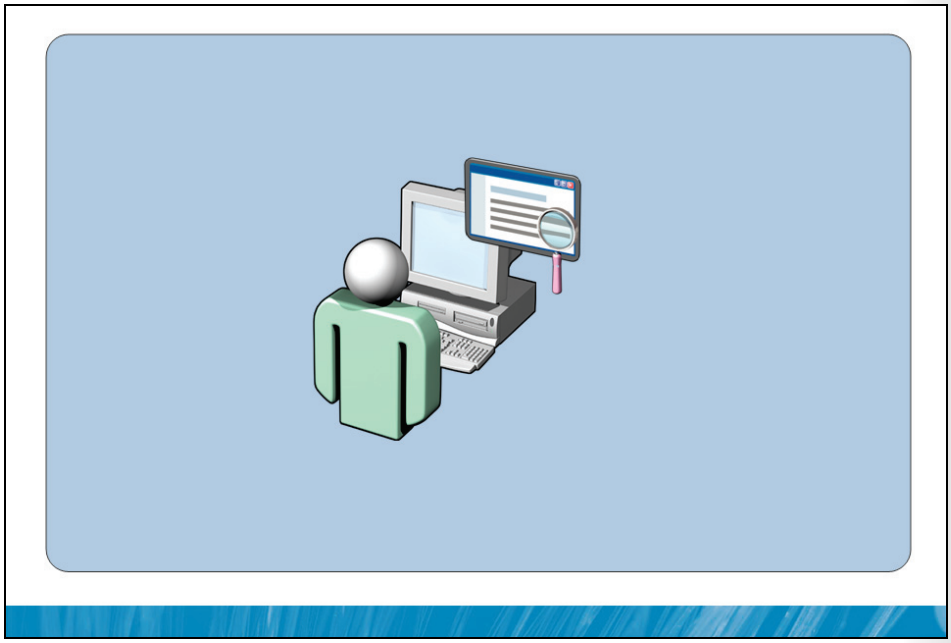
Lab Setup

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. makes use of technologies and programming languages other than Visual C# to drive some of the devices that it develops. In addition, Fabrikam, Inc. incorporates features from software products such as Microsoft Office into some of its applications. You have been asked to integrate some components written by using these technologies into the Visual C# software that supports the various devices.

Exercise 1: Integrating Code Written by Using a Dynamic Language into a Visual C# Application

Scenario

Fabrikam, Inc. has a sizable collection of Python and Ruby scripts that contain proven and thoroughly tested code. Although Visual C# is now the development language of choice for Fabrikam, Inc., it will be expensive and time-consuming to reimplement and fully test these scripts by using Visual C#. Instead, you have been asked to integrate the functionality that these scripts provide directly into your Visual C# applications.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

You will use the DLR to invoke an IronRuby script and an IronPython script (scripts for both languages are provided), create objects by using the types that are defined in these scripts, and call methods that these objects expose.

The main tasks for this exercise are as follows:

1. Examine the Python and Ruby code.
2. Open the starter project.
3. Create a Python object and call Python methods.
4. Test the Python code.
5. Create a Ruby object and call Ruby methods.
6. Test the Ruby code.

▶ Task 1: Examine the Python and Ruby code

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Visual Studio 2010.
3. Using Notepad, open the Shuffler.py file in the E:\Labfiles\Lab 15\Python folder.
4. In Notepad, examine the Python code.

   The Shuffler.py file contains a Python class called **Shuffler** that provides a method called **Shuffle**. The **Shuffle** method takes a parameter called *data* that contains a collection of items. The **Shuffle** method implements the Fisher-Yates-Durstenfeld algorithm to randomly shuffle the items in the **data** collection.

   The Python class also exposes a function called **CreateShuffler** that creates a new instance of the **Shuffler** class. You will use this method from Visual C# to create a **Shuffler** object.
5. Close Notepad.
6. Using Notepad, open the Trapezoid.rb file in the E:\Labfiles\Lab 15\Ruby folder.
7. In Notepad, examine the Ruby code.

   The Trapezoid.rb file contains a Ruby class called **Trapezoid** that models simple trapezoids. The constructor expects the angle of the lower-left vertex,

the length of the base, the length of the top, and the height of the trapezoid. The lengths of the remaining sides and angles are calculated.



Note: The **Trapezoid** class models a subset of possible trapezoids. The length of the base must be greater than the length of the top, and the specified vertex must be an acute angle.

The lengths of the sides, the angles of each vertex, and the height are exposed as properties.

The **to_s** method returns a string representation of the trapezoid.



Note: The **to_s** method is the Ruby equivalent of the **ToString** method in the .NET Framework. The Ruby binder in the DLR automatically translates a call to the **ToString** method on a Ruby object to a call to the **to_s** method.

The **area** method calculates the area of the trapezoid.

The Ruby file also provides a function called **CreateTrapezoid** that creates a new instance of the **Trapezoid** class.

8. Close Notepad.

▶ Task 2: Open the starter project

- Open the DynamicLanguageInterop solution in the E:\Labfiles\Lab 15\Starter \DynamicLanguageInterop folder.

▶ Task 3: Create a Python object and call Python methods

1. Examine the InteropTestWindow.xaml file.

   This window contains two tabs, labeled **Python Test** and **Ruby Test**.

   The **Python Test** tab enables you to type values into the **Data box** and specify whether this is text or numeric data. When you click **Shuffle**, the data will be packaged up into an array and passed to the **Shuffle** method of a Python **Shuffler** object. The shuffled data will be displayed in the **Shuffled Data** box.

   The functionality to create the Python object and call the **Shuffle** method has not yet been implemented; you will do this in this task.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronRuby runtime.

Assembly	Path
IronPython	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.dll
IronPython.Modules	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.Modules.dll
Microsoft.Dynamic	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Dynamic.dll
Microsoft.Scripting	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Scripting.dll

3. Review the task list.
4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task. This task is located near the top of the InteropTestWindow.xaml.cs file. This is the code behind the InteropTestWindow window.
5. After the comment, add **using** statements to bring the **IronPython.Hosting** and **Microsoft.Scripting.Hosting** namespaces into scope.
6. In the **InteropTestWindow** class, examine the string constants near the start of the class. In particular, note the **pythonLibPath** and **pythonCode** strings.

   The **pythonLibPath** constant specifies the folder where the Python libraries are installed. The **Shuffler** class makes use of a Python library called **random** that is located in this folder.

   The **pythonCode** constant specifies the name and location of the Python script that contains the **Shuffler** class.
7. In the task list, locate the **TODO: Create an instance of the Python runtime, and add a reference to the folder holding the "random" module** task, and then double-click this task. This task is located in the **ShuffleData** method.

   The **shuffle_Click** method calls the **ShuffleData** method when the user clicks the **Shuffle Data** button. The **shuffle_Click** method gathers the user input from the form and parses it into an array of objects. It then passes this array to the **ShuffleData** method. The purpose of the **ShuffleData** method is to create a Python **Shuffler** Python object and then call the **Shuffle** method by using the

MCT USE ONLY. STUDENT USE PROHIBITED

array as a parameter. When the **ShuffleData** method finishes, the **shuffle_Click** method displays the shuffled data in the Windows Presentation Foundation (WPF) window.

8. After the TODO comment, add code that performs the following tasks:
   a. Create a **ScriptEngine** object called **pythonEngine** by using the static **CreateEngine** method of the **Python** class.
   b. Obtain a reference to the search paths that the Python runtime uses; call the **GetSearchPaths** method of the **pythonEngine** object and store the result in an **ICollection<string>** collection object called **paths**.
   c. Add the path that is specified in the **pythonLibPath** string to the **paths** collection.
   d. Set the search paths that the **pythonEngine** object uses to the **paths** collection; use the **SetSearchPaths** method.
9. After the comment **TODO: Run the script and create an instance of the Shuffler class by using the CreateShuffler method in the script**, add code that performs the following tasks:
   a. Create a **dynamic** object called **pythonScript**. Initialize this object with the value that is returned by calling the **ExecuteFile** method of the **pythonEngine** object. Specify the **pythonCode** constant as the parameter to this method.

      This statement causes the Python runtime to load the Shuffler.py script. The **pythonScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.
   b. Create another **dynamic** object called **pythonShuffler**. Call the **CreateShuffler** method of the **pythonScript** object and store the result in the **pythonShuffler** object.

      This statement invokes the **CreateShuffler** function in the Python script. This function creates an instance of the **Shuffler** class and returns it. The **pythonShuffler** object then holds a reference to this object.



Note: The pythonScript variable is a **dynamic** object, so IntelliSense does not display the **CreateShuffler** method (or any other methods or properties).

10. After the comment **TODO: Shuffle the data**, add code that calls the **Shuffle** method of the **pythonShuffler** object. Pass the **data** array as the parameter to the **Shuffle** method.

MCT USE ONLY. STUDENT USE PROHIBITED

This statement runs the **Shuffle** method in the Python object. The DLR marshals the **data** array into a Python collection and then invokes the **Shuffle** method. When the method completes, the DLR unmarshals the shuffled collection back into the **data** array.

11. Build the application and correct any errors.

▶ Task 4: Test the Python code

1. Run the application.
2. In the Dynamic Language Interop Tests window, on the **Python Test** tab, in the **Data** box, type some random words that are separated by spaces.
3. Click the **Text** option button, and then click **Shuffle**. Verify that the shuffled version of the data appears in the **Shuffled Data** box.
4. Click **Shuffle** again. The data should be shuffled again and appear in a different sequence.
5. Replace the text in the **Data** box with integer values, click **Integer**, and then click **Shuffle**. Verify that the numeric data is shuffled.
6. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

▶ Task 5: Create a Ruby object and call Ruby methods

1. Examine the **Ruby Test** tab in the InteropTestWindow.xaml file.

   The **Ruby Test** tab enables you to specify the dimensions of a trapezoid (the angle of the first vertex, the length of the base, the length of the top, and the height) by using a series of slider controls. When you click the **Visualize** button, the application will create an instance of the Ruby **Trapezoid** class and display a graphical representation in the canvas in the lower part of the window. The dimensions and area of the trapezoid will be displayed in the text block that is to the right.

   The functionality to create the Ruby object and calculate its area and dimensions has not yet been implemented; you will do this in this task.

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronRuby runtime.

MCT USE ONLY. STUDENT USE PROHIBITED

Assembly	Path
IronRuby	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.dll
IronRuby.Libraries	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.Libraries.dll

3. Review the task list.
4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task.
5. Add a **using** statement to bring the **IronRuby** namespace into scope.
6. In the **InteropTestWindow** class, examine the **rubyCode** string constant near the start of the class.

   The **rubyCode** constant specifies the name and location of the Ruby script that contains the **Trapezoid** class.
7. In the task list, locate the **TODO: Retrieve the values specified by the user. These values are used to create the trapezoid** task, and then double-click this task. This task is located in the **visualize_Click** method. This method is called when the user clicks the **Visualize** button, after the user has specified the data for the trapezoid.
8. After the TODO comment, add code that performs the following tasks:
   a. Create an integer variable called vertexAInDegrees. Initialize this variable with the value of the **vertexA** slider control.



Hint: Use the **Value** property of a slider control to read the value. This value is returned as a **Double** value, so use a cast to convert it to an integer. This cast is safe because the slider controls are configured to return integer values in a small range, so no data will be lost.

   b. Create an integer variable called lengthSideAB. Initialize this variable with the value of the **sideAB** slider control.
   c. Create an integer variable called lengthSideCD. Initialize this variable with the value of the **sideCD** slider control.
   d. Create an integer variable called heightOfTrapezoid. Initialize this variable with the value of the **height** slider control.

MCT USE ONLY. STUDENT USE PROHIBITED

9. After the comment **TODO: Call the CreateTrapezoid method and build a trapezoid object**, add a statement that creates a dynamic variable called trapezoid and initializes it with the value that the **CreateTrapezoid** method returns. Pass the variables vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid as arguments to the **CreateTrapezoid** method.

   You will implement the **CreateTrapezoid** method in a later step. This method will create an instance of the Ruby **Trapezoid** class by using the specified data and return it.

10. After the comment **TODO: Display the lengths of each side, the internal angles, and the area of the trapezoid**, add a statement that calls the **DisplayStatistics** method. Pass the **trapezoid** object and the **trapezoidStatistics** text block as parameters to this method.

    You will implement the **DisplayStatistics** method in a later step. This method will call the **to_s** and **area** methods of the Ruby **Trapezoid** class and display the results in the **trapezoidStatistics** text block on the right of the **Ruby Test** tab in the WPF window.

11. After the comment **TODO: Display a graphical representation of the trapezoid**, add a statement that calls the **RenderTrapezoid** method. Pass the **trapezoid** object and the **trapezoidCanvas** canvas control as parameters to this method.

    The **RenderTrapezoid** method is already complete. This method queries the properties of the Ruby **Trapezoid** object and uses them to draw a representation of the trapezoid on the canvas in the lower part of the window.

12. In the task list, locate the **TODO: Create an instance of the Ruby runtime** task, and then double-click this task. This task is located in the **CreateTrapezoid** method.

13. At the start of this method, remove the statement that throws the **NotImplementedException** exception. After the comment, add a statement that creates a **ScriptRuntime** object called **rubyRuntime**. Initialize the rubyRuntime variable with the value that the static **CreateRuntime** method of the **Ruby** class returns.

14. After the comment **TODO: Run the Ruby script that defines the Trapezoid class**, add a statement that creates a **dynamic** object called **rubyScript**. Initialize the rubyScript variable with the value that the **UseFile** method of the **rubyRuntime** object returns. Pass the **rubyCode** constant as the parameter to the **UseFile** method.

MCT USE ONLY. STUDENT USE PROHIBITED

This statement causes the Ruby runtime to load the Trapezoid.rb script. The **rubyScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.

15. After the comment **TODO: Call the CreateTrapezoid method in the Ruby script to create a trapezoid object**, add a statement that creates a **dynamic** object called **rubyTrapezoid**. Initialize the rubyTrapezoid variable with the value that the **CreateTrapezoid** method of the **rubyScript** object returns. Pass the vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid variables as parameters to the **CreateTrapezoid** method.

    This statement invokes the **CreateTrapezoid** function in the Ruby script. The DLR marshals the arguments that are specified and passes them as parameters to the **CreateTrapezoid** function. This function creates an instance of the **Trapezoid** class and returns it. The **rubyTrapezoid** object then holds a reference to this object.



Note: The rubyScript variable is a **dynamic** object, so IntelliSense does not display the **CreateTrapezoid** method.

16. After the comment **TODO: Return the trapezoid object**, add a statement that returns the value in the rubyTrapezoid variable.
17. In the task list, locate the **TODO: Use a StringBuilder object to construct a string holding the details of the trapezoid** task, and then double-click this task. This task is located in the **DisplayStatistics** method.
18. After the comment, add a statement that creates a new **StringBuilder** object called **builder**.
19. After the comment **TODO: Call the to_s method of the trapezoid object to return the details of the trapezoid as a string**, add a statement that calls the **ToString** method of the trapezoid variable and appends the result to the end of the **builder** object.

    The DLR automatically converts the **ToString** method call into a call to the **to_s** method in the Ruby object. The **to_s** method constructs a Ruby string, which is unmarshaled into a .NET Framework string.
20. After the comment **TODO: Calculate the area of the trapezoid object by using the area method of the trapezoid class**, add code that calls the **area** method of the trapezoid variable, converts the result into a string, and appends this string to the end of the **builder** object.

MCT USE ONLY. STUDENT USE PROHIBITED

21. After the comment **TODO: Display the details of the trapezoid in the TextBlock control**, add a statement that sets the **Text** property of the **trapezoidStatistics** control to the string that is constructed by the **builder** object.
22. Build the application and correct any errors.

▶ Task 6: Test the Ruby code

1. Run the application.
2. In the Dynamic Language Interop Tests window, click the **Ruby Test** tab.
3. Set the **Vertex A** slider to **75**, set the **Length of Base** slider to **200**, set the **Length of Top** slider to **100**, set the **Height** slider to **150**, and then click **Visualize**.

   Verify that a representation of the trapezoid is displayed in the canvas in the lower half of the window and the statistics for the trapezoid appear in the text block that is to the right. The area of the trapezoid should be 22,500.
4. Experiment with different values for the slider controls, and then click **Visualize**. If you specify values that are outside the range for the set of trapezoids that the **Trapezoid** class can model, a message box should be displayed to indicate the problem. This error message is raised by the constructor in the **Trapezoid** class. The DLR catches the error and converts it into a .NET Framework **Exception** object. The **visualize_Click** method caches this exception and displays the error in a message box.
5. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

Exercise 2: Using a COM Component from a Visual C# Application

Scenario

One of the WPF applications that analyzes data that is retrieved from measuring devices needs to use this data to plot a range of graphs. Rather than write your own routines to plot and draw graphs, you will use the wide range of graphing capabilities already available through Office Excel.

You will use COM interop from a Visual C# application to invoke Office Excel and generate a graph with data that the Visual C# application provides.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

The main tasks for this exercise are as follows:

1. Examine the data files.
2. Open the starter project and examine the **StressData** type.
3. Examine the GraphWindow test harness.
4. Copy data to an Office Excel worksheet.
5, Generate an Office Excel graph.
6. Complete the test harness.
7. Test the application.

▶ Task 1: Examine the data files

1. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder, and then verify that this folder contains the following three text files:
   - 298K.txt
   - 318K.txt
   - 338K.txt
2. Using Notepad, open the 298K.txt file.

   This file contains results from the deflection tests for steel girders that were subjected to various pressures at a temperature of 298 Kelvin. The number on a line by itself at the top of the file is the temperature at which the tests were performed (298). The remaining lines contain pairs of numbers; the numbers in each pair are separated by a comma. These numbers are the pressure applied, which is measured in kiloNewtons (kN), and the deflection of the girder, which is measured in millimeters.
3. Close Notepad.
4. Using Notepad, open the 318K.txt file.

   This file is in the same format as the 298K.txt file. It contains the results of deflection tests that were performed at a temperature of 318 Kelvin. Notice that the final few lines do not contain any deflection data because the test was halted at a force of 1,000 kN.
5. Close Notepad.
6. Using Notepad, open the 338K.txt file.

This file is similar to the other two. It contains the results of deflection tests that were performed at a temperature of 338 Kelvin. The test was halted at a force of 800 kN.

7. Close Notepad.

▶ Task 2: Open the starter project and examine the StressData type

1. Using Visual Studio, open the GenerateGraph solution in the E:\Labfiles \Lab 15\Starter\GenerateGraph folder.
2. Open the StressData.cs file.

   The **StressData** type acts as a container for the stress data for a given temperature. It contains the following public properties:

   - **Temperature**. This is a short value that records the temperature of the test.
   - **Data**. This is a **Dictionary** collection that holds the data. The stress value is used as the key into the dictionary, and the item data is the deflection.

   The **StressData** class also overrides the **ToString** method, which returns a formatted string that lists the stress test data that is stored in the object.

▶ Task 3: Examine the GraphWindow test harness

1. Open the GraphWindow.xaml file.

   This window provides a simple test harness for reading the data from the data files and invoking Office Excel to generate a graph by using this data.

   When users click **Get Data**, they are prompted for the data file to load. The file is read into a new **StressData** object, and the contents of the file are displayed in the **TreeView** control that occupies the main part of the window. A user can click **Get Data** multiple times and load multiple files; they will all be read in and displayed. The **StressData** objects are stored in a **List** collection that is held in a private field in the **GraphWindow** class and is called **graphData**. This code has already been written for you.

   When a user clicks **Graph**, the data in the **graphData** collection will be used to generate an Office Excel graph. The information in each **StressData** object will be transferred to an Office Excel worksheet, and a line graph will then be generated to show the stress data for each temperature. A user can quickly examine this graph and spot any trends in the failure of girders.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Open the GraphWindow.xaml.cs code file.
3. Locate the **populateFromFile** method.

   This method uses a **StreamReader** object to read and parse the stress data from a file that is specified as a parameter, and it populates a **StressData** object that is also specified as a parameter.

   This method is complete.
4. Locate the **displayData** method.

   This method takes a populated **StressData** object and displays the items in this object in the **TreeView** control in the window.

   This method is also complete.
5. Locate the **getData_Click** method.

   This method runs when the user clicks the **Get Data** button. It uses an **OpenFileDialog** object to prompt the user for the name of a data file and then passes the file name together with a new **StressData** object to the **populateFromFile** method. It then adds the populated **StressData** object to the **graphData** collection before it calls the **displayData** method to add the data to the **TreeView** control in the window.

   This method is complete.
6. Locate the **generateGraph_Click** method.

   This method runs when the user clicks the **Generate** button. It prompts the user for the name of an Office Excel workbook to create. It will then create this new workbook and copy the data in the **graphData** collection into a worksheet in this workbook before it generates a graph.

   This method is not complete. You will add the missing functionality and complete the **transferDataToExcelSheet** and **generateExcelChart** helper methods that this code will use.

▶ Task 4: Copy data to an Office Excel worksheet

1. Add a reference to the Microsoft Excel 12.0 Object Library to the application. This is the COM object library that implements the Office Excel object model.
2. Review the task list.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the task list, locate the **TODO: Add the Microsoft.Office.Interop.Excel namespace** task, and then double-click this task. This task is located near the top of the GraphWindow.xaml.cs file.
4. Bring the **Microsoft.Office.Interop.Excel** namespace into scope, and give it an alias of **Excel**. This alias helps you to distinguish items in this namespace and avoid name clashes without having to specify the full namespace in ambiguous object references.
5. Locate the **transferDataToExcelSheet** method.

   The **generateGraph_Click** method will call this method. It takes three parameters:

   - An **Excel.Worksheet** object called **excelWS**. This object is a reference to the Office Excel worksheet that you will copy the data to.
   - An **Excel.Range** object called **dataRange**. This is an output parameter. You will use this object to indicate the area of the worksheet that contains the data after it has been copied.
   - A **List<StressData>** object called **excelData**. This is a collection of **StressData** objects that contain the data that you will copy to the Office Excel worksheet.

   This method returns **true** if it successfully copies the data to the Office Excel worksheet and **false** if an exception occurs.
6. In the **transferDataToExcelSheet** method, after the comment **TODO: Copy the data for the applied stresses to the first column in the worksheet**, add code that performs the following tasks:
   a. Declare an integer variable called rowNum and initialize it to **1**.
   b. Declare an integer variable called colNum and initialize it to **1**.
   c. Set the value of the cell at location rowNum, colNum in the **excelWS** worksheet object to the text "**Applied Stress**".



Hint: You can use the **Cells** property to read and write a cell in an **Excel** worksheet object. This property acts like a two-dimensional array.

   d. Use a **foreach** loop to iterate through the keys in the first **StressData** object in the **excelData** collection.

MCT USE ONLY. STUDENT USE PROHIBITED



Hint: Remember that the **StressData** object contains a **Dictionary** property called **Data**, and the key values in this dictionary are the applied stresses for the test (100, 200, 300, up to 1,500 kN). You can use the **Keys** property of a **Dictionary** object to obtain a collection of keys that you can iterate through.

e. In the body of the **foreach** loop, increment the rowNum variable, and store the value of each key found in the cell at location rowNum, colNum in the **excelWS** worksheet object.

7. Locate the comment **TODO: Give each column a header that specifies the temperature**.

   This comment is located in a **foreach** loop that iterates over each item in the **excelData** collection. These items are **StressData** objects, and each **StressData** object contains the data for the tests for a given temperature. When complete, the code in this **foreach** loop will copy the data for each **StressData** object to a new column in the **excelWS** worksheet object, and each column will have a header that specifies the temperature.

8. After the comment, add code that performs the following tasks:

   a. Increment the colNum variable so that it refers to the next column in the worksheet.

   b. Set the rowNum variable to **1**.

   c. Retrieve the temperature from the **deflectionData StressData** object, format it as a string with the letter "K" appended to the end (for Kelvin), and store this string in the cell at location rowNum, colNum in the **excelWS** worksheet object.

9. Locate the comment **TODO: Only copy the deflection value if it is not null**.

   This comment is located in a nested **foreach** loop that iterates over each value in a **StressData** object. Remember that not all stresses have a deflection value. Where this occurs, the data in the **StressData** object is null. The **if** statement detects whether the current deflection value is null.

10. After the comment, in the body of the **if** statement, add code that performs the following tasks:

    a. Increment the rowNum variable so that it refers to the next row in the worksheet.

    b. Copy the value of the deflection variable (that contains the deflection data) into the cell at location rowNum, colNum in the **excelWS** worksheet object.

MCT USE ONLY. STUDENT USE PROHIBITED

11. Locate the comment **TODO: Specify the range of cells in the spreadsheet containing the data in the dataRange variable**.

    This comment is located after all of the **foreach** loops have completed and all of the data has been copied to the worksheet.

12. After the comment, add a statement that populates the dataRange variable with information about the set of cells that have been filled.



Hint: You can determine the boundaries of the filled area of an Office Excel worksheet by querying the **UsedRange** property. This property returns an **Excel.Range** object.

13. Build the solution and correct any errors.

▶ Task 5: Generate an Office Excel graph

1. Locate the **generateExcelChart** method.

   The **generateGraph_Click** method will call this method after the data has been copied to the Office Excel worksheet. It takes three parameters:

   - A **string** object called **fileName**. When the graph has been created, the method will save the Office Excel workbook to a file by using this file name.
   - An **Excel.Workbook** object called **excelWB**. This is a reference to the Office Excel workbook containing the Office Excel worksheet that contains the data to use for the graph.
   - An **Excel.Range** object called **dataRange**. This range specifies the location in the Office Excel worksheet that contains the data to use for the graph.

2. In the **generateExcelChart** method, after the comment **TODO: Generate a line graph based on the data in the dataRange range**, add code that performs the following tasks:

   a. Add a new chart object to the Office Excel workbook, and store a reference to this chart object in an **Excel.Chart** variable called excelChart.



Hint: You can create a new chart by using the **Add** method of the **Charts** property of an Office Excel workbook object. The **Add** method takes no parameters and returns a reference to the chart object.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Call the **ChartWizard** method of the chart object to generate the chart. The following table lists the parameters that you should specify.

Parameter name	Value
Title	**"Applied Stress (kN) versus Deflection (mm)"**
Source	**dataRange**
Gallery	**Excel.XlChartType.xlLine**
PlotBy	**Excel.XlRowCol.xlColumns**
CategoryLabels	**1**
SeriesLabels	**1**
ValueTitle	**"Deflection"**
CategoryTitle	**"Applied Stress"**

3. After the comment **TODO: Save the Excel workbook**, add a statement that saves the Office Excel workbook by using the value in the *fileName* parameter.



Hint: Use the **SaveAs** method of the Office Excel Workbook object to save a workbook. This method takes a parameter called *Filename* that specifies the name of the file to use.

4. Build the solution and correct any errors.

▶ Task 6: Complete the test harness

1. Return to the **generateGraph_Click** method.
2. After the comment **TODO: If the user specifies a valid file name, start Excel and create a new workbook and worksheet to hold the data**, add code to perform the following tasks:
   a. Create a new **Excel.Application** object called **excelApp**.
   b. Make the application visible on the user's desktop by setting the **Visible** property of the **excelApp** object to **true**. (By default, Office Excel will run in the background.)

MCT USE ONLY. STUDENT USE PROHIBITED

c. Set the **AlertBeforeOverwriting** property of the **excelApp** object to **false**. This ensures that the **SaveAs** method always saves the workbook.

d. Set the **DisplayAlerts** property of the **excelApp** object to **false**.

e. Create a new workbook, and store a reference to this workbook in an **Excel.Workbook** variable called excelWB.



Hint: You can create a new workbook by using the **Add** method of the **Workbooks** property of an **Excel.Application** object. This method takes no parameters and returns a reference to the new workbook.

f. Create a variable called excelWS of type **Excel.Worksheet** and set it as the active worksheet in the new workbook.



Hint: You can obtain a reference to the active worksheet in an Office Excel workbook by using the **ActiveSheet** property.

3. After the comment **TODO: Copy the data from the graphData variable to the new worksheet and generate a graph**, add code to perform the following tasks:

   a. Create an **Excel.Range** object called **dataRange** and initialize it to null.

   b. Call the **transferDataToExcelSheet** method, and pass the **excelWS** object, the **dataRange** object, and the graphData variable as parameters. Note that the **dataRange** object should be an output parameter.

   c. If the value that the **transferDataToExcelSheet** method returns is **true**, call the **generateExcelChart** method. Pass the **FileName** property of the **SaveDialog** object, the **excelWB** object, and the **dataRange** object as parameters.

4. At the end of the **generateGraph_Click** method, in the **finally** block, after the comment **TODO: Close Excel and release any resources**, add code to check whether the excelApp variable is null; if it is not, close the Office Excel application.



Hint: Use the **Quit** method of an **Excel.Application** object to close Office Excel.

5. Build the solution and correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

▶ Task 7: Test the application

1. Start the application in Debug mode.
2. In the Graphing Data window, click **Get Data**.
3. In the **Graph Data** dialog box, click the **298K.txt** file, and then click **Open**.
4. In the Graphing Data window, in the tree view, expand the **Temperature: 298K** node. Verify that the data has been correctly loaded.
5. Repeat steps 2, 3, and 4 and load the data in the 318K.txt and 338K.txt files. Verify that the tree view lists the data from all three files.



Note: The **displayData** method displays the value –1 for any missing deflection data.

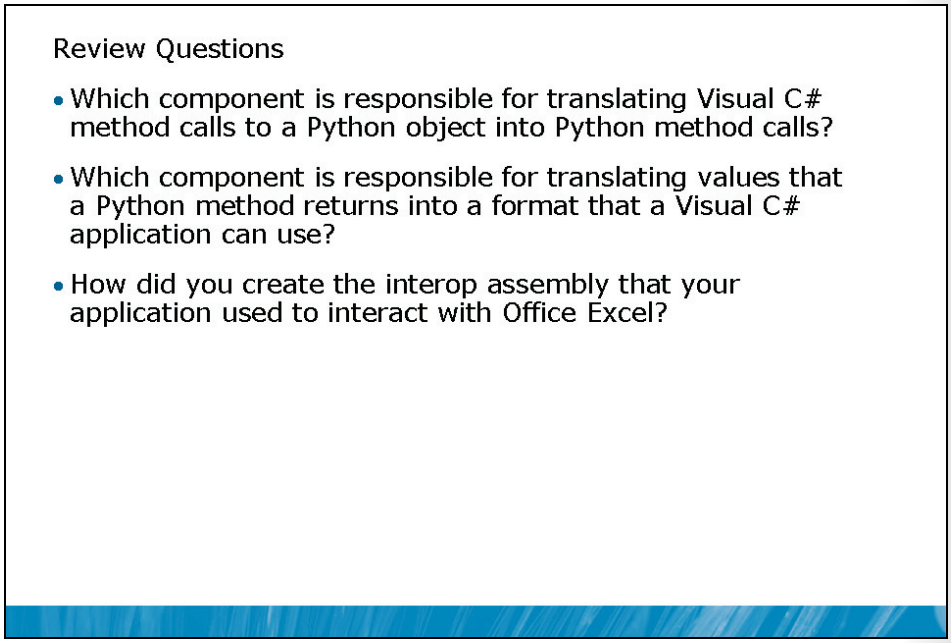
6. Click **Graph**.
7. In the **Graph Data** dialog box, accept the default file name, StressData.xlsx, for the name of the Office Excel workbook to be generated, and then click **Save**.

   You will see Office Excel start to run and your data copied across to a new worksheet. You will also briefly see the graph that is generated before the workbook is saved and Office Excel closes.
8. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder. Verify that this folder contains the Office Excel workbook StressData.xlsx.
9. Double-click the **StressData.xlsx** file to start Office Excel and open the workbook.

   The workbook should contain a chart that displays the stress test results by using the data and settings that you specified.
10. In Office Excel, click the **Sheet1** tab.

    This is the worksheet that your code generated. The first column contains the applied stress values, and the remaining three columns contain the deflections recorded at each of the three temperatures.
11. Close Office Excel.
12. Close the Graphing Data window.
13. Close Visual Studio.

Lab Review

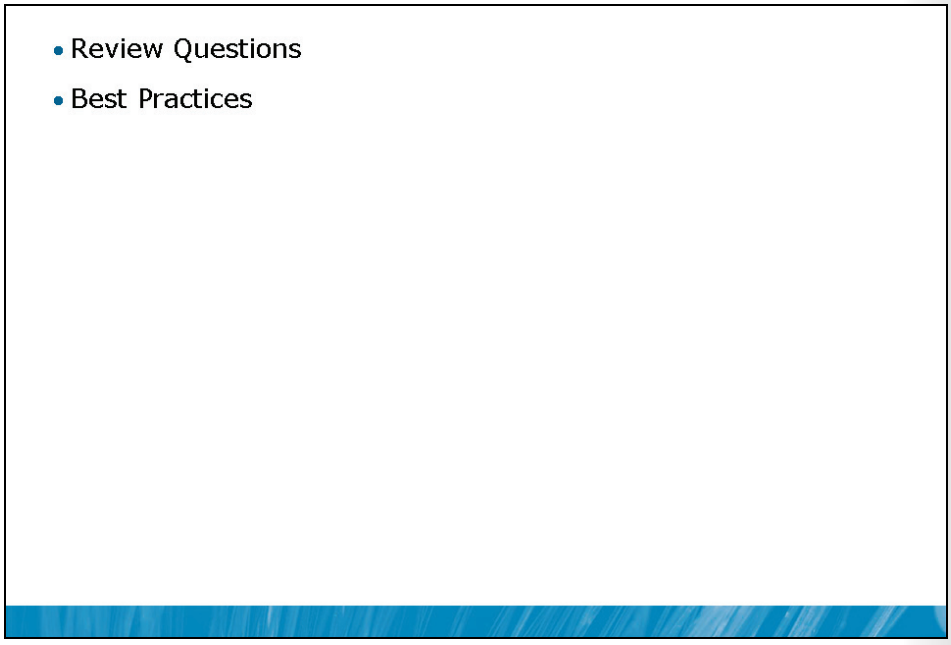


Review Questions

1. Which component is responsible for translating Visual C# method calls to a Python object into Python method calls?
2. Which component is responsible for translating values that a Python method returns into a format that a Visual C# application can use?
3. How did you create the interop assembly that your application used to interact with Office Excel?

MCT USE ONLY. STUDENT USE PROHIBITED

Module Review and Takeaways



Review Questions

1. What is the difference between the **dynamic** type and the **object** type?
2. What is the difference between an interop assembly and an RCW?

Best Practices Related to Integrating Visual C# Code with Dynamic Languages

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

- Do not use the **dynamic** keyword as a substitute for the **var** keyword. A variable defined as **var** is strongly typed and all references are checked at compile time. A variable defined as **dynamic** is only checked at run time and imposes an additional overhead at run time.
- Program defensively. Do not assume when you build an application that uses scripts that are based on dynamic languages that those scripts will be well-behaved. Be prepared to catch and handle exceptions that scripts cause.

MCT USE ONLY. STUDENT USE PROHIBITED

- Be prepared to catch and handle exceptions that are caused by missing scripts or unexpected versions of the runtime for the dynamic language.
- Only use scripts from trusted sources.

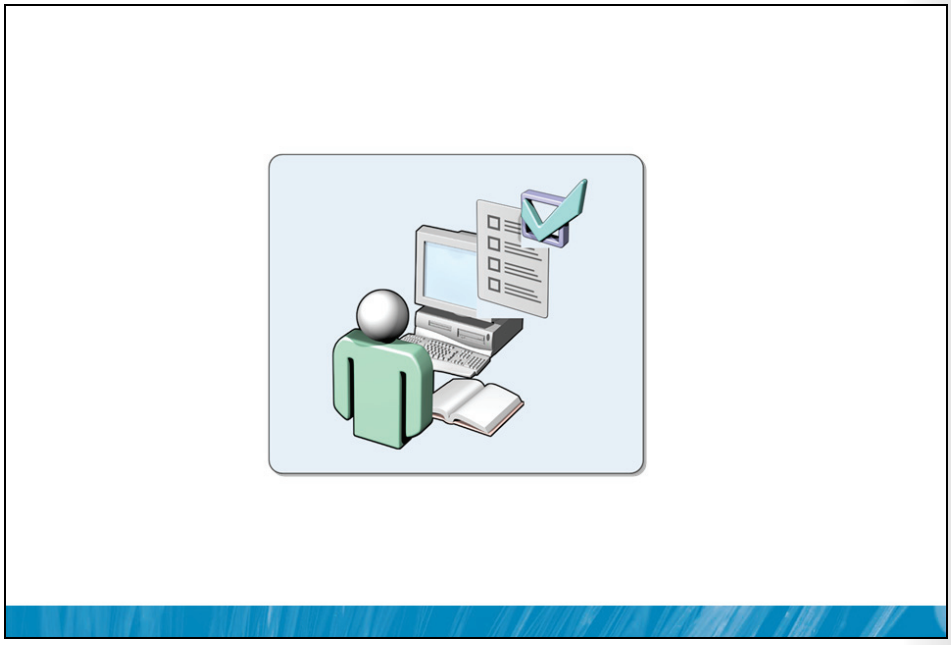
Best Practices Related to Accessing COM Components from Visual C#

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

- Ensure that any COM components that an application uses are installed and available on the computer that runs your application. Be prepared to catch and handle exceptions that missing COM components cause.
- Use PIA-less deployment.
- Only use COM components that trusted sources provide.

MCT USE ONLY. STUDENT USE PROHIBITED

Course Evaluation



Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Introducing C# and the .NET Framework

Lab 1: Introducing C# and the .NET Framework

Exercise 1: Building a Simple Console Application

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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Create a new console application project called ConsoleApplication in the E:\Labfiles\Lab 1\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
   b. In the **New Project** dialog box, in the Installed Templates pane, expand **Visual C#**, and then click **Windows**.
   c. In the Templates pane, click **Console Application**.
   d. Specify the following values for each of the properties in the dialog box, and then click **OK**:
      - Name: **ConsoleApplication**
      - Location: **E:\Labfiles\Lab 1\Ex1\Starter**
      - Solution name: **ConsoleApplication**
      - Create directory for solution: Select the check box.

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:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
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:
   a. On the **Debug** menu, click **Start Without Debugging**.
   b. In the console window, type some text, and then press ENTER.
   c. Verify that the text that you typed is echoed to the console.
   d. Press ENTER to return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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 reads 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)
    {
    }

    // 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:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

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:
   a. On the **Debug** menu, click **Start Without Debugging**.
   b. In the console window, type some text, and then press ENTER.
   c. Verify that the text that you typed is echoed to the console.
   d. Type some more text, and then press ENTER again.
   e. Verify that this line is also echoed to the console.
   f. Press CTRL+Z, and then verify that the application finishes.
   g. Press ENTER to return to Visual Studio.

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
        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.

MCT USE ONLY. STUDENT USE PROHIBITED

```
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:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
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
```

   a. On the **Debug** menu, click **Start Without Debugging**.
   b. In the console window, type **23.54367,25.6789** and then press ENTER.
   c. Verify that the text **x:23.54367, y:25.6789** is displayed on the console.
   d. Type some more text that consists of pairs of numbers that are separated by a comma, and then press ENTER again.
   e. Verify that this data is correctly formatted and displayed on the console.
   f. Press CTRL+Z.
   g. Press ENTER to return to Visual Studio.

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:

   - On the **Build** menu, click **Rebuild Solution**.

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

   a. Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt (2010)**.

   b. In the Visual Studio Command Prompt (2010) window, move to the E:\Labfiles\Lab 1\Ex1\Starter\ConsoleApplication\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:
   a. In Solution Explorer, right-click the **ConsoleApplication** project, and then click **Properties**.
   b. On the **Debug** tab, in the **Command line arguments** text box, type **< DataFile.txt**
   c. On the **File** menu, click **Save All**.
   d. Close the ConsoleApplication properties window.
7. Run the application in Debug mode from Visual Studio:
   - Click **Debug**, and then click **Start Debugging**.

   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:
   - In the Program.cs file, move the cursor to the closing brace (}) that corresponds to the end of the **Main** method, right-click, point to **Breakpoint**, and then click **Insert Breakpoint**.
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:
   a. Click **Debug**, and then click **Start Debugging**. The application will run, and then the program will stop when control reaches the end of the **Main** method and Visual Studio has the focus.
   b. In the Windows taskbar, click the icon for **ConsoleApplication.exe**.
   c. Verify that the output that is displayed in the ConsoleApplication.exe window is the same as before.
   d. Return to Visual Studio, and then on the **Debug** menu, click **Continue**. The application will finish.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Building a WPF Application

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 Windows Presentation Foundation (WPF) Application template:
  a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
  b. In the **New Project** dialog box, in the Project Types pane, expand **Visual C#**, and then click **Windows**.
  c. In the Templates pane, click **WPF Application**.
  d. Specify the following values for each of the properties in the dialog box, and then click **OK**:
     - Name: **WpfApplication**
     - Location: **E:\Labfiles\Lab 1\Ex2\Starter**
     - Solution name: **WpfApplication**
     - Create directory for solution: Select the check box

Task 2: Create the user interface

1. Add **TextBox**, **Button**, and **TextBlock** controls to the MainWindow window. Place them anywhere in the window:
   a. Click the **Toolbox** tab.
   b. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.
   c. Click the **TextBox** control, and then drag it anywhere onto the MainWindow window.
   d. Click the **Toolbox** tab.
   e. Click the **Button** control, and then drag it anywhere onto the MainWindow window.
   f. Click the **Toolbox** tab.
   g. Click the **TextBlock** control, and then drag it anywhere onto the MainWindow 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**	**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**

a. In the MainWindow window, click the **TextBox** control.

MCT USE ONLY. STUDENT USE PROHIBITED

b. In the Properties window, click the **textBox1** text adjacent to the **TextBox** prompt, and then change the name to **testInput**.

c. In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.

d. Repeat this process for the remaining properties of the **TextBox** control.

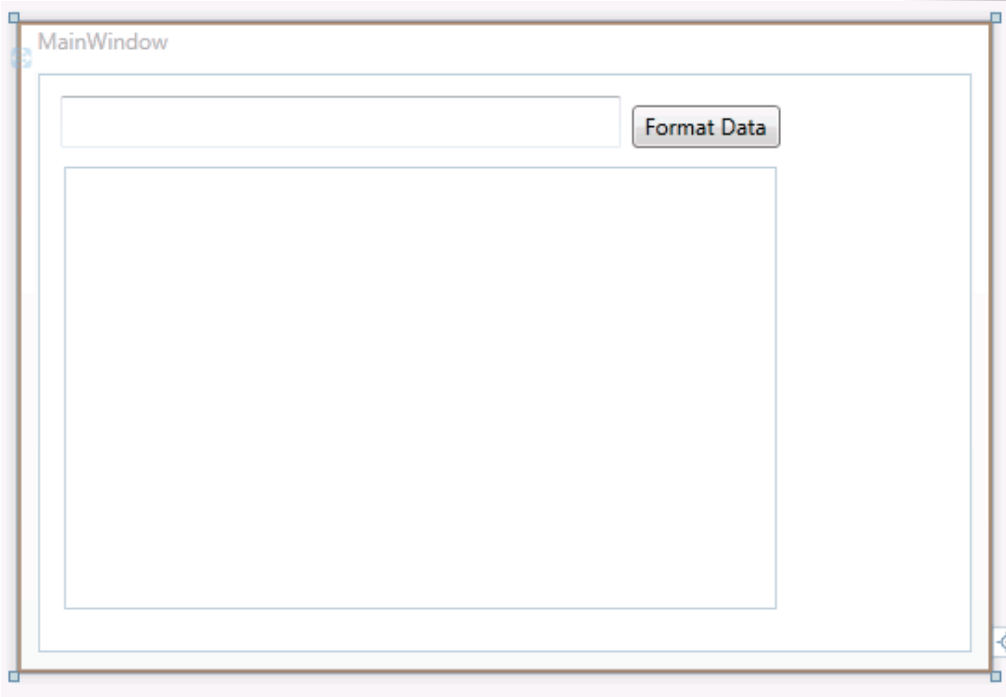
e. In the MainWindow window, click the **Button** control.

f. Follow the procedure described in steps b to e to set the specified properties for this control.

g. In the MainWindow window, click the **TextBlock** control.

h. Follow the procedure described in steps b to e to set the specified properties for this control.

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 format the data that the user enters

1. Create an event handler for the **Click** event of the button:
   a. In the MainWindow window, click the **Button** control.
   b. In the Properties window, click the **Events** tab.
   c. In the list of events, double-click the **Click** event.
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;
}
```

   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:
   - On the **Build** menu, click **Rebuild Solution**.
4. Run the application and verify that it works in a similar manner to the original console application in Exercise 1:
   a. On the **Debug** menu, click **Start Without Debugging**.
   b. In the MainWindow window, type **23.654,67.823** into the **TextBox** control.
   c. Click **Format Data**.
   d. Verify that **x:23.654 y:67.823** appears in the **TextBlock** control below the **TextBox** control.
5. Close the MainWindow window, and then return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   a. Display the MainWindow.xaml file.
   b. Click the title bar of the MainWindow window.
   c. In the Properties window, click the **Events** tab.
   d. In the list of events, double-click the **Loaded** event.
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 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**.

MCT USE ONLY. STUDENT USE PROHIBITED

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
```

- On the **Debug** menu, click **Start Debugging**.

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

Exercise 3: Verifying the Application

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
```

a. In Solution Explorer, double-click **DataFile.txt**.

b. Edit the data in the file so that it resembles the data shown.

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

d. Close the DataFile.txt window.



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:

   a. Display the MainWindow.xaml.cs file.

   b. Scroll down to the **Window_Loaded** event.

   c. Right-click the statement in the following code example, point to **Breakpoint**, and then click **Insert Breakpoint**.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
```

2. Start the application running in Debug mode:

   - On the **Debug** menu, click **Start Debugging**.

   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:

   - On the **Debug** menu, click **Step Into**, or press F11.

   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:

   - In the Locals window, verify that the value of line is **null**.

5. Step into the next statement:

   - On the **Debug** menu, click **Step Into**, or press F11.

MCT USE ONLY. STUDENT USE PROHIBITED

- 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:
   - In the Locals window, verify that the value of line is **1.2543,0.342**.
7. Step into the next statement:
   - On the **Debug** menu, click **Step Into**, or press F11.

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

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

8. Step into the next statement:
   - On the **Debug** menu, click **Step Into**, or press F11.
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:
   - In the Locals window, verify that the value of line is **1.2543 y:0.342**.
10. Step into the next statement:
    - On the **Debug** menu, click **Step Into**, or press F11.
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:
    - In the Locals window, verify that the value of line is **x:1.2543 y:0.342\n**.
12. Step into the next statement:
    - On the **Debug** menu, click **Step Into**, or press F11.

    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.

MCT USE ONLY. STUDENT USE PROHIBITED

a. In the Immediate window, type the expression in the following code example (including the question mark), and then press ENTER.

```
?formattedText.Text
```

b. Verify that the text "x:1.2534 y:0.342\n" is displayed.

14. Set another breakpoint at the end of the **while** loop:

- Right-click the closing brace at the end of the **while** loop, point to **Breakpoint**, and then click **Insert Breakpoint**.

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:

- On the **Debug** menu, click **Continue**, or press F5.

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:

- In the Locals window, verify that the value of line is **x:32525.7639 y:99811.9463\n**.

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:

a. In the Immediate window, on a blank line after the previous results, type the expression in the following code example (including the question mark), and then press ENTER.

```
?formattedText.Text
```

b. Verify that the text "x:1.2543 y:0.342\n x:32525.7639 y:99811.9463\n" is displayed.

18. Remove the breakpoint from the end of the **while** loop:

- Right-click the closing brace at the end of the **while** loop, point to **Breakpoint**, and then click **Delete Breakpoint**.

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:

a. On the **Debug** menu, click **Continue**, or press F5.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Verify that the **TextBlock** control displays the formatted results for every line in the DataFile.txt file.

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

Exercise 4: Generating Documentation for an Application

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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. Move to the **E:\Labfiles\Lab 1\Ex4 \Starter** folder, click **WpfApplication.sln**, and then click **Open**.

Task 2: Add XML comments to the application

1. Display the MainWindow.xaml.cs file:

   - In Solution Explorer, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

```
/// <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 **Window_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:
   a. In Solution Explorer, right-click the **WpfApplication** project, and then click **Properties**.
   b. On the **Build** tab, select the **XML Documentation file** check box, and then verify that the file name is set to: **bin\Debug\comments.XML**.
   c. On the **File** menu, click **Save All**.
   d. Close the WpfApplication properties window.
2. Build the solution, and then correct any errors:
   - On the **Build** menu, click **Rebuild Solution**.
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:

MCT USE ONLY. STUDENT USE PROHIBITED

a. Using Windows Explorer, move to the **E:\Labfiles\Lab 1\Ex4\Starter \WpfApplication\bin\Debug** folder.

b. Double-click the **comments.xml** file. The file should be displayed in Internet Explorer. Verify that it contains the text for the XML comments that you added to the WPF application (it will also contain other comments that Visual Studio has generated).

c. Close Internet Explorer.

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**:

   a. Click **Start**, point to **All Programs**, click **Accessories**, right-click **Command Prompt,** and then click **Run as administrator**.

   b. In the **User Account Control** dialog box, in the **Password** text box, type **Pa$$w0rd** and then click **Yes**.

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":

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

```
notepad builddoc.cmd
```

   b. Verify that Line 13 looks like the following code example.

```
set input="E:\Labfiles\Lab
1\Ex4\Starter\WpfApplication\bin\Debug\WpfApplication.exe"
```

   c. Close Notepad.

4. Run the builddoc.cmd script:

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

```
builddoc.cmd
```

MCT USE ONLY. STUDENT USE PROHIBITED

b. Press ENTER when the script prompts you to do so.

5. Open the test.chm file that the builddoc.cmd script generates:

   a. Using Windows Explorer, move to the **E:\Labfiles\Lab 1 \Ex4\HelpFile\Output** folder.

   b. Double-click the **test.chm** file.

6. Browse documentation that is generated for your application, and then close test.chm.

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

MCT USE ONLY. STUDENT USE PROHIBITED

Using C# Programming Constructs

Lab 2: Using C# Programming Constructs

Exercise 1: Calculating Square Roots with Improved Accuracy

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:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **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:
   a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
   b. In the **New Project** dialog box, in the Project Types pane, expand **Visual C#**, and then click **Windows**.
   c. In the Templates pane, click **WPF Application**.
   d. Specify the following values for each of the properties in the dialog box, and then click **OK**:
      - Name: **SquareRoots**
      - Location: **E:\Labfiles\Lab 2\Ex1\Starter**
      - Solution name: **SquareRoots**
      - Create directory for solution: Select the check box.

Task 2: Create the user interface

1. Add **TextBox**, **Button**, and two **Label** controls to the MainWindow window. Place them anywhere in the window:

a. In Solution Explorer, double-click **MainWindow.xaml**.

b. Click the **Toolbox** tab.

c. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.

d. Drag the **TextBox** control anywhere into the MainWindow window.

e. Click the **Toolbox** tab.

f. Drag the **Button** control anywhere into the MainWindow window.

g. Click the **Toolbox** tab.

h. Drag the **Label** control anywhere into the MainWindow window.

i. Click the **Toolbox** tab.

j. Drag the **Label** control anywhere into the MainWindow 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**

MCT USE ONLY. STUDENT USE PROHIBITED

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**

a. In the MainWindow window, click the **TextBox** control.

b. In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.

c. In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.

d. Repeat this process for the remaining properties of the **TextBox** control.

e. In the MainWindow window, click the **Button** control.

MCT USE ONLY. STUDENT USE PROHIBITED

f. Follow the procedure described in steps b to d to set the specified properties for this control.

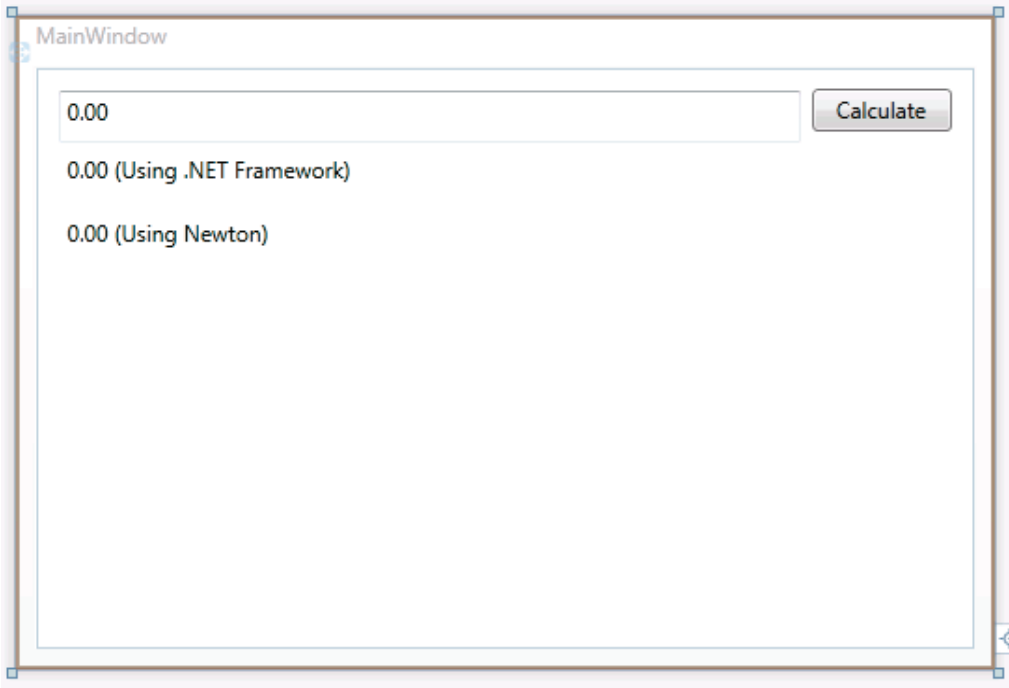
g. In the MainWindow window, click one of the **Label** controls.

h. Follow the procedure described in steps b to d to set the specified properties for this control.

i. In the MainWindow window, click the other **Label** control.

j. Follow the procedure described in steps b to d to set the specified properties for this control.

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:
   - In the MainWindow window, click the **Button** control.
   - In the Properties window, click the **Events** tab.

MCT USE ONLY. STUDENT USE PROHIBITED

- In the list of events, double-click the **Click** event.

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**. 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.

- Add the code in the following code example to the **calculateButton_Click** method.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox
    double numberDouble;
    if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }
}
```

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:

- Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Check that the user has entered a positive number
    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   - Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Use the .NET Framework Math.Sqrt method
    double squareRoot = Math.Sqrt(numberDouble);
}
```

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:
   - Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Format the result and display it
    frameworkLabel.Content = string.Format("{0} (Using the .NET
Framework)", squareRoot);
}
```

   At this point, your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox
    double numberDouble;
    if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Check that the user has entered a positive number
    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }

    // Use the .NET Framework Math.Sqrt method
    double squareRoot = Math.Sqrt(numberDouble);

    // Format the result and display it
    frameworkLabel.Content = string.Format("{0} (Using the .NET
Framework)", squareRoot);
}
```

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

   a. On the **Debug** menu, click **Start Debugging**.

   b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.

MCT USE ONLY. STUDENT USE PROHIBITED

c. Verify that the result matches the text in the **Expected result** column.

d. Repeat steps b and c for each row in the table.

7. Close the application and return to Visual Studio.

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:

   - Add the code in the following code example to the end of the **calculateButton_Click** method.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Newton's method for calculating square roots

    // Get user input as a decimal
    decimal numberDecimal;
    if (!decimal.TryParse(inputTextBox.Text, out numberDecimal))
    {
        MessageBox.Show("Please enter a decimal");
        return;
    }
}
```



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.

MCT USE ONLY. STUDENT USE PROHIBITED



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.

Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Specify 10 to the power of -28 as the minimum delta between
    // estimates. This is the minimum range supported by the decimal
    // type. When the difference between 2 estimates is less than this
    // value, then stop.
    decimal delta = Convert.ToDecimal(Math.Pow(10, -28));
}
```

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.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Take an initial guess at an answer to get started
    decimal guess = numberDecimal / 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**.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Estimate result for the first iteration
    decimal result = ((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

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...

    // While the difference between values for each current iteration
    // is not less than delta, then perform another iteration to
    // refine the answer.

    while (Math.Abs(result - guess) > delta)
    {
        // Use the result from the previous iteration
        // as the starting point
        guess = result;

        // Try again
        result = ((numberDecimal / guess) + guess) / 2;
    }
}
```

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.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the result
    newtonLabel.Content = string.Format("{0} (Using Newton)", result);
}
```

Your completed code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox

    double numberDouble;

  if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }

    // Check that the user has entered a positive number

    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }

    // Use the .NET Framework Math.Sqrt method

    double squareRoot = Math.Sqrt(numberDouble);

    // Format the result and display it

    frameworkLabel.Content = string.Format("{0} (Using .NET
Framework)", squareRoot);

    // Newton's method for calculating square roots

    // Get the user input as a decimal

    decimal numberDecimal;
    if (!decimal.TryParse(inputTextBox.Text, out numberDecimal))
    {
        MessageBox.Show("Please enter a decimal");
        return;
    }

    // Specify 10 to the power of -28 as the minimum delta between
    // estimates. This is the minimum range supported by the decimal
    // type. When the difference between 2 estimates is less than this
    // value, then stop.

    decimal delta = Convert.ToDecimal(Math.Pow(10, -28));

    // Take an initial guess at an answer to get started
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    decimal guess = numberDecimal / 2;

    // Estimate result for the first iteration
    decimal result = ((numberDecimal / guess) + guess) / 2);

    // While the difference between values for each current iteration
    // is not less than delta, then perform another iteration to
    // refine the answer.
    while (Math.Abs(result - guess) > delta)
    {
        // Use the result from the previous iteration
        // as the starting point
        guess = result;

        // Try again
        result = ((numberDecimal / guess) + guess) / 2;
    }

    // Display the result
    newtonLabel.Content = string.Format("{0} (Using Newton)", result);
}
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

a. On the **Debug** menu, click **Start Debugging**.

b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.

c. Repeat step b for each row in the table.

2. As a final test, try the value 0.0000000000000000000000000001 (27 zeroes after the decimal point). Can you explain the result?

   - The program halts and reports that **DivideByZeroException** was unhandled. This exception occurs because the value that is specified is smaller than the range of values that the **decimal** type allows, so the value in the guess variable is 0. You will see how to catch and handle exceptions in a later module.

3. Close the application and return to Visual Studio:

   - On the **Debug** menu, click **Stop Debugging**.

Exercise 2: Converting Integer Numeric Data to Binary

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:

  a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.

  b. In the **New Project** dialog box, in the Project Types pane, expand **Visual C#**, and then click **Windows**.

  c. In the Templates pane, click **WPF Application**.

  d. Specify the following values for each of the properties in the dialog box, and then click **OK**:

     - Name: **IntegerToBinary**.
     - Location: **E:\Labfiles\Lab 2\Ex2\Starter**
     - Solution name: **IntegerToBinary**
     - Create directory for solution: Select the check box.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Create the user interface

1. Add a **TextBox**, **Button**, and **Label** control to the MainWindow window. Place them anywhere in the window:
   a. In Solution Explorer, double-click **MainWindow.xaml**.
   b. Click the **Toolbox** tab.
   c. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.
   d. Drag the **TextBox** control anywhere into the MainWindow window.
   e. Click the **Toolbox** tab.
   f. Drag the **Button** control anywhere into the MainWindow window.
   g. Click the **Toolbox** tab.
   h. Drag the **Label** control anywhere into the MainWindow 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**

Control	Property	Value
	VerticalAlignment	**Top**
	Width	**75**
Label	**Name**	**binaryLabel**
	Content	**0**
	Height	**28**
	HorizontalAlignment	**Left**
	Margin	**12,41,0,0**
	VerticalAlignment	**Top**
	Width	**120**

a. In the MainWindow window, click the **TextBox** control.

b. In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.

c. In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.

d. Repeat this process for the remaining properties of the **TextBox** control.

e. In the MainWindow window, click the **Button** control.

f. Follow the procedure described in steps b to d to set the specified properties for this control.

g. In the MainWindow window, click one of the **Label** controls.

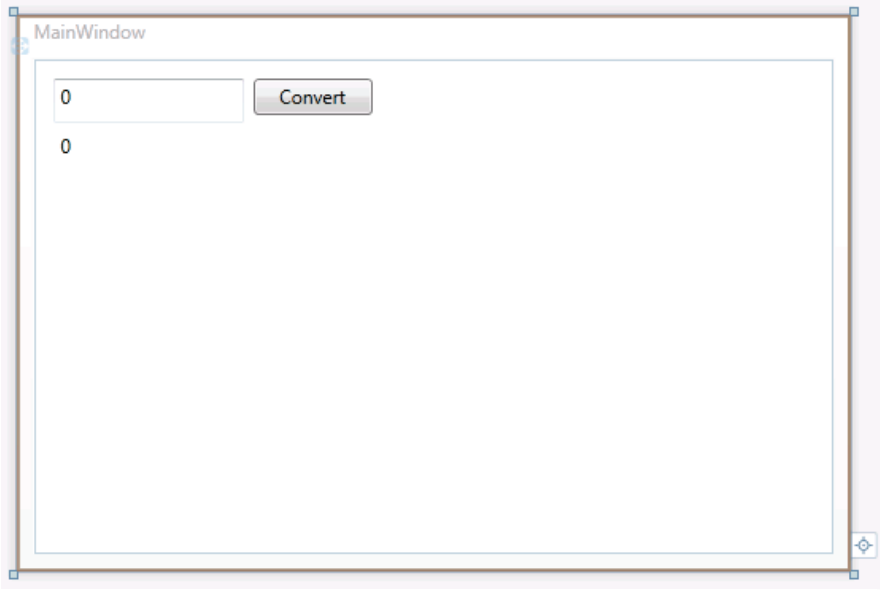
h. Follow the procedure described in steps b to d to set the specified properties for this control.

i. In the MainWindow window, click the other **Label** control.

j. Follow the procedure described in steps b to d to set the specified properties for this control.

MCT USE ONLY. STUDENT USE PROHIBITED

The MainWindow window should look like the following screen shot.



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:
   a. In the MainWindow window, click the **Button** control.
   b. In the Properties window, click the **Events** tab.
   c. In the list of events, double-click the **Click** event.
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:
   - Add the code in the following code example to the **convertButton_Click** method.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    // Get the integer entered by the user
    int i;
    if (!int.TryParse(inputTextBox.Text, out i))
    {
        MessageBox.Show("TextBox does not contain an integer");
        return;
    }
}
```

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:

   - Add the statements in the following code example to the **convertButton_Click** method, after the code that you added in the previous step.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Check that the user has not entered a negative number
    if (i < 0)
    {
        MessageBox.Show("Please enter a positive number or zero");
        return;
    }
}
```

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.

   Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Remainder will hold the remainder after dividing i by 2
    // after each iteration of the algorithm
    int remainder = 0;
}
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Binary will be used to construct the string of bits
    // that represent i as a binary value
    StringBuilder binary = new StringBuilder();
}
```

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.

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Generate the binary representation of i
    do
    {
        remainder = i % 2;
        i = i / 2;
        binary.Insert(0, remainder);
    }
    while (i > 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

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the result
    binaryLabel.Content = binary.ToString();
}
```

Your completed code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    // Get the integer entered by the user
    int i;
    if (!int.TryParse(inputTextBox.Text, out i))
    {
        MessageBox.Show("TextBox does not contain an integer");
        return;
    }

    // Check that the user has not entered a negative number
    if (i < 0)
    {
        MessageBox.Show("Please enter a positive number or zero");
        return;
    }

    // Remainder will hold the remainder after dividing i by 2
    // after each iteration of the algorithm
    int remainder = 0;

    // Binary will be used to construct the string of bits
    // that represent i as a binary value
    StringBuilder binary = new StringBuilder();

    // Generate the binary representation of i
    do
    {
        remainder = i % 2;
        i = i / 2;
        binary.Insert(0, remainder);
    }
    while (i > 0);

    // Display the result
    binaryLabel.Content = binary.ToString();
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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

   a. On the **Debug** menu, click **Start Debugging**.

   b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.

   c. Verify that the result matches the text in the **Expected result** column.

   d. Repeat steps b and c for each row in the table.

2. Close the application and return to Visual Studio:

   - On the **Debug** menu, click **Stop Debugging**.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Multiplying Matrices

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:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the

      E:\Labfiles\Lab 2\Ex3\Starter folder.

   c. Click **MatrixMultiplication.sln**, and then click **Open**.

2. Examine the user interface that the MainWindow window defines:

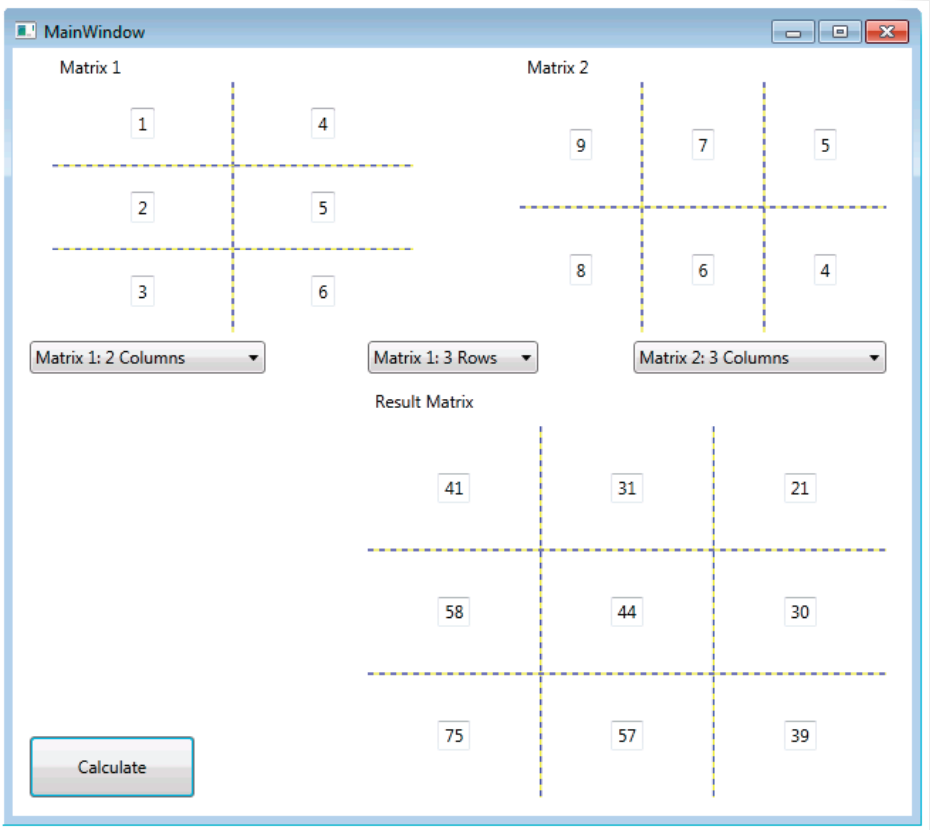
   - In Solution Explorer, double-click **MainWindow.xaml**.

   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.



Task 2: Define the matrix arrays and populate them with the data in the Grid controls

1. In Visual Studio, review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
public partial class MainWindow : Window
{
    // Declare three arrays of doubles to hold the 3 matrices:
    // The two input matrices and the result matrix
    double[,] matrix1;
    double[,] matrix2;
    double[,] result;
    ...
}
```

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**.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    // Retrieve the contents of the first two grids
    // into the first two matrices
    getValuesFromGrid(grid1, matrix1);
    getValuesFromGrid(grid2, 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

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Discover the dimensions of the input matrices
    // (Remember that the number of columns in the first matrix will
    // always be the same as the number of rows in the second matrix)
    int m1columns_m2rows = matrix1.GetLength(0);
    int m1rows = matrix1.GetLength(1);
    int m2columns = matrix2.GetLength(0);
    ...
}
```

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.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
    }
    ...
}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...

    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
        }
    }
    ...
}
```

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.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;
        }
    }
    ...
}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;
            // Iterate over the columns in the row in matrix1
            for (int cell = 0; cell < m1columns_m2rows; cell++)
            { }
        }
    }
    ...
}
```

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.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;

            // Iterate over the columns in the row in matrix1
            for (int cell = 0; cell < m1columns_m2rows; cell++)
            {
                // Multiply the value in the current column in the
                // current row in matrix1 with the value in the
                // current row in the current column in matrix2 and
                // add the result to accumulator
                accumulator +=
                    matrix1[cell, row] * matrix2[column, cell];
            }
        }
    }
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...

    // Calculate the value for each cell in the result matrix

    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;

            // Iterate over the columns in the row in matrix1

            for (int cell = 0; cell < m1columns_m2rows; cell++)
            {
                // Multiply the value in the current column in the
                // current row in matrix1 with the value in the
                // current row in the current column in matrix2 and
                // add the result to accumulator
                accumulator +=
                    matrix1[cell, row] * matrix2[column, cell];
            }

            result[column, row] = accumulator;
        }
    }
    ...
}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the results of your calculation in the third Grid
    initializeGrid(grid3, result);
}
```

2. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
3. Run the application in Debug mode:
   - On the **Debug** menu, click **Start Debugging**.
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.

   a. In the first **ComboBox** control, select **Matrix 1: 3 Columns**.
   b. In the second **ComboBox** control, select **Matrix 1: 2 Rows**.
   c. In the third **ComboBox** control, select **Matrix 2: 3 Columns**.
5. Specify the values for the cells in the matrices as shown in the following tables.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

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

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.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
    - On the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Declaring and Calling Methods

Lab 3: Declaring and Calling Methods

Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid's Algorithm

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 Microsoft® Visual Studio® 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 3\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, click **Add**.
   c. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 3\Snippets** folder, and then click **Select Folder**.
   d. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Euclid solution in the E:\Labfiles\Lab 3\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 3\Ex1\Starter** folder, click **Euclid.sln**, and then click **Open**.

Task 2: Implement Euclid's algorithm

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

   c. In the Task List window, double-click **TODO Exercise 1, Task 2**.

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.

   Your code should resemble the following code example.

```
...
static class GCDAlgorithms
{
    public static int FindGCDEuclid(int a, int b)
    {

    }
}
...
```

4. In the **FindGCDEuclid** method, add code that calculates and returns the greatest common divisor (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*.

   Your code should resemble the following code example.

```
...
static class GCDAlgorithms
{
    public static int FindGCDEuclid(int a, int b)
    {
        if (a == 0) return b;
        while (b != 0)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        {

            if (a > b)
            {
                a = a - b;
            }

            else
            {
                b = b - a;
            }

        }

        return 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 Windows® Presentation Foundation (WPF) window that you will use to test the **FindGCDEuclid** method and display the results:

   - In the Task List window, double-click **TODO Exercise 1, Task 3**.

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.

```
Euclid: result
```



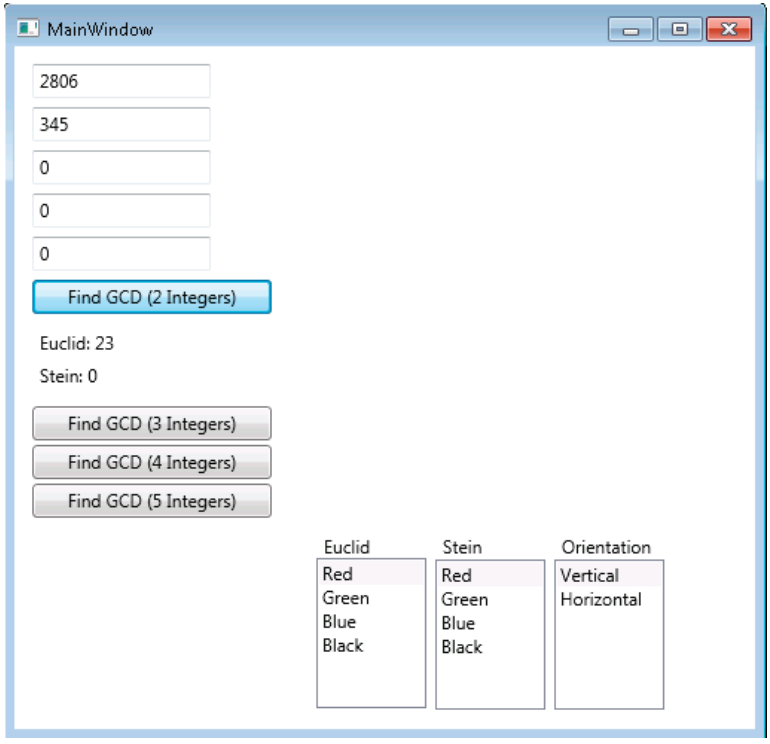
Hint: Set the **Content** property of a label control to display data in a label. Use the **String.Format** method to create a formatted string.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
if (sender == findGCD) // Euclid for two integers
{
    // Invoke the FindGCD method and display the result
    this.resultEuclid.Content =
        String.Format("Euclid: {0}",
        GCDAlgorithms.FindGCDEuclid(firstNumber, secondNumber));
}
...
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.
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.



MCT USE ONLY. STUDENT USE PROHIBITED

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	Result
0	0	0
0	10	10
25	10	5
25	100	25
26	100	2
27	100	1

8. Close the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Stop Debugging**.

Task 4: Create a unit test for the FindGCDEuclid method

1. Open the GCDAlgorithms.cs file:
   - In Solution Explorer, double-click **GCDAlgorithms.cs**.
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:
   a. In the **GCDAlgorithms** class, right-click the **FindGCDEuclid** method, and then click **Create Unit Tests**.
   b. In the **Create Unit Tests** dialog box, ensure that the **FindGCDEuclid(System.Int32, System.Int32)** check box is selected, ensure that the **Output project** is set to **Create a new Visual C# test project**, and then click **OK**.
   c. In the **New Test Project** dialog box, in the **Enter a name for your new project**, type **GCD Test Project** and then click **Create**.
   d. If the **You have made changes to your tests** dialog box is displayed, click **OK**.

MCT USE ONLY. STUDENT USE PROHIBITED

e. In the **Add InternalsVisibleTo Attribute** dialog box, click **Yes**.

3. In the GCD Test Project project, in the GCDAlgorithmsTest.cs file, locate the **FindGCDEuclidTest** method:
   - In the Code Editor window, in the **GCDAlgorithmsTest** class, 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.

   Your code should resemble the following code example.

```
...

[TestMethod()]

public void FindGCDEuclidTest()

{
    int a = 2806; // TODO: Initialize to an appropriate value
    int b = 345; // TODO: Initialize to an appropriate value
    int expected = 23; // TODO: Initialize to an appropriate value
    int actual;
    actual = GCDAlgorithms.FindGCDEuclid(a, b);
    Assert.AreEqual(expected, actual);
}

...
```

5. Open the Test View window and refresh the display if the unit test is not listed:
   a. On the **Test** menu, point to **Windows**, and then click **Test View**.
   b. If the **You have made changes to your tests** dialog box appears, click **OK**.
   c. In the Test View window, click the **Refresh** button.
6. Run the FindGCDEuclidTest test and verify that the test ran successfully:
   a. In the Test View window, right-click **FindGCDEuclidTest**, and then click **Run Selection**.
   b. In the Test Results window, verify that the **FindGCDEuclidTest** test passed.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Calculating the GCD of Three, Four, or Five Integers

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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 3\Ex2\Starter** folder, click **Euclid.sln**, and then click **Open**.

Task 2: Add overloaded methods to the GCDAlgorithms class

1. In Visual Studio, review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. Use the Task List window to navigate to the **TODO Exercise 2, Task 2** task:

   - In the Task List window, double-click **TODO Exercise 2, Task 2**.

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.

   Your code should resemble the following code example.

```
...
static class GCDAlgorithms
{
    ...
    public static int FindGCDEuclid(int a, int b, int c)
    {

    }
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
...
public static int FindGCDEuclid(int a, int b, int c)
{
    int d = FindGCDEuclid(a, b);
}
...
```

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.

   Your code should resemble the following code example.

```
...
public static int FindGCDEuclid(int a, int b, int c)
{
    int d = FindGCDEuclid(a, b);
    int e = FindGCDEuclid(d, c);
}
...
```

6. Add code to return the parameter e from the **FindGCDEuclid** method.

   Your code should resemble the following code example.

```
...
public static int FindGCDEuclid(int a, int b, int c)
{
    int d = FindGCDEuclid(a, b);
    int e = FindGCDEuclid(d, c);

    return e;

}
...
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
public static int FindGCDEuclid(int a, int b, int c, int d)
{
    int e = FindGCDEuclid(a, b, c);
    int f = FindGCDEuclid(e, d);
    return f;
}
...
```

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.

   Your code should resemble the following code example.

```
...
public static int FindGCDEuclid(int a, int b, int c, int d, int e)
{
    int f = FindGCDEuclid(a, b, c, d);
    int g = FindGCDEuclid(f, e);
    return g;
}
...
```

   At the end of this task, the **GCDAlgorithms** class should resemble the following code example.

```
...
static class GCDAlgorithms
{
    ...
    // TODO Exercise 2, Task 2
    // Add overloaded methods for 3,4, and 5 integers
    public static int FindGCDEuclid(int a, int b, int c)
    {
        int d = FindGCDEuclid(a, b);
        int e = FindGCDEuclid(d, c);
        return e;
    }
    public static int FindGCDEuclid(int a, int b, int c, int d)
    {
        int e = FindGCDEuclid(a, b, c);
        int f = FindGCDEuclid(e, d);
        return f;
    }
    public static int FindGCDEuclid(int a, int b, int c, int d, int e)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        int f = FindGCDEuclid(a, b, c, d);
        int g = FindGCDEuclid(f, e);
        return g;
    }
}
...
```

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:

   - In the Task List window, double-click **TODO Exercise 2, Task 3**.

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
```

   Your code should resemble the following code example.

```
...
else if (sender == findGCD3) // Euclid for three integers
{
    this.resultEuclid.Content =
        String.Format("Euclid: {0}",
        GCDAlgorithms.FindGCDEuclid(
            firstNumber,
            secondNumber,
            thirdNumber));
    this.resultStein.Content = "N/A";
}
...
```

3. Locate the **else if (sender == findGCD3)** block, the **else if (sender == findGCD4)** block, and the **else if (sender == findGCD5)** block, and modify

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
...
private void FindGCD_Click(object sender, RoutedEventArgs e)
{
    ...

    // TODO Exercise 2, Task 3
    // Call the overloaded methods for 3, 4 and 5 integers
    else if (sender == findGCD3) // Euclid for three integers
    {
        this.resultEuclid.Content =
            String.Format("Euclid: {0}",
            GCDAlgorithms.FindGCDEuclid(
            firstNumber, secondNumber, thirdNumber));
        this.resultStein.Content = "N/A";
    }
    else if (sender == findGCD4) // Euclid for four integers
    {
        this.resultEuclid.Content =
        String.Format("Euclid: {0}",
            GCDAlgorithms.FindGCDEuclid(
            firstNumber, secondNumber, thirdNumber, fourthNumber));
        this.resultStein.Content = "N/A";
    }
    else if (sender == findGCD5) // Euclid for five integers
    {
        this.resultEuclid.Content =
            String.Format("Euclid: {0}",
                GCDAlgorithms.FindGCDEuclid(firstNumber, secondNumber,
                thirdNumber, fourthNumber, fifthNumber));
        this.resultStein.Content = "N/A";
    }
}
...
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
5. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   - On the **Debug** menu, click **Stop Debugging**.

Task 4: Create unit tests for the overloaded methods

1. In Visual Studio, review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 4** task:
   - In the Task List window, double-click **TODO Exercise 2, Task 4**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Remove the **TODO Exercise 2, Task 4** comment and add a test method called **FindGCDEuclidTest1**.

   Your code should resemble the following code example.

```
...

[TestMethod()]
public void 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.

   Your code should resemble the following code example.

```
[TestMethod()]
public void FindGCDEuclidTest1()
{
    int a = 7396;
    int b = 1978;
    int c = 1204;
    int expected = 86;
}
```

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.

   Your code should resemble the following code example.

```
[TestMethod()]
public void FindGCDEuclidTest1()
{
    int a = 7396;
    int b = 1978;
    int c = 1204;
    int expected = 86;
    int actual = GCDAlgorithms.FindGCDEuclid(a, b, c);
}
```

6. Call the **AreEqual static** method of the **Assert** class, and pass the expected and actual variables as arguments.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
[TestMethod()]

public void FindGCDEuclidTest1()
{
    int a = 7396;
    int b = 1978;
    int c = 1204;
    int expected = 86;
    int actual = GCDAlgorithms.FindGCDEuclid(a, b, c);
    Assert.AreEqual(expected, actual);
}
```

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.

   Your code should resemble the following code example.

```
...
[TestClass()]
public class GCDAlgorithmsTest
{
    ...

    // Add unit tests for the new methods

    [TestMethod()]
    public void FindGCDEuclidTest1()
    {
        int a = 7396;
        int b = 1978;
        int c = 1204;
        int expected = 86;
        int actual = GCDAlgorithms.FindGCDEuclid(a, b, c);
        Assert.AreEqual(expected, actual);
    }

    [TestMethod()]
    public void FindGCDEuclidTest2()
    {
        int a = 7396;
        int b = 1978;
        int c = 1204;
        int d = 430;
        int expected = 86;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        int actual = GCDAlgorithms.FindGCDEuclid(a, b, c, d);
        Assert.AreEqual(expected, actual);
    }

    [TestMethod()]
    public void FindGCDEuclidTest3()
    {
        int a = 7396;
        int b = 1978;
        int c = 1204;
        int d = 430;
        int e = 258;
        int expected = 86;
        int actual = GCDAlgorithms.FindGCDEuclid(a, b, c, d, e);
        Assert.AreEqual(expected, actual);
    }
}
...
```

8. Open the Test View window and refresh the display if the unit test is not listed:
   a. On the **Test** menu, point to **Windows**, and then click **Test View**.
   b. In the Test View window, click the **Refresh** button.
9. Run the FindGCDEuclidTest, FindGCDEuclidTest1, FindGCDEuclidTest2, and FindGCDEuclidTest3 tests and verify that the tests ran successfully:
   a. In the Test View window, select the FindGCDEuclidTest, FindGCDEuclidTest1, FindGCDEuclidTest2, and FindGCDEuclidTest3 tests, right-click, and then click **Run Selection**.
   b. In the Test Results window, verify that the tests passed.

Exercise 3: Comparing the Efficiency of Two 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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 3\Ex3\Starter** folder, click **Stein.sln**, and then click **Open**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Implement Stein's algorithm

1. Open the GCDAlgorithms.cs file:
   - In Solution Explorer, double-click **GCDAlgorithms.cs**.
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.

   Your code should resemble the following code example.

```
...
static class GCDAlgorithms
{
    ...
    public static int FindGCDStein(int u, int v)
    {

    }
}
...
```

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 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;

// 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;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            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:
   - In Solution Explorer, double-click **MainWindow.xaml.cs**.
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 **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
```

Your code should resemble the following code example.

```
if (sender == findGCD) // Euclid for two integers
{
    // Do the calculations

    this.resultEuclid.Content =
    String.Format("Euclid: {0}",
        GCDAlgorithms.FindGCDEuclid(firstNumber, secondNumber));
    // Call the method that implements Stein's algorithm and display
    // the results

    this.resultStein.Content =
    String.Format("Stein: {0}",
       GCDAlgorithms.FindGCDStein(firstNumber, secondNumber));
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.
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:
   - On the **Debug** menu, click **Stop Debugging**.
7. Open the GCDAlgorithmsTest.cs file:
   - In Solution Explorer, double-click **GCDAlgorithmsTest.cs**.
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**.

   Your code should resemble the following code example.

```
...
[TestMethod()]
public void FindGCDSteinTest()
{

}
...
```

9. In the **FindGCDSteinTest** method, declare three variables called u, v, and expected, and assign them values **298467352**, **569484**, and **4** respectively.

   Your code should resemble the following code example.

```
[TestMethod()]
public void FindGCDSteinTest()
{
    int u = 298467352;
    int v = 569484;
    int expected = 4;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

    Your code should resemble the following code example.

```
[TestMethod()]
public void FindGCDSteinTest()
{
    int u = 298467352;
    int v = 569484;
    int expected = 4;
    int actual = GCDAlgorithms.FindGCDStein(u, v);
}
```

11. Call the **static AreEqual** method of the **Assert** class, and pass the expected and actual variables as arguments.

    Your code should resemble the following code example.

```
[TestMethod()]
public void FindGCDSteinTest()
{
    int u = 298467352;
    int v = 569484;
    int expected = 4;
    int actual = GCDAlgorithms.FindGCDStein(u, v);
    Assert.AreEqual(expected, actual);
}
```

12. Open the Test View window and refresh the display if the unit test is not listed:

    a. On the **Test** menu, point to **Windows**, and then click **Test View**.

    b. In the Test View window, click the **Refresh** button.

13. Run the FindGCDSteinTest test, and verify that the test ran successfully:

    a. In the Test View window, right-click the **FindGCDSteinTest** test, and then click **Run Selection**.

    b. In the Test Results window, verify that the test passed.

Task 4: Add code to test the performance of the algorithms

1. Open the GCDAlgorithms.cs file:

    - In Solution Explorer, double-click **GCDAlgorithms.cs**.

MCT USE ONLY. STUDENT USE PROHIBITED

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**.

   Your code should resemble the following code example.

```
...
static public int FindGCDEuclid(int a, int b, out long time)
{
}
...
```

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.

   Your code should resemble the following code example.

```
...

static public int FindGCDEuclid(int a, int b, out long time)
{
    time = 0;
    Stopwatch sw = new Stopwatch();
    sw.Start();
    ...
}

...
```

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.

   Your code should resemble the following code example.

```
...

static public int FindGCDEuclid(int a, int b, out long time)
{
    time = 0;
    Stopwatch sw = new Stopwatch();
    sw.Start();
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    ...
    sw.Stop();
    time = sw.ElapsedTicks;
    return a;
}
...
```

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.

   Your code should resemble the following code example.

```
...
static public int FindGCDStein(int u, int v, out long time)
{
    time = 0;
    Stopwatch sw = new Stopwatch();
    sw.Start();
    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)
    {
        sw.Stop();
        time = sw.ElapsedTicks;
        return u | v;
    }
    ...
    sw.Stop();
    time = sw.ElapsedTicks;
    return u;
}
...
```

7. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, double-click **MainWindow.xaml.cs**.

MCT USE ONLY. STUDENT USE PROHIBITED

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
```

   Your code should resemble the following code example.

```
...
if (sender == findGCD) // Euclid and Stein for two integers and graph
{
    long timeEuclid;
    long timeStein;

    // Do the calculations
    this.resultEuclid.Content =
        String.Format("Euclid: {0}, Time (ticks): {1}",
        GCDAlgorithms.FindGCDEuclid(firstNumber, secondNumber,
        out timeEuclid), timeEuclid);
    this.resultStein.Content = String.Format("Stein: {0}, Time
        (ticks): {1}", GCDAlgorithms.FindGCDStein(firstNumber,
        secondNumber, out timeStein), timeStein);
}
```

9. Comment out the code that calls the overloaded versions of the **FindGCDEuclid** method.

   Your code should resemble the following code example.

```
...
else if (sender == findGCD3) // Euclid for three integers
{
//    this.resultEuclid.Content = String.Format("Euclid: {0}",
//        GCDAlgorithms.FindGCDEuclid(firstNumber,
//        secondNumber, thirdNumber));
    this.resultStein.Content = "N/A";
}
else if (sender == findGCD4) // Euclid for four integers
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
{
//    this.resultEuclid.Content = String.Format("Euclid: {0}",
//        GCDAlgorithms.FindGCDEuclid(firstNumber,
//        secondNumber, thirdNumber, fourthNumber));
    this.resultStein.Content = "N/A";
}

else if (sender == findGCD5) // Euclid for five integers
{

//    this.resultEuclid.Content = String.Format("Euclid: {0}",
//        GCDAlgorithms.FindGCDEuclid(firstNumber,
//        secondNumber, thirdNumber, fourthNumber, fifthNumber));
    this.resultStein.Content = "N/A";
}

...
```

10. Open the GCDAlgorithmsTest.cs file:
    - In Solution Explorer, double-click **GCDAlgorithmsTest.cs**.
11. Modify the **FindGCDEuclidTest** and **FindGCDSteinTest** methods to use the new method signatures. Comment out the methods **FindGCDEuclidTest1**, **FindGCDEuclidTest2**, and **FindGCDEuclidTest3**.

    Your code should resemble the following code example.

```
[TestClass()]
public class GCDAlgorithmsTest
{
    ...

    [TestMethod()]
    public void FindGCDEuclidTest()
    {

        int a = 298467352;
        int b = 569484;
        long time;
        int expected = 4;
        int actual;
        actual = GCDAlgorithms.FindGCDEuclid(a, b, out time);
        Assert.AreEqual(expected, actual);
    }

    ...
    [TestMethod()]
    public void FindGCDSteinTest()
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        int u = 298467352;
        int v = 569484;
        long time;
        int expected = 4;
        int actual = GCDAlgorithms.FindGCDStein(u, v, out time);
        Assert.AreEqual(expected, actual);
    }

}
```

12. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.
13. Run the GreatestCommonDivisor application:
    - On the **Debug** menu, click Start **Debugging**.
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:
    - On the **Debug** menu, click **Stop Debugging**.
16. Open the Test View window and refresh the display if the unit test is not listed:
    a. On the **Test** menu, point to **Windows**, and then click **Test View**.
    b. In the Test View window, click the **Refresh** button.
17. Run the **FindGCDEuclidTest** and **FindGCDSteinTest** methods and verify that the tests ran successfully:
    a. In the Test View window, select the **FindGCDEuclidTest** and **FindGCDSteinTest** tests, right-click, and then click **Run Selection**.
    b. In the Test Results window, verify that the tests passed.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 4: Displaying Results Graphically

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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 3\Ex4\Starter** folder, click **Charting.sln**, and then click **Open**.

Task 2: Display the algorithm timings graphically

1. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, double-click **MainWindow.xaml.cs**.
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.

   Your code should resemble the following code example.

```
...
if (sender == findGCD) // Euclid and Stein for two integers and graph
{
    ...
    DrawGraph(timeEuclid, timeStein);
}
...
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.
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:
   - On the **Debug** menu, click **Stop Debugging**.

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**".

   Your code should resemble the following code example.

```
...

private void DrawGraph(long euclidTime, long steinTime,
    Orientation orientation = Orientation.Horizontal,
    string colorEuclid = "Red",
    string colorStein = "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")
{
    ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    BrushConverter bc = new BrushConverter();
    Brush bEuclid = (Brush)bc.ConvertFromString(colorEuclid);
    Brush bStein = (Brush)bc.ConvertFromString(colorStein);
    ...
}
...
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
5. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.
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:
   - On the **Debug** menu, click **Stop Debugging**.

Task 4: Modify the code that calls the DrawGraph method

1. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, double-click **MainWindow.xaml.cs**.
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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
if (sender == findGCD) // Euclid and Stein for two integers and graph
{
    ...

    // Get the preferred colors and orientation
    string selectedEuclidColor =
        ((ListBoxItem)this.euclidColor.SelectedItem).
        Content.ToString();
    string selectedSteinColor =
        ((ListBoxItem)this.steinColor.SelectedItem).
        Content.ToString();

    Orientation orientation;

    if (this.chartOrientation.SelectedIndex == 0)
    {
        orientation = Orientation.Vertical;
    }
    else
    {
        orientation = Orientation.Horizontal;
    }

    DrawGraph(timeEuclid, timeStein, orientation: orientation,
        colorStein: selectedSteinColor,
        colorEuclid: selectedEuclidColor);
}
...
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the GreatestCommonDivisor application:
   - On the **Debug** menu, click **Start Debugging**.
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:
   - On the **Debug** menu, click **Stop Debugging**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 5: Solving Simultaneous Equations (optional)

Task 1: Open the starter project

1. Open the SimultaneousEquations solution in the E:\Labfiles\Lab 3\Ex5 \Starter folder:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 3\Ex5 \Starter** folder, click **SimultaneousEquations.sln**, and then click **Open**.

2. Open the MainWindow.xaml file:

   - In Solution Explorer, expand the **GaussianElimination** project, and then double-click **MainWindow.xaml**.

   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:

   - In Solution Explorer, double-click **Gauss.cs**.

   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.

Your code should resemble the following code example.

```
...
private static double[] DeepCopy1D(double[] array)
{

}
...
```

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.

   Your code should resemble the following code example.

```
...
private static double[] DeepCopy1D(double[] array)
{
   // Get dimensions
    int columns = array.GetLength(0);

    // Initialize a new array
    double[] newArray = new double[columns];

    // Copy the values
    for (int i = 0; i < columns; i++)
    {
         newArray[i] = array[i];
    }
    return newArray;
}
...
```

4. In the **Gauss** class, declare another **private static** method called **DeepCopy2D**. The method should accept and return a two-dimensional **double** array.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
private static double[,] DeepCopy2D(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.

   Your code should resemble the following code example.

```
...

private static double[,] DeepCopy2D(double[,] array)
{
    // Get dimensions
    int columns = array.GetLength(0);
    int rows = array.GetLength(1);

    // Initialize a new array
    double[,] newArray = new double[columns, rows];

    // Copy the values
    for (int i = 0; i < columns; i++)
    {
        for (int j = 0; j < rows; j++)
        {
             newArray[i,j] = array[i,j];
        }
    }
    return newArray;
}

...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
...
public static double[] SolveGaussian(double[,] coefficients,
    double[] rhs)
{
    // TODO Exercise 5, Task 3
    // Make deep copies of the coefficients and rhs arrays
    double[,] a = DeepCopy2D(coefficients);
    double[] b = DeepCopy1D(rhs);

    ...
}
...
```

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.

Your code should resemble the following code example.

```
...

// TODO Exercise 5, Task 3
// Convert the equations to triangular form

double x, sum;

for (int k = 0; k < numberOfEquations - 1; k++)
{

    try
    {

        for (int i = k + 1; i < numberOfEquations; i++)
        {

            x = a[i, k] / a[k, k];
            for (int j = k + 1; j < numberOfEquations; j++)
```

```
                a[i, j] = a[i, j] - a[k, j] * x;

            b[i] = b[i] - b[k] * x;
        }
    }

    catch (DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
    }

}
...
```

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.

  Your code should resemble the following code example.

```
...

// TODO Exercise 5, Task 4
// Perform the back substitution and return the result

b[numberOfEquations - 1] = b[numberOfEquations - 1] /
a[numberOfEquations - 1, numberOfEquations - 1];

for (int i = numberOfEquations - 2; i >= 0; i--)
{
    sum = b[i];
    for (int j = i + 1; j < numberOfEquations; j++)
        sum = sum - a[i, j] * b[j];
    b[i] = sum / a[i, i];
}

return b;

...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 5: Test the solution

1. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, double-click **MainWindow.xaml.cs**.
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.

   Your code should resemble the following code example.

```
...
private void CmdSolve_Click(object sender, RoutedEventArgs e)
{
    ...
    // TODO Exercise 5, Step 5
    // Invoke the solveGaussian method
    answers = Gauss.SolveGaussian(coefficients, rhs);
    ...
} ...
```

3. Run the GaussianElimination application:
   - On the **Debug** menu, click **Start Debugging**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Handling Exceptions

Lab 4: Handling Exceptions

Exercise 1: Making a Method Fail-Safe

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 Microsoft® Visual Studio® 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Failsafe solution in the E:\Labfiles\Lab 4\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 4\Ex1\Starter** folder, click Failsafe.sln, and then click Open.
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.

   a. On the **Debug** menu, click **Start Debugging**.
   b. In the MainWindow window, click **Shutdown**, and then examine the unhandled exception message that appears in Visual Studio.
   c. In Visual Studio, on the **Debug** menu, click **Stop Debugging**.

Task 2: Examine the Switch class

1. If it is not already open, open the Switch.cs file in Visual Studio:

a. In Solution Explorer, in the Failsafe solution, expand the **SwitchDevice** project.

b. Right-click **Switch.cs**, and then click **View Code**.

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 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:

   a. In Solution Explorer, expand the **SwitchTestHarness** project.

   b. Expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

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.

   Your code should resemble the following code example.

```
...
// Step 1 - disconnect from the Power Generator
try
{

    if (sd.DisconnectPowerGenerator() ==
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        SwitchDevices.SuccessFailureResult.Fail)
    {
        this.textBlock1.Text +=
            "\nStep 1: Failed to disconnect power generation system";
    }
    else
    {
        this.textBlock1.Text +=
        "\nStep 1: Successfully disconnected power generation system";
    }
}
catch (SwitchDevices.PowerGeneratorCommsException ex)
{

}

// Step 2 - Verify the status of the Primary Coolant System
...
```

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.

Your code should resemble the following code example.

```
...
catch (SwitchDevices.PowerGeneratorCommsException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 1: " + ex.Message;
}
...
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

new line in the **textBlock1** control (note that this is step 2, not step 1 of the shutdown process).

Your code should resemble the following code example.

```
...
// Step 2 - Verify the status of the Primary Coolant System
try
{
    switch (sd.VerifyPrimaryCoolantSystem())
    {
        case SwitchDevices.CoolantSystemStatus.OK:
            this.textBlock1.Text +=
                "\nStep 2: Primary coolant system OK";
            break;
        case SwitchDevices.CoolantSystemStatus.Check:
            this.textBlock1.Text +=
                "\nStep 2: Primary coolant system requires manual
check";
            break;
        case SwitchDevices.CoolantSystemStatus.Fail:
            this.textBlock1.Text +=
                "\nStep 2: Problem reported with primary coolant
system";
            break;
    }
}
catch (SwitchDevices.CoolantPressureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 2: " + ex.Message;
}
catch (SwitchDevices.CoolantTemperatureReadException ex)
{
    this.textBlock1.Text
        += "\n*** Exception in step 2: " + ex.Message;
}

// Step 3 - Verify the status of the Backup Coolant System
...
```

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 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).

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
...
// Step 3 - Verify the status of the Backup Coolant System
try
{
    switch (sd.VerifyBackupCoolantSystem())
    {
        case SwitchDevices.CoolantSystemStatus.OK:
            this.textBlock1.Text +=
                "\nStep 3: Backup coolant system OK";
            break;
        case SwitchDevices.CoolantSystemStatus.Check:
            this.textBlock1.Text +=
                "\nStep 3: Backup coolant system requires manual
check";
            break;
        case SwitchDevices.CoolantSystemStatus.Fail:
            this.textBlock1.Text +=
                "\nStep 3: Backup reported with primary coolant
system";
            break;
    }
}
catch (SwitchDevices.CoolantPressureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 3: " + ex.Message;
}
catch (SwitchDevices.CoolantTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 3: " + ex.Message;
}

// Step 4 - Record the core temperature prior to shutting down the
reactor

...
```

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).

   Your code should resemble the following code example.

```
...
// Step 4 - Record the core temperature prior to shutting down the
reactor

try
{

    this.textBlock1.Text +=
        "\nStep 4: Core temperature before shutdown: " +
        sd.GetCoreTemperature();
}

catch (SwitchDevices.CoreTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 4: " + ex.Message;
}

// Step 5 - Insert the control rods into the reactor
...
```

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).

   Your code should resemble the following code example.

```
...
// Step 5 - Insert the control rods into the reactor

try
{
    if (sd.InsertRodCluster() ==
        SwitchDevices.SuccessFailureResult.Success)
    {
        this.textBlock1.Text +=
            "\nStep 5: Control rods successfully inserted";
    }
    else
    {
        this.textBlock1.Text +=
            "\nStep 5: Control rod insertion failed";
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
catch (SwitchDevices.RodClusterReleaseException ex)
{
    this.textBlock1.Text
        += "\n*** Exception in step 5: " + ex.Message;
}
// Step 6 - Record the core temperature after shutting down the
reactor
...
```

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).

   Your code should resemble the following code example.

```
...
// Step 6 - Record the core temperature after shutting down the
reactor

try
{
    this.textBlock1.Text +=
        "\nStep 6: Core temperature after shutdown: " +
        sd.GetCoreTemperature();
}
catch (SwitchDevices.CoreTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 6: " + ex.Message;
}

// Step 7 - Record the core radiation levels after shutting down the
reactor
...
```

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).

    Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
// Step 7 - Record the core radiation levels after shutting down the
reactor
try
{
    this.textBlock1.Text +=
        "\nStep 7: Core radiation level after shutdown: " +
        sd.GetRadiationLevel();
}
catch (SwitchDevices.CoreRadiationLevelReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 7: " + ex.Message;
}

// Step 8 - Broadcast "Shutdown Complete" message
...
```

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).

    Your code should resemble the following code example.

```
...
// Step 8 - Broadcast "Shutdown Complete" message
try
{
    sd.SignalShutdownComplete();
    this.textBlock1.Text +=
        "\nStep 8: Broadcasting shutdown complete message";
}
catch (SwitchDevices.SignallingException ex)
{
    this.textBlock1.Text +=
    "\n*** Exception in step 8: " + ex.Message;
}

this.textBlock1.Text +=
    "\nTexst sequence complete: " +
    DateTime.Now.ToLongTimeString();
...
```

12. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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.

a. On the **Debug** menu, click **Start Debugging**.

b. In the MainWindow window, click **Shutdown**.

c. Read through the messages that are displayed in the window, and verify that, where an exception occurred, a message appears that states "*** Exception in step *x* :*message*", where *x* is a step number, and *message* is an exception message.

d. Close the application and return to Visual Studio.

Exercise 2: Detecting an Exceptional Condition

Task 1: Open the MatrixMultiplication solution

1. In Visual Studio, open the MatrixMultiplication solution in the E:\Labfiles\Lab 4\Ex2\Starter folder:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the E:\Labfiles\Lab 4\Ex2\Starter folder, click **MatrixMultiplication.sln**, and then click **Open**.

2. Open the Matrix.cs file, and then locate the **MatrixMultiply** method:

   a. In Solution Explorer, in the MatrixMultiplication project, right-click **Matrix.cs**, and then click **View Code**.

   b. Examine 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.

Your code should resemble the following code example.

```
...
// Check the matrices are compatible
if (matrix1.GetLength(0) != matrix2.GetLength(1))
    throw new ArgumentException(
        "The number of columns in the first matrix must be the same as
the number of rows in the second matrix");

// Get the dimensions
...
```

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:

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
...
// Throw exceptions if either matrix contains a negative entry
if (matrix1[cell, row] < 0d)
    throw new ArgumentException(String.Format(
        "Matrix1 contains an invalid entry in cell[{0}, {1}]",
        cell, row));

accumulator += matrix1[cell, row] * matrix2[column, cell];
...
```

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.

   Your code should resemble the following code example.

```
...
    "Matrix1 contains an invalid entry"
        );

if (matrix2[column, cell] < 0d)
    throw new ArgumentException(String.Format(
        "Matrix2 contains an invalid entry in cell[{0}, {1}].",
        column, cell));

accumulator += matrix1[cell, row] * matrix2[column, cell];
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 3: Handle the exceptions that the MatrixMultiply method throws

1. Open the MainWindow Windows® Presentation Foundation (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:

   - In Solution Explorer, in the MatrixMultiplication project, double-click **MainWindow.xaml**.

2. Open the code file for the MainWindow WPF window:

   - In Solution Explorer, in the MatrixMultiplication project, expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

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**.

   Your code should resemble the following code example.

```
...
// Do the multiplication - checking for exceptions
try
{
    result = Matrix.MatrixMultiply(matrix1, matrix2);
}
catch (ArgumentException ex)
{

}

// Show the results
...
```

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.

Your code should resemble the following code example.

```
...
catch (ArgumentException ex)
{
    MessageBox.Show(ex.Message);
}

...
```

6. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
7. Start the application without debugging:
   - On the **Debug** menu, click **Start 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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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:

   - In Solution Explorer, expand the **Solution Items** folder, expand **Matrix Unit Test Project**, expand **Test References**, and then double-click **MatrixText.cs**.

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.

   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:

   a. On the **Build** menu, click **Build Solution**.

   b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

   c. Wait for the tests to run, and then in the Test Results window, verify that all tests passed.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Checking for Numeric Overflow

Task 1: Open the IntegerOverflow solution

1. Open the IntegerOverflow solution in the E:\Labfiles\Lab 4\Ex3\Starter folder:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the

   E:\Labfiles\Lab 4\Ex3\Starter folder, click **IntegerOverflow.sln**, and then click **Open**.

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 situations, it is better to catch the overflow error than to let an application proceed with incorrect data:

   a. On the **Debug** menu, click **Start Debugging**.

   b. In the MainWindow window, click **Multiply**.

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:

   - In Solution Explorer, in the IntegerMultiplySolution solution, expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

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.

   Your code should resemble the following code example.

```
...
    return;
}

try
{
    labelAnswer.Content = (x * y).ToString();
}
catch (OverflowException ex)
{
    MessageBox.Show(ex.Message);
}
...
```

4. Inside the **try** block, add a **checked** block around the line of code that performs the multiplication arithmetic.

   Your code should resemble the following code example.

```
...
try
{
    checked
    {
        labelAnswer.Content = (x * y).ToString();
    }
}
catch (Exception ex)
...
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 3: Test the application

1. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
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

MCT USE ONLY. STUDENT USE PROHIBITED

Reading and Writing Files

Lab 5: Reading and Writing Files

Exercise 1: Building a Simple File Editor

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:
   - Click Start, point to All Programs, click Microsoft Visual Studio 2010, and then click Microsoft Visual Studio 2010.
3. Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder, click **SimpleEditor.sln**, and then click **Open**.

Task 2: Display a dialog box to accept a file name from the user

1. Display the MainWindow.xaml window:
   - In Solution Explorer, expand the **FileEditor** project, and then double-click **MainWindow.xaml**.

   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:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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.

   Your code should resemble the following code example.

```
...
private string GetFileName()
{

}
...
```

5. In the method body, declare a new string member named **fname**, and then initialize it with the **String.Empty** value.

   Your code should resemble the following code example.

```
...
private string GetFileName()
{
    string fname = String.Empty;
}
...
```

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.

   Your code should resemble the following code example.

```
...
using System.Windows.Shapes;
using Microsoft.Win32;
...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED



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**".

Your code should resemble the following code example.

```
...
    string fname = string.Empty;

    OpenFileDialog openFileDlg = new OpenFileDialog();

    openFileDlg.InitialDirectory =
        @"E:\Labfiles\Lab 5\Ex1\Starter";

    openFileDlg.DefaultExt = ".txt";
    openFileDlg.Filter = "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.

Your code should resemble the following code example.

```
...
    bool? result = openFileDlg.ShowDialog();

    if (result == true)

    {
        fname = openFileDlg.FileName;
    }
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

9. At the end of the method, return the value in the fname variable.

   Your code should resemble the following code example.

```
...
        fname = openFileDlg.FileName;
    }
    return fname;
}
...
```

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:

   a. In Solution Explorer, right-click the **FileEditor** project, point to **Add**, and then click **Class**.

   b. In the **Add New Item - FileEditor** dialog box, in the **Name** box, type **TextFileOperations.cs** and then click **Add**.

2. At the top of the class file, add a statement to bring the **System.IO** namespace into scope.

   Your code should resemble the following code example.

```
...

using System.Text;
using System.IO;

namespace FileEditor

{
...
}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
class TextFileOperations
{
    public static string ReadTextFileContents(string fileName)
    {

    }
}
...
```

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.

Your code should resemble the following code example.

```
...
public static string ReadTextFileContents(string fileName)
{
    return File.ReadAllText(fileName);
}
...
```

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*.

   Your code should resemble the following code example.

```
...
    return File.ReadAllText(fileName);
}

    public static void WriteTextFileContents
        (string fileName, string text)
    {
    }
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
...

public static void WriteTextFileContents
    (string filename, string text)

{
    File.WriteAllText(fileName, text);
}

...
```

7. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
   - Review the **Error** list and check for 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.
   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 Windows® Presentation Foundation (WPF) window.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
private void OpenButton_Click(object sender, RoutedEventArgs e)
{
    // Call GetFileName to get the name of the file to load
    fileName = GetFileName();

    // Populate the editor text box with the file contents
    if (fileName != String.Empty)
    {
        editor.Text =
            TextFileOperations.ReadTextFileContents(fileName);
    }
}
...
```

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.

   Your code should resemble the following code example.

```
...
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
    // Write the contents of the editor TextBox back to the file
    if (fileName != String.Empty)
    {
        TextFileOperations.WriteTextFileContents(fileName,
                                                 editor.Text);
    }
}
...
```

5. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**.

6. Start the application without debugging:

MCT USE ONLY. STUDENT USE PROHIBITED

- On the **Debug** menu, click **Start 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:
    - In Windows Explorer, right-click **Commands.txt**, point to **Open with**, and then click **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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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:

- Uncomment the fourth line of code, to enable the method to call the **FileEditor.TextFileOperations.ReadTextFileContents** method.

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:

   - Uncomment the third line of code, to enable the method to call the **FileEditor.TextFileOperations.WriteTextFileContents** method.

5. Run all tests in the solution, and verify that all tests execute correctly:

   a. On the **Build** menu, click **Build Solution**.

   b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

   c. Wait for the tests to run, and then in the Test Results window, verify that all tests passed.

Exercise 2: Making the Editor XML Aware

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:

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 5\Ex2\Starter** folder, click **SimpleEditor.sln**, and then click **Open**.

Task 2: Add a new method to filter XML characters to the TextFileOperations class

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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.

   Your code should resemble the following code example.

```
...

public static string ReadAndFilterTextFileContents(string fileName)
{

}

...
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
public static string ReadAndFilterTextFileContents(string fileName)
{
    StringBuilder fileContents = new StringBuilder();
    int charCode;
}
...
```

5. Add a statement that instantiates a **StreamReader** object, named **fileReader**, by using the *fileName* parameter.

   Your code should resemble the following code example.

```
...
{
    StringBuilder fileContents = new StringBuilder();
    int charCode;

    StreamReader fileReader = new StreamReader(fileName);
}
...
```

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.

Your code should resemble the following code example.

```
...
    StreamReader fileReader = new StreamReader(fileName);
    while ((charCode = fileReader.Read()) != -1)
    {

    }
...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

charCode	Standard representation	Alternative representation
34	" (straight quotation mark)	"
38	& (ampersand)	&
39	' (apostrophe)	'
60	< (less than)	<
62	> (greater than)	>

Your code should resemble the following code example.

```
...
while ((charCode = fileReader.Read()) != -1)
{
    switch (charCode)
    {
        case 34: // "
            fileContents.Append("&quot;");
            break;

        case 38: // &
            fileContents.Append("&amp;");
            break;

        case 39: // '
            fileContents.Append("&apos;");
            break;

        case 60: // <
            fileContents.Append("&lt;");
            break;

        case 62: // >
            fileContents.Append("&gt;");
            break;
    }
}
...
```

8. Add a default **case** statement that appends the actual character read from the stream to the **fileContent StringBuilder** object.

MCT USE ONLY. STUDENT USE PROHIBITED



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.

Your code should resemble the following code example.

```
...
        case 62: // >
            fileContents.Append("&gt;");
            break;

        default:
            fileContents.Append((char)charCode);
            break;
    }

}
...
```

9. At the end of the method, return the contents of the **fileContent StringBuilder** object as a string.

   Your code should resemble the following code example.

```
...

public static string ReadAndFilterTextFileContents(string fileName)
{
    ...
    return fileContents.ToString();
}
...
```

10. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
...
if (filename != string.Empty)
{
    // Call the new read file contents method
    editor.Text =
        TextFileOperations.ReadAndFilterTextFileContents(filename);
}
...
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
4. Start the application without debugging:
   - On the **Debug** menu, click **Start Without Debugging**.
5. In the MainWindow window, click **Open**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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:

   - Uncomment the fourth line of code, to enable the method to call the **FileEditor.TextFileOperations.ReadAndFilterTextFileContents** method.

3. Run all tests in the solution, and verify that all tests execute correctly:

   a. On the **Build** menu, click **Build Solution**.

   b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

   c. Wait for the tests to run, and then in the Test Results window, verify that all tests passed.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Creating New Types

Lab 6: Creating New Types

Exercise 1: Using Enumerations to Specify Domains

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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Enumerations solution in the E:\Labfiles\Lab 6\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 6\Ex1\Starter** folder, click **Enumerations.sln**, and then click **Open**.

Task 2: Add enumerations to the StressTest namespace

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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**

Your code should resemble the following code example.

```
...
namespace StressTest
{
    public enum Material
    {
        StainlessSteel,
        Aluminum,
        ReinforcedConcrete,
        Composite,
        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**

   Your code should resemble the following code example.

```
...
namespace StressTest
{
    ...
    public enum CrossSection
    {
        IBeam,
        Box,
        ZShaped,
        CShaped
    }
}
...
```

5. Below the **CrossSection** enumeration, define a new enumeration named **TestResult**. The enumeration should have the following values:

MCT USE ONLY. STUDENT USE PROHIBITED

a. **Pass**

b. **Fail**

Your code should resemble the following code example.

```
...
namespace StressTest
{
    ...
    public enum TestResult
    {
        Pass,
        Fail
    }
}
...
```

6. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Retrieve the enumeration values

1. In the TestHarness project, display the MainWindow.xaml window:
   - In Solution Explorer, expand the **TestHarness** project, and then double-click **MainWindow.xaml**.

   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.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
...
if (materials.SelectedIndex == -1 ||
    crosssections.SelectedIndex == -1 ||
    testresults.SelectedIndex == -1)
{
    return;
}

Material selectedMaterial = (Material)materials.SelectedItem;
CrossSection selectedCrossSection =
    (CrossSection)crosssections.SelectedItem;
TestResult selectedTestResult = (TestResult)testresults.SelectedItem;

...
```

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**.

   Your code should resemble the following code example.

```
...
TestResult selectedTestResult = (TestResult)testresults.SelectedItem;

StringBuilder selectionStringBuilder = new StringBuilder();
...
```

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 "<*selectedMaterial*>" in this string with the corresponding value for the selectedMaterial variable that is shown in the following table.

MCT USE ONLY. STUDENT USE PROHIBITED

Material enumeration value	<*selectedMaterial*> string
Material.StainlessSteel	Stainless Steel
Material.Aluminum	Aluminum
Material.ReinforcedConcrete	Reinforced Concrete
Material.Composite	Composite
Material.Titanium	Titanium

Your code should resemble the following code example.

```
...
switch (selectedMaterial)
{
    case Material.StainlessSteel:
        selectionStringBuilder.Append("Material: Stainless Steel, ");
        break;
    case Material.Aluminum:
        selectionStringBuilder.Append("Material: Aluminum, ");
        break;
    case Material.ReinforcedConcrete:
        selectionStringBuilder.Append
            ("Material: Reinforced Concrete, ");
        break;
    case Material.Composite:
        selectionStringBuilder.Append("Material: Composite, ");
        break;
    case Material.Titanium:
        selectionStringBuilder.Append("Material: Titanium, ");
        break;
}
...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

Material enumeration value	<*selectedCrossSection*> string
CrossSection.IBeam	I-Beam
CrossSection.Box	Box
CrossSection.ZShaped	Z-Shaped
CrossSection.CShaped	C-Shaped

Your code should resemble the following code example.

```
...
switch (selectedCrossSection)
{
    case CrossSection.IBeam:
        selectionStringBuilder.Append("Cross-section: I-Beam, ");
        break;
    case CrossSection.Box:
        selectionStringBuilder.Append("Cross-section: Box, ");
        break;
    case CrossSection.ZShaped:
        selectionStringBuilder.Append("Cross-section: Z-Shaped, ");
        break;
    case CrossSection.CShaped:
        selectionStringBuilder.Append("Cross-section: C-Shaped, ");
        break;
}
...
```

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
TestResult.Fail	Fail

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
...
switch (selectedTestResult)
{
    case TestResult.Pass:
        selectionStringBuilder.Append("Result: Pass.");
        break;
    case TestResult.Fail:
        selectionStringBuilder.Append("Result: Fail.");
        break;
}
...
```

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.

   Your code should resemble the following code example.

```
...
private void selectionChanged
    (object sender, SelectionChangedEventArgs e)
{
    ...
    testDetails.Content = selectionStringBuilder.ToString();
}
...
```

Task 5: Test the solution

1. Build the application and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Using a Struct to Model a Simple Type

Task 1: Open the Structures solution

- Open the Structures solution in the E:\Labfiles\Lab 6\Ex2\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 6\Ex2\Starter** folder, click **Structures.sln**, and then click **Open**.

Task 2: Add the TestCaseResult structure

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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**.

   Your code should resemble the following code example.

```
...
public struct TestCaseResult
{
    public TestResult Result;

    public string ReasonForFailure;
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 3: Add an array of TestCaseResult objects to the user interface project

1. In the TestHarness project, display the MainWindow.xaml window:
   - In Solution Explorer, expand the **TestHarness** project, and then double-click **MainWindow.xaml**.

   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**.

   Your code should resemble the following code example.

```
...

public partial class MainWindow : Window
{
    TestCaseResult[] results;

    public MainWindow()
...

}

...
```

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**.

   Your code should resemble the following code example.

```
...

private void RunTests_Click(object sender, RoutedEventArgs e)
{
    reasonsList.Items.Clear();
    results = new TestCaseResult[10];
```

```
    // Fill the array with 10 TestCaseResult objects.

    int passCount = 0;
    ...
}
...
```

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 **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.

   Your code should resemble the following code example.

```
...

    for (int i = 0; i < results.Length; i++)
    {
        results[i] = TestManager.GenerateResult();
    }

...
```

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.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
for (int i = 0; i < results.Length; i++)
{
    if (results[i].Result == TestResult.Pass)
        passCount++;
    else
    {
        failCount++;
        reasonsList.Items.Add(results[i].ReasonForFailure);
    }
}
...
```

Task 6: Test the solution

1. Build the application and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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

Task 1: Open the Classes solution

- Open the Classes solution in the E:\Labfiles\Lab 6\Ex3\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the

     E:\Labfiles\Lab 6\Ex3\Starter folder, click Classes.sln, and then click Open.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Define the StressTestCase class

1. In the TestHarness project, display the MainWindow.xaml window:
   - In Solution Explorer, expand the **TestHarness** project, and then double-click **MainWindow.xaml**.

   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:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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**.

   Your code should resemble the following code example.

```
...
public class StressTestCase
{
    public Material GirderMaterial;
    public CrossSection CrossSection;
    public int LengthInMm;
    public int HeightInMm;
```

```
    public int WidthInMm;

    public TestCaseResult 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.

Your code should resemble the following code example.

```
...
public StressTestCase(Material girderMaterial,
                      CrossSection crossSection,
                      int lengthInMm,
                      int heightInMm,
                      int widthInMm)
{
    this.GirderMaterial = girderMaterial;
    this.CrossSection = crossSection;
    this.LengthInMm = lengthInMm;
    this.HeightInMm = heightInMm;
    this.WidthInMm = widthInMm;
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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, ...)
{
    ...
}
```

Your code should resemble the following code example.

```
...
public TestCaseResult testCaseResult;
public StressTestCase()
    : this (Material.StainlessSteel, CrossSection.IBeam, 4000, 20, 15)
{
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
...
public class StressTestCase
{
    ...
    public void PerformStressTest()
    {
    }
}
...
```

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"**

   Your code should resemble the following code example.

```
...
public void PerformStressTest()
{
    string[] failureReasons =
    {
        "Fracture detected",
        "Beam snapped",
        "Beam dimensions wrong",
        "Beam warped",
        "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.

Your code should resemble the following code example.

```
...
if (Utility.Rand.Next(10) == 9)
{
    TestCaseResult.Result = TestResult.Fail;
    int failureCode = Utility.Rand.Next(5);
    TestCaseResult.ReasonForFailure = failureReasons[failureCode];
}
...
```

4. If the **Rand** method returns a value other than 9, add code to set the **TestCaseResult.Result** member value to **TestResult.Pass**.

Your code should resemble the following code example.

```
...

if (Utility.Rand.Next(10) == 9)
{
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
else
{
    TestCaseResult.Result = TestResult.Pass;
}

...
```

5. Below the **PerformStressTest** method, add a public method named **GetStressTestResult**, which accepts no parameters and returns a **TestCaseResult** object.

   Your code should resemble the following code example.

```
...
public class StressTestCase
{
    ...
    public TestCaseResult GetStressTestResult()
    {

    }
}
...
```

6. In the **GetStressTestResult** method, add code to return a reference to the **TestCaseResult** member.

   Your code should resemble the following code example.

```
...
public TestCaseResult GetStressTestResult()
{
    return TestCaseResult;
}
...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
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.

Your code should resemble the following code example.

```
...
public class StressTestCase
{
    ...
    public override string ToString()
    {
        return String.Format("Material: {0}, CrossSection: {1},
Length: {2}mm, Height: {3}mm, Width: {4}mm",
            GirderMaterial.ToString(),
            CrossSection.ToString(),
            LengthInMm,
            HeightInMm,
            WidthInMm);
    }
} ...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
...
public partial class MainWindow : Window
{
    ...
    private StressTestCase[] CreateTestCases()
    {
    }
} ...
```

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.

   Your code should resemble the following code example.

```
...
private StressTestCase[] CreateTestCases()
{
    StressTestCase[] stressTestCases = new StressTestCase[10];
}
...
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

Array position	Material	CrossSection	Length	Height	Width
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

Your code should resemble the following code example.

```
private StressTestCase[] CreateTestCases()
{
    ...

    stressTestCases[0] = new StressTestCase();
    stressTestCases[1] = new StressTestCase
        (Material.Composite, CrossSection.CShaped, 3500, 100, 20);
    stressTestCases[2] = new StressTestCase();
    stressTestCases[3] = new StressTestCase
        (Material.Aluminium, CrossSection.Box, 3500, 100, 20);
    stressTestCases[4] = new StressTestCase();
    stressTestCases[5] = new StressTestCase
        (Material.Titanium, CrossSection.CShaped, 3600, 150, 20);
    stressTestCases[6] = new StressTestCase
        (Material.Titanium, CrossSection.ZShaped, 4000, 80, 20);
    stressTestCases[7] = new StressTestCase
        (Material.Titanium, CrossSection.Box, 5000, 90, 20);
    stressTestCases[8] = new StressTestCase();
    stressTestCases[9] = new StressTestCase
        (Material.StainlessSteel, CrossSection.Box, 3500, 100, 20);

}
```

5. At the end of the method, return the **stressTestCases** array.

Your code should resemble the following code example.

```
...
public partial class MainWindow : Window

{
    ...
    private StressTestCase[] CreateTestCases()
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        ...
        return stressTestCases;
    }
}
...
```

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**.

   Your code should resemble the following code example.

```
...
private void doTests_Click(object sender, RoutedEventArgs e)
{
    testList.Items.Clear();
    resultList.Items.Clear();

    StressTestCase[] stressTestCases = CreateTestCases();
}
...
```

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.

   Your code should resemble the following code example.

```
...
private void doTests_Click(object sender, RoutedEventArgs e)
{
    ...
    StressTestCase[] stressTestCases = CreateTestCases();
    StressTestCase currentTestCase;
    TestCaseResult currentTestResult;
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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 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.

   Your code should resemble the following code example.

```
...
for (int i = 0; i < stressTestCases.Length; i++)
{
    currentTestCase = stressTestCases[i];
    currentTestCase.PerformStressTest();
    testList.Items.Add(currentTestCase);
    currentTestResult = currentTestCase.GetStressTestResult();
    resultList.Items.Add(currentTestResult.Result + " " +
        currentTestResult.ReasonForFailure);
}  ...
```

Task 8: Test the solution

1. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

5. Close the application, and then return to Visual Studio

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 that 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

a. On the **Build** menu, click **Build Solution**.

b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

c. Wait for the tests to run, and in the Test Results window, verify that all of the tests passed.

Exercise 4: Using a Nullable Struct

Task 1: Open the NullableStructs solution

- Open the NullableStructs solution in the E:\Labfiles\Lab 6\Ex4\Starter folder:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the

     E:\Labfiles\Lab 6\Ex4\Starter folder, click **NullableStructs.sln**, and then click **Open**.

Task 2: Modify the TestCaseResult field to make it nullable

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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.

   Your code should resemble the following code example.

```
...
public TestCaseResult? TestCaseResult;
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

  Your code should resemble the following code example.

```
...
public StressTestCase(Material girderMaterial,
    CrossSection crossSection,
    int lengthInMm,
    int heightInMm,
    int widthInMm)
{
    this.GirderMaterial = girderMaterial;
    this.CrossSection = crossSection;
    this.LengthInMm = lengthInMm;
    this.HeightInMm = heightInMm;
    this.WidthInMm = widthInMm;
    this.TestCaseResult = 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.

   Your code should resemble the following code example.

```
...
public void PerformStressTest()
{
    TestCaseResult currentTestCase = new TestCaseResult();

    // List of possible reasons for a failure.
    string[] failureReasons = { "Fracture detected",
    ...
}
...
```

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.

Your code should resemble the following code example.

```
...
public void PerformStressTest()
{
    ...
    if (Utility.rand.Next(10) == 9)
    {
        currentTestCase.Result = TestResult.Fail;
        currentTestCase.ReasonForFailure =
            failureReasons[Utility.rand.Next(5)];
        TestCaseResult = currentTestCase;
    }
    ...
}
...
```

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.

Your code should resemble the following code example.

```
...
public void PerformStressTest()
{
    ...
    else
    {
        currentTestCase.Result = TestResult.Pass;
        TestCaseResult = currentTestCase;
    }
    ...
} ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 5: Modify the GetStressTestResult method

- In the **GetStressTestResult** method, modify the method definition to return a nullable **TestCaseResult** value.

  Your code should resemble the following code example.

```
...
public TestCaseResult? GetStressTestResult()
{
    ...
} ...
```

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**.

   Your code should resemble the following code example.

```
...
StressTestCase currentStressTest;
TestCaseResult? currentTestResult;
for (int i = 0; i < stressTestCases.Length; i++)}
...
```

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.

   Your code should resemble the following code example.

```
...
for (int i = 0; i < stressTestCases.Length; i++)
{
    currentStressTest = stressTestCases[i];
    currentStressTest.PerformStressTest();
    testList.Items.Add(currentStressTest.ToString());
    currentTestResult = currentStressTest.GetStressTestResult();

    if (currentTestResult.HasValue)
    {
```

```
        resultList.Items.Add(
            currentTestResult.Value.Result.ToString() + " " +
            currentTestResult.Value.ReasonForFailure);
    }
}
```

Task 7: Test the solution

1. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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 the 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

MCT USE ONLY. STUDENT USE PROHIBITED

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** 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:
   a. On the **Build** menu, click **Build Solution**.
   b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.
   c. Wait for the tests to run, and in the Test Results window, verify that all of the tests passed.
5. Close Visual Studio:
   - On the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Encapsulating Data and Methods

Lab 7: Encapsulating Data and Methods

Exercise 1: Hiding Data Members

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:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 7\Ex1\Starter** folder, click **StressTesting.sln**, and then click **Open**.

Task 2: Declare fields in the StressTestCase class as private

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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.

   Your code should resemble the following code example.

```
...
/// <summary>
/// Girder material type (enumeration type)
/// </summary>
private Material girderMaterial;

/// <summary>
/// Girder cross-section (enumeration type)
/// </summary>
private CrossSection crossSection;

/// <summary>
/// Girder length in millimeters
/// </summary>
private int lengthInMm;

/// <summary>
/// Girder height in millimeters
/// </summary>
private int heightInMm;

/// <summary>
/// Girder width in millimeters
/// </summary>
private int widthInMm;

/// <summary>
/// Details of test result (structure type)
/// Made nullable
/// </summary>
private TestCaseResult? testCaseResult;
...
```

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:

   a. On the **Build** menu, click **Build Solution**.

   b. If the error list is not automatically displayed, on the **View** menu, click **Error List**.

   c. If the error list is not showing errors, in the error list pane, click **Errors**.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Comment out the code that caused the errors that are shown in the error list. These errors are caused by six statements in the **doTests_Click** method:

   a. In the error list, double-click the first error. This error is located in the StressTest Test Harness solution, in the MainWindow.xaml.cs file.

   b. In the **MainWindow** class, in the **doTests_Click** method, comment out the six lines of code that raise errors.

   Your code should resemble the following code example.

```
...
private void doTests_Click(object sender, RoutedEventArgs e)
{
    ...
    //Material m = stc.girderMaterial;
    //CrossSection c = stc.crossSection;
    //int l = stc.lengthInMm;
    //int h = stc.heightInMm;
    //int w = stc.widthInMm;
    //tcr = stc.testCaseResult.Value;

    stc.PerformStressTest();
    ...
}
...
```

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.

3. In the **StressTestCaseConstructorTest** method, comment out the five **Assert** statements that cause errors.

   Your code should resemble the following code example.

```
...
public void StressTestCaseConstructorTest()
{
    ...
    //Assert.AreEqual(Material.Composite, target.girderMaterial);
    //Assert.AreEqual(CrossSection.CShaped, target.crossSection);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    //Assert.AreEqual(5000, target.lengthInMm);
    //Assert.AreEqual(32, target.heightInMm);
    //Assert.AreEqual(18, target.widthInMm);
}
...
```

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
```

Your code should resemble the following code example.

```
public void StressTestCaseConstructorTest()

    {

    ...

    string expected = "Material: Composite, CrossSection: CShaped,
Length: 5000mm, Height: 32mm, Width: 18mm, No Stress Test Performed";
    StressTestCase target = new StressTestCase(
        girderMaterial,
        crossSection,
        lengthInMm,
        heightInMm,
        widthInMm);

    ...
}
```

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.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public void StressTestCaseConstructorTest()
    {
    ...
    StressTestCase target = new StressTestCase(
        girderMaterial,
        crossSection,
        lengthInMm,
        heightInMm,
        widthInMm);
    ...
    Assert.AreEqual(expected, target.ToString());
}
```

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.

   Your code should resemble the following code example.

```
public void StressTestCaseConstructorTest1()
{
    string expected = "Material: StainlessSteel, CrossSection: IBeam,
Length: 4000mm, Height: 20mm, Width: 15mm, No Stress Test Performed";
    StressTestCase target = new StressTestCase();
    //Assert.AreEqual(Material.StainlessSteel, target.girderMaterial);
    //Assert.AreEqual(CrossSection.IBeam, target.crossSection);
    //Assert.AreEqual(4000, target.lengthInMm);
    //Assert.AreEqual(20, target.heightInMm);
    //Assert.AreEqual(15, target.widthInMm);
    Assert.AreEqual(expected, target.ToString());
}
```

6. Rebuild the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**.

7. Run all of the tests in the solution, and then verify that all of the tests execute successfully:

   a. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Wait for the tests to run, and in the Test Results window, verify that all of the tests pass.

Exercise 2: Using Static Members to Share Data

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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 7\Ex2\Starter** folder, click **StressTesting.sln**, and then click **Open**.

Task 2: Create a struct to hold the number of successes and failures

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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**.

   Your code should resemble the following code example.

```
...
    public struct TestStatistics
    {
        private int numberOfTestsPerformed;
        private int numberOfFailures;
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
public struct TestStatistics
{
    ...
    private int numberOfFailures;

    public void IncrementTests(bool success)
    {
        numberOfTestsPerformed++;
        if (!success)
        {
            numberOfFailures++;
        }
    }
}
```

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.

   Your code should resemble the following code example.

```
public struct TestStatistics
{
    ...
    public int GetNumberOfTestsPerformed()
    {
        return numberOfTestsPerformed;
    }
}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public struct TestStatistics
{
    ...

    public int GetNumberOfFailures()
    {
        return numberOfFailures;
    }
}
```

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.

   Your code should resemble the following code example.

```
public struct TestStatistics
{
    ...

    internal void ResetCounters()
    {
        numberOfFailures = 0;
        numberOfTestsPerformed = 0;
    }

}
```

8. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**. 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 to the StressTestCase class** comment, and then declare a new private static member of type **TestStatistics** named **statistics**.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public class StressTestCase
{
    ...
    private TestCaseResult? testCaseResult;
    private static TestStatistics 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.

   Your code should resemble the following code example.

```
public class StressTestCase
{
    ...
    private static TestStatistics statistics;
    public static TestStatistics GetStatistics()
    {
        return statistics;
    }
    ...
}
```

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.

   Your code should resemble the following code example.

```
public class StressTestCase
{
    ...
    public static TestStatistics GetStatistics()
    {
        return statistics;
    }

    public static void ResetStatistics()
    {
        statistics.ResetCounters();
    }
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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 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.

   Your code should resemble the following code example.

```
public void PerformStressTest()
{
    ...
    if (Utility.rand.Next(10) == 9)
    {
        ...
        tcr.reasonForFailure = failureReasons[Utility.rand.Next(5)];

        statistics.IncrementTests(false);
    }
    else
    {
        tcr.result = TestResult.Pass;

        statistics.IncrementTests(true);
    }
    ...
}
```

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 comments 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.

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
private void doTests_Click(object sender, RoutedEventArgs e)
{
    ...
    TestStatistics statistics = StressTestCase.GetStatistics();
    statisticsLabel1.Content = string.Format(
        "Number of tests: {0}, Failures: {1}",
        statistics.GetNumberOfTestsPerformed(),
        statistics.GetNumberOfFailures());
    statistics.IncrementTests(true);
    statistics = StressTestCase.GetStatistics();
    statisticsLabel2.Content = string.Format(
        "Number of tests: {0}, Failures: {1}",
        statistics.GetNumberOfTestsPerformed(),
        statistics.GetNumberOfFailures());
}
```

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:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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 was 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:
   a. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.
   b. Wait for the tests to run, and in the Test Results window, verify that all of the tests pass.

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Implementing an Extension Method

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:

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 7\Ex3\Starter** folder, click **StressTesting.sln**, and then click **Open**.

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.

Your code should resemble the following code example.

```
public static class Extensions
{
    public static string ToBinaryString(this long i)
    {
    }
}
```

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.

   Your code should resemble the following code example.

```
public static string ToBinaryString(this System.Int64 i)
{
    long remainder = 0;
    StringBuilder binary = new StringBuilder("");

    while (i > 0)
    {
        remainder = i % 2;
        i = i / 2;
        binary.Insert(0, remainder);
    }
    return binary.ToString();
}
```

Task 3: Modify the TestCaseResult struct to include a long field

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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**.

   Your code should resemble the following code example.

```
public struct TestCaseResult
{
    ...
    public string reasonForFailure;

    public long failureData;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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.

Your code should resemble the following code example.

```
public void PerformStressTest()
{
    ...
    tcr.reasonForFailure = failureReasons[Utility.rand.Next(5)];

    tcr.failureData = Utility.Rand.Next(int.MaxValue);

    statistics.IncrementTests(false);
    ...
}
```

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 **ToBinaryString** method call to the end of the string that is added to the **resultList.Items** collection.

   Your code should resemble the following code example.

```
private void doTests_Click(object sender, RoutedEventArgs e)
{
    ...
    {
        ...
        if (stc.GetStressTestResult().HasValue)
        {
            tcr = (TestCaseResult)stc.GetStressTestResult().Value;

            // Modified in Exercise 3 to use extension method.
            resultList.Items.Add(tcr.result.ToString()
                + " " + tcr.reasonForFailure
                + " " + tcr.failureData.ToBinaryString());
        }
    }
    ...
}
```

Task 6: Test the solution

1. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
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

MCT USE ONLY. STUDENT USE PROHIBITED

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:

   a. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

   b. Wait for all of the tests to run, and in the Test Results window, verify that all of the tests pass.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Inheriting from Classes and Implementing Interfaces

Lab 8: Inheriting from Classes and Implementing Interfaces

Exercise 1: Defining an 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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 8\Snippets folder.
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** drop-down, click **Visual C#**.
   c. In the **Code Snippets Manager** dialog box, click **Add**.
   d. In the Code Snippets Directory dialog box, browse to the E:\Labfiles\Lab 8\Snippets folder, and then click Select Folder.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module8 solution in the E:\Labfiles\Lab 8\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 8\Ex1\Starter** folder, click **Module8.sln**, and then click **Open**.

Task 2: Create the IMeasuringDevice interface

1. Open the IMeasuringDevice code file:

- In Solution Explorer, double-click **IMeasuringDevice.cs**.

2. In the **MeasuringDevice** namespace, declare the **IMeasuringDevice** interface. The **IMeasuringDevice** interface must be accessible to code in other assemblies.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
    }
}
```

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.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into a metric value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to metric units.</returns>
        decimal MetricValue();
    }
}
```

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:

   - Add the code in the following code example to the interface.

```
/// <summary>
/// Converts the raw data collected by the measuring device into an
/// imperial value.
/// </summary>
/// <returns>The latest measurement from the device converted to
/// imperial units.</returns>
decimal ImperialValue();
```

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   - Add the code in the following code example to the interface.

```
/// <summary>
/// Starts the measuring device.
/// </summary>
void StartCollecting();
```

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:
   - Add the method in the following code example to the interface.

```
/// <summary>
/// Stops the measuring device.
/// </summary>
void StopCollecting();
```

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:
   - Add the method in the following code example to the interface.

```
/// <summary>
/// Enables access to the raw data from the device in whatever units
/// are native to the device.
/// </summary>
/// <returns>The raw data from the device in native format.</returns>
int[] GetRawData();
```

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

   At the end of this exercise, your code should resemble the following code example.

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

namespace MeasuringDevice
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
{
    public interface IMeasuringDevice
    {
        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into a metric value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to metric units.</returns>
        decimal MetricValue();

        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into an imperial value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to imperial units.</returns>
        decimal ImperialValue();

        /// <summary>
        /// Starts the measuring device.
        /// </summary>
        void StartCollecting();

        /// <summary>
        /// Stops the measuring device.
        /// </summary>
        void StopCollecting();

        /// <summary>
        /// Enables access to the raw data from the device in whatever
        /// units are native to the device.
        /// </summary>
        /// <returns>The raw data from the device in native
        /// format.</returns>
        int[] GetRawData();
    }
}
```

Exercise 2: Implementing an Interface

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:

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 8\Ex2\Starter** folder, click **Module8.sln**, and then click **Open**.

Task 2: Create the Units enumeration

The Units enumeration will contain two values, **Metric** and **Imperial**. Metric measurements are used in the International System of Units (SI), and include measurements in kilograms and meters. Imperial measurements were originally used in the British Empire, and are similar to customary system units in the United States.

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. In the task list, double-click the task **TODO: Implement the Units enumeration**. This task is located in the UnitsEnumeration.cs file:

   - In the task list, double-click **TODO: Implement the Units enumeration**.

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.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public enum Units
    {
    }
}
```

4. Add the values **Metric** and **Imperial** to the enumeration.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public enum Units
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        Metric, Imperial
    }
}
```

5. Comment your code to make it easier for developers who use the enumeration.

   Your code should resemble the following code example.

```
/// <summary>
/// Public enumeration used in measuring device classes to specify the
/// units used by the device.
/// </summary>
public enum Units
{
    Metric, Imperial
}
```

6. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

   At the end of this task, your code should resemble the following code example.

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

namespace MeasuringDevice
{
    /// <summary>
    /// Public enumeration used in measuring device classes to specify
    /// the units used by the device.
    /// </summary>
    public enum Units
    {
        Metric, Imperial
    }
}
```

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:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- In the task list, double-click the task **TODO: Implement the MeasureLengthDevice class**.

2. Remove the TODO comment and add a **public** class named **MeasureLengthDevice**.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public class MeasureLengthDevice
    {
    }
}
```

3. Modify the **MeasureLengthDevice** class declaration to implement the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
public class MeasureLengthDevice : IMeasuringDevice
```

4. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface:

   - Right-click **IMeasuringDevice**, point to **Implement Interface**, and then click **Implement Interface**.

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 want, but you do not need to modify it:

   - At the start of the file, after the existing **using** statements, add the statement in the following code example.

```
using DeviceControl;
```

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.

Your code should resemble the following code example.

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private DeviceType measurementType;
```

7. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.LENGTH**.

   Your modified code should resemble the following code example.

```
private const DeviceType measurementType = 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.

   Your code should resemble the following code example.

```
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
    GetMeasurements();
}
```

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;
            }
        }
    });
}
```

- To use the Mod8GetMeasurementsMethod snippet, add a blank line immediately after the closing brace of the **StartCollecting** method, type **Mod8GetMeasurementsMethod** and then press the TAB key.

The **GetMeasurements** method retrieves measurements from the emulated device. In this module, you will use the code in the **GetMeasurements** method

MCT USE ONLY. STUDENT USE PROHIBITED

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**.

    Your code should resemble the following code example.

```
public void StopCollecting()
{
    if(controller != 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**.

    Your code should resemble the following code example.

```
public void StopCollecting()
{
    if(controller != null)
    {
        controller.StopDevice();
        controller = null;
    }
}
```

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.

    Your code should resemble the following code example.

```
public int[] GetRawData()
{
    return dataCaptured;
}
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 25.4.

Your code should resemble the following code example.

```
public decimal MetricValue()
{
    decimal metricMostRecentMeasure;

    if (unitsToUse == Units.Metric)
    {
        metricMostRecentMeasure =
            Convert.ToDecimal(mostRecentMeasure);
    }
    else
    {
        // Imperial measurements are in inches.
        // Multiply imperial measurement by 25.4 to convert from
        // inches to millimeters.
        // Convert from an integer value to a decimal.
        decimal decimalImperialValue =
            Convert.ToDecimal(mostRecentMeasure);
        decimal conversionFactor = 25.4M;
        metricMostRecentMeasure =
            decimalImperialValue * conversionFactor;
    }

    return metricMostRecentMeasure;
}
```



Note: This code performs the process of converting from imperial to metric step by step. You can perform this conversion in a single statement as shown below. However, you should consider that code should be as self-documenting as possible so that it can be maintained more easily.

```
public decimal MetricValue()
{
    return (unitsToUse == units.Metric) ?
           (decimal)mostRecentMeasure :
           (decimal)mostRecentMeasure * 25.4M;
}
```

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

MCT USE ONLY. STUDENT USE PROHIBITED

the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 0.03937.

Your code should resemble the following code example.

```
public decimal ImperialValue()
{
    decimal imperialMostRecentMeasure;
    if (unitsToUse == Units.Imperial)
    {
        imperialMostRecentMeasure =
            Convert.ToDecimal(mostRecentMeasure);
    }
    else
    {
        // Metric measurements are in millimeters.
        // Multiply metric measurement by 0.03937 to convert from
        // millimeters to inches.
        // Convert from an integer value to a decimal.
        decimal decimalMetricValue =
            Convert.ToDecimal(mostRecentMeasure);
        decimal conversionFactor = 0.03937M;
        imperialMostRecentMeasure =
        decimalMetricValue * conversionFactor;
    }

    return imperialMostRecentMeasure;
}
```

16. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.

    Your code should resemble the following code example.

```
public class MeasureLengthDevice : IMeasuringDevice
{
    public MeasureLengthDevice(Units deviceUnits)
    {
        unitsToUse = deviceUnits;
    }

    ...

}
```

17. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

At the end of this task, your code should resemble the following code example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DeviceControl;
namespace MeasuringDevice
{
    public class MeasureLengthDevice : IMeasuringDevice
    {
        public MeasureLengthDevice(Units deviceUnits)
        {
            unitsToUse = deviceUnits;
        }

        public decimal MetricValue()
        {
            decimal metricMostRecentMeasure;

            if (unitsToUse == Units.Metric)
            {
                metricMostRecentMeasure =
                    Convert.ToDecimal(mostRecentMeasure);
            }
            else
            {
                // Imperial measurements are in inches.
                // Multiply imperial measurement by 25.4 to convert
                // from inches to millimeters.
                // Convert from an integer value to a decimal.
                decimal decimalImperialValue =
                    Convert.ToDecimal(mostRecentMeasure);
                decimal conversionFactor = 25.4M;
                metricMostRecentMeasure =
                    decimalImperialValue * conversionFactor;
            }

            return metricMostRecentMeasure;
        }

        public decimal ImperialValue()
        {
            decimal imperialMostRecentMeasure;

            if (unitsToUse == Units.Imperial)
            {
                imperialMostRecentMeasure =
                    Convert.ToDecimal(mostRecentMeasure);
            }
```

```
            else
            {
                // Metric measurements are in millimeters.
                // Multiply metric measurement by 0.03937 to convert
                // from millimeters to inches.
                // Convert from an integer value to a decimal.
                decimal decimalMetricValue =
                    Convert.ToDecimal(mostRecentMeasure);
                decimal conversionFactor = 0.03937M;
                imperialMostRecentMeasure =
                    decimalMetricValue * conversionFactor;
            }

            return imperialMostRecentMeasure;
        }

        public void StartCollecting()
        {
            controller =
                DeviceController.StartDevice(measurementType);
            GetMeasurements();
        }

        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;
                            }
                        }
                    });
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        public void StopCollecting()
        {
            if (controller != null)
            {
                controller.StopDevice();
                controller = null;
            }
        }

        public int[] GetRawData()
        {
            return dataCaptured;
        }

        private Units unitsToUse;
        private int[] dataCaptured;
        private int mostRecentMeasure;
        private DeviceController controller;
        private const DeviceType measurementType = DeviceType.LENGTH;
    }
}
```

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. Review the task list:
    a. If the task list is not already visible, on the **View** menu, click **Task List**.
    b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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:
    - In the task list, double-click the first **TODO: Add code to instantiate the device field** item.
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

MCT USE ONLY. STUDENT USE PROHIBITED

MeasureLengthDevice class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureLengthDevice** constructor.

Your code should resemble the following code example.

```
private void createInstance_Click(object sender, RoutedEventArgs e)
{
    if((bool)metricChoice.IsChecked)
    {
        device = new MeasureLengthDevice(Units.Metric);
    }
    else
    {
        device = new MeasureLengthDevice(Units.Imperial);
    }
}
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 5: Test the MeasureLengthDevice class by using the test harness

1. Set the Exercise2TestHarness project to be the default startup project:
   - In Solution Explorer, right-click the **Exercise2TestHarness** project, and then click **Set as StartUpProject**.
2. Start the Exercise2TestHarness application:
   - On the **Debug** menu, click **Start Without Debugging**.
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

MCT USE ONLY. STUDENT USE PROHIBITED

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

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:
    a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 8\Ex3\Starter** folder, click **Module8.sln**, and then click **Open**.

Task 2: Create the MeasureMassDevice class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the MeasureMassDevice.cs file:
   - In Solution Explorer, double-click **MeasureMassDevice.cs**.
3. Replace the TODO comment with a **public** class named **MeasureMassDevice**.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public class MeasureMassDevice
    {
    }
}
```

4. Modify the **MeasureMassDevice** class declaration to implement the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
public class MeasureMassDevice : IMeasuringDevice
{
}
```

5. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface:
   - Right-click **IMeasuringDevice**, point to **Implement Interface**, and then click **Implement Interface**.
6. Bring the **DeviceControl** namespace into scope:
   - At the start of the file, after the existing **using** statements, add the statement in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
using DeviceControl;
```

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

Your code should resemble the following code example.

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private DeviceType measurementType;
```

8. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.MASS**.

Your modified code should resemble the following code example.

```
private const DeviceType measurementType = 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.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
}
```

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.

    Your code should resemble the following code example.

```
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
    GetMeasurements();

}
```

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;
            }
        }
    });
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

- To use the Mod8GetMeasurementsMethod snippet, add a blank line immediately after the closing brace of the **StartCollecting** method, type **Mod8GetMeasurementsMethod** and then press the TAB key.

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**.

    Your code should resemble the following code example.

```
public void StopCollecting()
{
    if(controller != 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**.

    Your code should resemble the following code example.

```
public void StopCollecting()
{
    if(controller != null)
    {
        controller.StopDevice();
        controller = 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.

    Your code should resemble the following code example.

```
public int[] GetRawData()
{
    return dataCaptured;
}
```

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.

Your code should resemble the following code example.

```
public decimal MetricValue()
{
    decimal metricMostRecentMeasure;

    if (unitsToUse == Units.Metric)
    {
        metricMostRecentMeasure =
            Convert.ToDecimal(mostRecentMeasure);
    }

    else
    {
        // Imperial measurements are in pounds.
        // Multiply imperial measurement by 0.4536 to convert from
        // pounds to kilograms.
        // Convert from an integer value to a decimal.
        decimal decimalImperialValue =
            Convert.ToDecimal(mostRecentMeasure);
        decimal conversionFactor = 0.4536M;
        metricMostRecentMeasure =
            decimalImperialValue * conversionFactor;
    }

    return metricMostRecentMeasure;
}
```

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.

    Your code should resemble the following code example.

```
public decimal ImperialValue()
{
    decimal imperialMostRecentMeasure;

    if (unitsToUse == Units.Imperial)
    {
        imperialMostRecentMeasure =
            Convert.ToDecimal(mostRecentMeasure);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }

    else
    {
        // Metric measurements are in kilograms.
        // Multiply metric measurement by 2.2046 to convert from
        // kilograms to pounds.
        // Convert from an integer value to a decimal.

        decimal decimalMetricValue =
            Convert.ToDecimal(mostRecentMeasure);
        decimal conversionFactor = 2.2046M;
        imperialMostRecentMeasure =
            decimalMetricValue * conversionFactor;
    }

    return imperialMostRecentMeasure;
}
```

17. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.

    Your code should resemble the following code example.

```
public class MeasureMassDevice : IMeasuringDevice
{
    public MeasureMassDevice(Units deviceUnits)
    {
        unitsToUse = deviceUnits;
    }

    ...

}
```

18. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

    At the end of this task, your code should resemble the following code example.

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

using DeviceControl;

namespace MeasuringDevice
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
{
    public class MeasureMassDevice : IMeasuringDevice
    {
        public MeasureMassDevice(Units DeviceUnits)
        {
            unitsToUse = DeviceUnits;
        }

        public decimal MetricValue()
        {
            decimal metricMostRecentMeasure;

            if (unitsToUse == Units.Metric)
            {
                metricMostRecentMeasure =
                    Convert.ToDecimal(mostRecentMeasure);
            }
            else
            {
                // Imperial measurements are in pounds.
                // Multiply imperial measurement by 0.4536 to convert
                // from pounds to kilograms.
                // Convert from an integer value to a decimal.
                decimal decimalImperialValue =
                    Convert.ToDecimal(mostRecentMeasure);
                decimal conversionFactor = 0.4536M;
                metricMostRecentMeasure =
                    decimalImperialValue * conversionFactor;
            }

            return metricMostRecentMeasure;
        }

        public decimal ImperialValue()
        {
            decimal imperialMostRecentMeasure;

            if (unitsToUse == Units.Imperial)
            {
                imperialMostRecentMeasure =
                    Convert.ToDecimal(mostRecentMeasure);
            }
            else
            {
                // Metric measurements are in kilograms.
                // Multiply metric measurement by 2.2046 to convert
                // from kilograms to pounds.
                // Convert from an integer value to a decimal.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            decimal decimalMetricValue =
                Convert.ToDecimal(mostRecentMeasure);
            decimal conversionFactor = 2.2046M;
            imperialMostRecentMeasure =
                decimalMetricValue * conversionFactor;
        }

        return imperialMostRecentMeasure;
    }
    public void StartCollecting()
    {
        controller =
            DeviceController.StartDevice(measurementType);
        GetMeasurements();
    }

    public void StopCollecting()
    {
        if (controller != null)
        {
            controller.StopDevice();
            controller = null;
        }
    }

    public int[] GetRawData()
    {
        return dataCaptured;
    }

    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)
```

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

```
                            {
                                x = 0;
                            }
                        }
                    });
        }

        private Units unitsToUse;
        private int[] dataCaptured;
        private int mostRecentMeasure;
        private DeviceController controller;
        private const DeviceType measurementType = DeviceType.MASS;
    }
}
```

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. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
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:
   - In the task list, double-click the first **TODO: Instantiate the device field by using the new MeasureMassDevice class** item.
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.

   Your code should resemble the following code example.

```
case "Mass Device":
    if((bool)metricChoice.IsChecked)
    {
        device = new MeasureMassDevice(Units.Metric);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }
    else
    {
        device = new MeasureMassDevice(Units.Imperial);
    }

    break;
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. 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:
   - In Solution Explorer, right-click the **Exercise3TestHarness** project, and then click **Set as StartUpProject**.
2. Start the Exercise3TestHarness application:
   - On the **Debug** menu, click **Start Without Debugging**.
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.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   - In Solution Explorer, double-click **MeasureDataDevice.cs**.
2. Remove the TODO comment and add an **abstract** class named **MeasureDataDevice**.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public abstract class MeasureDataDevice
    {
    }
}
```

3. Modify the **MeasureDataDevice** class declaration to implement the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
public abstract class MeasureDataDevice : IMeasuringDevice
```

4. Bring the **DeviceControl** namespace into scope:
   - At the start of the file, after the existing **using** statements, add the statement in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
using DeviceControl;
```

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.

   Your code should resemble the following code example.

```
public abstract class MeasureDataDevice : IMeasuringDevice
{
    public abstract decimal MetricValue();
}
```

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.

   Your code should resemble the following code example.

```
public abstract class MeasureDataDevice : IMeasuringDevice
{
    public abstract decimal MetricValue();
    public abstract decimal ImperialValue();
}
```

7. In the MeasureLengthDevice.cs file, locate and copy the code for the **StartCollecting** method, and then add this method to the **MeasureDataDevice** class:

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Solution Explorer, double-click **MeasureLengthDevice.cs**.

b. In the MeasureLengthDevice.cs file, locate and highlight the code in the following code example, and then press CTRL+C.

```
/// <summary>
/// Starts the measuring device.
/// </summary>

public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
    GetMeasurements();
}
```

c. Return to the MeasureDataDevice.cs file.

d. In the **MeasureDataDevice** class, add two blank lines after the declaration in the following code example.

```
public abstract decimal ImperialValue();
```

e. Press CTRL+V.

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:

a. In Solution Explorer, double-click **MeasureLengthDevice.cs**.

b. In the MeasureLengthDevice.cs file, locate and highlight the code in the following code example, and then press CTRL+C.

```
/// <summary>
/// Stops the measuring device.
/// </summary>
public void StopCollecting()
{
    if (controller != null)
    {
        controller.StopDevice();
        controller = null;
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

c. Return to the MeasureDataDevice.cs file.

d. In the **MeasureDataDevice** class, add two blank lines after the **StartCollecting** method.

e. Press CTRL+V.

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:

a. In Solution Explorer, double-click **MeasureLengthDevice.cs**.

b. In the MeasureLengthDevice.cs file, locate and highlight the code in the following code example, and then press CTRL+C.

```
/// <summary>
/// Enables access to the raw data from the device in whatever units
are native to the device.
/// </summary>
/// <returns>The raw data from the device in native format.</returns>
public int[] GetRawData()
{
    return dataCaptured;
}
```

c. Return to the MeasureDataDevice.cs file.

d. In the **MeasureDataDevice** class, add two blank lines after the **StopCollecting** method.

e. Press CTRL+V.

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:

a. In Solution Explorer, double-click **MeasureLengthDevice.cs**.

b. In the MeasureLengthDevice.cs file, locate and highlight the code in the following code example, and then press CTRL+C.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];

    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        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;
            }
        }

    });

}
```

c. Return to the MeasureDataDevice.cs file.

d. In the **MeasureDataDevice** class, add two blank lines after the **GetRawData** method.

e. Press CTRL+V.

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

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Solution Explorer, double-click **MeasureLengthDevice.cs**.

b. In the MeasureLengthDevice.cs file, locate and highlight the code in the following code example, and then press CTRL+C.

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private const DeviceType measurementType = DeviceType.LENGTH;
```

c. Return to the MeasureDataDevice.cs file.

d. In the **MeasureDataDevice** class, add two blank lines after the **GetMeasurements** method.

e. Press CTRL+V.

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:

    - Change each of the accessors from **private** to **protected**. Your code should resemble the following code example.

```
protected Units unitsToUse;
protected int[] dataCaptured;
protected int mostRecentMeasure;
protected DeviceController controller;
protected const DeviceType measurementType = DeviceType.LENGTH;
```

13. Modify the declaration of the **measurementType** field so that it is no longer constant and not instantiated when it is declared:

    - Modify the last line of code in the previous code example so that it resembles the following code example.

```
protected DeviceType measurementType;
```

14. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   - In the MeasureLengthDevice.cs file, change the class declaration as shown in the following code example.

```
public class MeasureLengthDevice : MeasureDataDevice, IMeasuringDevice
```

2. Remove the **StartCollecting** method from the **MeasureLengthDevice** class:
   - Remove the code in the following code example.

```
/// <summary>
/// Starts the measuring device.
/// </summary>
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
    GetMeasurements();
}
```

3. Remove the **StopCollecting** method from the **MeasureLengthDevice** class:
   - Remove the code in the following code example.

```
/// <summary>
/// Stops the measuring device.
/// </summary>
public void StopCollecting()
{
    if (controller != null)
    {
        controller.StopDevice();
        controller = null;
    }
}
```

4. Remove the **GetRawData** method from the **MeasureLengthDevice** class:

- Remove the code in the following code example.

```
/// <summary>
/// Enables access to the raw data from the device in whatever units
are native to the device.
/// </summary>
/// <returns>The raw data from the device in native format.</returns>

public int[] GetRawData()
{
    return dataCaptured;
}
```

5. Remove the **GetMeasurements** method from the **MeasureLengthDevice** class:

- Remove the code in the following code example.

```
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;
            }

        }

    });
}
```

6. Remove the fields in the following table from the **MeasureLengthDevice** class.

MCT USE ONLY. STUDENT USE PROHIBITED

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

- Remove the declarations in the following code example.

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private const DeviceType measurementType = DeviceType.LENGTH;
```

7. Modify the constructor to set the **measurementType** field to **DeviceType.LENGTH**:

   - Modify the code to resemble the following code example.

```
public MeasureLengthDevice(Units deviceUnits)
{
    unitsToUse = deviceUnits;
    measurementType = DeviceType.LENGTH;
}
```

8. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class:

   - Modify the code to resemble the following code example.

```
public override decimal MetricValue()
{
   ...
}
```

9. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class:

   - Modify the code to resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public override decimal ImperialValue()
{
   ...
}
```

10. In the MeasureMassDevice.cs file, modify the declaration of the **MeasureMassDevice** class so that it inherits from the **MeasureDataDevice** class:
    - On the **MeasureMassDevice.cs** tab, change the class declaration as shown in the following code example.

```
public class MeasureMassDevice : MeasureDataDevice, IMeasuringDevice
```

11. Remove the **StartCollecting** method from the **MeasureMassDevice** class:
    - Remove the code in the following code example.

```
public void StartCollecting()
{
    controller = DeviceController.StartDevice(measurementType);
    GetMeasurements();
}
```

12. Remove the **StopCollecting** method from the **MeasureMassDevice** class:
    - Remove the code in the following code example.

```
public void StopCollecting()
{
    if (controller != null)
    {
        controller.StopDevice();
        controller = null;
    }
}
```

13. Remove the **GetRawData** method from the **MeasureMassDevice** class:
    - Remove the code in the following code example.

```
public int[] GetRawData()
{
    return dataCaptured;
}
```

14. Remove the **GetMeasurements** method from the **MeasureMassDevice** class:

MCT USE ONLY. STUDENT USE PROHIBITED

- Remove the code in the following code example.

```
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;
            }
        }

    });

}
```

15. Remove the fields in the following table from the **MeasureMassDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

- Remove the code in the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private const DeviceType measurementType = DeviceType.MASS;
```

16. Modify the constructor to set the **measurementType** field to **DeviceType.MASS**:
    - Modify the code to resemble the following code example.

```
public MeasureMassDevice(Units deviceUnits)
{
    unitsToUse = deviceUnits;
    measurementType = DeviceType.MASS;
}
```

17. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class:
    - Modify the code to resemble the following code example.

```
public override decimal MetricValue()
{
   ...
}
```

18. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class:
    - Modify the code to resemble the following code example.

```
public override decimal ImperialValue()
{
   ...
}
```

19. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. 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.

MCT USE ONLY. STUDENT USE PROHIBITED

1. Start the Exercise3TestHarness application:
   - On the **Debug** menu, click **Start Without Debugging**.
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.
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:
    - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Managing the Lifetime of Objects and Controlling Resources

Lab 9: Managing the Lifetime of Objects and Controlling Resources

Exercise 1: Implementing the IDisposable Interface

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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010** .
3. Import the code snippets from the E:\Labfiles\Lab 9\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, click **Add**.
   c. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 9\Snippets** folder, and then click **Select Folder**.
   d. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module9 solution in the E:\Labfiles\Lab 9\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9\Ex1\Starter** folder, click **Module9.sln**, and then click **Open**.

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** interface, rather than editing the existing interface to ensure compatibility with existing code.

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the ILoggingMeasuringDevice.cs file:
   - In Solution Explorer, double-click **ILoggingMeasuringDevice.cs**.
3. Remove the TODO comment and declare an interface named **ILoggingMeasuringDevice**. The interface must be accessible from code in different assemblies.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface ILoggingMeasuringDevice
    {
    }
}
```

4. Modify the interface to inherit from the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
public interface ILoggingMeasuringDevice : IMeasuringDevice
```

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.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface ILoggingMeasuringDevice : IMeasuringDevice
    {
        /// <summary>
        /// Returns the file name of the logging file for the device.
        /// </summary>
        /// <returns>The file name for the logging file.</returns>
        string GetLoggingFile();
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

6. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. 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:
   - In Solution Explorer, double-click **MeasureDataDevice.cs**.
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.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    // TODO: Modify this class to implement the IDisposable interface.

    public abstract class MeasureDataDevice : ILoggingMeasuringDevice
    {
        ...
    }

}
```

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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
protected DeviceType measurementType;
protected string loggingFileName;
// TODO: Add methods to implement the ILoggingMeasuringDevice
interface.

...
```

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.

   Your code should resemble the following code example.

```
...

protected string loggingFileName;
private TextWriter loggingFileWriter;

// TODO: Add methods to implement the ILoggingMeasuringDevice
interface.

...
```

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.

   Your code should resemble the following code example.

```
...

private TextWriter loggingFileWriter;

public string GetLoggingFile()
{
    return loggingFileName;
}

...
```

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.

MCT USE ONLY. STUDENT USE PROHIBITED

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:

   - To use the code snippet, delete the TODO comment, type **Mod9InstantiateLoggingFileWriter** and then press the TAB key.

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.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
public void StopCollecting()
{
    if (controller != null)
    {
        controller.StopDevice();
        controller = null;
    }

    // New code to write to the log.
    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine("Collecting Stopped.");
    }
}
```

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.

    Your code should resemble the following code example.

```
while (controller != null)
{
    System.Threading.Thread.Sleep(timer.Next(1000, 5000));
    dataCaptured[x] = controller != null ?
        controller.TakeMeasurement() : dataCaptured[x];
    mostRecentMeasure = dataCaptured[x];
    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine
            ("Measurement Taken: {0}", mostRecentMeasure.ToString());
    }
    x++;
    if (x == 10)
    {
        x = 0;
    }
}
```

14. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

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.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{

    public abstract class MeasureDataDevice
        : ILoggingMeasuringDevice, IDisposable

    {
        ...
    }

}
```

2. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IDisposable** interface:

   - Right-click **IDisposable**, point to **Implement Interface**, and then click **Implement 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**.

   Your code should resemble the following code example.

```
protected virtual void Dispose(bool disposing)
{
    if (disposing)
    {
        // Check that the log file is closed; if it is not closed, log
        // a message and close it.
        if (loggingFileWriter != null)
        {
            loggingFileWriter.WriteLine("Object Disposed");
            loggingFileWriter.Flush();
            loggingFileWriter.Close();
            loggingFileWriter = null;
        }
    }
}
```

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.

   Your code should resemble the following code example.

```
public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

5. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

   At the end of this task, the **MeasuringDataDevice** class should resemble the following code example.

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

using DeviceControl;
using System.IO;

namespace MeasuringDevice
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
public abstract class MeasureDataDevice
    : ILoggingMeasuringDevice, IDisposable
{
    /// <summary>
    /// Converts the raw data collected by the measuring device
    /// into a metric value.
    /// </summary>
    /// <returns>The latest measurement from the device converted
    /// to metric units.</returns>
    public abstract decimal MetricValue();

    /// <summary>
    /// Converts the raw data collected by the measuring device
    /// into an imperial value.
    /// </summary>
    /// <returns>The latest measurement from the device converted
    /// to imperial units.</returns>
    public abstract decimal ImperialValue();

    /// <summary>
    /// Starts the measuring device.
    /// </summary>
    public void StartCollecting()
    {
        controller =
            DeviceController.StartDevice(measurementType);

        // 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 whether 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");
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            }
        }
        else
        {
            loggingFileWriter.WriteLine
                ("Log file status checked - Already open");
            loggingFileWriter.WriteLine("Collecting Started");
        }
        GetMeasurements();
    }

    /// <summary>
    /// Stops the measuring device.
    /// </summary>
    public void StopCollecting()
    {
        if (controller != null)
        {
            controller.StopDevice();
            controller = null;
        }

        // New code to write to the log.
        if (loggingFileWriter != null)
        {
            loggingFileWriter.WriteLine("Collecting Stopped");
        }
    }

    /// <summary>
    /// Enables access to the raw data from the device in whatever
    /// units are native to the device.
    /// </summary>
    /// <returns>The raw data from the device in native
    /// format.</returns>
    public int[] GetRawData()
    {
        return dataCaptured;
    }

    private void GetMeasurements()
    {
        dataCaptured = new int[10];
        System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
        {
            int x = 0;
            Random timer = new Random();

            while (controller != null)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            {
                System.Threading.Thread.Sleep
                    (timer.Next(1000, 5000));
                dataCaptured[x] = controller != null
                    ? controller.TakeMeasurement()
                    : dataCaptured[x];
                mostRecentMeasure = dataCaptured[x];

                if (loggingFileWriter != null)
                {
                    loggingFileWriter.WriteLine
                        ("Measurement Taken: {0}",
                        mostRecentMeasure.ToString());
                }

                x++;
                if (x == 10)
                {
                    x = 0;
                }
            }
        });
    }

    protected Units unitsToUse;
    protected int[] dataCaptured;
    protected int mostRecentMeasure;
    protected DeviceController controller;
    protected DeviceType measurementType;

    // New fields and method to implement the logging
    // functionality.

    protected string loggingFileName;
    private TextWriter loggingFileWriter;

    /// <summary>
    /// Returns the file name of the logging file for the device.
    /// </summary>
    /// <returns>The file name of the logging file.</returns>
    public string GetLoggingFile()
    {
        return loggingFileName;
    }

    // New methods to implement the IDisposable interface.

    /// <summary>
    /// Dispose method required for the IDispose interface.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        /// </summary>
        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {

            if (disposing)
            {
                // Check that the log file is closed; if it is not
                // closed, log a message and close it.

                if (loggingFileWriter != null)
                {
                    loggingFileWriter.WriteLine("Object Disposed");
                    loggingFileWriter.Flush();
                    loggingFileWriter.Close();
                    loggingFileWriter = null;
                }

            }

        }
    }
}
```

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:
   - In Solution Explorer, double-click **MeasureMassDevice.cs**.
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.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
/// <summary>
/// Construct a new instance of the MeasureMassDevice class.
/// </summary>
/// <param name="DeviceUnits">Specifies the units used natively by the
/// device.</param>
/// <param name="LogFileName">Specifies the required file name used
/// for logging in the class.</param>
public MeasureMassDevice(Units deviceUnits, string logFileName)
{
    unitsToUse = deviceUnits;
    measurementType = DeviceType.MASS;
    loggingFileName = logFileName;
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Exercise 2: Managing Resources Used by an Object

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:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9\Ex2\Starter** folder, click **Module9.sln**, and then click **Open**.

Task 2: Test the logging functionality by using the test harness

1. Run the Exercise2 Test Harness application:
   - On the **Debug** menu, click **Start Debugging**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.
   b. In Notepad, on the **File** menu, click **Open**.
   c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.
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:
   - On the **File** menu, click **Exit**.
7. Run the Exercise2 Test Harness application again, click **Get Measurements**, and then wait for the data to appear:
   a. In Visual Studio, on the **Debug** menu, click **Start Debugging**.
   b. In the Exercise 2 window, click **Get Measurements**.
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.

MCT USE ONLY. STUDENT USE PROHIBITED

9. Stop the Exercise 2 application:
   - On the **Debug** menu, click **Stop Debugging**.

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:
   - In Solution Explorer, expand the **Exercise2 Test Harness** project, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.
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.

   Your code should resemble the following code example.

```
private void createInstance_Click(object sender, RoutedEventArgs e)
{
    using (MeasureMassDevice device =
        new MeasureMassDevice(Units.Metric,
            @"E:\Labfiles\Lab 9\LogFile.txt"))
    {
        device.StartCollecting();
        loggingFileNameBox.Text = device.GetLoggingFile();
        System.Threading.Thread.Sleep(20000);
        metricValueBox.Text = device.MetricValue().ToString();
        imperialValueBox.Text = device.ImperialValue().ToString();
        rawDataValues.ItemsSource = device.GetRawData();
        device.StopCollecting();
    }
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Verify that the object is disposed of correctly

1. Run the Exercise2 Test Harness application:
   - On the **Debug** menu, click **Start Debugging**.
2. Click **Get Measurements**, and then wait until the data appears.

MCT USE ONLY. STUDENT USE PROHIBITED

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:
   a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.
   b. In Notepad, on the **File** menu, click **Open**.
   c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.
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**.
6. Close Notepad:
   - On the **File** menu, click **Exit**.
7. Run the Exercise2 Test Harness application again:
   - On the **Debug** menu, click **Start Debugging**.
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.

   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:
    a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.
    b. In Notepad, on the **File** menu, click **Open**.
    c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 9** folder, click **LogFile.txt**, and then click **Open**.
11. Review the contents of the log file.

MCT USE ONLY. STUDENT USE PROHIBITED

The file contains the most recent values displayed on the form.

12. Close Notepad:
    - On the **File** menu, click **Exit**.
13. Close the Exercise 2 window.
14. Close Visual Studio:
    - On the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Encapsulating Data and Defining Overloaded Operators

Lab A: Creating and Using Properties

Exercise 1: Defining Properties in an Interface

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 Microsoft Visual Studio 2010:
   - Click **Start**, point **to All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 10\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** list, select **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 10\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab A\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab A\Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Add properties to the IMeasuringDeviceWithProperties interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List.**
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the IMeasuringDeviceWithProperties.cs file:
   - In Solution Explorer, double-click **IMeasuringDeviceWithProperties.cs.**
3. Remove the comment **TODO: Add properties to the interface.**:
   - Delete the following line of code.

```
// TODO: Add properties to the interface.
```

4. Add a read-only property to the interface of type **Units** called **UnitsToUse**.

   Your code should resemble the following code example.

```
interface IMeasuringDeviceWithProperties : ILoggingMeasuringDevice
{
    Units UnitsToUse { get; }
}
```

5. Add a read-only property to the interface of type **int[]** called **DataCaptured**.

   Your code should resemble the following code example.

```
interface IMeasuringDeviceWithProperties : ILoggingMeasuringDevice
{
    Units UnitsToUse { get; }
    int[] DataCaptured { get; }
}
```

6. Add a read-only property to the interface of type **int** called **MostRecentMeasure**.

   Your code should resemble the following code example.

```
interface IMeasuringDeviceWithProperties : ILoggingMeasuringDevice
{
    Units UnitsToUse { get; }
    int[] DataCaptured { get; }
    int MostRecentMeasure { get; }
}
```

7. Add a read/write property to the interface of type **string** called **LoggingFileName**.

   Your code should resemble the following code example.

```
interface IMeasuringDeviceWithProperties : ILoggingMeasuringDevice
{
    Units UnitsToUse { get; }

    int[] DataCaptured { get; }

    int MostRecentMeasure { get; }

    string LoggingFileName { get; set; }
}
```

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Exercise 2: Implementing Properties in a Class

Task 1: Open the starter project



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **IMeasuringDeviceWithProperties** interface successfully, proceed directly to **Task 2: Update the MeasureDataDevice class to implement the IMeasuringDeviceWithProperties interface**.

- Open the Module10 solution in the E:\Labfiles\Lab 10\Lab A\Ex2\Starter folder. This solution contains a completed version of the **IMeasuringDeviceWithProperties** interface:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab A \Ex2\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Update the MeasureDataDevice class to implement the IMeasuringDeviceWithProperties interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the MeasureDataDevice.cs file:
   - In Solution Explorer, double-click **MeasureDataDevice.cs**.
3. Remove the comment **TODO: Implement the IMeasuringDeviceWithProperties interface.**:
   - Delete the following line of code.

```
// TODO: Implement the IMeasuringDeviceWithProperties interface.
```

4. Modify the class declaration to implement the **IMeasuringDeviceWithProperties** interface instead of the **ILoggingMeasuringDevice** interface.

   The **IMeasuringDeviceWithProperties** interface inherits from the **ILoggingMeasuringDevice** interface, so modifying the declaration will not break compatibility with existing applications; the class can still be cast as an instance of the **ILoggingMeasuringDevice** interface.

   Your code should resemble the following code example.

```
public abstract class MeasureDataDevice :
IMeasuringDeviceWithProperties, IDisposable
{
    ...
}
```

5. Remove the comment **TODO: Add properties specified by the IMeasuringDeviceWithProperties interface.**:
   - Delete the following line of code.

MCT USE ONLY. STUDENT USE PROHIBITED

You will use the Implement Interface Wizard in the next step to add the properties.

```
// TODO: Add properties specified by the
    IMeasuringDeviceWithProperties interface.
```

6. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDeviceWithProperties** interface:
   - Right-click **IMeasuringDeviceWithProperties**, point to **Implement Interface**, and then click **Implement Interface**.
7. Locate the **UnitsToUse** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **UnitsToUse** property to return the **unitsToUse** field.

   Your code should resemble the following code example.

```
public Units UnitsToUse
{
    get
    {
        return unitsToUse;
    }
}
```

8. Locate the **DataCaptured** property **get** accessor, and then remove the default that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **DataCaptured** property to return the **dataCaptured** field.

   Your code should resemble the following code example.

```
public int[] DataCaptured
{
    get
    {
        return dataCaptured;
    }
}
```

9. Locate the **MostRecentMeasure** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **MostRecentMeasure** property to return the **mostRecentMeasure** field.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public int MostRecentMeasure
{
    get
    {
        return mostRecentMeasure;
    }
}
```

10. Locate the **LoggingFileName** property **get** accessor, and then remove the default body that throws a **NotImplementedException** exception. Add code to the **get** accessor of the **LoggingFileName** property to return the **loggingFileName** field.

    Your code should resemble the following code example.

```
public string LoggingFileName
{
    get
    {
        return loggingFileName;
    }
    set
    {
        throw new NotImplementedException();
    }
}
```

11. Modify the **set** accessor of the **LoggingFileName** property as shown in the following code example.



Note: A code snippet is available, called Mod10LoggingFileNamePropertySetAccessor, that you can use to add this code.

```
if (loggingFileWriter == null)
{
    // If the file has not been opened simply update the file name.
    loggingFileName = value;
}
else
{
    // If the file has been opened close the current file first,
    // then update the file name and open the new file.
    loggingFileWriter.WriteLine("Log File Changed");
    loggingFileWriter.WriteLine("New Log File: {0}", value);
    loggingFileWriter.Close();
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Now update the logging file and open the new file.
    loggingFileName = value;

    // 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");
    }

    loggingFileWriter.WriteLine("Log File Changed Successfully");

}
```

- To use the Mod10LoggingFileNamePropertySetAccessor snippet, remove the statement that throws the **NotImplementedException** exception, and after the opening brace of the **set** accessor, type **Mod10LoggingFileNamePropertySetAccessor** and then press the TAB key.

The **set** accessor for the **LoggingFileName** property checks whether the log file is currently open. If the log file has not been opened, the **set** accessor simply updates the local field. However, if the log file has been opened, the accessor closes the current log file and opens a new log file with the new file name in addition to updating the local field.

12. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Using Properties Exposed by a Class

Task 1: Add the test harness to the solution

The test harness application for this lab is a simple Windows® Presentation Foundation (WPF) application that is designed to test the functionality of the **MeasureDataDevice** class that you have just modified. 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. Add the test harness to the solution. The test harness is a project called Exercise3TestHarness, located in the E:\Labfiles\Lab 10\Lab A\Ex3 \Starter\Exercise3TestHarness folder:

    a. In Solution Explorer, right-click the **Solution 'Module 10'** node, point to **Add**, and then click **Existing Project**.

    b. In the **Add Existing Project** dialog box, move to the **E:\Labfiles\Lab 10 \Lab A\Ex3\Starter\Exercise3TestHarness** folder, click the **Exercise3TestHarness** project file, and then click **Open**.

2. Set the Exercise3TestHarness project as the startup project for the solution:

    - In Solution Explorer, right-click **Exercise3TestHarness**, and then click **Set as Startup Project**.

Task 2: Update the test harness

1. Review the task list:

    a. If the task list is not already visible, on the **View** menu, click **Task List**.

    b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. Review the user interface for the test application:

    - In Solution Explorer, double-click **MainWindow.xaml**.

    The test harness application includes functionality to enable you to test the properties you developed in the previous exercise. The **Start Collecting** button creates a new instance of the **MeasureMassDevice** object and starts collecting measurements from the emulated device. The application includes text boxes that display the output from the application. It also includes an **Update** button to enable you to update the file name of the log file. Finally, the test harness includes a button to stop the collection of measurements from the emulated device and dispose of the object.

3. Open the MainWindow.xaml.cs file:
   - In Solution Explorer, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.



Note: In the following steps, you will store values in the **Text** property of **TextBox** controls in the WPF window. This is a **string** property. In some of the steps, you may need to call the **ToString** method to convert the property to a **string**.

4. Remove the comment **TODO: Add code to set the unitsBox to the current units.**:
   - Delete the following line of code.

```
// TODO: Add code to set the unitsBox to the current units.
```

5. Locate the following line of code.

```
unitsBox.Text = "";
```

6. Update the code you located in the previous step to set the **Text** property of the **unitsBox** object to the **UnitsToUse** property of the **device** object.

   Your code should resemble the following code example.

```
unitsBox.Text = device.UnitsToUse.ToString();
```

7. Remove the comment **TODO: Add code to set the mostRecentMeasureBox to the value from the device.**:
   - Delete the following line of code.

```
// TODO: Add code to set the mostRecentMeasureBox to the value from
    the device.
```

8. Locate the following line of code.

```
mostRecentMeasureBox.Text = "";
```

9. Update the code you located in the previous step to set the **Text** property of the **mostRecentMeasureBox** object to the **MostRecentMeasure** property of the **device** object.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
mostRecentMeasureBox.Text = device.MostRecentMeasure.ToString();
```

10. Remove the comment **TODO: Update to use the LoggingFileName property.**:
    - Delete the following line of code.

```
// TODO: Update to use the LoggingFileName property.
```

11. Locate the following line of code.

```
loggingFileNameBox.Text =
    device.GetLoggingFile().Replace(labFolder, "");
```

12. Update the code you located in the previous step to set the **Text** property of the **loggingFileNameBox** object to the **LoggingFileName** property of the **device** object. Your code should call the **Replace** method of the **string** class in the same way as the code you are updating.

    Your code should resemble the following code example.

```
loggingFileNameBox.Text = device.LoggingFileName.Replace(labFolder,
"");
```

13. Remove the comment **TODO: Update to use the DataCaptured property.**:
    - Delete the following line of code.

```
// TODO: Update to use the DataCaptured property.
```

14. Locate the following line of code.

```
rawDataValues.ItemsSource = device.GetRawData();
```

15. Update the code you located in the previous step to set the **ItemsSource** property of the **rawDataValues** object to the **DataCaptured** property of the **device** object.

    Your code should resemble the following code example.

```
rawDataValues.ItemsSource = device.DataCaptured;
```

16. In the **updateButton_Click** method, remove the comment **TODO: Add code to update the log file name property of the device** and add code to set the

MCT USE ONLY. STUDENT USE PROHIBITED

LoggingFileName property of the **device** object to the concatenation of the **labFolder** field and the **Text** property of the **loggingFileNameBox** box.

Your code should resemble the following code example.

```
if (device != null)
{
    device.LoggingFileName = labFolder + loggingFileNameBox.Text;
}
```

17. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Test the properties by using the test harness

1. Start the Exercise3TestHarness application:
    - On the **Debug** menu, click **Start Without Debugging**.
2. Click **Start Collecting**. This action causes the application to pause for 10 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.
3. Using Windows Explorer, move to the **E:\Labfiles\Lab 10\Lab A** folder, and then verify that the default logging file, LogFile.txt, has been created:
    a. In the taskbar, click the **Windows Explorer** icon.
    b. In Windows Explorer, move to the **E:\Labfiles\Lab 10\Lab A** folder.
4. Return to the Exercise3TestHarness window. Wait at least a further 10 seconds to ensure that the emulated device has generated some additional values before you perform the following steps.
5. Change the log file to **LogFile2.txt**, and then click **Update**:
    - In the **Logging File** box, type **LogFile2.txt** and then click **Update**.

    The **Update** button calls the code you added to set the **LoggingFileName** property of the device; because the device is running, and therefore logging values to the log file, the code will close the current log file and open a new one with the name you specified.
6. Wait at least 10 seconds to ensure that the emulated device has generated some additional values before you perform the following steps.
7. Using Windows Explorer, move to the **E:\Labfiles\Lab 10\Lab A** folder, and then verify that the new logging file, LogFile2.txt, has been created.

MCT USE ONLY. STUDENT USE PROHIBITED

8. Return to the Exercise3TestHarness window, and then click **Stop Collecting / Dispose Object**.
9. Close the Exercise3TestHarness window.
10. Close Visual Studio:
    - In Visual Studio, on the **File** menu, click **Exit**.
11. Using Notepad, open the LogFile.txt file in the E:\Labfiles\Lab 10\Lab A folder:
    a. Click **Start**, point to **All Programs**, click **Accessories**, and then click **Notepad**.
    b. In Notepad, on the **File** menu, click **Open**.
    c. In the **Open** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 10\Lab A** folder, click **LogFile.txt**, and then click **Open**.
12. Review the contents of the LogFile.txt file.

    The file includes the values originally displayed in the test harness in addition to some not displayed. The file then indicates that the log file has changed and gives the name of the new log file.
13. Open the LogFile2.txt file in the E:\Labfiles\Lab 10\Lab A folder:
    a. On the **File** menu, click **Open**.
    b. In the **Open** dialog box, in the **File name** box, click **LogFile2.txt**, and then click **Open**.
14. Review the contents of the LogFile2.txt file.

    The file indicates that the log file has changed successfully. The file then includes any measurements taken after the log file changed and finally indicates that collection has stopped and the object was disposed of.
15. Close Notepad:
    - On the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Encapsulating Data and Defining Overloaded Operators

Lab B: Creating and Using Indexers

Exercise 1: Implementing an Indexer to Access Bits in a Control Register

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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab B\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab B \Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Add an indexer to the ControlRegister class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the ControlRegister.cs file:
   - In Solution Explorer, double-click **ControlRegister.cs**.
3. Remove the comment **TODO: Add an indexer to enable access to individual bits in the control register** and add a **public** indexer to the class. The indexer should take an **int** called **index** as the parameter and return an **int**.

   Your code should resemble the following code example.

```
public int this[int index]
{
}
```

4. Add a **get** accessor to the indexer. In the **get** accessor, add code to determine whether the bit specified by the *index* parameter in the **registerData** object is set to 1 or 0 and return the value of this bit.



Hint: Use the logical **AND** operator (**&**) and the left-shift operator (**<<**) to determine whether the result of left-shifting the value in the **registerData** object by the value of the **index** object is zero or non-zero. If the result is zero, return 0; otherwise, return 1. You can use the following code example to assist you with this step.

```
// Incomplete—Use this as part of your solution.
(registerData & (1 << index)) != 0
```

Your code should resemble the following code example.

```
public int this[int index]
{
    get
    {
        bool isSet = (registerData & (1 << index)) != 0;
        return isSet ? 1 : 0;
    }
}
```

5. Add a **set** accessor to the indexer. In the **set** accessor, add code to verify that the parameter specified is either 1 or 0. Throw an **ArgumentException** exception with the message "Argument must be 1 or 0" if it is not one of these values.

Your code should resemble the following code example.

```
public int this[int index]
{
    get
    {
        bool isSet = (registerData & (1 << index)) != 0;
        return isSet ? 1 : 0;
    }
    set
    {
        if (value != 0 && value !=1)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        {
            throw new ArgumentException("Argument must be 1 or 0");
        }
    }
}
```

6. In the **set** accessor, if **value** is 1, add code to set the bit specified by the **index** object in the **registerData** field to 1; otherwise, set this bit to 0.



Hint: Use the compound assignment operators **|=** and **&=** to set a specified bit in an integer value to 1 or 0. Use the expression **(1 << index)** to determine which bit in the integer value to set.

Your code should resemble the following code example.

```
public int this[int index]
{
    get
    {
        bool isSet = (registerData & (1 << index)) != 0;
        return isSet ? 1 : 0;
    }
    set
    {
        if (value != 0 && value !=1)
        {
            throw new ArgumentException("Argument must be 1 or 0");
        }

        if (value == 1)
            registerData |= (1 << index);
        else
            registerData &= ~(1 << index);
    }
}
```

7. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Using an Indexer Exposed by a Class

Task 1: Add the test harness to the solution

The test harness application for this lab is a simple console application that is designed to test the functionality of the **ControlRegister** class to which you have added an indexer. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class you have developed.

1. Add the test harness to the solution. The test harness is a project called Exercise2TestHarness, located in the E:\Labfiles\Lab 10\Lab B\Ex2 \Starter\Exercise2TestHarness folder:
    a. In Solution Explorer, right-click the **Solution 'Module 10'** node, point to **Add**, and then click **Existing Project**.
    b. In the **Add Existing Project** dialog box, move to the **E:\Labfiles\Lab 10 \Lab B\Ex2\Starter\Exercise2TestHarness** folder, click the **Exercise2TestHarness** project file, and then click **Open**.
2. Set the Exercise2TestHarness project as the startup project for the solution:
    - In Solution Explorer, right-click **Exercise2TestHarness**, and then click **Set as Startup Project**.

Task 2: Update the test harness

1. Review the task list:
    a. If the task list is not already visible, on the **View** menu, click **Task List**.
    b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the Program.cs file:
    - In Solution Explorer, double-click **Program.cs**.
3. Remove the TODO comment:
    - Delete the following line of code.

```
// TODO: Add code to test the ControlRegister class and indexer.
```

4. Add code to create a new instance of the **ControlRegister** class called **register**.

    Your code should resemble the following code example.

```
ControlRegister register = new ControlRegister();
```

5. Add code to set the **RegisterData** property of the **register** object to **8**.

   Your code should resemble the following code example.

```
ControlRegister register = new ControlRegister();
register.RegisterData = 8;
```

6. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: A code snippet is available, called Mod10WriteRegisterData, that you can use to add this code.

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

   - To use the Mod10WriteRegisterData snippet, on a blank line, type **Mod10WriteRegisterData** and then press the TAB key.

7. Add a statement to write the message "Set Bit 1 to 1" to the console.

   Your code should resemble the following code example.

```
ControlRegister register = new ControlRegister();
register.RegisterData = 8;

Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Console.WriteLine();

Console.WriteLine("Set Bit 1 to 1");
```

8. Add a statement to set the bit at index **1** in the **register** object to **1**.

   Your code should resemble the following code example.

```
...

Console.WriteLine("Set Bit 1 to 1");
register[1] = 1;
```

9. Add code to write a blank line to the console.

   Your code should resemble the following code example.

```
...

Console.WriteLine("Set Bit 1 to 1");
register[1] = 1;
Console.WriteLine();
```

10. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: You can use the Mod10WriteRegisterData code snippet to add this code.

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

   - To use the Mod10WriteRegisterData snippet, on a blank line, type **Mod10WriteRegisterData** and then press the TAB key.

11. Add a statement to write the message "Set Bit 0 to 1" to the console.

    Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
Console.WriteLine("Set Bit 1 to 1");
register[1] = 1;
Console.WriteLine();
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();

Console.WriteLine("Set Bit 0 to 1");
```

12. Add code to set the bit at index **0** in the **register** object to **1**.

    Your code should resemble the following code example.

```
...

Console.WriteLine("Set Bit 0 to 1");
register[0] = 1;
```

13. Add code to write a blank line to the console.

    Your code should resemble the following code example.

```
...

Console.WriteLine("Set Bit 0 to 1");
register[0] = 1;
Console.WriteLine();
```

14. Add the following code, which writes the current value for the **RegisterData** property and uses the indexer to write the first eight bits of the **ControlRegister** object to the console.



Note: You can use the Mod10WriteRegisterData code snippet to add this code.

```
Console.WriteLine("RegisterData: {0}", register.RegisterData);
Console.WriteLine("Bit 0: {0}", register[0].ToString());
Console.WriteLine("Bit 1: {0}", register[1].ToString());
Console.WriteLine("Bit 2: {0}", register[2].ToString());
Console.WriteLine("Bit 3: {0}", register[3].ToString());
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Console.WriteLine("Bit 4: {0}", register[4].ToString());
Console.WriteLine("Bit 5: {0}", register[5].ToString());
Console.WriteLine("Bit 6: {0}", register[6].ToString());
Console.WriteLine("Bit 7: {0}", register[7].ToString());
Console.WriteLine();
```

- To use the Mod10WriteRegisterData snippet, on a blank line, type **Mod10WriteRegisterData** and then press the TAB key.

15. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Test the ControlRegister class by using the test harness

1. Start the Exercise2TestHarness application:
    - On the **Debug** menu, click **Start Without Debugging**.
2. Verify that the output from the console appears correctly. The output should resemble the following code example.

```
RegisterData : 8
Bit 0: 0
Bit 1: 0
Bit 2: 0
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0

Set Bit 1 to 1

RegisterData : 10
Bit 0: 0
Bit 1: 1
Bit 2: 0
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0

Set Bit 0 to 1

RegisterData : 11
Bit 0: 1
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Bit 1: 1
Bit 2: 0
Bit 3: 1
Bit 4: 0
Bit 5: 0
Bit 6: 0
Bit 7: 0
```

3. Close the Exercise2TestHarness window.
4. Close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Encapsulating Data and Defining Overloaded Operators

Lab C: Overloading Operators

Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types

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 Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab C\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab C \Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Create a Matrix class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Remove the comment **TODO: Add the Matrix class** and add a **public Matrix** class to the **MatrixOperators** namespace.

   Your code should resemble the following code example.

```
namespace MatrixOperators
{
    public class Matrix
    {
        // TODO Add an addition operator to the Matrix class.

        // TODO Add a subtraction operator to the Matrix class.

        // TODO Add a multiplication operator to the Matrix class.
    }

    // TODO: Add the MatrixNotCompatibleException exception class.

}
```

4. Add a two-dimensional array of integers named **data** to the **Matrix** class.

   Your code should resemble the following code example.

```
public class Matrix
{
    int[,] data;

    // TODO Add an addition operator to the Matrix class.

    // TODO Add a subtraction operator to the Matrix class.

    // TODO Add a multiplication operator to the Matrix class.
}
```

5. Add a **public** constructor to the **Matrix** class. The constructor should take a single integer parameter called *size* and initialize the **data** array to a square array by using the value passed to the constructor as the size of each dimension of the array.

   Your code should resemble the following code example.

```
public class Matrix
{
    int[,] data;

    public Matrix(int size)
    {
        data = new int[size, size];
    }
    ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

6. After the constructor, add the following code to add an indexer to the class. You can either type this code manually, or you can use the Mod10MatrixClassIndexer code snippet.

```
public int this[int RowIndex, int ColumnIndex]
{
    get
    {
        if (RowIndex > data.GetUpperBound(0) ||
            ColumnIndex > data.GetUpperBound(0))
        {
            throw new IndexOutOfRangeException();
        }

        else
        {
            return data[RowIndex, ColumnIndex];
        }

    }
    set
    {
        if (RowIndex > data.GetUpperBound(0) ||
            ColumnIndex > data.GetUpperBound(0))
        {
            throw new IndexOutOfRangeException();
        }

        else
        {
            data[RowIndex, ColumnIndex] = value;
        }

    }
}
```

   The indexer takes two parameters, one that indicates the row, and another that indicates the column. The indexer checks that the values are in range for the current matrix (that they are not bigger than the matrix) and then returns the value of the indexed item from the data array.

   - To use the code snippet, type **Mod10MatrixClassIndexer** and then press the TAB key.

7. After the indexer, add the following code to override the **ToString** method of the **Matrix** class. You can either type this code manually, or you can use the Mod10MatrixClassToStringMethod code snippet.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public override string ToString()
{
    StringBuilder builder = new StringBuilder();

    // Iterate over every row in the matrix.
    for (int x = 0; x < data.GetLength(0); x++)
    {
        // Iterate over every column in the matrix.
        for (int y = 0; y < data.GetLength(1); y++)
        {
            builder.AppendFormat("{0}\t", data[x, y]);
        }
        builder.Append(Environment.NewLine);
    }
    return builder.ToString();
}
```

- To use the code snippet, type **Mod10MatrixClassToStringMethod** and then press the TAB key.

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Create a MatrixNotCompatibleException exception class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. If it is not already open, open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Remove the comment **TODO: Add the MatrixNotCompatibleException exception class** and add a **public MatrixNotCompatibleException** class to the **MatrixOperators** namespace.

   Your code should resemble the following code example.

```
namespace MatrixOperators
{
    public class Matrix
    {
        ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }
    public class MatrixNotCompatibleException
    {
    }
}
```

4. Modify the **MatrixNotCompatibleException** class to inherit from the **Exception** class.

   Your code should resemble the following code example.

```
public class MatrixNotCompatibleException : Exception
{ }
```

5. Add a field of type **Matrix** called **firstMatrix** to the **MatrixNotCompatibleException** class and instantiate it to **null**.

   Your code should resemble the following code example.

```
public class MatrixNotCompatibleException : Exception
{
    Matrix firstMatrix = null;
}
```

6. Add a field of type **Matrix** called **secondMatrix** to the **MatrixNotCompatibleException** class and instantiate it to **null**.

   Your code should resemble the following code example.

```
public class MatrixNotCompatibleException : Exception
{
    Matrix firstMatrix = null;
    Matrix secondMatrix = null;
}
```

7. Add a property of type **Matrix** called **FirstMatrix** to the **MatrixNotCompatibleException** class, and then add a **get** accessor that returns the **firstMatrix** field.

   Your code should resemble the following code example.

```
public class MatrixNotCompatibleException : Exception
{
    Matrix firstMatrix = null;
    Matrix secondMatrix = null;
    public Matrix FirstMatrix
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        get
        {
            return firstMatrix;
        }
    }
}
```

8. Add a property of type **Matrix** called **SecondMatrix** to the **MatrixNotCompatibleException** class, and then add a **get** accessor that returns the **secondMatrix** field.

   Your code should resemble the following code example.

```
public class MatrixNotCompatibleException : Exception
{
    Matrix firstMatrix = null;
    Matrix secondMatrix = null;

    public Matrix FirstMatrix
    {
        get
        {
            return firstMatrix;
        }
    }

    public Matrix SecondMatrix
    {
        get
        {
            return secondMatrix;
        }
    }
}
```

9. Add the following constructors to the **MatrixNotCompatibleException** class. You can either type this code manually, or you can use the Mod10MatrixNotCompatibleExceptionClassConstructors code snippet.

```
public MatrixNotCompatibleException()
    : base()
{
}

public MatrixNotCompatibleException(string message)
    : base(message)
{
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
public MatrixNotCompatibleException(string message,
    Exception innerException)
    : base(message, innerException)
{
}

public MatrixNotCompatibleException(SerializationInfo info,
    StreamingContext context)
    : base(info, context)
{
}
```

- To use the code snippet, type **Mod10MatrixNotCompatibleExceptionClassConstructors** and then press the TAB key.

10. Add a constructor to the **MatrixNotCompatibleException** class. The constructor should take two **Matrix** objects and a **string** object as parameters. The constructor should use the **string** object to call the **base** constructor and instantiate the **matrix1** and **matrix2** fields by using the *Matrix* parameters.

    Your code should resemble the following code example.

```
public MatrixNotCompatibleException(Matrix matrix1,
    Matrix matrix2, string message)
    : base(message)
{
    firstMatrix = matrix1;
    secondMatrix = matrix2;
}
```

11. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

    At the end of the exercise, your code should resemble the following code example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace MatrixOperators
{
    public class Matrix
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        int[,] data;
        public Matrix(int Size)
        {
            data = new int[Size, Size];
        }
        public int this[int RowIndex, int ColumnIndex]
        {
            get
            {
                if (RowIndex > data.GetUpperBound(0) ||
                    ColumnIndex > data.GetUpperBound(0))
                {
                    throw new IndexOutOfRangeException();
                }
                else
                {
                    return data[RowIndex, ColumnIndex];
                }
            }
            set
            {
                if (RowIndex > data.GetUpperBound(0) ||
                    ColumnIndex > data.GetUpperBound(0))
                {
                    throw new IndexOutOfRangeException();
                }
                else
                {
                    data[RowIndex, ColumnIndex] = value;
                }
            }
        }
        public override string ToString()
        {
            StringBuilder builder = new StringBuilder();

            // Iterate over every row in the matrix.
            for (int x = 0; x < data.GetLength(0); x++)
            {
                // Iterate over every column in the matrix.
                for (int y = 0; y < data.GetLength(1); y++)
                {
                    builder.AppendFormat("{0}\t", data[x, y]);
                }
                builder.Append(Environment.NewLine);
            }

            return builder.ToString();
        }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }
    public class MatrixNotCompatibleException : Exception
    {
        Matrix firstMatrix = null;
        Matrix secondMatrix = null;
        public Matrix FirstMatrix
        {
            get
            {
                return firstMaxtrix;
            }
        }
        public Matrix SecondMatrix
        {
            get
            {
                return secondMaxtrix;
            }
        }
        public MatrixNotCompatibleException()
            : base()
        {
        }
        public MatrixNotCompatibleException(string message)
            : base(message)
        {
        }
        public MatrixNotCompatibleException(string message,
            Exception innerException)
            : base(message, innerException)
        {
        }

        public MatrixNotCompatibleException(SerializationInfo info,
            StreamingContext context)
            : base(info, context)
        {
        }

        public MatrixNotCompatibleException(Matrix matrix1,
            Matrix matrix2, string message)
            : base(message)
        {
            firstMatrix = matrix1;
            secondMatrix = matrix2;
        }
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Implementing Operators for the Matrix Type

Task 1: Open the starter project



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **Matrix** and **MatrixNotCompatibleException** types successfully, proceed directly to **Task 2: Add an addition operator to the Matrix class**.

- Open the Module10 solution in the E:\Labfiles\Lab 10\Lab C\Ex2\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 10\Lab C\Ex2\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Add an addition operator to the Matrix class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Replace the comment **TODO Add an addition operator to the Matrix class** with an overload of the **+** operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.

   Your code should resemble the following code example.

```
public static Matrix operator +(Matrix matrix1, Matrix matrix2)
{
}
```

4. Add code to the **+** operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new

MatrixNotCompatibleException exception, by using the matrices and the message "Matrices not the same size" as parameters.

Your code should resemble the following code example.

```
public static Matrix operator +(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

5. If both matrices are the same size, add code that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as either of the source matrices.

   Your code should resemble the following code example.

```
public static Matrix operator +(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

6. Add code to iterate over every item in the first matrix. For each item in the first matrix, calculate the sum of this item and the corresponding item in the second matrix, and store the result in the corresponding position in the **newMatrix** matrix.



Hint: Use a **for** loop to iterate over the rows in the first matrix and a nested **for** loop to iterate over the columns in each row.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public static Matrix operator +(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));
        // Iterate over every row in the matrix.
        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                newMatrix.data[x, y] =
                    matrix1.data[x, y] + matrix2.data[x, y];
            }
        }
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the **+** operator.

   Your code should resemble the following code example.

```
public static Matrix operator +(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));

        // Iterate over every row in the matrix.
        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                newMatrix.data[x, y] =
                    matrix1.data[x, y] + matrix2.data[x, y];
            }
        }
        return newMatrix;
    }
    else
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Add a subtraction operator to the Matrix class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. If it is not already open, open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Replace the comment **TODO Add a subtraction operator to the Matrix class** with an overload of the **-** operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.

   Your code should resemble the following code example.

```
public static Matrix operator -(Matrix matrix1, Matrix matrix2)
{
}
```

4. Add code to the **-** operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new **MatrixNotCompatibleException** exception, by using the matrices and the message "Matrices not the same size" as parameters.

   Your code should resemble the following code example.

```
public static Matrix operator -(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
    }
    else
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

5. If both matrices are the same size, add code that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as either of the source matrices.

   Your code should resemble the following code example.

```
public static Matrix operator -(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

6. Add code to iterate over every item in the first matrix. For each item in the first matrix, calculate the difference between this item and the corresponding item in the second matrix, and store the result in the corresponding position in the **newMatrix** matrix.

   Your code should resemble the following code example.

```
public static Matrix operator -(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));

        // Iterate over every row in the matrix.
        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                newMatrix.data[x, y] =
                    matrix1.data[x, y] - matrix2.data[x, y];
            }
        }
    }
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the **-** operator.

   Your code should resemble the following code example.

```
public static Matrix operator -(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));

        // Iterate over every row in the matrix.

        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                newMatrix.data[x, y] =
                    matrix1.data[x, y] - matrix2.data[x, y];
            }
        }

        return newMatrix;
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

8. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Add a multiplication operator to the Matrix class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.

MCT USE ONLY. STUDENT USE PROHIBITED

b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. If it is not already open, open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Replace the comment **TODO Add a multiplication operator to the Matrix class** with an overload of the * operator that takes two **Matrix** objects as parameters and returns an instance of the **Matrix** class.

   Your code should resemble the following code example.

```
public static Matrix operator *(Matrix matrix1, Matrix matrix2)
{
}
```

4. Add code to the * operator to check that each of the matrices are the same size (the **Matrix** class only supports square matrices, so you only need to check one dimension of the matrix). If they are not the same size, throw a new **MatrixNotCompatibleException** exception, by using the matrices and the message "Matrices not the same size" as parameters.

   Your code should resemble the following code example.

```
public static Matrix operator *(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

5. Add code to the conditional block that creates a new instance of the **Matrix** class named **newMatrix** and initialize it to a matrix with the same size as the source matrices.

   Your code should resemble the following code example.

```
public static Matrix operator *(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

6. Add code to iterate over every item in the first matrix and calculate the product of the two matrices, storing the result in the **newMatrix** matrix. Remember that to calculate each element $x_{a,b}$ in **newMatrix**, 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.

   Your code should resemble the following code example.

```
public static Matrix operator *(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));

        // Iterate over every row in the matrix.
        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                int temp = 0;
                for (int z = 0; z < matrix1.data.GetLength(0); z++)
                {
                    temp += matrix1.data[x, z] * matrix2.data[z, y];
                }
                newMatrix.data[x, y] = temp;
            }
        }
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

7. After the code that calculates the values for the **newMatrix** object, add a statement that returns the **newMatrix** object as the result of the * operator.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public static Matrix operator *(Matrix matrix1, Matrix matrix2)
{
    if (matrix1.data.GetLength(0) == matrix2.data.GetLength(0))
    {
        Matrix newMatrix = new Matrix(matrix1.data.GetLength(0));

        // Iterate over every row in the matrix.
        for (int x = 0; x < matrix1.data.GetLength(0); x++)
        {
            // Iterate over every column in the matrix.
            for (int y = 0; y < matrix1.data.GetLength(1); y++)
            {
                int temp = 0;
                for (int z = 0; z < matrix1.data.GetLength(0); z++)
                {
                    temp += matrix1.data[x, z] * matrix2.data[z, y];
                }
                newMatrix.data[x, y] = temp;
            }
        }
        return newMatrix;
    }
    else
    {
        throw new MatrixNotCompatibleException
            (matrix1, matrix2, "Matrices not the same size");
    }
}
```

8. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

Exercise 3: Testing the Operators for the Matrix Type

Task 1: Add the test harness to the solution

The test harness application for this lab is a simple console application that is designed to test the functionality of the **Matrix** class. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class you have developed.

1. Add the test harness to the solution. The test harness is a project called Exercise3TestHarness, located in the E:\Labfiles\Lab 10\Lab C\Ex3 \Starter\Exercise3TestHarness folder:

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Solution Explorer, right-click the **Solution 'Module 10'** node, point to **Add**, and then click **Existing Project**.

b. In the **Add Existing Project** dialog box, move to the **E:\Labfiles\Lab 10 \Lab C\Ex3\Starter\Exercise3TestHarness** folder, click the **Exercise3TestHarness** project file, and then click **Open**.

2. Set the Exercise3TestHarness project as the startup project for the solution:

   - In Solution Explorer, right-click **Exercise3TestHarness**, and then click **Set as Startup Project**.

Task 2: Add code to test the operators in the Matrix class

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. Open the Program.cs file:

   - In Solution Explorer, double-click **Program.cs**.

3. Review the **Main** method.

   This method creates two 3×3 square matrices called **matrix1** and **matrix2** and populates them with sample data. The method then displays their contents to the console by using the **ToString** method.

4. Remove the TODO comment:

   - Delete the following line of code.

```
// TODO: Add code to test the operators in the Matrix class.
```

5. Add a statement to write the message "Matrix 1 + Matrix 2:" to the console.

   Your code should resemble the following code example.

```
Console.WriteLine("Matrix 1 + Matrix 2:");
```

6. Add a statement to create a new **Matrix** object called **matrix3** and populate it with the sum of the **matrix1** and **matrix2** objects.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
Matrix matrix3 = matrix1 + matrix2;
```

7. Add code to write the contents of the **matrix3** matrix to the console, followed by a blank line.

   Your code should resemble the following code example.

```
Console.WriteLine(matrix3.ToString());
Console.WriteLine();
```

8. Add a statement to write the message "Matrix 1 - Matrix 2:" to the console.

   Your code should resemble the following code example.

```
Console.WriteLine("Matrix 1 - Matrix 2:");
```

9. Add code to create a new **Matrix** object called **matrix4** and populate it with the difference between the **matrix1** and **matrix2** objects (subtract **matrix2** from **matrix1**).

   Your code should resemble the following code example.

```
Matrix matrix4 = matrix1 - matrix2;
```

10. Add code to write the contents of the **matrix4** matrix to the console, followed by a blank line.

    Your code should resemble the following code example.

```
Console.WriteLine(matrix4.ToString());
Console.WriteLine();
```

11. Add a statement to write the message "Matrix 1 × Matrix 2:" to the console.

    Your code should resemble the following code example.

```
Console.WriteLine("Matrix 1 x 2:");
```

12. Add code to create a new **Matrix** object called **matrix5** and populate it with the product of the **matrix1** and **matrix2** objects.

    Your code should resemble the following code example.

```
Matrix matrix5 = matrix1 * matrix2;
```

MCT USE ONLY. STUDENT USE PROHIBITED

13. Add code to write the contents of the **matrix5** matrix to the console, followed by a blank line.

    Your code should resemble the following code example.

```
Console.WriteLine(matrix5.ToString());
Console.WriteLine();
```

14. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Test the matrix operators by using the test harness

1. Start the Exercise3TestHarness application:

    - On the **Debug** menu, click **Start Without Debugging**.

2. Verify that the output from the console appears correctly. The output should resemble the following.

```
Matrix 1:
1     2       3
4     5       6
7     8       9

Matrix 2:
9     8       7
6     5       4
3     2       1

Matrix 1 + 2:
10    10      10
10    10      10
10    10      10

Matrix 1 - 2:
-8    -6      -4
-2    0       2
4     6       8

Matrix 1 x 2:
30    24      18
84    69      54
138   114     90
```

3. Close the console window.

MCT USE ONLY. STUDENT USE PROHIBITED

4. Close Visual Studio:
    - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Decoupling Methods and Handling Events

Lab 11: Decoupling Methods and Handling Events

Exercise 1: Raising and Handling Events

Task 1: Open the Events solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Events solution in the E:\Labfiles\Lab 11\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 11\Ex1\Starter** folder, click **Events.sln**, and then click **Open**.

Task 2: Create a new interface that extends the IMeasuringDevice interface

1. In the MeasuringDevice project, add a new interface named **IEventEnabledMeasuringDevice** in a file named IEventEnabledMeasuringDevice.cs:
   a. In Solution Explorer, right-click the **MeasuringDevice** project, point to **Add**, and then click **New Item**.
   b. In the **Add New Item - MeasuringDevice** dialog box, under **Installed Templates**, click **Code**, and in the template list, click **Interface**.
   c. In the **Name** box, type **IEventEnabledMeasuringDevice.cs** and then click **Add**.



Note: Creating a new interface that extends an existing interface is good programming practice, because it preserves the structure of the original interface for backward

compatibility with preexisting code. All preexisting code can reference the original interface, and new code can reference the new interface and take advantage of any new functionality.

2. Modify the interface definition so that the **IEventEnabledMeasuringDevice** interface extends the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
...
interface IEventEnabledMeasuringDevice : IMeasuringDevice
{
}
...
```

3. In the **IEventEnabledMeasuringDevice** interface, add an event named **NewMeasurementTaken** by using the base **EventHandler** delegate.

   Your code should resemble the following code example.

```
...
interface IEventEnabledMeasuringDevice : IMeasuringDevice
{
    event EventHandler NewMeasurementTaken;
}
...
```

4. Build the application to enable Microsoft IntelliSense to reflect your changes:
   - On the **Build** menu, click **Build Solution**, and then correct any errors.

Task 3: Add the NewMeasurementTaken event to the MeasureDataDevice class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Locate the **TODO - Modify the class definition to implement the extended interface** task, and then double-click this task. This task is located in the MeasureDataDevice class file.

MCT USE ONLY. STUDENT USE PROHIBITED

3. Remove the **TODO - Modify the class definition to implement the extended interface** comment, and then modify the class definition to implement the **IEventEnabledMeasuringDevice** interface instead of the **IMeasuringDevice** interface:

   - At the top of the code file, in the class definition, replace **IMeasuringDevice** with **IEventEnabledMeasuringDevice**.

   Your code should resemble the following code example.

```
...
public abstract class MeasureDataDevice :
    IEventEnabledMeasuringDevice, IDisposable
{ ...
```

4. In the task list, locate the **TODO - Add the NewMeasurementTaken event** task, and then double-click this task. This task is located at the end of the **MeasureDataDevice** class.

5. Remove the **TODO - Add the NewMeasurementTaken event** comment, and then declare an event named **NewMeasurementTaken** by using the same signature as the interface.

   Your code should resemble the following code example.

```
...
// class implementation of the NewMeasurementTaken event.
public event EventHandler NewMeasurementTaken;

// TODO - Add an OnMeasurementTaken method.
...
```

6. Below the event, remove the **TODO - Add an OnMeasurementTaken method** comment, and then add a protected virtual method named **OnNewMeasurementTaken**. The method should accept no parameters and have a void return type. The **MeasureDataDevice** class will use this method to raise the **NewMeasurementTaken** event.

   Your code should resemble the following code example.

```
...
    // Method to raise the NewMeasurementTaken event.
    protected virtual void OnNewMeasurementTaken()
    {
    }
} ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

7. In the **OnNewMeasurementTaken** method, add code to check that there is a subscriber for the **NewMeasurementTaken** event; if so, raise the event. The signature of the **EventHandler** delegate defines two parameters: an *object* parameter that indicates the object that raised the event and an *EventArgs* parameter that provides any additional data that is passed to the event handler. Set the *object* parameter to **this** and the *EventArgs* parameter to **null**.



Note: It is good programming practice to check that there are subscribers for an event before you raise it. If an event has no subscribers, the related delegate is null, and the Microsoft .NET Framework runtime will throw an exception if the event is raised.

Your code should resemble the following code example.

```
...
protected virtual void OnNewMeasurementTaken()
{
    if (NewMeasurementTaken != null)
    {
        NewMeasurementTaken(this, null);
    }
}
...
```

Task 4: Add a BackgroundWorker member to the MeasureDataDevice class

1. In the task list, locate the **TODO - Declare a BackgroundWorker to generate data** task, and then double-click this task. This task is located near the top of the **MeasureDataDevice** class.
2. Remove the **TODO - Declare a BackgroundWorker to generate data** comment, and then add a private **BackgroundWorker** member named **dataCollector** to the class.

   Your code should resemble the following code example.

```
...
// BackgroundWorker member to generate measurements.
private BackgroundWorker dataCollector;

/// <summary>
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 5: Add the GetMeasurements method to the MeasureDataDevice class

The **GetMeasurements** method will initialize the **dataCollector BackgroundWorker** member to poll for new measurements and raise the **NewMeasurementTaken** event each time it detects a new measurement.

1. In the task list, locate the **TODO - Implement the GetMeasurements method** task, and then double-click this task.
2. Remove the **TODO - Implement the GetMeasurements method** comment, and then add a new private method named **GetMeasurements** to the class. This method should take no parameters and not return a value.

   Your code should resemble the following code example.

```
...
// Add a GetMeasurements method to configure and start the
// BackgroundWorker.

private void GetMeasurements()
{
}
...
```

3. In the **GetMeasurements** method, add code to perform the following actions:
   a. Instantiate the **dataCollector BackgroundWorker** member.
   b. Specify that the **dataCollector BackgroundWorker** member supports cancellation.
   c. Specify that the **dataCollector BackgroundWorker** member reports progress while running.



Hint: Set the **WorkerSupportsCancellation** and **WorkerReportsProgress** properties.

Your code should resemble the following code example.

```
...
private void GetMeasurements()
{
    dataCollector = new BackgroundWorker();
    dataCollector.WorkerSupportsCancellation = true;
    dataCollector.WorkerReportsProgress = true;
} ...
```

4. Add the following code to instantiate a **DoWorkEventHandler** delegate that refers to a method called **dataCollector_DoWork**. Attach the delegate to the **DoWork** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_DoWork** method when the **DoWork** event is raised.



Hint: Use IntelliSense to generate a code stub for the **dataCollector_DoWork** method. To do this, type the first part of the line of code, up to the **+=** operators, and then press the TAB key twice. Visual Studio uses a built-in code snippet to complete the line of code and then add a method stub. You can do this each time you hook up an event handler to an event by using the **+=** compound assignment operator.

```
...
    dataCollector.WorkerReportsProgress = true;

    dataCollector.DoWork +=
        new DoWorkEventHandler(dataCollector_DoWork);
}
...
```

5. Using the same technique as in the previous step, instantiate a **ProgressChangedEventHandler** delegate that refers to a method called **dataCollector_ProgressChanged**. Attach this delegate to the **ProgressChanged** event property of the **dataCollector** member. The **dataCollector** object will call the **dataCollector_ProgressChanged** method when the **ProgressChanged** event is raised.

   Your code should resemble the following code example.

```
...
    dataCollector.DoWork +=
        new DoWorkEventHandler(dataCollector_DoWork);

    dataCollector.ProgressChanged += new
        ProgressChangedEventHandler(dataCollector_ProgressChanged);
}

...
```

6. Add code to start the **dataCollector BackgroundWorker** object running asynchronously.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...

    dataCollector.ProgressChanged += new
        ProgressChangedEventHandler(dataCollector_ProgressChanged);

    dataCollector.RunWorkerAsync();

}

...
```

Task 6: Implement the dataCollector_DoWork method

1. Underneath the **GetMeasurements** method, locate the **dataCollector_DoWork** method.

   This method was generated during the previous task. It runs on a background thread, and its purpose is to collect and store measurement data.

2. In the **dataCollector_DoWork** method, remove the statement that raises the **NotImplementedException** exception and add code to perform the following actions:

   a. Instantiate the **dataCaptured** array with a new integer array that contains 10 items.

   b. Define an integer i with an initial value of zero. You will use this variable to track the current position in the **dataCaptured** array.

   c. Add a **while** loop that runs until the **dataCollector**.**CancellationPending** property is **false**.

   Your code should resemble the following code example.

```
...
void dataCollector_DoWork(object sender, DoWorkEventArgs e)
{

    dataCaptured = new int[10];

    int i = 0;
    while (!dataCollector.CancellationPending)
    {
    }
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the **while** loop, add code to perform the following actions:
   a. Invoke the **controller**.**TakeMeasurement** method, and store the result in the **dataCaptured** array at the position that the integer **i** indicates. The **TakeMeasurement** method of the **controller** object blocks until a new measurement is available.
   b. Update the **mostRecentCapture** property to contain the value in the **dataCaptured** array at the position that the integer **i** indicates.
   c. If the value of the disposed variable is **true**, terminate the **while** loop. This step ensures that the measurement collection stops when the **MeasureDataDevice** object is destroyed.

   Your code should resemble the following code example.

```
...

while (!dataCollector.CancellationPending)
{
    dataCaptured[i] = controller.TakeMeasurement();
    mostRecentMeasure = dataCaptured[i];

    if (disposed)
    {
        break;
    }
}
...
```

4. Add code to the **while** loop after the statements that you added in the previous step to perform the following actions:
   a. Check whether the **loggingFileWriter** property is null.
   b. If the **loggingFileWriter** property is not null, call the **loggingFileWriter**.**Writeline** method, passing a string parameter of the format "Measurement - *mostRecentMeasure*" where *mostRecentMeasure* is the value of the mostRecentMeasure variable.



Note: The **loggingFileWriter** property is a simple **StreamWriter** object that writes to a text file. This property is initialized in the **StartCollecting** method. You can use the **WriteLine** method to write to a **StreamWriter** object.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
        break;
    }

    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine
            ("Measurement - {0}", mostRecentMeasure.ToString());
    }
}
...
```

5. Add a line of code to the end of the **while** loop to invoke the **dataCollector**.**ReportProgress** method, passing zero as the parameter.

   The **ReportProgress** method raises the **ReportProgress** event and is normally used to return the percentage completion of the tasks assigned to the **BackgroundWorker** object. You can use the **ReportProgress** event to update progress bars or time estimates in the user interface (UI). In this case, because the task will run indefinitely until canceled, you will use the **ReportProgress** event as a mechanism to prompt the UI to refresh the display with the new measurement.

   Your code should resemble the following code example.

```
...
        loggingFileWriter.WriteLine
            ("Measurement - {0}", mostRecentMeasure.ToString());
    }

    dataCollector.ReportProgress(0);
}

...
```

6. Add code to the end of the **while** loop to perform the following actions:

   a. Increment the integer **i**.

   b. If the value of the integer is greater than nine, reset **i** to zero.

      You are using the integer i as a pointer to the next position to write to in the **dataCaptured** array. This array has space for 10 measurements. When element 9 is filled, the device will start to overwrite data beginning at element 0.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
...
    dataCollector.ReportProgress(0);

    i++;
    if (i > 9)
    {
        i = 0;
    }
}
...
```

Task 7: Implement the dataCollector_ProgressChanged method

1. Locate the **dataCollector_ProgressChanged** method.

   This method was generated during an earlier task. It runs when the **ProgressChanged** event is raised. In this exercise, this event occurs when the **dataCollector_DoWork** method takes and stores a new measurement.

2. In the event handler, delete the exception code and then invoke the **OnNewMeasurementTaken** method, passing no parameters.

   The **OnNewMeasurementTaken** method raises the **NewMeasurementTaken** event that you defined earlier. You will modify the UI to subscribe to this event so that when it is raised, the UI can update the displayed information.

   Your code should resemble the following code example.

```
...
void dataCollector_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    OnNewMeasurementTaken();
}
...
```

Task 8: Call the GetMeasurements method to start collecting measurements

1. In the task list, locate the **TODO - Call the GetMeasurements method** task, and then double-click this task. This task is located in the **StartCollecting** method.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Remove the **TODO - Call the GetMeasurements method** comment, and add a line of code to invoke the **GetMeasurements** method.

   Your code should resemble the following code example.

```
    ...
    else
    {
        loggingFileWriter.WriteLine("Log file status checked – Already
open");
        loggingFileWriter.WriteLine("Collecting Started");
    }

    GetMeasurements();
}
...
```

Task 9: Call the CancelAsync method to stop collecting measurements

1. In the task list, locate the **TODO - Cancel the data collector** task, and then double-click this task. This task is located in the **StopCollecting** method.
2. Remove the **TODO - Cancel the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **CancelAsync** method to stop the work performed by the **dataCollector BackgroundWorker** object.

   Your code should resemble the following code example.

```
...
    }

    // Stop the data collection BackgroundWorker.

    if (dataCollector != null)
    {
        dataCollector.CancelAsync();
    }
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

Task 10: Dispose of the BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - Dispose of the data collector** task, and then double-click this task. This task is located in the **Dispose** method of the **MeasureDataDevice** class.
2. Remove the **TODO - Dispose of the data collector** comment and add code to perform the following actions:
   a. Check that the **dataCollector** member is not null.
   b. If the **dataCollector** member is not null, call the **Dispose** method to dispose of the **dataCollector** instance.

   Your code should resemble the following code example.

```
...
    }
    // Dispose of the dataCollector BackgroundWorker object.
    if (dataCollector != null)
    {
        dataCollector.Dispose();
    }
}
...
```

Task 11: Update the UI to handle measurement events

1. In the task list, locate the **TODO - Declare a delegate to reference NewMeasurementEvent** task, and then double-click this task. This task is located in the code behind the MainWindow.xaml window.
2. Remove the comment and add code to define a delegate of type **EventHandler** named **newMeasurementTaken**.

   Your code should resemble the following code example.

```
...
MeasureMassDevice device;

EventHandler newMeasurementTaken;

private void startCollecting_Click(object sender, RoutedEventArgs e)
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the **startCollecting_Click** method, remove the comment **TODO - use a delegate to refer to the event handler**, and add code to initialize the **newMeasurementTaken** delegate with a new **EventHandler** delegate that is based on a method named **device_NewMeasurementTaken**. You will create the **device_NewMeasurementTaken** method in the next task.



Note: You cannot use IntelliSense to automatically generate the stub for the **device_NewMeasurementTaken** method, as you did in earlier tasks.

Your code should resemble the following code example.

```
...

// Hook up the delegate to an event handler method.

newMeasurementTaken = new EventHandler(device_NewMeasurementTaken);

// TODO - Hook up the event handler to the event.

...
```

4. In the **startCollecting_Click** method, remove the **TODO - Hook up the event handler to the event** comment, and add code to connect the **newMeasurementTaken** delegate to the **NewMeasurementTaken** event of the **device** object. The **device** object is an instance of the **MeasureMassDevice** class, which inherits from the **MeasureDataDevice** abstract class.



Hint: To connect a delegate to an event, use the **+=** compound assignment operator on the event.

Your code should resemble the following code example.

```
...

newMeasurementTaken = new EventHandler(device_NewMeasurementTaken);
device.NewMeasurementTaken += newMeasurementTaken;
loggingFileNameBox.Text = device.GetLoggingFile();

...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 12: Implement the device_NewMeasurementTaken event-handling method

1. In the task list, locate the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** task, and then double-click this task.
2. Remove the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** comment, and add a private event-handler method named **device_NewMeasurementTaken**. The method should not return a value, but should take the following parameters:
   a. An **object** object named **sender**.
   b. An **EventArgs** object named **e**.

   Your code should resemble the following code example.

```
...
private void device_NewMeasurementTaken(object sender, EventArgs e)
{
}
private void updateButton_Click(object sender, RoutedEventArgs e)
...
```

3. In the **device_NewMeasurementTaken** method, add code to check that the **device** member is not null. If the **device** member is not null, perform the following tasks:
   a. Update the **Text** property of the **mostRecentMeasureBox** text box with the value of the **device.MostRecentMeasure** property.



Hint: Use the **ToString** method to convert the value that the **device.MostRecentMeasure** property returns from an integer to a string.

   b. Update the **Text** property of the **metricValueBox** text box with the value that the **device.MetricValue** method returns.
   c. Update the **Text** property of the **imperialValueBox** text box with the value that the **device.ImperialValue** method returns.
   d. Reset the **rawDataValues.ItemsSource** property to **null**.
   e. Set the **rawDataValues.ItemsSource** property to the value that the **device.GetRawData** method returns.



Note: The final two steps are both necessary to ensure that the data-binding mechanism that the **Raw Data** box uses on the Windows Presentation Foundation (WPF) window updates the display correctly.

Your code should resemble the following code example.

```
...
void device_NewMeasurementTaken(object sender, EventArgs e)
{
    if (device != null)
    {
        mostRecentMeasureBox.Text =
            device.MostRecentMeasure.ToString();
        metricValueBox.Text = device.MetricValue().ToString();
        imperialValueBox.Text = device.ImperialValue().ToString();
        rawDataValues.ItemsSource = null;
        rawDataValues.ItemsSource = device.GetRawData();
    }
}
...
```

Task 13: Disconnect the event handler

1. In the task list, locate the **TODO - Disconnect the event handler** task, and then double-click this task. This task is located in the **stopCollecting_Click** method, which runs when the user clicks the **Stop Collecting** button.
2. Remove the **TODO - Disconnect the event handler** comment, and add code to disconnect the **newMeasurementTaken** delegate from the **device.NewMeasurementTaken** event.



Hint: To disconnect a delegate from an event, use the **-=** compound assignment operator on the event.

Your code should resemble the following code example.

```
...
    device.StopCollecting();
    device.NewMeasurementTaken -= newMeasurementTaken;
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 14: Test the solution

1. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**.
2. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
3. Click **Start Collecting**, and verify that measurement values begin to appear in the **Raw Data** box.

   The **MeasureMassDevice** object used by the application takes metric measurements and stores them, before raising the **NewMeasurementTaken** event. The event calls code that updates the UI with the latest information. Continue to watch the **Raw Data** list box to see the buffer fill with data and then begin to overwrite earlier values.
4. Click **Stop Collecting**, and verify that the UI no longer updates.
5. Click **Start Collecting** again. Verify that the **Raw Data** list box is cleared and that new measurement data is captured and displayed.
6. Click **Stop Collecting**.
7. Close the application, and then return to Visual Studio.

Exercise 2: Using Lambda Expressions to Specify Code

Task 1: Open the Events solution

- Open the Events solution in the E:\Labfiles\Lab 11\Ex2\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 11\Ex2\Starter** folder, click **Events.sln**, and then click **Open**.



Note: The Events solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1; however, it includes an updated task list to enable you to complete this exercise.

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Define a new EventArgs class to support heartbeat events

1. In the MeasuringDevice project, add a new code file named HeartBeatEvent.cs:
   a. In Solution Explorer, right-click the **MeasuringDevice** project, point to **Add**, and then click **New Item**.
   b. In the **Add New Item - MeasuringDevice** dialog box, in the template list click **Code File**.
   c. In the **Name** box, type **HeartBeatEvent** and then click **Add**.
2. In the code file, add a **using** directive to bring the **System** namespace into scope.

   Your code should resemble the following code example.

```
using System;
```

3. Define a new class named **HeartBeatEventArgs** in the **MeasuringDevice** namespace. The class should extend the **EventArgs** class.



Note: A custom event arguments class can contain any number of properties; these properties store information when the event is raised, enabling an event handler to receive event-specific information when the event is handled.

Your code should resemble the following code example.

```
...

namespace MeasuringDevice
{

    public class HeartBeatEventArgs : EventArgs
    {

    }

}
```

4. In the **HeartBeatEventArgs** class, add a read-only automatic **DateTime** property named **TimeStamp**.

   Your code should resemble the following code example.

```
...
public class HeartBeatEventArgs : EventArgs
{
    public DateTime TimeStamp { get; private set; }
}
...
```

5. Add a constructor to the **HeartBeatEventArgs** class. The constructor should accept no arguments, and initialize the **TimeStamp** property to the date and time when the class is constructed. The constructor should also extend the base class constructor.

   Your code should resemble the following code example.

```
public class HeartBeatEventArgs : EventArgs
{
    ...

    public HeartBeatEventArgs()
        : base()
    {
        this.TimeStamp = DateTime.Now;
    }
}
```

Task 3: Declare a new delegate type

- Below the **HeartBeatEventArgs** class, declare a **public delegate** type named **HeartBeatEventHandler**. The delegate should refer to a method that does not return a value, but that has the following parameters:

  a. An **object** parameter named *sender*.

  b. A **HeartBeatEventArgs** parameter named *args*.

  Your code should resemble the following code example.

```
...
// Delegate defining the HeartBeat event signature.
public delegate void HeartBeatEventHandler
    (object sender, HeartBeatEventArgs args);
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 4: Update the IEventEnabledMeasuringDevice interface

1. In the task list, locate the **TODO - Define the new event in the interface** task and then double-click this task. This task is located in the **IEventEnabledMeasuringDevice** interface:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
   c. Double-click the **TODO - Define the new event in the interface** task.
2. Remove this comment and add an event called **HeartBeat** to the interface. The event should specify that subscribers use the **HeartBeatEventHandler** delegate type to specify the method to run when the event is raised.

   Your code should resemble the following code example.

```
...
// Event that fires every heartbeat.
event HeartBeatEventHandler HeartBeat;

// TODO - Define the HeartBeatInterval property in the interface.
...
```

3. Remove the **TODO - Define the HeartBeatInterval property in the interface** comment, and then add a read-only integer property called **HeartBeatInterval** to the interface.

   Your code should resemble the following code example.

```
...
    // Read-only heartbeat interval - set in constructor.
    int HeartBeatInterval { get; }
}
...
```

Task 5: Add the HeartBeat event and HeartBeatInterval property to the MeasureDataDevice class

1. In the task list, locate the **TODO - Add the HeartBeatInterval property** task, and then double-click this task. This task is located in the **MeasureDataDevice** class.

2. Remove the **TODO - Add the HeartBeatInterval property** comment, and add a protected integer member named **heartBeatIntervalTime**.

   Your code should resemble the following code example.

```
...
// Heartbeat interval in milliseconds.
protected int heartBeatIntervalTime;
// TODO - Add the HeartBeat event.
...
```

3. Add code to implement the public integer property **HeartBeatInterval** that the **IEventEnabledMeasuringDevice** interface defines. The property should return the value of the **heartBeatInterval** member when the **get** accessor method is called. The property should have a **private set** accessor method to enable the constructor to set the property.

   Your code should resemble the following code example.

```
...
protected int heartBeatIntervalTime;
public int HeartBeatInterval
{
    get
    {
        return heartBeatIntervalTime;
    }
}


// TODO - Add the HeartBeat event.
...
```

4. Remove the **TODO - Add the HeartBeat event** comment, and add the **HeartBeat** event that the **IEventEnabledMeasuringDevice** interface defines.

   Your code should resemble the following code example.

```
...
// Event that fires every heartbeat
public event HeartBeatEventHandler HeartBeat;

// TODO - Add the OnHeartBeat method to fire the event.
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

5. Remove the **TODO - Add the OnHeartBeat method to fire the event** comment, and add a protected virtual void method named **OnHeartBeat** that takes no parameters.

   Your code should resemble the following code example.

```
...
// Overrideable method to fire the OnHeartBeat event.
protected virtual void OnHeartBeat()
{
}

// TODO - Declare the BackgroundWorker to generate the heartbeat.
...
```

6. In the **OnHeartBeat** method, add code to perform the following actions:

   a. Check whether the **HeartBeat** event has any subscribers.

   b. If the event has subscribers, raise the event, passing the current object and a new instance of the **HeartBeatEventArgs** object as parameters.

   Your code should resemble the following code example.

```
...
protected virtual void OnHeartBeat()
{
    if (HeartBeat != null)
    {
        HeartBeat(this, new HeartBeatEventArgs());
    }
}
...
```

Task 6: Use a BackgroundWorker object to generate the heartbeat

1. Remove the **TODO - Declare the BackgroundWorker to generate the heartbeat** comment, and then define a private **BackgroundWorker** object named **heartBeatTimer**.

   Your code should resemble the following code example.

```
...

// Background worker object to host the heartbeat thread.
private BackgroundWorker heartBeatTimer;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
// TODO - Create a method to configure the background Worker by using
// Lambda Expressions.

...
```

2. Remove the **TODO - Create a method to configure the BackgroundWorker using Lambda Expressions** comment, and declare a private method named **StartHeartBeat** that accepts no parameters and does not return a value.

   Your code should resemble the following code example.

```
...
    // Start the BackgroundWorker that fires the heartbeat.
    private void StartHeartBeat()
    {

    }
}
...
```

3. In the **StartHeartBeat** method, add code to perform the following actions:

   a. Instantiate the **heartBeatTimer BackgroundWorker** object.

   b. Configure the **heartBeatTimer** object to support cancellation.

   c. Configure the **heartBeatTimer** object to support progress notification.

   Your code should resemble the following code example.

```
...
private void StartHeartBeat()
{
    heartBeatTimer = new BackgroundWorker();
    heartBeatTimer.WorkerSupportsCancellation = true;
    heartBeatTimer.WorkerReportsProgress = true;
}
...
```

4. Add a handler for the **heartBeatTimer DoWork** event by using a lambda expression to define the actions to be performed. The lambda expression should take two parameters (use the names *o* and *args*). In the lambda expression body, add a **while** loop that continually iterates and contains code to perform the following actions:

   a. Use the static **Thread.Sleep** method to put the current thread to sleep for the length of time that the **HeartBeatInterval** property indicates.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Check the value of the **disposed** property. If the value is **true**, terminate the loop.

c. Call the **heartBeatTimer**.**ReportProgress** method, passing zero as the parameter.



Note: Use the **+=** compound assignment operator to specify that the method will handle the **DoWork** event, define the signature of the lambda expression, and then use the **=>** operator to denote the start of the body of the lambda expression.

Your code should resemble the following code example.

```
...
heartBeatTimer.WorkerReportsProgress = true;
heartBeatTimer.DoWork += (o, args) =>
    {
        while (true)
        {
            Thread.Sleep(HeartBeatInterval);

            if (disposed)
            {
                break;
            }
            heartBeatTimer.ReportProgress(0);
        }
    };
...
```

5. Add a handler for the **heartBeatTimer**.**ReportProgress** event by using another lambda expression to create the method body. In the lambda expression body, add code to call the **OnHeartBeat** method, which raises the **HeartBeat** event.

   Your code should resemble the following code example.

```
...
    heartBeatTimer.ReportProgress(0);
    }
};
heartBeatTimer.ProgressChanged += (o, args) =>
    {
        OnHeartBeat();
    };
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

6. At the end of the **StartHeartBeat** method, add a line of code to start the **heartBeatTimer BackgroundWorker** object running asynchronously.

   Your code should resemble the following code example.

```
...
    heartBeatTimer.ProgressChanged += (o, args) =>
        {
            OnHeartBeat();
        };
    heartBeatTimer.RunWorkerAsync();
}
...
```

Task 7: Call the StartHeartBeat method when the MeasureDataDevice object starts running

1. In the task list, locate the **TODO - Call StartHeartBeat() from StartCollecting method** task, and then double-click this task. This task is located in the **StartCollecting** method.
2. Remove this comment, and add a line of code to invoke the **StartHeartBeat** method.

   Your code should resemble the following code example.

```
...
    loggingFileWriter.WriteLine("Collecting Started");
}

StartHeartBeat();

GetMeasurements();
...
```

Task 8: Dispose of the heartBeatTimer BackgroundWorker object when the MeasureDataDevice object is destroyed

1. In the task list, locate the **TODO - dispose of the heartBeatTimer BackgroundWorker** task, and then double-click this task. This task is located in the **Dispose** method.

MCT USE ONLY. STUDENT USE PROHIBITED

2. Remove the comment and add code to check that the **heartBeatTimer BackgroundWorker** object is not null. If the **heartBeatTimer** object is not null, call the **Dispose** method of the **BackgroundWorker** object.

   Your code should resemble the following code example.

```
...
    if (dataCollector != null)
    {
        dataCollector.Dispose();
    }

    if (heartBeatTimer != null)
    {
        heartBeatTimer.Dispose();
    }
}
...
```

   You have now updated the **MeasureDataDevice** abstract class to implement event handlers by using lambda expressions. To enable the application to benefit from these changes, you must modify the **MeasureMassDevice** class, which extends the **MeasureDataDevice** class.

Task 9: Update the constructor for the MeasureMassDevice class

1. Open the MeasureMassDevice class file:
   - In Solution Explorer, in the MeasuringDevice project, double-click **MeasureMassDevice.cs**.
2. At the start of the class, modify the signature of the constructor to take an additional **integer** value named **heartBeatInterval**.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice
    (Units deviceUnits, string logFileName, int heartBeatInterval)
{
    unitsToUse = DeviceUnits;
    measurementType = DeviceType.MASS;
    loggingFileName = LogFileName;
}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. Modify the body of the constructor to store the value of the **HeartBeatInterval** member in the **heartBeatInterval** member.

   Your code should resemble the following code example.

```
...
    loggingFileName = LogFileName;
    heartBeatIntervalTime = heartBeatInterval;
}
...
```

4. Below the existing constructor, remove the **TODO – Add a chained constructor that calls the previous constructor** comment, and add a second constructor that accepts the following parameters:

   a. A **Units** instance named **deviceUnits**.

   b. A **string** instance named **logFileName**.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice(Units deviceUnits, string logFileName) {}
...
```

5. Modify the new constructor to implicitly call the existing constructor. Pass a value of **1000** as the *heartBeatInterval* parameter value.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice(Units deviceUnits, string logFileName)
    : this(deviceUnits, logFileName, 1000) { }
...
```

Task 10: Handle the HeartBeat event in the UI

1. In the task list, locate the **TODO - Use a lambda expression to handle the HeartBeat event in the UI** task, and then double-click the task. This task is located in the **startCollecting_Click** method in the code behind the MainWindow window in the Monitor project.

2. Remove the comment, and add a lambda expression to handle the **device**.**HeartBeat** event. The lambda expression should take two parameters (name them *o* and *args*). In the body of the lambda expression, add code to

MCT USE ONLY. STUDENT USE PROHIBITED

update the **heartBeatTimeStamp** label with the text "HeartBeat Timestamp: *timestamp*" where timestamp is the value of the **args.TimeStamp** property.



Hint: Set the **Content** property of a label to modify the text that the label displays.

Your code should resemble the following code example.

```
...
device.HeartBeat += (o, args) =>
    {
        heartBeatTimeStamp.Content =
            string.Format("HeartBeat Timestamp: {0}", args.TimeStamp);
    };
...
```

Task 11: Test the solution

1. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**.
2. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
3. Click **Start Collecting**, and verify that values begin to appear as before. Also note that the **HeartBeat Timestamp** value now updates once per second.
4. Click **Stop Collecting**, and verify that the **RawData** list box no longer updates. Note that the timestamp continues to update, because your code does not terminate the timestamp heartbeat when you stop collecting.
5. Click **Dispose Object**, and verify that the timestamp no longer updates.
6. Close the application, and then return to Visual Studio.
7. Close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Using Collections and Building Generic Types

Lab A: Using Collections

Exercise 1: Optimizing a Method by Caching Data

Task 1: Open the Collections solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Collections solution in the E:\Labfiles\Lab 12\Lab A\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab A\Ex1\Starter** folder, click **Collections.sln**, and then click **Open**.

Task 2: Modify the Gauss class to implement the memoization pattern

1. In the TestHarness project, display the MainWindow.xaml window:
   - In Solution Explorer, expand the **TestHarness** project, and then double-click **MainWindow.xaml**.

   The MainWindow window implements a simple test harness to enable you to test the method that you will use to perform Gaussian elimination. This is a Windows® Presentation Foundation (WPF) application that enables a user to enter the coefficients for four simultaneous equations that consist of four variables (w, x, y, and z). It then uses Gaussian elimination to find a solution for these equations. The results are displayed in the lower part of the screen.
2. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

3. In the task list, locate the **TODO - Add a static Hashtable** task, and then double-click this task.

   This task is located in the GaussianElimination project, in the **Gauss** class.

4. At the top of the Gauss.cs file, at the end of the list of **using** statements, add a statement to bring the **System.Collections** namespace into scope.

   Your code should resemble the following code example.

```
...

using System.Text;
using System.Collections;
namespace GaussianElimination

...
```

5. Remove the comment, and then add code to define a static **Hashtable** object named **results**.

   Your code should resemble the following code example.

```
...

public static class Gauss
{
    static Hashtable results;

    /// <summary>

    ...
}

...
```

6. At the beginning of the **SolveGaussian** method, before the statements that create a deep copy of the parameters, add code to ensure that the **results Hashtable** object is initialized. Create a new instance of this object if it is currently **null**.

   Your code should resemble the following code example.

```
...

public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    if (results == null)
    {
        results = new Hashtable();
    }

    ...
}
 ...
```

7. Add code to generate a hash key that is based on the method parameters by performing the following tasks:
   a. Define a new **StringBuilder** object named **hashString**.
   b. Iterate through the **coefficients** array, and append each value in the array to the **hashString StringBuilder** object.
   c. Iterate through the **rhs** array, and append each value in the array to the **hashString StringBuilder** object.
   d. Define a new **string** object named **hashValue**, and initialize it to the value that the **hashString.ToString** method returns.



Hint: This procedure generates a hash key by simply concatenating the values that are passed into the method. You can use more advanced hashing algorithms to generate better hashes. The **System.Security.Cryptography** namespace includes many classes that you can use to implement hashing.

Your code should resemble the following code example.

```
...

public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...

    StringBuilder hashString = new StringBuilder();

    foreach (double coefficient in coefficients)
    {
        hashString.Append(coefficient);
    }

    foreach (double value in rhs)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            hashString.Append(value);
        }

        string hashValue = hashString.ToString();

        ...
}
...
```

8. Add code to check whether the **results** object already contains a key that has the value in the **hashValue** string. If it does, return the value that is stored in the **Hashtable** collection class that corresponds to the **hashValue** key. If the **results** object does not contain the **hashValue** key, the method should use the existing logic in the method to perform the calculation.



Hint: A **HashTable** object stores and returns values as objects. You must cast the value that is returned from a **HashTable** object to the appropriate type before you work with it. In this case, cast the returned value to an array of **double** values.

Your code should resemble the following code example.

```
...
public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...
    string hashValue = hashString.ToString();

    if (results.Contains(hashValue))
    {
        return (double[])results[hashValue];
    }

    else
    {
        // Make a deep copy of the parameters

        ...
        return rhsCopy;
    }

}
...
```

MCT USE ONLY. STUDENT USE PROHIBITED

9. In the task list, locate the **TODO - Store the result of the calculation in the Hashtable** task, and then double-click this task.

   This task is located near the end of the **SolveGaussian** method.

10. Remove the comment, and then add code to the method to store the **rhsCopy** array in the **HashTable** object, specifying the **hashValue** object as the key.

    Your code should resemble the following code example.

```
...
public static double[] SolveGaussian
   (double[,] coefficients, double[] rhs)
{
    ...
    else{
        ...
        System.Threading.Thread.Sleep(5000);
        results.Add(hashValue, rhsCopy);
        return rhsCopy;
    }
}
...
```

Task 3: Test the solution

1. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
2. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
3. 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.

- $2w + x - y + z = 8$
- $-3w - x + 2y + z = -11$
- $-2w + x - 2y = -3$
- $3w - x + 2y - 2z = -5$

MCT USE ONLY. STUDENT USE PROHIBITED

Observe that the operation takes approximately five seconds to complete.

4. Verify that the following results are displayed:
   - w = 4
   - x = −17
   - y = −11
   - z = 6
5. Modify the third equation to match the following equation, and then click **Solve** again:
   - −2w + x − 2y **+ 3z** = −3

   Observe that this operation also takes approximately five seconds to complete.
6. Verify that the following results are displayed:
   - w = −2
   - x = 25
   - y = 7
   - z = −6
7. Undo the change to the third equation so that all of the equations match those in Step 3, and then click **Solve**. Observe that this time, the operation takes much less time to complete because it uses the solution that was generated earlier.
8. Close the application, and then close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Using Collections and Building Generic Types

Lab B: Building Generic Types

Exercise 1: Defining a Generic Interface

Task 1: Open the GenericTypes solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 12\Lab B\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** drop-down list box, select **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 12\Lab B\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B\Ex1\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Define the generic IBinaryTree interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

2. In the task list, locate the **TODO – Define the IBinaryTree interface** task, and then double-click this task. This task is located in the IBinaryTree.cs file.
3. In the **BinaryTree** namespace, define a new generic public interface named **IBinaryTree**. This interface should take a single type parameter named **TItem**. Specify that the type parameter must implement the generic **IComparable** interface.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public interface IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
    }
}
```

4. In the **IBinaryTree** interface, define the following public methods:
   a. An **Add** method, which takes a **TItem** object named **newItem** as a parameter and does not return a value.
   b. A **Remove** method, which takes a **TItem** object named **itemToRemove** as a parameter and does not return a value.
   c. A **WalkTree** method, which takes no parameters and does not return a value.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public interface IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
        void Add(TItem newItem);

        void Remove(TItem itemToRemove);

        void WalkTree();
    }
}
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 2: Implementing a Generic Interface

Task 1: Open the GenericTypes solution



Note: Perform this task only if you have not been able to complete Exercise 1. If you have defined the **IBinaryTree** interface successfully, proceed directly to **Task 2: Create the Tree class**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex2\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B \Ex2\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Create the Tree class

1. In the BinaryTree project, add a new class named **Tree**:
   a. In Solution Explorer, right-click the **BinaryTree** project, point to **Add**, and then click **Class**.
   b. In the **Add New Item - BinaryTree** dialog box, in the **Name** box, type **Tree** and then click **Add**.
2. Modify the **Tree** class definition. This class should be a public generic class that takes a single type parameter called **TItem** and implements the **IBinaryTree** interface. The **TItem** type parameter must implement the generic **IComparable** interface.

   Your code should resemble the following code example.

```
...
namespace BinaryTree
{
    public class Tree<TItem> : IBinaryTree<TItem>
        where TItem : IComparable<TItem>
    {
    }
}
...
```

3. Add the following automatic properties to the **Tree** class:

a. A **TItem** property named **NodeData**.

b. A generic **Tree<TItem>** property named **LeftTree**.

c. A generic **Tree<TItem>** property named **RightTree**.

Your code should resemble the following code example.

```
...
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    public TItem NodeData { get; set; }

    public Tree<TItem> LeftTree { get; set; }

    public Tree<TItem> RightTree { get; set; }
}

...
```

4. Add a public constructor to the **Tree** class. The constructor should take a single **TItem** parameter called *nodeValue*. The constructor should initialize the **NodeData** member by using the *nodeValue* parameter, and then set the **LeftTree** and **RightTree** members to **null**.

   Your code should resemble the following code example.

```
...
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...

    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }
}

...
```

5. After the constructor, define a method called **Add**. This method should take a **TItem** object as a parameter, but not return a value.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...
    public Tree(TItem nodeValue)
    {
        this.NodeData = nodeValue;
        this.LeftTree = null;
        this.RightTree = null;
    }

    public void Add(TItem newItem)
    {
    }
}
...
```

6. In the **Add** method, add code to insert the **newItem** object into the tree in the appropriate place by performing the following tasks:

    a. Compare the value of the **newItem** object with the value of the **NodeData** property. Both items implement the **IComparable** interface, so use the **CompareTo** method of the **NodeData** property. The **CompareTo** method returns zero if both items have the same value, a positive value if the value of the **NodeData** property is greater than the value of the **newItem** object, and a negative value if the value of the **NodeData** property is less than the value of the **newItem** object.

    b. If the value of the **newItem** object is less than the value of the **NodeData** property, perform the following actions to insert a **newItem** object into the left subtree:

        i. If the **LeftTree** property is **null**, initialize it and pass the **newItem** object to the constructor.

        ii. If the **LeftTree** property is not **null**, recursively call the **Add** method of the **LeftTree** property and pass the **newItem** object as the parameter.

    c. If the value of the **newItem** object is greater than or equal to the value of the **NodeData** property, perform the following actions to insert the **newItem** object into the right subtree:

        i. If the **RightTree** property is **null**, initialize it and pass the value of the **newItem** object to the constructor.

MCT USE ONLY. STUDENT USE PROHIBITED

ii. If the **RightTree** property is not **null**, recursively call the **Add** method of the **RightTree** property and pass the **newItem** object as the parameter.

Your code should resemble the following code example.

```
public void Add(TItem newItem)
{
    TItem currentNodeValue = this.NodeData;

    // Check if the item should be inserted in the left tree.
    if (currentNodeValue.CompareTo(newItem) > 0)
    {
        // Is the left tree null?
        if (this.LeftTree == null)
        {
            this.LeftTree = new Tree<TItem>(newItem);
        }
        else // Call the Add method recursively.
        {
            this.LeftTree.Add(newItem);
        }
    }
    else // Insert in the right tree.
    {
        // Is the right tree null?
        if (this.RightTree == null)
        {
            this.RightTree = new Tree<TItem>(newItem);
        }
        else // Call the Add method recursively.
        {
            this.RightTree.Add(newItem);
        }
    }
}
```

7. After the **Add** method, add another public method called **WalkTree** that does not take any parameters and does not return a value.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IBinaryTree<TItem>
    where TItem : IComparable<TItem>
{
    ...

    public void Add(TItem newItem)
    {
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        ...
    }

    public void WalkTree()
    {
    }
}
...
```

8. In the **WalkTree** method, add code that visits each node in the tree in order and displays the value that each node holds by performing the following tasks:

   a. If the value of the **LeftTree** property is not **null**, recursively call the **WalkTree** method on the **LeftTree** property.

   b. Display the value of the **NodeData** property to the console by using a **Console.WriteLine** statement.

   c. If the value of the **RightTree** property is not **null**, recursively call the **WalkTree** method on the **RightTree** property.

   Your code should resemble the following code example.

```
public void WalkTree()
{
    // Recursive descent of the left tree.

    if (this.LeftTree != null)
    {
        this.LeftTree.WalkTree();
    }

    Console.WriteLine(this.NodeData.ToString());

    // Recursive descent of the right tree.

    if (this.RightTree != null)
    {
        this.RightTree.WalkTree();
    }

}
```

9. After the **WalkTree** method, add the **Remove** method to delete a value from the tree, as the following code example shows. It is not necessary for you to fully understand how this method works, so you can either type this code manually or use the Mod12Remove code snippet.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public void Remove(TItem itemToRemove)
{

    // Cannot remove null.
    if (itemToRemove == null)
    {
        return;
    }

    // Check if the item could be in the left tree.
    if (this.NodeData.CompareTo(itemToRemove) > 0
        && this.LeftTree != null)
    {
        // Check the left tree.
        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.LeftTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The LeftTree property has no children - set the
            // LeftTree property to null.
            if (this.LeftTree.LeftTree == null
                && this.LeftTree.RightTree == null)
            {
                this.LeftTree = null;
            }
            else // Remove LeftTree.
            {
                RemoveNodeWithChildren(this.LeftTree);
            }
        }
        else
        {
            // Keep looking - call the Remove method recursively.
            this.LeftTree.Remove(itemToRemove);
        }
    }

    // Check if the item could be in the right tree.?
    if (this.NodeData.CompareTo(itemToRemove) < 0
        && this.RightTree != null)
    {
        // Check the right tree.

        // Check 2 levels down the tree - cannot remove
        // 'this', only the LeftTree or RightTree properties.
        if (this.RightTree.NodeData.CompareTo(itemToRemove) == 0)
        {
            // The RightTree property has no children – set the
            // RightTree property to null.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            if (this.RightTree.LeftTree == null
                && this.RightTree.RightTree == null)
            {
                this.RightTree = null;
            }

            else // Remove the RightTree.
            {
                RemoveNodeWithChildren(this.RightTree);
            }

        }

        else
        {
            // Keep looking - call the Remove method recursively.
            this.RightTree.Remove(itemToRemove);
        }
    }

    // This will only apply at the root node.
    if (this.NodeData.CompareTo(itemToRemove) == 0)
    {
        // No children - do nothing, a tree must have at least
        // one node.
        if (this.LeftTree == null && this.RightTree == null)
        {
            return;
        }

        else // The root node has children.
        {
            RemoveNodeWithChildren(this);
        }

    }
}
```

- To use the code snippet, type **Mod12Remove** and then press the TAB key twice.

10. After the **Remove** method, add the **RemoveNodeWithChildren** method to remove a node that contains children from the tree, as the following code example shows. This method is called by the **Remove** method. Again, it is not necessary for you to understand how this code works, so you can either type this code manually or use the Mod12RemoveNodeWithChildren code snippet.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void RemoveNodeWithChildren(Tree<TItem> node)
{
    // Check whether the node has children.
    if (node.LeftTree == null && node.RightTree == null)
    {
        throw new ArgumentException("Node has no children");
    }

    // The tree node has only one child - replace the
    // tree node with its child node.
    if (node.LeftTree == null ^ node.RightTree == null)
    {
        if (node.LeftTree == null)
        {
            node.CopyNodeToThis(node.RightTree);
        }
        else
        {
            node.CopyNodeToThis(node.LeftTree);
        }
    }
    else
    // The tree node has two children - replace the tree node's value
    // with its "in order successor" node value and then remove the
    // in order successor node.
    {
        // Find the in order successor - the leftmost descendant of
        // its RightTree node.
        Tree<TItem> successor = GetLeftMostDescendant(node.RightTree);

        // Copy the node value from the in order successor.
        node.NodeData = successor.NodeData;

        // Remove the in order successor node.
        if (node.RightTree.RightTree == null &&
            node.RightTree.LeftTree == null)
        {
            node.RightTree = null; // The successor node had no
                                   // children.
        }
        else
        {
            node.RightTree.Remove(successor.NodeData);
        }
    }
}
```

- To use the code snippet, type **Mod12RemoveNodeWithChildren** and then press the TAB key twice.

MCT USE ONLY. STUDENT USE PROHIBITED

11. After the **RemoveNodeWithChildren** method, add the **CopyNodeToThis** method, as the following code example shows. The **RemoveNodeWithChildren** method calls this method to copy another node's property values into the current node. You can either type this code manually or use the Mod12CopyNodeToThis code snippet.

```
private void CopyNodeToThis(Tree<TItem> node)
{
    this.NodeData = node.NodeData;
    this.LeftTree = node.LeftTree;
    this.RightTree = node.RightTree;
}
```

    - To use the code snippet, type **Mod12CopyNodeToThis** and then press the TAB key twice.

12. After the **CopyNodeToThis** method, add the **GetLeftMostDescendant** method, as the following code example shows. The **RemoveNodeWithChildren** method also calls this method to retrieve the leftmost descendant of a tree node. You can either type this code manually or use the Mod12GetLeftMostDescendant code snippet.

```
private Tree<TItem> GetLeftMostDescendant(Tree<TItem> node)
{
    while (node.LeftTree != null)
    {
        node = node.LeftTree;
    }
    return node;
}
```

    - To use the code snippet, type **Mod12GetLeftMostDescendant** and then press the TAB key twice.

13. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Exercise 3: Implementing a Test Harness for the BinaryTree Project

Task 1: Open the GenericTypes solution



Note: Perform this task only if you have not been able to complete Exercise 2. If you have defined the **IBinaryTree** interface and built the **Tree** class successfully, proceed directly to **Task 3: Complete the test harness**.

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B \Ex3\Starter** folder, click **GenericTypes.sln**, and then click **Open**.

Task 2: Import the TestHarness project



Note: Perform this task only if you have completed Exercise 2 successfully.

1. Import the TestHarness project in the E:\Labfiles\Lab 12\Lab B\Ex3\Starter \TestHarness folder into the GenericTypes solution:
   a. In Visual Studio, in Solution Explorer, right-click **Solution 'GenericTypes' (1 Project)**, point to **Add**, and then click **Existing Project**.
   b. In the **Add Existing Project** dialog box, move to the **E:\Labfiles\Lab 12 \Lab B\Ex3\Starter\TestHarness** folder, click **TestHarness.csproj**, and then click **Open**.
2. In the TestHarness project, update the reference to the BinaryTree project:
   a. In Solution Explorer, in the TestHarness project, expand **References**, right-click **BinaryTree**, and then click **Remove**.
   b. Right-click **References**, and then click **Add Reference**.
   c. In the **Add Reference** dialog box, on the **Projects** tab, click **BinaryTree**, and then click **OK**.

3. Set the TestHarness project as the startup project:
   - In Solution Explorer, right-click **TestHarness**, and then click **Set as Startup Project**.

Task 3: Complete the test harness

1. Open the Program.cs file:
   - In Solution Explorer, in the TestHarness project, double-click **Program.cs**.
2. In the **Main** method, add code to instantiate a new **IBinaryTree** object named **tree**, using **int** as the type parameter. Pass the value **5** to the constructor. This code creates a new binary tree of integers and adds an initial node that contains the value **5**.

   Your code should resemble the following code example.

```
...

static void Main(string[] args)
{
    IBinaryTree<int> tree = new Tree<int>(5);
}

...
```

3. Add code to the **Main** method to add the following values to the tree, in the following order:
   a. **1**
   b. **4**
   c. **7**
   d. **3**
   e. **4**

   Your code should resemble the following code example.

```
...

static void Main(string[] args)
{
    IBinaryTree<int> tree = new Tree<int>(5);
    tree.Add(1);
    tree.Add(4);
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    tree.Add(7);
    tree.Add(3);
    tree.Add(4);
}
...
```

4. Add code to the **Main** method to perform the following actions:

    a. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    b. Print the message "Add 15" to the console, and then add the value **15** to the tree.

    c. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    d. Print the message "Remove 5" to the console, and then remove the value **5** from the tree.

    e. Print the message "Current Tree: " to the console, and then invoke the **WalkTree** method on the **tree** object.

    f. Pause at the end of the method until ENTER is pressed.

    Your code should resemble the following code example.

```
...
static void Main(string[] args)
{
    ...
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.WriteLine("Add 15");
    tree.Add(15);
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.WriteLine("Remove 5");
    tree.Remove(5);
    Console.WriteLine("Current Tree: ");
    tree.WalkTree();
    Console.ReadLine();
}...
```

5. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 4: Test the BinaryTree project

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. Verify that the output in the console window resembles the following code example. Note that the data in the binary tree is sorted and is displayed in ascending order.

```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
3
4
4
7
15
```

3. Press ENTER to close the console window, and then return to Visual Studio.

Exercise 4: Implementing a Generic Method

Task 1: Open the GenericTypes solution

- Open the GenericTypes solution in the E:\Labfiles\Lab 12\Lab B\Ex4\Starter folder:
  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 12\Lab B \Ex4\Starter** folder, click **GenericTypes.sln**, and then click **Open**.



Note: The GenericTypes solution in the Ex4 folder is functionally the same as the code that you completed in Exercise 3. However, it includes an updated task list and a new test project to enable you to complete this exercise.

Task 2: Create the BuildTree method

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Add the BuildTree generic method** task, and then double-click this task. This task is located at the end of the **Tree** class.
3. Remove the **TODO - Add the BuildTree generic method** comment, and then add a public static generic method named **BuildTree** to the **Tree** class. The type parameter for the method should be called **TreeItem**, and the method should return a generic **Tree<TreeItem>** object. The **TreeItem** type parameter must represent a type that implements the generic **IComparable** interface.

   The method should take two parameters: a **TreeItem** object called **nodeValue** and a **params** array of **TreeItem** objects called **values**.

   Your code should resemble the following code example.

```
public static Tree<TreeItem> BuildTree<TreeItem>
    (TreeItem nodeValue, params TreeItem[] values)
    where TreeItem : IComparable<TreeItem>
{

}
```

4. In the **BuildTree** method, add code to construct a new **Tree** object by using the data that is passed in as the parameters by performing the following actions:
   a. Define a new **Tree** object named **tree** by using the **TreeItem** type parameter, and initialize the new **Tree** object by using the **nodeValue** parameter.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Iterate through the **values** array, and add each value in the array to the **tree** object.

c. Return the **tree** object at the end of the method.

Your code should resemble the following code example.

```
public static Tree<TreeItem> BuildTree<TreeItem>
    (TreeItem nodeValue, params TreeItem[] values)
    where TreeItem : IComparable<TreeItem>
{

    Tree<TreeItem> tree = new Tree<TreeItem>(nodeValue);

    foreach (TreeItem item in values)
    {
        tree.Add(item);
    }
    return tree;

}
```

Task 3: Modify the test harness to use the BuildTree method

1. In the task list, locate the **TODO - Modify the test harness to use the BuildTree method** task, and then double-click this task.

   This task is located in the **Main** method of the Program.cs class file in the TestHarness project.

2. In the **Main** method, remove the existing code that instantiates the **tree** object and adds the first five values to the tree. Replace this code with a statement that calls the **BuildTree** method to create a new **Tree** object named **tree**, based on the integer type, with the following integer values:

   a. **1**

   b. **4**

   c. **7**

   d. **3**

   e. **4**

   f. **5**

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
static void Main(string[] args)
{
    IBinaryTree<int> tree =
        Tree<int>.BuildTree<int>(1, new int[]{4, 7, 3, 4, 5});
    Console.WriteLine("Current Tree: ");
    ...
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
4. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
5. Verify that the output in the console window resembles the following code example.

```
Current Tree:
1
3
4
4
5
7
Add 15
Current Tree:
1
3
4
4
5
7
15
Remove 5
Current Tree:
1
3
4
4
7
15
```

6. Press ENTER to close the console window.
7. Close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Building and Enumerating Custom Collection Classes

Lab 13: Building and Enumerating Custom Collection Classes

Exercise 1: Implementing the IList<TItem> Interface

Task 1: Open the CustomCollections solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 13\Ex1\Starter** folder, click **CustomCollections.sln**, and then click **Open**.

Task 2: Modify the Tree class to implement the IList<TItem> interface

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Implement the generic IList<TItem> interface** task, and then double-click this task.

   This task is located in the **Tree** class.
3. Remove the **TODO - Implement the generic IList<TItem> interface** comment, and then modify the class definition to implement the generic **IList**

interface. Specify the value **TItem** as the type parameter (this is the type parameter that the **Tree** class references).

Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>,
    IBinaryTree<TItem> where TItem : IComparable<TItem>
{
    ...
}
```

4. Add method and property stubs that implement the **IList** interface:
   - In the class definition, right-click **IList**, point to **Implement Interface**, and then click **Implement Interface**.

     Visual Studio generates method stubs for each method that is defined in the interface, and adds them to the end of the class file. You will add code to complete some of these methods later in this exercise.

Task 3: Add support for indexing items in the Tree class

1. In the task list, locate the **TODO - Add a member to define node position** task, and then double-click this task.
2. Remove the **TODO - Add a member to define node position** comment, and then add code to define a private integer member named **position**.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>, IBinaryTree<TItem> where
TItem : IComparable<TItem>
{
    ...
    public Tree<TItem> RightTree { get; set; }

    // Add a private integer variable position to define
    // the node's position in the tree.
    private int position;
    ...
}
```

3. In the class constructor, add code to initialize the **position** member to −1.



Note: The **position** member is the index for items in the tree. When you add or remove items from the tree, you will invalidate the **position** member of any following elements

MCT USE ONLY. STUDENT USE PROHIBITED

in the tree. By setting the **position** member to –1, you indicate to the tree that the index has become invalid. When the application attempts to use the index to perform an action, and encounters a negative value, the application can rebuild the index by invoking the **IndexTree** method that you will add later.

Your code should resemble the following code example.

```
public Tree(TItem nodeValue)
{
    ...
    this.position = -1;
}
```

4. At the beginning of the **Add** method, add code to set the **position** member to −1.

   Your code should resemble the following code example.

```
public void Add(TItem newItem)
{
    // If we're adding something, the position field will become
    // invalid. Reset position to -1.
    this.position = -1;
    ...
}
```

5. In the task list, locate the **TODO - Set the position member to -1** task, and then double-click this task.

   This task is located in the **Remove** method.

6. Remove the **TODO - Set the position member to -1** comment, and then add code to set the **position** member to −1.

   Your code should resemble the following code example.

```
public void Remove(TItem itemToRemove)
{
    ...
    // If we're deleting something, the position field will become
    // invalid. Reset position to -1
    this.position = -1;
    ...
}
```

7. In the task list, locate the **TODO - Add methods to enable indexing the tree** task, and then double-click this task.

MCT USE ONLY. STUDENT USE PROHIBITED

This task is located at the end of the **Tree** class, in the **Utility methods** code region.

8. Delete the **TODO - Add methods to enable indexing the tree** comment, and then add a method named **IndexTree**. This method should accept an integer parameter named *index*, and return an integer value. Add code to the method to perform the following actions:

   a. If the **LeftTree** property is not **null**, call the **IndexTree** method of the **LeftTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.

   b. Update the local **position** member with the value of the *index* parameter.

   c. Increment the *index* parameter.

   d. If the **RightTree** property is not **null**, call the **IndexTree** method of the **RightTree** property and assign the result to the *index* parameter. Pass the current value of the index variable to the **IndexTree** method.

   e. At the end of the method, return the value of the *index* parameter.

   Your code should resemble the following code example.

```
public class Tree<TItem> : IList<TItem>, IBinaryTree<TItem> where
TItem : IComparable<TItem>
{
    ...
    #region Utility methods
    ...
    private int IndexTree(int index)
    {
        if (this.LeftTree != null)
        {
            index = this.LeftTree.IndexTree(index);
        }
        this.position = index;

        index++;

        if (this.RightTree != null)
        {
            index = this.RightTree.IndexTree(index);
        }
        return index;
    }
    #endregion
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

9. After the **IndexTree** method, add a private method named **GetItemAtIndex**. This method should accept an integer parameter named *index*, and return a **Tree<TItem>** object. In the method, add code to perform the following actions:

   a. If the value of the **position** member is **−1**, call the local **IndexTree** method. Pass **0** as the parameter to the **IndexTree** method.

   b. If the value of the **position** member is greater than the value of the index variable, call the **GetItemAtIndex** method of the **LeftTree** property and return the value that is generated . Pass the value of the *index* parameter to the **GetItemAtIndex** method.

   c. If the value of the **position** member is less than the value of the index variable, call the **GetItemAtIndex** method of the **RightTree** property and return the value that is generated. Pass the value of the *index* parameter to the **GetItemAtIndex** method.

   d. At the end of the method, return a reference to the current object.

   Your code should resemble the following code example.

```
private Tree<TItem> GetItemAtIndex(int index)
{
    // Add the index values if they're not already there
    if (this.position == -1)
    {
        this.IndexTree(0);
    }

    if (this.position > index)
    {
        return this.LeftTree.GetItemAtIndex(index);
    }
    if (this.position < index)
    {
        return this.RightTree.GetItemAtIndex(index);
    }
    return this;
}
```

10. After the **GetItemAtIndex** method, add a private method named **GetCount**. This method should accept an integer parameter named *accumulator*, and return an integer value. Add code to the method to perform the following actions:

MCT USE ONLY. STUDENT USE PROHIBITED

a. If the **LeftTree** property is not **null**, call the **GetCount** method of the **LeftTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.

b. Increment the value in the accumulator variable.

c. If the **RightTree** property is not **null**, call the **GetCount** method of the **RightTree** property and store the result in the accumulator variable. Pass the current value of the accumulator variable to the **GetCount** method.

d. At the end of the method, return the value of the accumulator variable.

Your code should resemble the following code example.

```
private int GetCount(int accumulator)
{
    if (this.LeftTree != null)
    {
        accumulator = LeftTree.GetCount(accumulator);
    }
    accumulator++;
    if (this.RightTree != null)
    {
        accumulator = RightTree.GetCount(accumulator);
    }
    return accumulator;
}
```

Task 4: Implement the IList<T> interface methods and properties

1. Locate the **IndexOf** method. This method accepts a **TItem** object named **item**, and returns an integer value.

   This method should iterate through the tree and return a value that indicates the index of the **TItem** object in the tree. The method currently throws a **NotImplementedException** exception.

2. Replace the code in the **IndexOf** method with code to perform the following actions:

   a. If the *item* parameter is **null**, return the value **−1**.

   b. If the value of the **position** member is **−1**, call the **IndexTree** method and pass the value **0** as a parameter to the **IndexTree** method.

   c. Compare the value of the *item* parameter to the local **NodeData** property value:

MCT USE ONLY. STUDENT USE PROHIBITED

i. If the value of the *item* parameter is less than the value in the **NodeData** property, and if the *LeftTree* parameter is **null**, return **-1**. Otherwise, return the result of a recursive call to the **LeftTree.IndexOf** method, passing the **item** value to the **IndexOf** method.

ii. If the value of the *item* parameter is greater than the value in the **NodeData** property, and if the *RightTree* parameter is **null**, return **-1**. Otherwise, return the result of a recursive call to the **RightTree.IndexOf** method, passing the **item** value to the **IndexOf** method.



Hint: Use the **CompareTo** method to compare the value in the *item* parameter and the value in the **NodeData** property.

d. At the end of the method, return the value of the local **position** member.

Your code should resemble the following example.

```
public int IndexOf(TItem item)
{
    if (item == null) return -1;

    // Add the index values if they're not already there
    if (this.position == -1)
        this.IndexTree(0);

    // Find the item - searching the tree for a matching Node.
    if (item.CompareTo(this.NodeData) < 0)
    {
        if (this.LeftTree == null)
        {
            return -1;
        }
        return this.LeftTree.IndexOf(item);
    }
    if (item.CompareTo(this.NodeData) > 0)
    {
        if (this.RightTree == null)
        {
            return -1;
        }
        return this.RightTree.IndexOf(item);
    }
    return this.position;
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. Locate the **this** indexer.

   The **this** indexer should return the **TItem** object at the index that the *index* parameter specifies. Currently, both **get** and **set** accessors throw a **NotImplementedException** exception.

4. Replace the code in the **get** accessor with code to perform the following actions:

   a. If the value of the *index* parameter is less than zero, or greater than the value of the **Count** property, throw an **ArgumentOutOfRangeException** exception with the following parameters:

      i. A string value, "index".

      ii. The *index* parameter value.

      iii. A string value, "Indexer out of range".

   b. At the end of the **get** accessor, call the **GetItemAtIndex** method. Pass the value of the index variable to the **GetItemAtIndex** method. Return the value of the **NodeData** property from the item that is retrieved by calling the **GetItemAtIndex** method.

   Your code should resemble the following code example.

```
public TItem this[int index]
{
    get
    {
        if (index < 0 || index >= Count)
        {
            throw new ArgumentOutOfRangeException
                ("index", index, "Indexer out of range");
        }

        return GetItemAtIndex(index).NodeData;
    }
    set
    {
        throw new NotImplementedException();
    }
}
```

5. Locate the **Clear** method. This method accepts no parameters, and does not return a value.

   This method should clear the contents of the tree and return it to a default state. Currently, the method throws a **NotImplementedException** exception.

MCT USE ONLY. STUDENT USE PROHIBITED

6. Replace the code in the **Clear** method with code to perform the following actions:

   a. Set the **LeftTree** property to **null**.

   b. Set the **RightTree** property to **null**.

   c. Set the **NodeData** property to the default value for a **TItem** object.

   Your code should resemble the following code example:

```
public void Clear()
{
    LeftTree = null;
    RightTree = null;
    NodeData = default(TItem);
}
```

7. Locate the **Contains** method. This method accepts a *TItem* parameter, *item*, and returns a Boolean value.

   This method should iterate through the tree and return a Boolean value that indicates whether a node that matches the value of the *item* parameter exists in the tree. Currently, the method throws a **NotImplementedException** exception.

8. Replace the code in the **Contains** method with code to perform the following actions:

   a. If the value of the **NodeData** property is the same as the value of the *item* parameter, return **true**.

   b. If the value of the **NodeData** property is greater than the value of the *item* parameter, and if the **LeftTree** property is not **null**, return the result of a recursive call to the **LeftTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   c. If the value of the **NodeData** property is less than the value of the *item* parameter, and if the **RightTree** property is not **null**, return the result of a recursive call to the **RightTree**.**Contains** method, passing the *item* parameter to the **Contains** method.

   d. At the end of the method, return **false**.

   Your code should resemble the following code example.

```
public bool Contains(TItem item)
{
    if (NodeData.CompareTo(item) == 0)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        return true;
    }

    if (NodeData.CompareTo(item) > 0)
    {
        if (this.LeftTree != null)
            return this.LeftTree.Contains(item);
    }
    else
    {
        if (this.RightTree != null)
            return this.RightTree.Contains(item);
    }

    return false;
}
```

9. Locate the **Count** property.

   This property is read-only, and should return an integer that represents the total number of items in the tree. Currently, the **get** accessor throws a **NotImplementedException** exception.

10. Replace the code in the **get** accessor with code to invoke the **GetCount** method, by passing **0** to the method call. Return the value that the **GetCount** method calculates.

    Your code should resemble the following code example.

```
public int Count
{
    get
    {
        return this.GetCount(0);
    }
}
```

11. Locate the **IsReadOnly** property.

    This property should return a Boolean value that signifies whether the tree is read-only.

12. Replace the code in the **get** accessor with a statement that returns the Boolean value **false**.

    Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
public bool IsReadOnly
{
    get
    {
        return false;
    }
}
```

13. Locate the **ICollection<TItem>.Remove** method. This method accepts a *TItem* parameter named *item*, and returns a Boolean value.

    This method should check whether a node with a value that matches the *item* parameter exists in the tree, and if so, remove the item from the tree. If an item is removed, the method should return **true**; otherwise, the method should return **false**.



Note: This version of the **Remove** method is fully qualified with the name of the interface. This is to disambiguate it from the local **Remove** method that is defined elsewhere in the **Tree** class.

14. In the **ICollection<TItem>.Remove** method, replace the existing code with statements that perform the following actions:

    a. If the tree contains a node that matches the value in the *item* parameter, call the local **Remove** method, and then return **true**.

    b. At the end of the method, return **false**.

    Your code should resemble the following code example.

```
bool ICollection<TItem>.Remove(TItem item)
{
    if (this.Contains(item) == true)
    {
        this.Remove(item);
        return true;
    }
    return false;
}
```

15. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 5: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file and examine the **Main** method.

   The BinaryTreeTestHarness project contains code that you will use to test the completed **BinaryTree** class. You will continue to extend the **BinaryTree** class in the following exercises, so the **BinaryTree** class is not currently complete. For this reason, this exercise does not use some methods in the test harness.

   The **Main** method contains method calls to each of the test methods that you are about to examine:

   - In Solution Explorer, in the BinaryTreeTestHarness project, double-click **Program.cs**.

2. Examine the **TestIntegerTree** method.

   The **TestIntegerTree** method tests the **Remove** and **Contains** methods, and the indexer functionality of the **BinaryTree** class. First, the method invokes the **CreateATreeOfIntegers** method to build a sample tree that contains 10 values. Then, the method invokes the **WalkTree** method, which prints each node value to the console in numerical order.



Note: The **CreateATreeOfIntegers** method creates a **Tree** object that contains the values 10, 5, 11, 5, –12, 15, 0, 14, –8, and 10 in the order that the method adds them.

   The method then invokes the **Count** method and prints the result to the console. The method casts the tree to an **ICollection** object, and then calls the **Remove** method to remove the value **11** from the tree. The method again prints the result of the **Count** method to the console to prove that an item has been removed.



Note: The **BinaryTree** method contains two **Remove** methods, and in this case, the test method should invoke the interface-defined **ICollection.Remove** method. To enable the test method to do this, it must cast the **Tree** object to an **ICollection** object.

   The method then tests the **Contains** method by invoking the **Contains** method with the value **11** (which has just been removed) and then −12 (which is known to exist in the list).

Finally, the method tests the tree indexer by first retrieving the index of the value **5** in the tree and printing the index to the console, and then using the same index to retrieve the value **5** from that position in the tree.

3. Examine the **TestDeleteRootNodeInteger** method.

   The **TestDeleteRootNodeInteger** method tests the functionality of the **Remove** method when it attempts to remove the tree root node. When the root node value is removed from the tree, the next available node should be copied into its place to enable the tree to continue to function.

   In this test, the root node has the value **10**. There is a second node with the value **10**, so the **Remove** method must be invoked twice to remove both values.

   The method first invokes the **CreateATreeOfIntegers** method to build a sample tree, and then prints the tree to the console by invoking the **WalkTree** method. The method then casts the **Tree** object to an **ICollection** object, and then invokes the **Remove** method twice to remove both values of **10**. Finally, the method again invokes the **WalkTree** method to verify that the tree still functions correctly.

4. Examine the **TestStringTree** method.

   This method uses similar logic to the **TestIntegerTree** method to test the **Count**, **Remove**, **Contains**, and indexer method functionality. This method uses a **BinaryTree** object that contains the string values "k203", "h624", "p936", "h624", "a279", "z837", "e762", "r483", "d776", and "k203". In this test, the **Remove** method is tested by using the "p936" string value, and the indexer is tested by using the "h624" string value.

5. Examine the **TestDeleteRootNodeString** method.

   This method uses similar logic to the **TestDeleteRootNodeInteger** method to test the **Remove** method functionality, using the same string-based tree as the **TestStringTree** method. In this test, the "k203" string value is removed twice to test root node removal.

6. Examine the **TestTestResultTree** method.

   This method uses similar logic to the **TestIntegerTree** and **TestStringTree** methods to test the **Count**, **Remove**, **Contains**, and indexer method functionality, but it uses a **BinaryTree** object based on the **TestResult** type.

MCT USE ONLY. STUDENT USE PROHIBITED



Note: The **TestResult** class implements the **IComparable** interface, and uses the **Deflection** property to compare instances of the **TestResult** object. Therefore, items in this tree are indexed by their **Deflection** property value.

In this case, the **Remove** method is tested with the **TestResult** object that has a **Deflection** value of **226**. The indexer is tested with the **TestResult** object that has a **Deflection** value of **114**.

7. Examine the **TestDeleteRootNodeTestResult** method.

   This method uses similar logic to the **TestDeleteRootNodeInteger and TestDeleteRootNodeString** methods to test the **Remove** method functionality, using the same **TestResult**-based tree as the **TestTestResultTree** method. In this test, the **TestResult** object that has a **Deflection** value of **190** is removed twice to test root node removal.

8. Run the BinaryTreeTestHarness application:

   - On the **Debug** menu, click **Start Without Debugging**.

9. Verify that the output in the console window resembles the following code example.

```
TestIntegerTree()
WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10
Remove(11)

Count: 9

Contains(11): False

Contains(-12): True
IndexOf(5): 3

tree[3]: 5
```

MCT USE ONLY. STUDENT USE PROHIBITED

Download from www.eBookTM.com

Note that:

a. The console shows the output of the **TestIntegerTree** method.

b. The tree is displayed in numerical order by the **WalkTree** method.

c. Initially, the list contains 10 items, and then after the **Remove** method is called, the tree contains nine items.

d. The **Remove** method removes the value **11**, so the result of the **Contains** method is **false**. Note also that the **Contains** method verifies the presence of the value **−12**.

e. The **IndexOf** method reports that the value **5** is in position 3 in the list. This is confirmed by retrieving the value in position 3, which is shown to be **5**.

10. Press ENTER, and then verify that the output in the console window resembles the following code example.

```
TestDeleteRootNodeInteger()
Before
-12
-8
0
5
5
10
10
11
14
15

Remove 10 twice

After
-12
-8
0
5
5
11
14
15
```

Note that the tree shows two instances of the value **10** in the first list. Then, after those values are removed, the list no longer contains them. Also note that, after removing the root node value, the tree retains the remaining values and continues to function as expected.

MCT USE ONLY. STUDENT USE PROHIBITED

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This is the same test as the one you performed in step 9, but it is performed by using string data. Items in the list are displayed in alphabetical order.

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
p936
r483
z837
```

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

MCT USE ONLY. STUDENT USE PROHIBITED

This test is the same as the one you performed in steps 9 and 11, but this test is based on **TestResult** objects. Items are displayed in numerical order based on the value of the **Deflection** property.

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

15. Press ENTER twice to return to Visual Studio.

Exercise 2: Implementing an Enumerator by Writing Code

Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex2\Starter folder:



Note: The CustomCollections solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 13\Ex2 \Starter** folder, click **CustomCollections.sln**, and then click **Open**.

Task 2: Create the TreeEnumerator class

- In the BinaryTree project, add a new class named **TreeEnumerator**. This class should implement the **IEnumerator** interface, and should take a type parameter, **TItem**, where the **TItem** type implements the **IComparable** interface:

  a. In Solution Explorer, right-click the **BinaryTree** project, point to **Add**, and then click **Class**.

  b. In the **Add New Item - BinaryTree** dialog box, in the **Name** box, type **TreeEnumerator** and then click **Add**.

  c. In the TreeEnumerator.cs file, modify the **TreeEnumerator** class definition to make the class generic, based on a type parameter called **TItem**. Specify that the class implements the **IEnumerator** generic interface, and that the **TItem** type parameter implements the **IComparable** generic interface.

  Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{

}
```

Task 3: Add class-level variables and a constructor

1. In the **TreeEnumerator** class, add the following members:

   a. A **Tree<TItem>** object named **currentData**, initialized to a **null** value.

   This member will store the initial **Tree** object data that is passed to the class when it is constructed, and will be used to populate the internal queue with data. The data is also stored to enable the internal queue to reset.

   b. A **TItem** object named **currentItem**, initialized to a default **TItem** object.

   This member will store the last item that is removed from the queue.

MCT USE ONLY. STUDENT USE PROHIBITED

c. A private **Queue<TItem>** object named **enumData**, initialized to a **null** value.

This member holds an internal queue of items that the enumerator will iterate over. You will populate this queue with the items in the **Tree** object.

Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{
    private Tree<TItem> currentData = null;

    private TItem currentItem = default(TItem);

    private Queue<TItem> enumData = null;
}
```

2. Add a constructor. The constructor should accept a *Tree<TItem>* parameter named *data*, and should initialize the **currentData** member with the value of this parameter.

Your code should resemble the following code example.

```
class TreeEnumerator<TItem> : IEnumerator<TItem>
    where TItem : IComparable<TItem>
{
    ...
    public TreeEnumerator(Tree<TItem> data)
    {
        this.currentData = data;
    }
}
```

Task 4: Add a method to populate the queue

- Below the constructor, add a new private method named **Populate**. The method should accept a *Queue<TItem>* parameter named *enumQueue*, and a *Tree<TItem>* parameter named *tree*. It should not return a value. Add code to the method to perform the following actions:

a. If the **LeftTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**LeftTree** property as parameters to the method.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Add the **tree**.**NodeData** property value of the *tree* parameter to the **enumQueue** queue.

c. If the **RightTree** property of the *tree* parameter is not **null**, recursively call the **Populate** method, passing the *enumQueue* parameter and the **tree**.**RightTree** property as parameters to the method.

This code walks the tree and fills the queue with each item that is found, in order.

Your code should resemble the following code example.

```
private void Populate(Queue<TItem> enumQueue, Tree<TItem> tree)
{
    if (tree.LeftTree != null)
    {
        Populate(enumQueue, tree.LeftTree);
    }

    enumQueue.Enqueue(tree.NodeData);

    if (tree.RightTree != null)
    {
        Populate(enumQueue, tree.RightTree);
    }
}
```

Task 5: Implement the IEnumerator<T> and IEnumerator methods

1. In the class definition, right-click **IEnumerator**, point to **Implement Interface**, and then click **Implement Interface Explicitly**.

   Visual Studio will generate stubs for the methods and properties that the **IEnumerator<>**, **IEnumerator**, and **IDisposable** interfaces expose.

2. Locate the **Current** property.

   This property should return the last **TItem** object that was removed from the queue.

3. In the **get** accessor of the **Current** property, replace the existing code with code to perform the following actions:

   a. If the **enumData** member is **null**, throw a new **InvalidOperationException** exception with the message "Use MoveNext before calling Current".

   b. Return the value of the **currentItem** member.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
TItem IEnumerator<TItem>.Current
{
    get
    {
        if (this.enumData == null)
        {
            throw new InvalidOperationException("Use MoveNext before
calling Current");
        }
        return this.currentItem;
    }
}
```

4. Locate the **MoveNext** method. The method accepts no parameters and returns a Boolean value.

   The **MoveNext** method should ensure that the internal queue is initialized, retrieve the next item from the internal queue, and then store it in the **currentItem** property. If the operation succeeds, the method returns **true**, otherwise, it returns **false**.

5. In the **MoveNext** method, replace the existing code with code to perform the following actions:

   a. If the **enumData** object is **null**, create a new queue object, and then invoke the **Populate** method, passing the new **queue** object and the **currentData** member as parameters to the method call.

   b. If the **enumData** object contains any values, retrieve the first item in the queue, store it in the **currentItem** member, and then return the Boolean value **true**.

   c. At the end of the method, return the Boolean value **false**.

   Your code should resemble the following code example.

```
bool System.Collections.IEnumerator.MoveNext()
{
    if (this.enumData == null)
    {
        this.enumData = new Queue<TItem>();
        Populate(this.enumData, this.currentData);
    }

    if (this.enumData.Count > 0)
    {
        this.currentItem = this.enumData.Dequeue();
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            return true;
        }
        return false;
    }
```

6. Locate the **Reset** method. This method accepts no parameters, and does not return a value.

   This method should reset the enumerator to its initial state. You do this by repopulating the internal queue with the data from the **Tree** object.

7. In the **Reset** method, replace the existing code with code that invokes the **Populate** method, passing the **enumData** and **currentData** members as parameters to the method.

   Your code should resemble the following code example.

```
void System.Collections.IEnumerator.Reset()
{
    Populate(this.enumData, this.currentData);
}
```

8. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**.

Task 6: Implement the IDisposable interface

1. In the **TreeEnumerator** class, locate the **Dispose** method. This method accepts no parameters and does not return a value.

   The method should dispose of the class, relinquishing any resources that may not be reclaimed if they are not disposed of explicitly, such as file streams and database connections.



Note: The **Queue** object does not implement the **IDisposable** interface, so you will use the **Dispose** method of the **TreeEnumerator** class to clear the queue of any data.

2. In the **Dispose** method, replace the existing code with code that clears the **enumQueue** queue object.



Hint: Use the **Clear** method of the **Queue** class to empty a **Queue** object.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
void IDisposable.Dispose()
{
    this.enumData.Clear();
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 7: Modify the Tree class to return a TreeEnumerator object

1. In the task list, locate the **TODO - Update the Tree class to return the TreeEnumerator class** task, and then double-click this task.

   This task is located in the **Tree** class.

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

   c. Double-click the **TODO - Update the Tree class to return the TreeEnumerator class** task.

2. Remove the comment. In the **GetEnumerator** method, replace the existing code with code that creates and initializes a new **TreeEnumerator** object. Specify the **TItem** type as the type parameter, and pass the current object as the parameter to the **TreeEnumerator** constructor. Return the **TreeEnumerator** object that is created.

   Your code should resemble the following code example.

```
public IEnumerator<TItem> GetEnumerator()
{
    return new TreeEnumerator<TItem>(this);
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 8: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

MCT USE ONLY. STUDENT USE PROHIBITED

This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 1. However, it has been updated to test the enumerator functionality that you just added:

- In Solution Explorer, in the BinaryTreeTestHarness project, double-click **Program.cs**.

2. Examine the **TestIteratorsIntegers** method.

   This method tests the iterator functionality that you just implemented, by using the same integer tree as in Exercise 1. The method builds the tree by invoking the **CreateATreeOfIntegers** method, and then uses a **foreach** statement to iterate through the list and print each value to the console. The method then attempts to iterate through the tree in reverse order, and print each item to the console.



Note: You will add the functionality to enable reverse iteration of the tree in the next exercise. It is expected that attempting to reverse the tree will throw a **NotImplementedException** exception. The **TestIteratorsIntegers** method will catch this exception when it occurs, and print a message to the console.

3. Examine the **TestIteratorsStrings** method.

   This method uses similar logic to the **TestIteratorsIntegers** method to test the iterator functionality of the **BinaryTree** object, but it uses the same string-based tree as the one you used in Exercise 1. The method uses the **CreateATreeOfStrings** method to build the tree, iterates through the tree, and then prints all items to the console. This method also attempts to display the data in the tree in reverse order, and will encounter a **NotImplementedException** exception (you will implement this feature in the next exercise).

4. Examine the **TestIteratorsTestResults** method.

   This method uses similar logic to the **TestIteratorsIntegers** and **TestIteratorsStrings** methods to test the iterator functionality of the **BinaryTree** object. It uses a **TestResult**-based tree by invoking the **CreateATreeOfTestResults** method as in Exercise 1.

5. Run the BinaryTreeTestHarness application:

   - On the **Debug** menu, click **Start Without Debugging**.

6. Verify that the output in the console window matches the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
5
10
10
11
14
15

Count: 10

Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercise 1, and confirms that you have not compromised existing functionality by adding the iterator functionality.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Remove 10 twice

After
-12
-8
0
5
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercise 1, and again confirms that existing functionality works as expected.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that the items in the list are displayed in numerical order, and note that the **Reverse** method displays a message that indicates that the **Reverse** functionality is not yet implemented.

9. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestStringTree()

WalkTree()
a279
d776
e762
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercise 1.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice

After
a279
d776
e762
h624
h624
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
p936
r483
z837
```

This output matches the **TestDeleteRootNodeString** method output from Exercise 1.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in step 8. It uses string data to verify the iterator functionality, and all items are displayed in alphabetical order.

12. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercise 1.

13. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercise 1.

MCT USE ONLY. STUDENT USE PROHIBITED

14. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Not Implemented. You will implement this functionality in Exercise 3
```

Note that this represents the same test as you performed in steps 8 and 11. It uses **TestResult** object data to verify the iterator functionality, and all items are displayed in numerical order based on the value of the **Deflection** property.

15. Press ENTER twice to return to Visual Studio.

Exercise 3: Implementing an Enumerator by Using an Iterator

Task 1: Open the CustomCollections solution

- Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex3\Starter folder:



Note: The CustomCollections solution in the Ex3 folder is functionally the same as the code that you completed in Exercise 2. However, it includes an updated task list and an updated test harness to enable you to complete this exercise.

a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

b. In the Open Project dialog box, move to the E:\Labfiles\Lab 13\Ex3\Starter folder, click CustomCollections.sln, and then click Open.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Add an enumerator to return an enumerator that iterates through data in reverse order

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Add a method to return the list in reverse order** task, and then double-click this task.

   This task is located at the end of the **Tree** class.
3. Remove the task comment, and then add a new public method named **Reverse**. The method should accept no parameters, and return an **IEnumerable** collection based on the **TItem** type parameter.

   Your code should resemble the following code example.

```
public IEnumerable<TItem> Reverse()
{

}
```

4. Add code to the method to perform the following actions:
   a. If the **RightTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **RightTree** property, and then yield each item that is found.



Hint: The **yield** statement is used in an **iterator** block to return a value to the enumerator object, or to signal the end of an iteration.

   b. Yield the value in the **NodeData** property of the current item.
   c. If the **LeftTree** property is not **null**, iterate through the items that are returned by calling the **Reverse** method of the **LeftTree** property, and then yield each item that is found.

   Your code should resemble the following code example.

```
public IEnumerable<TItem> Reverse()
{
    if (this.RightTree != null)
    {
```

```
        foreach (TItem item in this.RightTree.Reverse())
        {
            yield return item;
        }
    }

    yield return this.NodeData;

    if (this.LeftTree != null)
    {
        foreach (TItem item in this.LeftTree.Reverse())
        {
            yield return item;
        }
    }

}
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 3: Use the BinaryTreeTestHarness application to test the solution

1. In the BinaryTreeTestHarness project, open the Program.cs file.

   This version of the BinaryTreeTestHarness project contains the same code and performs the same tests as in Exercise 2. Now that you have implemented the **Reverse** method in the **BinaryTree** object, the test application should not encounter the **NotImplementedException** exception in the **TestIteratorsIntegers**, **TestIteratorsStrings**, and **TestIteratorsTestResults** methods.

2. Run the BinaryTreeTestHarness application:
   - On the **Debug** menu, click **Start Without Debugging**.

3. Verify that the output in the console window matches the following code example.

```
TestIntegerTree()

WalkTree()
-12
-8
0
5
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
5
10
10
11
14
15

Count: 10

Remove(11)

Count: 9

Contains(11): False

Contains(-12): True

IndexOf(5): 3

tree[3]: 5
```

This output matches the **TestIntegerTree** method output from Exercises 1 and 2, and confirms that you have not compromised existing functionality by adding the reverse iterator functionality.

4. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeInteger()

Before
-12
-8
0
5
5
10
10
11
14
15

Remove 10 twice

After
-12
-8
0
5
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
5
11
14
15
```

This output matches the **TestDeleteRootNodeInteger** method output from Exercises 1 and 2, and again confirms that the existing functionality works as expected.

5. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsIntegers()

In ascending order
-12
-8
0
5
5
10
10
11
14
15

In descending order
15
14
11
10
10
5
5
0
-8
-12
```

This output is similar to the **TestIteratorsIntegers** method in Exercise 2, but the **Reverse** method is now implemented, so the tree is also displayed in descending numerical order.

6. Press ENTER, and then verify that the output in the console window matches the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
TestStringTree()

WalkTree()
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Count: 10

Remove("p936")

Count: 9

Contains("p936"): False

Contains("a279"): True

IndexOf("h624"): 3

tree[3]: h624
```

This output matches the **TestStringTree** method output from Exercises 1 and 2.

7. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeString()

Before
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

Remove k203 twice
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
After
a279
d776
e762
h624
h624
p936
r483
z837
```

This output matches the **TestDeleteRootNodeString** method output from Exercises 1 and 2.

8. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsStrings()

In ascending order
a279
d776
e762
h624
h624
k203
k203
p936
r483
z837

In descending order
z837
r483
p936
k203
k203
h624
h624
e762
d776
a279
```

This test uses string data to verify the iterator functionality, and all items are displayed in alphabetical order, and then reverse alphabetical order.

9. Press ENTER, and then verify that the output in the console window matches the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
TestTestResultTree()

WalkTree()
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

Count: 10

Remove(def266)

Count: 9

Contains(def266): False

Contains(def0): True

IndexOf(def114): 3

tree[3]: Deflection: 114, AppliedStress: 40, Temperature: 200, Date:
3/18/2010
```

This output matches the **TestTestResultTree** method output from Exercises 1 and 2.

10. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestDeleteRootNodeTestResults()

Before
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
Remove def190 twice

After
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
```

This output matches the **TestDeleteRootNodeTestResults** method output from Exercises 1 and 2.

11. Press ENTER, and then verify that the output in the console window matches the following code example.

```
TestIteratorsTestResults()

In ascending order
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010

In descending order
Deflection: 342, AppliedStress: 100, Temperature: 200, Date: 3/18/2010
Deflection: 304, AppliedStress: 90, Temperature: 200, Date: 3/18/2010
Deflection: 266, AppliedStress: 80, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 70, Temperature: 200, Date: 3/18/2010
Deflection: 190, AppliedStress: 60, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 50, Temperature: 200, Date: 3/18/2010
Deflection: 114, AppliedStress: 40, Temperature: 200, Date: 3/18/2010
Deflection: 76, AppliedStress: 30, Temperature: 200, Date: 3/18/2010
Deflection: 38, AppliedStress: 20, Temperature: 200, Date: 3/18/2010
Deflection: 0, AppliedStress: 10, Temperature: 200, Date: 3/18/2010
```

This test uses **TestResult** object data to verify iterator functionality. Therefore, all items are displayed in numerical order based on the value of the **Deflection** property, and then the list is reversed to display data in descending numerical order based on the value of the **Deflection** property.

MCT USE ONLY. STUDENT USE PROHIBITED

12. Press ENTER twice to return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Using LINQ to Query Data

Lab 14: Using LINQ to Query Data

Exercise 1: Using the LINQ Query Operators

Task 1: Open the starter solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 14\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** list, click **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 14\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 14\Ex1\Starter** folder, click **StressDataAnalyzer.sln**, and then click **Open**.
5. Examine the user interface (UI) for the StressDataAnalyzer application. Note the following features of the application:
   - The stress test data is generated by a stress test device. The data is stored in a binary data file, and this application reads the data from this file when the application starts to run. The application holds the data in memory by using a **Tree** object.

- The UI contains two main areas. The upper area enables the user to specify criteria to match stress data. The lower area displays the data.
- The stress test data criteria are:
  i. The date that the test was performed.
  ii. The temperature at which the test was performed.
  iii. The stress that was applied during the test.
  iv. The deflection that resulted from applying the stress.

  Each criterion is specified as a range by using the slider controls.
- After selecting the criteria to match, the user clicks **Display** to generate a Language-Integrated Query (LINQ) query that fetches the matching data from the **Tree** object in memory and shows the results.

a. In Solution Explorer, expand the **StressDataAnalyzer** project.

b. Double-click the **DataAnalyzer.xaml** file.

Task 2: Declare variables to specify the stress data file name and the Tree object

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. In the task list, locate the **TODO - Declare filename and tree variables** task, and then double-click this task. This task is located in the **DataAnalyzer.xaml.cs** class.
3. Delete the **TODO - Declare filename and tree variables** comment, and then add code to declare the following variables:
   a. A private constant **string** object named stressDataFilename. Initialize the object with the string "E:\Labfiles\Lab 14\StressData.dat". This is the name of the data file that holds the stress data.
   b. A private **Tree** object named stressData that is based on the **TestResult** type. This **Tree** object will hold the data that is read from the stress data file. Initialize this object to null.

MCT USE ONLY. STUDENT USE PROHIBITED

The **TestResult** type is a struct that contains the following four fields, corresponding to the data for each stress test record:

- **TestDate**. This is a **DateTime** field that contains the date on which the stress test was performed.
- **Temperature**. This is a **short** field that contains the temperature, in Kelvin, at which the test was performed.
- **AppliedStress**. This is another **short** field that specifies the stress, in kiloNewtons (kN), that was applied during the test.
- **Deflection**. This is another **short** field that specifies the deflection of the girder, in millimeters (mm), when the stress was applied.

The **TestResult** type implements the **IComparable** interface. The comparison of test data is based on the value of the **Deflection** field.

Your code should resemble the following code example.

```
public partial class DataAnalyzer : Window
{

    // Declare a string variable to hold the name of the file
    // that contains the stress test data.
    private const string stressDataFilename =
        @"E:\Labfiles\Lab 14\StressData.dat";

    // Declare a Tree variable to hold the loaded data.
    private Tree<TestResult> stressData = null;

    public DataAnalyzer()
    ...
}
```

Task 3: Add a method to read the test data

1. In the task list, locate the **TODO - Add a method to read the contents of the StressData file** task, and then double-click this task.
2. Delete the **TODO - Add a method to read the contents of the StressData file** comment, and then add the method in the following code example, which is named **ReadTestData**. This method reads the stress data from the file and populates the **Tree** object. It is not necessary for you to fully understand how this method works, so you can either type this code manually, or you can use the Mod14ReadTestData code snippet.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void ReadTestData()
{
    // Open a stream over the file that holds the test data.
    using (FileStream readStream =
        File.Open(stressDataFilename, FileMode.Open))
    {
        // The data is serialized as TestResult instances.
        // Use a BinaryFormatter object to read the stream and
        // deserialize the data.
        BinaryFormatter formatter = new BinaryFormatter();
        TestResult initialNode =
            (TestResult)formatter.Deserialize(readStream);

        // Create the binary tree and use the first item retrieved
        // as the root node. (Note: The tree will likely be
        // unbalanced, because it is probable that most nodes will
        // have a value that is greater than or equal to the value in
        // this root node - this is because of the way in which the
        // test results are generated and the fact that the TestResult
        // class uses the deflection as the discriminator when it
        // compares instances.)
        stressData = new Tree<TestResult>(initialNode);

        // Read the TestResult instances from the rest of the file
        // and add them into the binary tree.
        while (readStream.Position < readStream.Length)
        {
            TestResult data =
                (TestResult)formatter.Deserialize(readStream);
            stressData.Insert(data);
        }
    }
}
```

- To use the code snippet, type **Mod14ReadTestData** and then press the TAB key twice.

Task 4: Read the test data by using a BackgroundWorker object

1. In the **Window_Loaded** method, add code to perform the following tasks:
   a. Create a **BackgroundWorker** object named **workerThread**.
   b. Configure the **workerThread** object; the object should not report progress or support cancellation.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{

    // Read the test data and populate the binary tree.
    // Use a BackgroundWorker object to avoid tying up the UI.

    BackgroundWorker workerThread = new BackgroundWorker();
    workerThread.WorkerReportsProgress = false;
    workerThread.WorkerSupportsCancellation = false;

}
```

2. In the **Window_Loaded** method, add an event handler for the **workerThread.DoWork** event. When the event is raised, the event handler should invoke the **ReadTestData** method.

   Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...

    workerThread.DoWork += (o, args) =>
        {
            this.ReadTestData();
        };

}
```

3. Add an event handler for the **workerThread**.**RunWorkerComplete** event. When the event is raised, the event handler should perform the following tasks:

   a. Enable the **displayResults** button.

   b. Display the message 'Ready' in the **statusMessage** StatusBarItem in the status bar at the bottom of the Windows Presentation Foundation (WPF) window.



Hint: Set the **Content** property of a status bar item to display a message in that item.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...
    workerThread.RunWorkerCompleted += (o, args) =>
        {
            this.displayResults.IsEnabled = true;
            this.statusMessage.Content = "Ready";
        };
}
```

4. At the end of the **Window_Loaded** method, add code to perform the following tasks:

    a. Start the **workerThread BackgroundWorker** object running asynchronously.

    b. Display the message "Reading Test Data..." in the **statusMessage** item in the status bar at the bottom of the WPF window.

    Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...
    workerThread.RunWorkerAsync();
    this.statusMessage.Content = "Reading test data ...";
}
```

Task 5: Define the LINQ query

1. In the task list, locate the **TODO - Define the LINQ query** task, and then double-click this task. This task is located in the **CreateQuery** method.
2. Replace the existing code in the method with code that defines an **IEnumerable<TestResult>** object called **query**. Initialize the query variable with a LINQ query that retrieves all of the **TestResult** objects in the **stressData** tree that meet the following criteria. The query should order returned values by the **TestDate** property. The query should evaluate each object by using the following criteria:

    a. The value of the **TestDate** property is greater than or equal to the *dateStart* parameter value.

    b. The value of the **TestDate** property is less than or equal to the *dateEnd* parameter value.

MCT USE ONLY. STUDENT USE PROHIBITED

c. The value of the **Temperature** property is greater than or equal to the **temperatureStart** parameter value.

d. The value of the **Temperature** property is less than or equal to the *temperatureEnd* parameter value.

e. The value of the **AppliedStress** property is greater than or equal to the *appliedStressStart* parameter value.

f. The value of the **AppliedStress** property is less than or equal to the *appliedStressEnd* parameter value.

g. The value of the **Deflection** property is greater than or equal to the *deflectionStart* parameter value.

h. The value of the **Deflection** property is less than or equal to the *deflectionEnd* parameter value.

Your code should resemble the following code example.

```
private IEnumerable<TestResult> CreateQuery
    (DateTime dateStart, DateTime dateEnd, short temperatureStart,
     short temperatureEnd, short appliedStressStart, short
     appliedStressEnd, short deflectionStart, short deflectionEnd)
{

    IEnumerable<TestResult> query =
        from result in stressData
        where result.TestDate >= dateStart &&
              result.TestDate <= dateEnd &&
              result.Temperature >= temperatureStart &&
              result.Temperature <= temperatureEnd &&
              result.AppliedStress >= appliedStressStart &&
              result.AppliedStress <= appliedStressEnd &&
              result.Deflection >= deflectionStart &&
              result.Deflection <= deflectionEnd
        orderby result.TestDate
        select result;
}
```

3. At the end of the method, return the **query** object.

Your code should resemble the following code example.

```
private IEnumerable<TestResult> CreateQuery
    (DateTime dateStart, DateTime dateEnd, short temperatureStart,
     short temperatureEnd, short appliedStressStart, short
     appliedStressEnd, short deflectionStart, short deflectionEnd)
{
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    ...
        select result;

    return query;
}
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 6: Execute the query

1. In the task list, locate the **TODO - Execute the LINQ query** task, and then double-click this task. This task is located in the **FormatResults** method. This method takes an enumerable collection of **TestResult** objects as a parameter and generates a string that contains a formatted list of **TestResult** objects. The parameter is the item that the **CreateQuery** method returns. Iterating through this list runs the LINQ query.
2. Delete the **TODO - Execute the LINQ query** comment, and then add code to the **FormatResults** method to perform the following task:
   - For each item that the query returns, format and append the details of each item to the **builder** StringBuilder object. Each item should be formatted to display the following properties in a double-tab delimited format:
     i. **TestDate**
     ii. **Temperature**
     iii. **AppliedStress**
     iv. **Deflection**

   Your code should resemble the following code example.

```
private string FormatResults(IEnumerable<TestResult> query)
{
    ...
    builder.Append
        ("Test Date\t\tTemperature\tApplied Stress\tDeflection\n");

    // Iterate through the results and format each item found.
    foreach (var item in query)
    {
        builder.Append(String.Format("{0:d}\t\t{1}\t\t{2}\t\t{3}\n",
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            item.TestDate,
            item.Temperature,
            item.AppliedStress,
            item.Deflection));
    }

    // Return the string that is constructed by using the
    // StringBuilder object.

    return builder.ToString();

}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 7: Run the query by using a BackgroundWorker object

1. In the task list, locate the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** task, and then double-click this task. This task is located in the **DisplayResults_Click** method. This method calls the **CreateQuery** method to generate the LINQ query that matches the criteria that the user specifies, and it then runs the query to generate and format the results by using a **BackgroundWorker** object called **workerThread**.
2. Delete the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** comment, and then define an event handler for the **workerThread.DoWork** event. Add code to the event handler to invoke the **FormatResults** method, passing the **query** object as the parameter to the method. Store the value that the method returns in the *Result* parameter of the **DoWork** event handler.

   Your code should resemble the following code example.

```
private void DisplayResults_Click(object sender, RoutedEventArgs e)
{

    try
    {
        ...

        workerThread.WorkerSupportsCancellation = false;

        // Return the formatted string as the result of the background
        // operation.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        workerThread.DoWork += (o, args) =>
            {
                args.Result = FormatResults(query);
            };

        ...
    }

    ...
}
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 8: Display the results

1. Below the event handler for the **DoWork** event, add an event handler for the **workerThread**.**RunWorkerComplete** event. Add code to the event handler to perform the following tasks:
   a. Update the **results**.**Text** property with the value of the *Result* parameter of the **RunWorkerComplete** event handler.
   b. Enable the **displayResults** button.
   c. Update the **statusMessage** status bar item to "Ready".

   Your code should resemble the following code example.

```
private void DisplayResults_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...

        workerThread.DoWork += (o, args) =>
        {
            args.Result = FormatResults(query);
        };

        // When the BackgroundWorker object has completed reading
        // the test data, display the results, set the status bar
        // to "Ready", and enable the displayResults button.

        workerThread.RunWorkerCompleted += (o, args) =>
        {
            this.results.Text = args.Result as string;
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
            this.displayResults.IsEnabled = true;
            this.statusMessage.Content = "Ready";
        };
        ...
    }
    ...
}
```

2. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 9: Test the solution

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. Click **Display**, and make a note of the **Time (ms)** value that is displayed next to the **Display** button.
3. Click **Display** two more times. The times for these operations will probably be lower than the time that the initial query took because the various internal data structures have already been initialized. Make a note of these times.



Note: The time that is displayed is the time that is required to fetch the data by using the LINQ query, but not the time that is taken to format and display this data. This is why the "Fetching results" message appears for several seconds after the data has been retrieved.

4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 40,641 values.
5. Use the **DatePicker** and slider controls to modify the search criteria to the values in the following table, and then click **Display** again.

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

6. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values. Note the time that it

MCT USE ONLY. STUDENT USE PROHIBITED

took to complete the search—the time should be less than the times that you recorded in Step 3. Keep a note of these values for comparison in Exercise 2.

7. Close the Stress Data Analyzer window, and then return to Visual Studio.

   Currently, any search through the data uses all four criteria—date, temperature, applied stress, and deflection—regardless of the values that are specified in the UI. If the user does not change the default values for any criteria, the LINQ query that the application generates still contains criteria for each field. This is rather inefficient. However, you can construct dynamic LINQ queries to enable you to generate a custom query that is based only on the criteria that are specified at run time. You will implement this functionality in the next exercise.

Exercise 2: Building Dynamic LINQ Queries

Task 1: Open the StressDataAnalyzer solution

1. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex2\Starter folder:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 14\Ex2\Starter** folder, click **StressDataAnalyzer.sln**, and then click **Open**.

2. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

3. Examine the modified UI for the StressDataAnalyzer application. Note the following features of the application:

   - The UI is an extended version of that used in Exercise 1. The user can specify which criteria to apply by using check boxes. Any criteria that are not selected are not included in the LINQ query.
   - The user can change the order in which the data is displayed by selecting the appropriate option button in the **Order By** section of the window.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- The user can limit the number of items that a query returns by selecting the **Limit** check box and by using the slider control to specify the number of items.

a. In Solution Explorer, expand the **StressDataAnalyzer** project.

b. Double-click the **DataAnalyzer.xaml** file.

Task 2: Dynamically build a lambda expression for the query criteria

1. In the task list, locate the **TODO - Complete the BuildLambdaExpressionForQueryCriteria method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionForQueryCriteria** method.

   The **BuildLambdaExpressionForQueryCriteria** method dynamically constructs a lambda expression from the values that are passed in as parameters. There are 12 parameters, which are divided into four groups. The *dateRangeSpecified* parameter is a Boolean value that indicates whether the user has selected the date criteria in the window, and the *startDate* and *endDate* parameters contain the start date and end date values that the user specifies. If the *dateRangeSpecified* parameter is **false**, the date is not included in the criteria for matching stress data. The same logic applies to the remaining parameters.

   The value that the **BuildLambdaExpressionForQueryCriteria** method returns is an **Expression** object. The **Expression** type represents a strongly typed lambda expression as a data structure in the form of an expression tree. The type parameter is a delegate that indicates the form of the lambda expression. In the **BuildLambdaExpressionForQueryCriteria** method, the lambda expression takes a **TestResult** object and returns a Boolean value that indicates whether this object should be included in the results that are generated by running the lambda expression.

   The existing code in this method creates a reference to an **Expression** object named **lambda**. You will add code to populate this object with an expression tree that represents a lambda expression that matches the query criteria that the 12 parameters specify. If the user does not specify any query criteria, this method returns a null value.



Note: The **Expression** type is located in the **System.Linq.Expressions** namespace. The application creates an alias for this namespace called **Expressions**. You cannot refer to the **Expression** type without the qualifying namespace in a WPF application because the WPF assemblies also contain a type called **Expression**.

2. Delete the **TODO - Complete the BuildLambdaExpressionForQueryCriteria method** comment, and then add code to perform the following tasks:

   a. Create a **Type** reference for the **TestResult** type named **testResultType**.



Hint: Creating a type reference in this way enables you to repeatedly refer to an object type without repeatedly calling the **typeof** method. The **typeof** method is a relatively costly method compared to retrieving an object reference.

   b. Create an **Expressions.ParameterExpression** object named **itemBeingQueried** by using the **Expressions.Expression.Parameter** static method. Specify the **testResultType** type reference as the type of the parameter, and use the string **"item"** as the name of the parameter.



Hint: The string that is passed as the second parameter to the method call defines how your lambda expression will refer to the object that is being queried. In this example, one part of the resultant expression will resemble **"item.TestDate >= startDate"**.

Your code should resemble the following code example.

```
private Expressions.Expression<Func<TestResult, bool>>
    BuildLambdaExpressionForQueryCriteria
    (...)
{
    ...
    if (dateRangeSpecified || temperatureRangeSpecified ||
        appliedStressRangeSpecified || deflectionRangeSpecified)
    {
        // Create the expression that defines the parameter for the
        // lambda expression.
        // The type is TestResult, and the lambda expression refers to
        // it with the name "item".
        Type testResultType = typeof(TestResult);
        Expressions.ParameterExpression itemBeingQueried =
            Expressions.Expression.Parameter(testResultType, "item");
        ...
    }
    ...
}
```

3. Add code to the method to create the following **Expressions.BinaryExpression** objects; each object should have an initial value of null:

MCT USE ONLY. STUDENT USE PROHIBITED

a. **dateCondition**

b. **temperatureCondition**

c. **appliedStressCondition**

d. **deflectionCondition**

You will populate these expression objects with query criteria that match the parameters that are passed in to the method. You will then combine these expression objects together to form the complete lambda expression tree.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...
    // Create expressions for each of the possible conditions.
    Expressions.BinaryExpression dateCondition = null;
    Expressions.BinaryExpression temperatureCondition = null;
    Expressions.BinaryExpression appliedStressCondition = null;
    Expressions.BinaryExpression deflectionCondition = null;
    ...
}
...
```

4. Add code to the method to invoke the **BuildDateExpressionBody** method, and store the result in the **dateCondition** object. Pass the following values as parameters to the method call:

   a. **dateRangeSpecified**

   b. **startDate**

   c. **endDate**

   d. **testResultType**

   e. **itemBeingQueried**



Note: The **BuildDateExpressionBody** method returns a **BinaryExpression** object that checks the stress test data against the **startDate** and **endDate** values. You will update the **BuildDateExpressionBody** method in the following task.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...

    // Build Boolean expressions for each of the possible criteria
    // that the user specifies.
    // These method calls may return null if the user did not
    // specify criteria for a property.
    dateCondition = BuildDateExpressionBody(
        dateRangeSpecified, startDate, endDate,
        testResultType, itemBeingQueried);
    ...
}
```

5. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **temperatureCondition** object. Pass the following values as parameters to the method call:
   a. **temperatureRangeSpecified**
   b. **fromTemperature**
   c. **toTemperature**
   d. **testResultType**
   e. A string that contains the value "**Temperature**"
   f. **itemBeingQueried**



Note: The **BuildNumericExpressionBody** method also returns a **BinaryExpression** object that will form part of the dynamic LINQ query. In this case, the data that this part of the query checks will contain numeric data rather than a **DateTime** value, and the name of the field that is being checked is **Temperature**. You will update the **BuildNumericExpressionBody** method later in the lab.

Your code should resemble the following code example.

```
dateCondition = BuildDateExpressionBody(
     dateRangeSpecified, startDate, endDate,
     testResultType, itemBeingQueried);

temperatureCondition = BuildNumericExpressionBody(
    temperatureRangeSpecified, fromTemperature, toTemperature,
    testResultType, "Temperature", itemBeingQueried);
```

MCT USE ONLY. STUDENT USE PROHIBITED

6. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **appliedStressCondition** object. Pass the following values as parameters to the method call:
   a. **appliedStressRangeSpecified**
   b. **fromStressRange**
   c. **toStressRange**
   d. **testResultType**
   e. A string that contains the value "**AppliedStress**"
   f. **itemBeingQueried**

   Your code should resemble the following code example.

```
temperatureCondition = BuildNumericExpressionBody(
    temperatureRangeSpecified, fromTemperature, toTemperature,
    testResultType, "Temperature", itemBeingQueried);

appliedStressCondition = BuildNumericExpressionBody(
    appliedStressRangeSpecified, fromStressRange, toStressRange,
    testResultType, "AppliedStress", itemBeingQueried);
```

7. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **deflectionCondition** object. Pass the following values as parameters to the method call:
   a. **deflectionRangeSpecified**
   b. **fromDeflection**
   c. **toDeflection**
   d. **testResultType**
   e. A string that contains the value "**Deflection**"
   f. **itemBeingQueried**

   Your code should resemble the following code example.

```
appliedStressCondition = BuildNumericExpressionBody(
    appliedStressRangeSpecified, fromStressRange, toStressRange,
    testResultType, "AppliedStress", itemBeingQueried);

deflectionCondition = BuildNumericExpressionBody(
    deflectionRangeSpecified, fromDeflection, toDeflection,
    testResultType, "Deflection", itemBeingQueried);
```

MCT USE ONLY. STUDENT USE PROHIBITED

8. Add code to the method to invoke the **BuildLambdaExpressionBody** method, and store the result in a new **Expressions.Expression** object named **body**. Pass the **dateCondition**, **temperatureCondition**, **appliedStressCondition**, and **deflectionCondition** objects as parameters to the method.



Note: The **BuildLambdaExpressionBody** method takes the four expression objects, each of which evaluate a single property in a **TestResult** object, and combines them into a complete lambda expression that evaluates all of the properties that the user specifies criteria for. You will complete the **BuildLambdaExpressionBody** method later in the lab.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...

    // Combine the Boolean expressions together into a single body.
    Expressions.Expression body = BuildLambdaExpressionBody(
        dateCondition, temperatureCondition,
        appliedStressCondition, deflectionCondition);
    ...

}
```

9. Add code to the method to invoke the **Expression.Lambda** generic method, and store the response in the **lambda** object. The **Expression.Lambda** method should construct a lambda expression from the body of the lambda expressions in the **body Expression** object and the **itemBeingQueried ParameterExpression** object. Specify the delegate type **Func<TestResult, bool>** as the type parameter of the method.



Hint: The static **Expression.Lambda** method constructs an expression tree that represents a completed lambda expression, including the data that is being queried by the expression.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Build the lambda expression by using the parameter and the
    // body expressions.

    lambda = Expressions.Expression.Lambda<Func<TestResult, bool>>(
        body, itemBeingQueried);

}
```

10. Build the project and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Dynamically build the date expression tree

1. In the task list, locate the **TODO - Complete the BuildDateExpressionBody method** task, and then double-click this task. This task is located in the **BuildDateExpressionBody** method.

   The existing code in this method defines a **BinaryExpression** object named **dateCondition**. This object will be used to return the expression tree that evaluates date values. The method then checks that the *dateRangeSpecified* parameter is **true**. You will add code to this conditional statement to build an expression tree that is equivalent to the condition in the following code example.

```
item.TestDate >= startDate && item.TestDate <= endDate
```

   If the user did not specify any date criteria, this method returns a null expression tree.

2. Delete the **TODO - Complete the BuildDateExpressionBody method** comment, and then add code to create a new **MemberInfo** object named **testDateProperty**. Call the **GetProperty** method to generate code that retrieves the **TestDate** property from the *testResultType* type parameter. Pass the string "TestDate" as the parameter to the **GetProperty** method.

   Your code should resemble the following code example.

```
if (dateRangeSpecified)
{

    // Generate the expression:
    //
    //    item.TestDate >= startDate
    //
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    MemberInfo testDateProperty =
        testResultType.GetProperty("TestDate");

    ...
}
```

3. Add code to the method to create a **MemberExpression** object named **testDateMember**. Populate the object with the value that is returned by calling the **Expression**.**MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testDateProperty** value as parameters to the method.



Note: A **MemberExpression** object is an expression that represents access to a property of the object that is being queried. In this case, the object represents the ***item*.TestDate** property.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    MemberInfo testDateProperty =
        testResultType.GetProperty("TestDate");

    Expressions.MemberExpression testDateMember =
        Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testDateProperty);
}
```

4. Add code to create an **Expressions.ConstantExpression** object named **lowerDate**, and populate the object with the result of calling the **Expression.Expressions.Constant** method. Pass the *startDate* parameter as a parameter to the method call.



Note: A **ConstantExpression** object is an expression that represents the results of evaluating a constant value. In this case, the object represents the value in the startDate variable.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.MemberExpression testDateMember =
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
        Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testDateProperty);

    Expressions.ConstantExpression lowerDate =
        Expressions.Expression.Constant(startDate);
}
```

5. Add code to create an **Expressions.BinaryExpression** object named **lowerDateCondition**, and populate the object with the result of calling the **Expressions.Expression.GreaterThanOrEqual** method. Pass the **testDateMember** and **lowerDate** objects as parameters to the method call.



Note: The **GreaterThanOrEqual** method generates a binary expression that combines the **testDateMember** object (representing the "this.startDate" portion of the expression) and the **lowerDate** object (representing the "startDate" portion of the expression) to generate a tree for the expression **"this.startDate >= startDate"**.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.ConstantExpression lowerDate =
        Expressions.Expression.Constant(startDate);
    Expressions.BinaryExpression lowerDateCondition =
        Expressions.Expression.GreaterThanOrEqual(
            testDateMember, lowerDate);
}
```

6. By using the same principles that you saw in Steps 4 and 5, add code to perform the following tasks:
   a. Create a **ConstantExpression** object named **upperDate** by passing the *endDate* parameter as a parameter to the method call.
   b. Create a **BinaryExpression** object named **upperDateCondition** by invoking the **Expression.LessThanOrEqual** method. Pass the **testDateMember** and **upperDate** objects as parameters to the method call.



Note: This code should build the second part of the date evaluation expression, which represents **"endDate <= testDateMember"**.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.BinaryExpression lowerDateCondition =
        Expressions.Expression.GreaterThanOrEqual
        (testDateMember, lowerDate);
    // Generate the expression:
    //
    //    item.Testdate <= endDate
    //
    Expressions.ConstantExpression upperDate =
        Expressions.Expression.Constant(endDate);
    Expressions.BinaryExpression upperDateCondition =
        Expressions.Expression.LessThanOrEqual(
            testDateMember, upperDate);
}
```

7. Add code to combine the expressions in the **lowerDate** and **upperDate** **ExpressionTree** objects into a single Boolean expression tree that returns **true** if both conditions are true or **false** otherwise; call the **Expressions**.**Expression.AndAlso** static method to combine the expressions together, and store the result in the **dateCondition** object.



Note: The **Expressions.Expression.AndAlso** method combines the two discrete expressions that you just created, **"item.TestDate >= startDate"** and **"Item.TestDate <= endDate"** into a **BinaryExpression** object that represents the expression **"item.TestDate >= startDate && Item.TestDate <= endDate"**.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    // Combine the expressions with the && operator.
    dateCondition = Expressions.Expression.AndAlso(
        lowerDateCondition, upperDateCondition);
}
```

8. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 4: Dynamically build numeric expression trees

1. In the task list, locate the **TODO - Complete the BuildNumericExpressionBody method** task, and then double-click this task. This task is located in the **BuildNumericExpressionBody** method.

   The existing code in this method defines a **BinaryExpression** object named **booleanCondition**. This object will be used to return the expression tree that evaluates conditions based on **short** integer values. You will add code to this conditional statement to build an expression tree that is equivalent to the expression in the following code example, where *propertyName* represents the value of the *propertyName* parameter.

```
item.PropertyName >= lowerRange && item.PropertyName <= upperRange
```

2. Delete the **TODO - Complete the BuildNumericExpressionBody method** comment, and then add code to generate the first half of the expression by performing the following tasks:

   a. Create a new **MemberInfo** object named **testProperty**. Call the **GetProperty** method to generate code that retrieves the property that the *propertyName* parameter specifies from the *testResultType* type parameter.

   b. Create a **MemberExpression** object named **testMember** that represents access to the property that the **testProperty** object specifies; call the static **Expression.MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testProperty** object as parameters.

   c. Create a **ConstantExpression** object named **lowerValue** by invoking the **Expression.Constant** method. Pass the *lowerRange* parameter as a parameter to the method call.

   d. Create a **BinaryExpression** object named **lowerValueCondition**, which combines the **testMember** and **lowerValue** expression objects into a **GreaterThanOrEqual** binary expression.



Hint: Your code should build the first half of the target expression, which represents **"item.***PropertyName* **>= lowerRange"**, where *PropertyName* represents the value of the *propertyName* parameter. Your code should use similar syntax to that used to generate the expression in Task 3

Your code should resemble the following code example.

```
private Expressions.BinaryExpression BuildNumericExpressionBody(bool
rangeSpecified, short lowerRange, short upperRange, Type
testResultType, string propertyName, Expressions.ParameterExpression
itemBeingQueried)
{
    ...
    if (rangeSpecified)
    {
        // Generate the expression:
        //
        //    item.<Property> >= lowerRange
        //
        MemberInfo testProperty =
            testResultType.GetProperty(propertyName);

        Expressions.MemberExpression testMember =
            Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testProperty);

        Expressions.ConstantExpression lowerValue =
            Expressions.Expression.Constant(lowerRange);

        Expressions.BinaryExpression lowerValueCondition =
            Expressions.Expression.GreaterThanOrEqual(
                testMember, lowerValue);
        ...
    }
    ...
}
```

3. Add code to generate the second half of the target expression by performing the following tasks:

    a. Create a **ConstantExpression** object named **upperValue** by invoking the static **Expression**.**Constant** method. Pass the *upperRange* parameter as a parameter to the method call.

    b. Create a **BinaryExpression** object named **upperValueCondition**, which combines the **testMember** and **upperValue** expression objects into a **LessThanOrEqual** binary expression.



Hint: Your code should build the second half of the target expression, which represents "**item**.*PropertyName* **<= upperRange**", where *PropertyName* represents the value of the *propertyName* parameter. Your code should again use similar syntax to that used to generate the expression in Task 3.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
if (rangeSpecified)
{
    ...
    // Generate the expression:
    //
    //    item.<Property> <= upperRange
    //
    Expressions.ConstantExpression upperValue =
        Expressions.Expression.Constant(upperRange);

    Expressions.BinaryExpression upperValueCondition =
        Expressions.Expression.LessThanOrEqual(
            testMember, upperValue);
}
```

4. At the end of the method, add code to set the **booleanCondition** object to an expression that combines the **lowerValueCondition** and **upperValueCondition** expressions by using the static **Expressions.Expression.AndAlso** method.

   Your code should resemble the following code example.

```
if (rangeSpecified)
{
    ...

    // Combine the expressions with &&
    booleanCondition =
        Expressions.Expression.AndAlso(
            lowerValueCondition, upperValueCondition);
}
```

5. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 5: Combine the expression trees

1. In the task list, locate the **TODO - Complete the BuildLambdaExpressionBody method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionBody** method.

   This method takes four parameters that define the expression trees for each of the possible criteria that the user can enter. If any criteria are missing, the corresponding expression tree is null. The purpose of this method is to

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

combine these expression trees together into an overall expression tree that includes all of the criteria that the user enters.

The existing code in this method creates an **Expression** object named **body**. This object will contain the combined binary expressions that form the body of the LINQ query. If the user did not specify any criteria, the **body** object is assigned an expression tree that contains the Boolean constant **true**. This expression tree causes all items to be retrieved.

2. Delete the **TODO - Complete the BuildLambdaExpressionBody method** comment, and then add code to check whether the *dateCondition* parameter is null. If not, set the **body** object to the **dateCondition** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...
    Expressions.Expression body = null;
    if (dateCondition != null)
    {
        body = dateCondition;
    }
    ...
}
```

3. Add code to check whether the *temperatureCondition* parameter is null, and if not, perform the following tasks:

   a. If the **body** object is null, set the **body** object to the **temperatureCondition** expression tree.

   b. If the **body** object is not null, combine the expression tree in the body object with the expression tree in the *temperatureCondition* parameter by using the static **Expressions.Expression**.**AndAlso** method. Assign the result to the **body** object.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...
    // Add the temperatureCondition expression.
    if (temperatureCondition != null)
    {
        if (body == null)
        {
            body = temperatureCondition;
```

```
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                        body, temperatureCondition);
        }
    }
    ...
}
```

4. Repeat the logic in Step 3 and add the expression tree that the *appliedStressCondition* parameter defines to the **body** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...

    // Repeat the same logic for the remaining condition expressions.
    if (appliedStressCondition != null)
    {
        if (body == null)
        {
            body = appliedStressCondition;
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                        body, appliedStressCondition);
        }
    }
    ...
}
```

5. Repeat the logic in Step 3 and add the expression tree that the *deflectionCondition* parameter defines to the **body** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...

    // Repeat the same logic for the remaining condition expressions.
    ...

    if (deflectionCondition != null)
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    {
        if (body == null)
        {
            body = deflectionCondition;
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                        body, deflectionCondition);
        }
    }
    ...
}
```

6. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 6: Build a lambda expression for the OrderBy statement

1. In the task list, locate the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionForOrderBy** method.

   The purpose of this method is to construct an expression that specifies the order in which the data should be retrieved. The parameter to this method is a value from the **OrderByKey** enumeration. This enumeration is defined as part of the application and contains the following values: **ByDate**, **ByTemperature**, **ByAppliedStress**, **ByDeflection**, and **None**. The following code example shows the form of the lambda expression that this method generates.

```
item => item.Property
```

   In this example, Property references the property from the **TestResult** type that corresponds to the parameter that is passed into the method. If the user does not specify a sort key, this method returns a null value.

2. Delete the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** comment, and then add code to the method to create the **ParameterExpression** object that defines the parameter for the lambda expression by performing the following tasks:

MCT USE ONLY. STUDENT USE PROHIBITED



Note: You will need to create a **Type** reference to the **TestResult** object, and the lambda expression should refer to the object **item**.

a. Create a **Type** reference named **testResultType** by using the **typeOf** operator and passing a **TestResult** object as a parameter.

b. Create a **ParameterExpression** object named **itemBeingQueried** by using the static **Expressions.Expression.Parameter** method. Specify the **testResultType** object and a string that contains the text **"item"** as parameters to the method.

Your code should resemble the following code example.

```
private Expressions.Expression<Func<TestResult, ValueType>>
    BuildLambdaExpressionForOrderBy(OrderByKey orderByKey)
{
    ...

    if (orderByKey != OrderByKey.None)
    {
        // Create the expression that defines the parameter for the
        // lambda expression.
        // The type is TestResult, and the lambda expression refers to
        // it with the name "item".

        Type testResultType = typeof(TestResult);
        Expressions.ParameterExpression itemBeingQueried =
            Expressions.Expression.Parameter(testResultType, "item");
    ...

    }

    ...
}
```

3. In the **BuildLambdaExpressionForOrderBy** method, replace the **TODO - Create a MemberExpression and MemberInfo object** comment with code to perform the following tasks:

   a. Create a **MemberExpression** object named **sortKey**, and initialize this object to null.

   b. Create a **MemberInfo** object named **property**, and initialize this object to null.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
if (orderByKey != OrderByKey.None)
{
    ...

    // Create the expression that will define the sort key that
    // the lambda expression returns.

    // This expression will reference one of the properties in the
    // TestResult structure depending on the key that the user
    // specifies.

    Expressions.MemberExpression sortKey = null;
    MemberInfo property = null;

    ...
}
```

4. Replace the **TODO - Evaluate the orderByKey parameter to determine the property to sort by** comment with code to evaluate the *orderByKey* parameter. Use the **GetProperty** method of the testResultType variable to generate code that retrieves the corresponding property value from the item that is specified as the parameter to the lambda expression. Store the result in the property variable. The following table lists the name of each property to use, depending on the value of the *orderByKey* parameter.

orderByKey value	testResultType property to use
ByDate	**"TestDate"**
ByTemperature	**"Temperature"**
ByAppliedStress	**"AppliedStress"**
ByDeflection	**"Deflection"**



Note: Near the beginning of the **BuildLambdaExpressionForOrderBy** method, a conditional statement prevents the method from performing this code if the *orderByKey* parameter has the value **OrderByKey**.**None**; therefore, you do not need to check for this value.

Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
if (orderByKey != OrderByKey.None)
{
    ...

    MemberInfo property = null;

    switch(orderByKey)
    {
        case OrderByKey.ByDate:
            // If the user selected the date column, set the property
            // object to TestDate.
            property = testResultType.GetProperty("TestDate");
            break;
        case OrderByKey.ByTemperature:
            // If the user selected the temperature column, set the
            // property object to Temperature.
            property = testResultType.GetProperty("Temperature");
            break;
        case OrderByKey.ByAppliedStress:
            // If the user selected the applied stress column, set the
            // property object to AppliedStress.
            property = testResultType.GetProperty("AppliedStress");
            break;
        case OrderByKey.ByDeflection:
            // If the user selected the deflection column, set the
            // property object to Deflection.
            property = testResultType.GetProperty("Deflection");
            break;
    }

...

}
```

5. Replace the **TODO - Construct the expression that specifies the OrderBy field** comment with code that retrieves the value that the property variable specifies from the item that the itemBeingQueried variable specifies. To do this, call the static **Expressions.Expression.MakeMemberAccess** method, and pass the **itemBeingQueried** expression tree and the **property** object as parameters to this method.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Construct an expression that specifies the value in the field
    // that the property object references in the TestResult object.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    sortKey = Expressions.Expression.MakeMemberAccess(
        itemBeingQueried, property);
    ...
}
```

6. Replace the **TODO - Create a UnaryExpression object to convert the sortKey object to a ValueType** comment with code to create a new **UnaryExpression** object named **convert** by invoking the static **Expressions.Expression.Convert** method. Pass the **sortKey** object and the type of the **ValueType** type as parameters to the method call. This step is necessary because the possible sort keys are all value types, and they must be converted to **ValueType** objects for the ordering to function correctly.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Cast the sortKey object to a ValueType object (ValueType is the
    // ancestor of all value types, including DateTime and short).
    Expressions.UnaryExpression convert =
        Expressions.Expression.Convert(sortKey, typeof(ValueType));
    ...
}
```

7. Replace the **TODO - Create the OrderBy lambda expression** comment with code to combine the converted unary expression that contains the sort key and the itemBeingQueried variable into a lambda expression by using the static **Expression.Lambda** generic method. Specify the type **Func<TestResult, ValueType>** as the type parameter to the **Lambda** method; the resulting lambda expression takes a **TestResult** object as the parameter and returns a **ValueType** object.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Build the lambda expression by using the parameter and the
    // expression that contains the sort key.
    lambda = Expressions.Expression.Lambda
        <Func<TestResult, ValueType>>(convert, itemBeingQueried);
}
```

8. Build the project and correct any errors:

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

- On the **Build** menu, click **Build Solution**. Correct any errors.

Task 7: Examine the CreateQuery method

- In the task list, locate the **TODO - Examine the CreateQuery method** task, and then double-click this task. This task is located in the **CreateQuery** method.

  This method is the starting point for the lambda expression generation. The method accepts parameters that indicate which query criteria the lambda expression should include and the upper and lower ranges for each of these criteria.

  The method first calls the **BuildLambdaExpressionForQueryCriteria** method to construct a lambda expression that incorporates the query criteria. It then calls the **BuildLambdaExpressionForOrderBy** method to construct the lambda expression that defines the sort order for retrieving the data. Note that, at this point, it is possible that either of these expressions may still be null if the user either did not specify any criteria or did not specify a sort key.

  After the method creates the expression objects, it creates an **IEnumerable** generic collection named **query** that is based on the **TestResult** type, and it initializes the object with the data in the **stressData** parameter.

  If the lambda expression that specifies the query criteria is not null, the method then filters the data in the **IEnumerable** collection by invoking the **Where** LINQ extension method on the collection. The parameter to the **Where** method is the lambda expression that contains the query criteria. Note that the **Compile** method of an **Expression<TDelegate>** object converts the expression tree into a compiled lambda expression that the common language runtime (CLR) can execute.

  If the lambda expression that defines the sort order is not null, this method then applies this lambda expression to the **IEnumerable** collection by using the **OrderBy** LINQ extension method. As before, the **Compile** method converts the expression tree that defines the sort key into code that can be executed by using the CLR.

  If the user specifies that the query should return a limited number of rows, the **Take** LINQ extension method is applied to the **IEnumerable** collection with the limit that the user specifies.

  Finally, the **IEnumerable** collection is returned to the caller. Note that this method does not run the LINQ query. This action occurs in the

DisplayResults_Click method, when the code calls the **Count** method of the **IEnumerable** collection.

Task 8: Test the solution

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. In the Stress Data Analyzer window, click **Display** to display all results with no query criteria, sort key, or limit to the number of items that are returned. Note the time that it takes to execute the query.



Note: This test is different from the test that you performed at the end of the first exercise. In the original application, the LINQ query used a lambda expression that contained criteria for all properties, whereas this test does not use any criteria. Therefore, the operation should be faster.

3. Select the **Test Date** and **Temperature** check boxes, modify the search criteria to the values in the following table, and then click **Display** again.

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values, as in the test in Exercise 1. However, the time it takes to execute the query should again be less than the time that you recorded in Exercise 1.
5. Clear the **Test Date** and **Temperature** check boxes, and then select the **Limit?** check box. Set the limit value to 2,000, and then click **Display**.

   Note that when the number of rows is reduced, the time it takes to execute the query is substantially reduced.
6. In the **Order By** section, select **Temperature**, and then click **Display** again.

   Note that the expression takes substantially longer to execute when a sort key is included in the expression.
7. Close the Stress Data Analyzer window, and then return to Visual Studio.

MCT USE ONLY. STUDENT USE PROHIBITED

8. Close Visual Studio:
    - In Visual Studio, on the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Integrating Visual C# Code with Dynamic Languages and COM Components

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

Exercise 1: Integrating Code Written by Using a Dynamic Language into a Visual C# Application

Task 1: Examine the Python and Ruby code

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Using Notepad, open the Shuffler.py file in the E:\Labfiles\Lab 15\Python folder:
   a. Using Windows® Explorer, move to the **E:\Labfiles\Lab 15\Python** folder.
   b. Right-click **Shuffler.py**, and then click **Open**.
   c. In the **Windows** dialog box, click **Select a program from a list of installed programs**, and then click **OK**.
   d. In the **Open with** dialog box, click **Notepad**, and then click **OK**.

   In Notepad, examine the Python code.

   The Shuffler.py file contains a Python class called **Shuffler** that provides a method called **Shuffle**. The **Shuffle** method takes a parameter called *data* that contains a collection of items. The **Shuffle** method implements the Fisher-Yates-Durstenfeld algorithm to randomly shuffle the items in the **data** collection.

The Python class also exposes a function called **CreateShuffler** that creates a new instance of the **Shuffler** class. You will use this method from Microsoft Visual C#® to create a **Shuffler** object.

4. Close Notepad.
5. Using Notepad, open the Trapezoid.rb file in the E:\Labfiles\Lab 15\Ruby folder:
   a. Using Windows Explorer, move to the **E:\Labfiles\Lab 15\Ruby** folder.
   b. Right-click **Trapezoid.rb**, and then click **Open**.
   c. In the **Windows** dialog box, click **Select a program from a list of installed programs**, and then click **OK**.
   d. In the **Open with** dialog box, click **Notepad**, and then click **OK**.
6. In Notepad, examine the Ruby code.

   The Trapezoid.rb file contains a Ruby class called **Trapezoid** that models simple trapezoids. The constructor expects the angle of the lower-left vertex, the length of the base, the length of the top, and the height of the trapezoid. The lengths of the remaining sides and angles are calculated.



Note: The **Trapezoid** class models a subset of possible trapezoids. The length of the base must be greater than the length of the top, and the specified vertex must be an acute angle.

The lengths of the sides, the angles of each vertex, and the height are exposed as properties.

The **to_s** method returns a string representation of the trapezoid.



Note: The **to_s** method is the Ruby equivalent of the **ToString** method in the Microsoft .NET Framework. The Ruby binder in the dynamic language runtime (DLR) automatically translates a call to the **ToString** method on a Ruby object to a call to the **to_s** method.

The **area** method calculates the area of the trapezoid.

The Ruby file also provides a function called **CreateTrapezoid** that creates a new instance of the **Trapezoid** class.

7. Close Notepad.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 2: Open the starter project

- Open the DynamicLanguageInterop solution in the E:\Labfiles\Lab 15\Starter \DynamicLanguageInterop folder:

  a. On the **File** menu, point to **Open**, and then click Project/Solution.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 15\Starter \DynamicLanguageInterop** folder.

  c. Click **DynamicLanguageInterop.sln**, and then click **Open**.

Task 3: Create a Python object and call Python methods

1. Examine the InteropTestWindow.xaml file:

   - In Solution Explorer, expand the **DynamicLanguageInterop** project, and then double-click the **InteropTestWindow.xaml** file.

   This window contains two tabs, labeled **Python Test** and **Ruby Test**.

   The **Python Test** tab enables you to type values into the **Data** box and specify whether this is text or numeric data. When you click **Shuffle**, the data will be packaged up into an array and passed to the **Shuffle** method of a Python **Shuffler** object. The shuffled data will be displayed in the **Shuffled Data** box.

   The functionality to create the Python object and call the **Shuffle** method has not yet been implemented; you will do this in this task.

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronPython runtime.

Assembly	Path
IronPython	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.dll
IronPython.Modules	C:\Program Files\IronPython 2.6 for .NET 4.0\IronPython.Modules.dll
Microsoft.Dynamic	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Dynamic.dll
Microsoft.Scripting	C:\Program Files\IronPython 2.6 for .NET 4.0\Microsoft.Scripting.dll

MCT USE ONLY. STUDENT USE PROHIBITED

a. In Solution Explorer, right-click **References**, and then click **Add Reference**.

b. In the **Add Reference** dialog box, click **Browse**.

c. Move to the **C:\Program Files\IronPython 2.6 for .NET 4.0** folder.

d. Select **IronPython.dll**, **IronPython.Modules.dll**, **Microsoft.Dynamic.dll**, and **Microsoft.Scripting.dll**, and then click **OK**.

3. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the task list is displaying **User Tasks**, in the drop-down list box click **Comments**.

4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task. This task is located near the top of the InteropTestWindow.xaml.cs file. This is the code behind the InteropTestWindow window.

5. After the comment, add **using** statements to bring the **IronPython.Hosting** and **Microsoft.Scripting.Hosting** namespaces into scope.

   Your code should resemble the following code example.

```
// TODO: Add Namespaces containing IronPython and IronRuby runtime
support and interop types
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
...
```

6. In the **InteropTestWindow** class, examine the string constants near the start of the class. In particular, note the **pythonLibPath** and **pythonCode** strings.

   The **pythonLibPath** constant specifies the folder where the Python libraries are installed. The **Shuffler** class makes use of a Python library called **random** that is located in this folder.

   The **pythonCode** constant specifies the name and location of the Python script that contains the **Shuffler** class.

7. In the task list, locate the **TODO: Create an instance of the Python runtime, and add a reference to the folder holding the "random" module** task, and then double-click this task. This task is located in the **ShuffleData** method.

   The **shuffle_Click** method calls the **ShuffleData** method when the user clicks the **Shuffle Data** button. The **shuffle_Click** method gathers the user input

from the form and parses it into an array of objects. It then passes this array to the **ShuffleData** method. The purpose of the **ShuffleData** method is to create a Python **Shuffler** Python object and then call the **Shuffle** method by using the array as a parameter. When the **ShuffleData** method finishes, the **shuffle_Click** method displays the shuffled data in the Windows Presentation Foundation (WPF) window.

8. After the TODO comment, add code that performs the following tasks:

   a. Create a **ScriptEngine** object called **pythonEngine** by using the static **CreateEngine** method of the **Python** class.

   b. Obtain a reference to the search paths that the Python runtime uses; call the **GetSearchPaths** method of the **pythonEngine** object and store the result in an **ICollection<string>** collection object called **paths**.

   c. Add the path that is specified in the **pythonLibPath** string to the **paths** collection.

   d. Set the search paths that the **pythonEngine** object uses to the **paths** collection; use the **SetSearchPaths** method.

   Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    // TODO: Create an instance of the Python runtime, and add a
reference to the folder holding the "random" module
    // The Python script references this module
    ScriptEngine pythonEngine = Python.CreateEngine();
    ICollection<string> paths = pythonEngine.GetSearchPaths();
    paths.Add(pythonLibPath);
    pythonEngine.SetSearchPaths(paths);
    ...
}
```

9. After the comment **TODO: Run the script and create an instance of the Shuffler class by using the CreateShuffler method in the script**, add code that performs the following tasks:

   a. Create a **dynamic** object called **pythonScript**. Initialize this object with the value that is returned by calling the **ExecuteFile** method of the **pythonEngine** object. Specify the **pythonCode** constant as the parameter to this method.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

This statement causes the Python runtime to load the Shuffler.py script. The **pythonScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.

b. Create another **dynamic** object called **pythonShuffler**. Call the **CreateShuffler** method of the **pythonScript** object and store the result in the **pythonShuffler** object.

This statement invokes the **CreateShuffler** function in the Python script. This function creates an instance of the **Shuffler** class and returns it. The **pythonShuffler** object then holds a reference to this object.



Note: The pythonScript variable is a **dynamic** object, so Microsoft IntelliSense® does not display the **CreateShuffler** method (or any other methods or properties).

Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    ...
    // TODO: Run the script and create an instance of the Shuffler
class by using the CreateShuffler method in the script
    dynamic pythonScript = pythonEngine.ExecuteFile(pythonCode);
    dynamic pythonShuffler = pythonScript.CreateShuffler();
    ...
}
```

10. After the comment **TODO: Shuffle the data**, add code that calls the **Shuffle** method of the **pythonShuffler** object. Pass the **data** array as the parameter to the **Shuffle** method.

This statement runs the **Shuffle** method in the Python object. The DLR marshals the **data** array into a Python collection and then invokes the **Shuffle** method. When the method completes, the DLR unmarshals the shuffled collection back into the **data** array.

Your code should resemble the following code example.

```
private void ShuffleData(object[] data)
{
    ...
    // TODO: Shuffle the data.
    pythonShuffler.Shuffle(data);
}
```

Download from www.eBookTM.com

11. Build the application and correct any errors:
    - On the **Build** menu, click **Build Solution**.

Task 4: Test the Python code

1. Run the application:
    - On the **Debug** menu, click **Start Without Debugging**.
2. In the Dynamic Language Interop Tests window, on the **Python Test** tab, in the **Data** box, type some random words that are separated by spaces.
3. Click the **Text** option button, and then click **Shuffle**. Verify that the shuffled version of the data appears in the **Shuffled Data** box.
4. Click **Shuffle** again. The data should be shuffled again and appear in a different sequence.
5. Replace the text in the **Data** box with integer values, click **Integer**, and then click **Shuffle**. Verify that the numeric data is shuffled.
6. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

Task 5: Create a Ruby object and call Ruby methods

1. Examine the **Ruby Test** tab in the InteropTestWindow.xaml file.

   The **Ruby Test** tab enables you to specify the dimensions of a trapezoid (the angle of the first vertex, the length of the base, the length of the top, and the height) by using a series of slider controls. When you click the **Visualize** button, the application will create an instance of the Ruby **Trapezoid** class and display a graphical representation in the canvas in the lower part of the window. The dimensions and area of the trapezoid will be displayed in the text block that is to the right.

   The functionality to create the Ruby object and calculate its area and dimensions has not yet been implemented; you will do this in this task.

   a. In Solution Explorer, double-click the **InteropTestWindow.xaml** file.

   b. Click the **Ruby Test** tab.

2. Add references to the assemblies listed in the following table. The DLR uses these assemblies to provide access to the IronRuby runtime.

MCT USE ONLY. STUDENT USE PROHIBITED

Assembly	Path
IronRuby	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.dll
IronRuby.Libraries	C:\Program Files\IronRuby 1.0v4\bin\IronRuby.Libraries.dll

   a. In Solution Explorer, right-click **References**, and then click **Add Reference**.

   b. In the **Add Reference** dialog box, click **Browse**.

   c. Move to the **C:\Program Files\IronRuby 1.0v4\bin** folder.

   d. Select **IronRuby.dll** and **IronRuby.Libraries.dll**, and then click **OK**.

3. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

4. In the task list, locate the **TODO: Add Namespaces containing IronPython and IronRuby runtime support and interop types** task, and then double-click this task.

5. Add a **using** statement to bring the **IronRuby** namespace into scope.

   Your code should resemble the following code example.

```
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
using IronRuby;
...
```

6. In the **InteropTestWindow** class, examine the **rubyCode** string constant near the start of the class.

   The **rubyCode** constant specifies the name and location of the Ruby script that contains the **Trapezoid** class.

7. In the task list, locate the **TODO: Retrieve the values specified by the user. These values are used to create the trapezoid** task, and then double-click this task. This task is located in the **visualize_Click** method. This method is called when the user clicks the **Visualize** button, after the user has specified the data for the trapezoid.

MCT USE ONLY. STUDENT USE PROHIBITED

8. After the TODO comment, add code that performs the following tasks:

   a. Create an integer variable called vertexAInDegrees. Initialize this variable with the value of the **vertexA** slider control.



Hint: Use the **Value** property of a slider control to read the value. This value is returned as a **Double** value, so use a cast to convert it to an integer. This cast is safe because the slider controls are configured to return integer values in a small range, so no data will be lost.

   b. Create an integer variable called lengthSideAB. Initialize this variable with the value of the **sideAB** slider control.

   c. Create an integer variable called lengthSideCD. Initialize this variable with the value of the **sideCD** slider control.

   d. Create an integer variable called heightOfTrapezoid. Initialize this variable with the value of the **height** slider control.

   Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        // TODO: Retrieve the values specified by the user. These
values are used to create the trapezoid.
        int vertexAInDegrees = (int)vertexA.Value;
        int lengthSideAB = (int)sideAB.Value;
        int lengthSideCD = (int)sideCD.Value;
        int heightOfTrapezoid = (int)height.Value;
        ...
    }
    ...
}
```

9. After the comment **TODO: Call the CreateTrapezoid method and build a trapezoid object**, add a statement that creates a dynamic variable called trapezoid and initializes it with the value that the **CreateTrapezoid** method returns. Pass the variables vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid as arguments to the **CreateTrapezoid** method.

   You will implement the **CreateTrapezoid** method in a later step. This method will create an instance of the Ruby **Trapezoid** class by using the specified data and return it.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Call the CreateTrapezoid method and build a trapezoid
object.
       dynamic trapezoid = CreateTrapezoid(vertexAInDegrees,
           lengthSideAB, lengthSideCD, heightOfTrapezoid);
        ...
    }
    ...
}
```

10. After the comment **TODO: Display the lengths of each side, the internal angles, and the area of the trapezoid**, add a statement that calls the **DisplayStatistics** method. Pass the **trapezoid** object and the **trapezoidStatistics** text block as parameters to this method.

    You will implement the **DisplayStatistics** method in a later step. This method will call the **to_s** and **area** methods of the Ruby **Trapezoid** class and display the results in the **trapezoidStatistics** text block on the right of the **Ruby Test** tab in the WPF window.

    Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Display the lengths of each side, the internal
angles, and the area of the trapezoid.
        DisplayStatistics(trapezoid, this.trapezoidStatistics);
        ...
    }
    ...
}
```

11. After the comment **TODO: Display a graphical representation of the trapezoid**, add a statement that calls the **RenderTrapezoid** method. Pass the **trapezoid** object and the **trapezoidCanvas** canvas control as parameters to this method.

MCT USE ONLY. STUDENT USE PROHIBITED

The **RenderTrapezoid** method is already complete. This method queries the properties of the Ruby **Trapezoid** object and uses them to draw a representation of the trapezoid on the canvas in the lower part of the window.

Your code should resemble the following code example.

```
private void visualize_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...
        // TODO: Display a graphical representation of the trapezoid.
        RenderTrapezoid(trapezoid, this.trapezoidCanvas);
    }
    ...
}
```

12. In the task list, locate the **TODO: Create an instance of the Ruby runtime** task, and then double-click this task. This task is located in the **CreateTrapezoid** method.

13. At the start of this method, remove the statement that throws the **NotImplementedException** exception. After the comment, add a statement that creates a **ScriptRuntime** object called **rubyRuntime**. Initialize the rubyRuntime variable with the value that the static **CreateRuntime** method of the **Ruby** class returns.

    Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    // TODO: Create an instance of the Ruby runtime.
    ScriptRuntime rubyRuntime = Ruby.CreateRuntime();

    ...
}
```

14. After the comment **TODO: Run the Ruby script that defines the Trapezoid class**, add a statement that creates a **dynamic** object called **rubyScript**. Initialize the rubyScript variable with the value that the **UseFile** method of the **rubyRuntime** object returns. Pass the **rubyCode** constant as the parameter to the **UseFile** method.

    This statement causes the Ruby runtime to load the Trapezoid.rb script. The **rubyScript** object contains a reference to this script that you can use to invoke functions and access classes that are defined in this script.

MCT USE ONLY. STUDENT USE PROHIBITED

Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Run the Ruby script that defines the Trapezoid class.
    dynamic rubyScript = rubyRuntime.UseFile(rubyCode);
    ...
}
```

15. After the comment **TODO: Call the CreateTrapezoid method in the Ruby script to create a trapezoid object**, add a statement that creates a **dynamic** object called **rubyTrapezoid**. Initialize the rubyTrapezoid variable with the value that the **CreateTrapezoid** method of the **rubyScript** object returns. Pass the vertexAInDegrees, lengthSideAB, lengthSideCD, and heightOfTrapezoid variables as parameters to the **CreateTrapezoid** method.

    This statement invokes the **CreateTrapezoid** function in the Ruby script. The DLR marshals the arguments that are specified and passes them as parameters to the **CreateTrapezoid** function. This function creates an instance of the **Trapezoid** class and returns it. The **rubyTrapezoid** object then holds a reference to this object.



Note: The rubyScript variable is a **dynamic** object, so IntelliSense does not display the **CreateTrapezoid** method.

Your code should resemble the following code example.

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Call the CreateTrapezoid method in the Ruby script to
create a trapezoid object.
    dynamic rubyTrapezoid = rubyScript.CreateTrapezoid(
        vertexAInDegrees, lengthSideAB, lengthSideCD,
        heightOfTrapezoid);
    ...
}
```

16. After the comment **TODO: Return the trapezoid object**, add a statement that returns the value in the rubyTrapezoid variable.

    Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
private dynamic CreateTrapezoid(int vertexAInDegrees, int
    lengthSideAB, int lengthSideCD, int heightOfTrapezoid)
{
    ...
    // TODO: Return the trapezoid object.

    return rubyTrapezoid;

}
```

17. In the task list, locate the **TODO: Use a StringBuilder object to construct a string holding the details of the trapezoid** task, and then double-click this task. This task is located in the **DisplayStatistics** method.

18. After the comment, add a statement that creates a new **StringBuilder** object called **builder**.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    // TODO: Use a StringBuilder object to construct a string holding
the details of the trapezoid.

    StringBuilder builder = new StringBuilder();

    ...
}
```

19. After the comment **TODO: Call the to_s method of the trapezoid object to return the details of the trapezoid as a string**, add a statement that calls the **ToString** method of the trapezoid variable and appends the result to the end of the **builder** object.

    The DLR automatically converts the **ToString** method call into a call to the **to_s** method in the Ruby object. The **to_s** method constructs a Ruby string, which is unmarshaled into a .NET Framework string.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...

    // TODO: Call the to_s method of the trapezoid object to return
the details of the trapezoid as a string.
```

MCT USE ONLY. STUDENT USE PROHIBITED

```
    // Note: The ToString method invokes to_s.
    builder.Append(trapezoid.ToString());
    ...
}
```

20. After the comment **TODO: Calculate the area of the trapezoid object by using the area method of the trapezoid class**, add code that calls the **area** method of the trapezoid variable, converts the result into a string, and appends this string to the end of the **builder** object.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...
    // TODO: Calculate the area of the trapezoid object by using the
area method of the trapezoid class
    // and append it to the string holding the details of the
trapezoid
    builder.Append(String.Format("\nArea:\t\t{0}",
        trapezoid.area().ToString()));
    ...
}
```

21. After the comment **TODO: Display the details of the trapezoid in the TextBlock control**, add a statement that sets the **Text** property of the **trapezoidStatistics** control to the string that is constructed by the **builder** object.

    Your code should resemble the following code example.

```
private void displayStatistics(dynamic trapezoid,
    TextBlock trapezoidStatistics)
{
    ...
    // TODO: Display the details of the trapezoid in the TextBlock
control
    trapezoidStatistics.Text = builder.ToString();
}
```

22. Build the application and correct any errors:

    - On the **Build** menu, click **Build Solution**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

Task 6: Test the Ruby code

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. In the Dynamic Language Interop Tests window, click the **Ruby Test** tab.
3. Set the **Vertex A** slider to **75**, set the **Length of Base** slider to **200**, set the **Length of Top** slider to **100**, set the **Height** slider to **150**, and then click **Visualize**.

   Verify that a representation of the trapezoid is displayed in the canvas in the lower half of the window and the statistics for the trapezoid appear in the text block that is to the right. The area of the trapezoid should be 22,500.
4. Experiment with different values for the slider controls, and then click **Visualize**. If you specify values that are outside the range for the set of trapezoids that the **Trapezoid** class can model, a message box should be displayed to indicate the problem. This error message is raised by the constructor in the **Trapezoid** class. The DLR catches the error and converts it into a .NET Framework **Exception** object. The **visualize_Click** method caches this exception and displays the error in a message box.
5. Close the Dynamic Language Interop Tests window, and then return to Visual Studio.

Exercise 2: Using a COM Component from a Visual C# Application

Task 1: Examine the data files

1. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder, and then verify that this folder contains the following three text files:
   - 298K.txt
   - 318K.txt
   - 338K.txt
2. Using Notepad, open the 298K.txt file:
   - In Windows Explorer, right-click the **298K.txt** file, and then click **Open**.

   This file contains results from the deflection tests for steel girders that were subjected to various pressures at a temperature of 298 Kelvin. The number on

a line by itself at the top of the file is the temperature at which the tests were performed (298). The remaining lines contain pairs of numbers; the numbers in each pair are separated by a comma. These numbers are the pressure applied, which is measured in kiloNewtons (kN), and the deflection of the girder, which is measured in millimeters.

3. Close Notepad.
4. Using Notepad, open the 318K.txt file.

   This file is in the same format as the 298K.txt file. It contains the results of deflection tests that were performed at a temperature of 318 Kelvin. Notice that the final few lines do not contain any deflection data because the test was halted at a force of 1,000 kN.

5. Close Notepad.
6. Using Notepad, open the 338K.txt file.

   This file is similar to the other two. It contains the results of deflection tests that were performed at a temperature of 338 Kelvin. The test was halted at a force of 800 kN.

7. Close Notepad.

Task 2: Open the starter project and examine the StressData type

1. Using Visual Studio, open the GenerateGraph solution in the E:\Labfiles\Lab 15\Starter\GenerateGraph folder:
   a. On the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 15\Starter\GenerateGraph** folder.
   c. Click **GenerateGraph.sln**, and then click **Open**.
2. Open the StressData.cs file:
   - In Solution Explorer, double-click **StressData.cs**.

   The **StressData** type acts as a container for the stress data for a given temperature. It contains the following public properties:

   - **Temperature**. This is a short value that records the temperature of the test.
   - **Data**. This is a **Dictionary** collection that holds the data. The stress value is used as the key into the dictionary, and the item data is the deflection.

MCT USE ONLY. STUDENT USE PROHIBITED

The **StressData** class also overrides the **ToString** method, which returns a formatted string that lists the stress test data that is stored in the object.

Task 3: Examine the GraphWindow test harness

1. Open the GraphWindow.xaml file:

   - In Solution Explorer, double-click **GraphWindow.xaml**.

   This window provides a simple test harness for reading the data from the data files and invoking Microsoft Office Excel® to generate a graph by using this data.

   When users click **Get Data**, they are prompted for the data file to load. The file is read into a new **StressData** object, and the contents of the file are displayed in the **TreeView** control that occupies the main part of the window. A user can click **Get Data** multiple times and load multiple files; they will all be read in and displayed. The **StressData** objects are stored in a **List** collection that is held in a private field in the **GraphWindow** class and is called **graphData**. This code has already been written for you.

   When a user clicks **Graph**, the data in the **graphData** collection will be used to generate an Office Excel graph. The information in each **StressData** object will be transferred to an Office Excel worksheet, and a line graph will then be generated to show the stress data for each temperature. A user can quickly examine this graph and spot any trends in the failure of girders.

2. Open the GraphWindow.xaml.cs code file:

   - In Solution Explorer, expand **GraphWindow.xaml**, and then double-click **GraphWindow.xaml.cs**.

3. Locate the **populateFromFile** method.

   This method uses a **StreamReader** object to read and parse the stress data from a file that is specified as a parameter, and it populates a **StressData** object that is also specified as a parameter.

   This method is complete.

4. Locate the **displayData** method.

   This method takes a populated **StressData** object and displays the items in this object in the **TreeView** control in the window.

   This method is also complete.

5. Locate the **getData_Click** method.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED

This method runs when the user clicks the **Get Data** button. It uses an **OpenFileDialog** object to prompt the user for the name of a data file and then passes the file name together with a new **StressData** object to the **populateFromFile** method. It then adds the populated **StressData** object to the **graphData** collection before it calls the **displayData** method to add the data to the **TreeView** control in the window.

This method is complete.

6. Locate the **generateGraph_Click** method.

   This method runs when the user clicks the **Generate** button. It prompts the user for the name of an Office Excel workbook to create. It will then create this new workbook and copy the data in the **graphData** collection into a worksheet in this workbook before it generates a graph.

   This method is not complete. You will add the missing functionality and complete the **transferDataToExcelSheet** and **generateExcelChart** helper methods that this code will use.

Task 4: Copy data to an Office Excel worksheet

1. Add a reference to the Microsoft Excel 12.0 Object Library to the application. This is the COM object library that implements the Office Excel object model:
   a. In Solution Explorer, right-click **References**, and then click **Add Reference**.
   b. In the **Add Reference** dialog box, click the **COM** tab.
   c. In the list of COM components, scroll down and click **Microsoft Excel 12.0 Object Library**, and then click **OK**.
2. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
3. In the task list, locate the **TODO: Add the Microsoft.Office.Interop.Excel namespace** task, and then double-click this task. This task is located near the top of the GraphWindow.xaml.cs file.
4. Bring the **Microsoft.Office.Interop.Excel** namespace into scope, and give it an alias of **Excel**. This alias helps you to distinguish items in this namespace and

avoid name clashes without having to specify the full namespace in ambiguous object references.

Your code should resemble the following code example.

```
// TODO: Add the Microsoft.Office.Interop.Excel namespace.
using Excel = Microsoft.Office.Interop.Excel;
```

5. Locate the **transferDataToExcelSheet** method.

   The **generateGraph_Click** method will call this method. It takes three parameters:

   - An **Excel.Worksheet** object called **excelWS**. This object is a reference to the Office Excel worksheet that you will copy the data to.
   - An **Excel.Range** object called **dataRange**. This is an output parameter. You will use this object to indicate the area of the worksheet that contains the data after it has been copied.
   - A **List<StressData>** object called **excelData**. This is a collection of **StressData** objects that contain the data that you will copy to the Office Excel worksheet.

   This method returns **true** if it successfully copies the data to the Office Excel worksheet and **false** if an exception occurs.

6. In the **transferDataToExcelSheet** method, after the comment **TODO: Copy the data for the applied stresses to the first column in the worksheet**, add code that performs the following tasks:

   a. Declare an integer variable called rowNum and initialize it to **1**.

   b. Declare an integer variable called colNum and initialize it to **1**.

   c. Set the value of the cell at location rowNum, colNum in the **excelWS** worksheet object to the text "**Applied Stress**".



Hint: You can use the **Cells** property to read and write a cell in an **Excel** worksheet object. This property acts like a two-dimensional array.

   d. Use a **foreach** loop to iterate through the keys in the first **StressData** object in the **excelData** collection.

MCT USE ONLY. STUDENT USE PROHIBITED



Hint: Remember that the **StressData** object contains a **Dictionary** property called **Data**, and the key values in this dictionary are the applied stresses for the test (100, 200, 300, up to 1,500 kN). You can use the **Keys** property of a **Dictionary** object to obtain a collection of keys that you can iterate through.

e. In the body of the **foreach** loop, increment the rowNum variable, and store the value of each key found in the cell at location rowNum, colNum in the **excelWS** worksheet object.

Your code should resemble the following code example.

```
private bool transferDataToExcelSheet(Excel.Worksheet excelWS,
    out Excel.Range dataRange, List<StressData> excelData)
{
    try
    {
        // TODO: Copy the data for the applied stresses to the first
column in the worksheet.
        // This should be a list of values: 100, 200, 300, ..., 1500
        // Each set of data in the list in the graphData object uses
the same set of stresses.
        int rowNum = 1;
        int colNum = 1;
        excelWS.Cells[rowNum, colNum] = "Applied Stress";
        foreach (short appliedStress in excelData[0].Data.Keys)
        {
            rowNum++;
            excelWS.Cells[rowNum, colNum] = appliedStress;
        }
        ...
    }
}
```

7. Locate the comment **TODO: Give each column a header that specifies the temperature**.

   This comment is located in a **foreach** loop that iterates over each item in the **excelData** collection. These items are **StressData** objects, and each **StressData** object contains the data for the tests for a given temperature. When complete, the code in this **foreach** loop will copy the data for each **StressData** object to a new column in the **excelWS** worksheet object, and each column will have a header that specifies the temperature.

8. After the comment, add code that performs the following tasks:

   a. Increment the colNum variable so that it refers to the next column in the worksheet.

MCT USE ONLY. STUDENT USE PROHIBITED

b. Set the rowNum variable to **1**.

c. Retrieve the temperature from the **deflectionData StressData** object, format it as a string with the letter "K" appended to the end (for Kelvin), and store this string in the cell at location rowNum, colNum in the **excelWS** worksheet object.

Your code should resemble the following code example.

```
foreach (StressData deflectionData in excelData)
{
    // TODO: Give each column a header that specifies the temperature.
    colNum++;
    rowNum = 1;
    excelWS.Cells[rowNum, colNum] = String.Format("{0}K",
        deflectionData.Temperature);
    ...
}
```

9. Locate the comment **TODO: Only copy the deflection value if it is not null**.

   This comment is located in a nested **foreach** loop that iterates over each value in a **StressData** object. Remember that not all stresses have a deflection value. Where this occurs, the data in the **StressData** object is null. The **if** statement detects whether the current deflection value is null.

10. After the comment, in the body of the **if** statement, add code that performs the following tasks:

    a. Increment the rowNum variable so that it refers to the next row in the worksheet.

    b. Copy the value of the deflection variable (that contains the deflection data) into the cell at location rowNum, colNum in the **excelWS** worksheet object.

    Your code should resemble the following code example.

```
// Copy the deflection data to this column in the worksheet
foreach (short? deflection in deflectionData.Data.Values)
{
    // TODO: Only copy the deflection value if it is not null
    if (deflection != null)
    {
        rowNum++;
        excelWS.Cells[rowNum, colNum] = deflection;
    }
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

11. Locate the comment **TODO: Specify the range of cells in the spreadsheet containing the data in the dataRange variable**.

    This comment is located after all of the **foreach** loops have completed and all of the data has been copied to the worksheet.

12. After the comment, add a statement that populates the dataRange variable with information about the set of cells that have been filled.



Hint: You can determine the boundaries of the filled area of an Office Excel worksheet by querying the **UsedRange** property. This property returns an **Excel.Range** object.

Your code should resemble the following code example.

```
...
// TODO: Specify the range of cells in the spreadsheet containing the
data in the dataRange variable.
dataRange = excelWS.UsedRange;
...
```

13. Build the solution and correct any errors:
    - On the **Build** menu, click **Build Solution**.

Task 5: Generate an Office Excel graph

1. Locate the **generateExcelChart** method.

   The **generateGraph_Click** method will call this method after the data has been copied to the Office Excel worksheet. It takes three parameters:

   - A **string** object called **fileName**. When the graph has been created, the method will save the Office Excel workbook to a file by using this file name.
   - An **Excel.Workbook** object called **excelWB**. This is a reference to the Office Excel workbook containing the Office Excel worksheet that contains the data to use for the graph.
   - An **Excel.Range** object called **dataRange**. This range specifies the location in the Office Excel worksheet that contains the data to use for the graph.

2. In the **generateExcelChart** method, after the comment **TODO: Generate a line graph based on the data in the dataRange range**, add code that performs the following tasks:

MCT USE ONLY. STUDENT USE PROHIBITED

a. Add a new chart object to the Office Excel workbook, and store a reference to this chart object in an Excel.Chart variable called excelChart.



Hint: You can create a new chart by using the **Add** method of the **Charts** property of an Office Excel workbook object. The **Add** method takes no parameters and returns a reference to the chart object.

b. Call the **ChartWizard** method of the chart object to generate the chart. The following table lists the parameters that you should specify.

Parameter name	Value
Title	**"Applied Stress (kN) versus Deflection (mm)"**
Source	**dataRange**
Gallery	**Excel.XlChartType.xlLine**
PlotBy	**Excel.XlRowCol.xlColumns**
CategoryLabels	**1**
SeriesLabels	**1**
ValueTitle	**"Deflection"**
CategoryTitle	**"Applied Stress"**

Your code should resemble the following code example.

```
private static void generateExcelChart(string fileName, Excel.Workbook
    excelWB, Excel.Range dataRange)
{
    // TODO: Generate a line graph based on the data in the dataRange
range.
    Excel.Chart excelChart = excelWB.Charts.Add();
    excelChart.ChartWizard(
        Title: "Applied Stress (kN) versus Deflection (mm)",
        Source: dataRange, Gallery: Excel.XlChartType.xlLine,
        PlotBy: Excel.XlRowCol.xlColumns, CategoryLabels: 1,
        SeriesLabels: 1, ValueTitle: "Deflection",
        CategoryTitle: "Applied Stress");
        ...
}
```

MCT USE ONLY. STUDENT USE PROHIBITED

3. After the comment **TODO: Save the Excel workbook**, add a statement that saves the Office Excel workbook by using the value in the *fileName* parameter.



Hint: Use the **SaveAs** method of the Office Excel Workbook object to save a workbook. This method takes a parameter called *Filename* that specifies the name of the file to use.

Your code should resemble the following code example.

```
private static void generateExcelChart(string fileName, Excel.Workbook
    excelWB, Excel.Range dataRange)
{
    ...
    // TODO: Save the Excel workbook
    excelWB.SaveAs(Filename: fileName);
}
```

4. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 6: Complete the test harness

1. Return to the **generateGraph_Click** method.
2. After the comment **TODO: If the user specifies a valid file name, start Excel and create a new workbook and worksheet to hold the data**, add code to perform the following tasks:
   a. Create a new **Excel.Application** object called **excelApp**.
   b. Make the application visible on the user's desktop by setting the **Visible** property of the **excelApp** object to **true**. (By default, Office Excel will run in the background.)
   c. Set the **AlertBeforeOverwriting** property of the **excelApp** object to **false**. This ensures that the **SaveAs** method always saves the workbook.
   d. Set the **DisplayAlerts** property of the **excelApp** object to **false**.
   e. Create a new workbook, and store a reference to this workbook in an **Excel.Workbook** variable called excelWB.

MCT USE ONLY. STUDENT USE PROHIBITED



Hint: You can create a new workbook by using the **Add** method of the **Workbooks** property of an **Excel.Application** object. This method takes no parameters and returns a reference to the new workbook.

f. Create a variable called excelWS of type **Excel.Worksheet** and set it as the active worksheet in the new workbook.



Hint: You can obtain a reference to the active worksheet in an Office Excel workbook by using the **ActiveSheet** property.

You code should resemble the following code example.

```
if (saveDialog.ShowDialog().Value)
{
    // TODO: If the user specifies a valid file name, start Excel
    // and create a new workbook and worksheet to hold the data

    excelApp = new Excel.Application();
    excelApp.Visible = true;
    excelApp.AlertBeforeOverwriting = false;
    excelApp.DisplayAlerts = false;
    Excel.Workbook excelWB = excelApp.Workbooks.Add();
    Excel.Worksheet excelWS = excelWB.ActiveSheet;

    ...
}
```

3. After the comment **TODO: Copy the data from the graphData variable to the new worksheet and generate a graph**, add code to perform the following tasks:
   a. Create an **Excel.Range** object called **dataRange** and initialize it to null.
   b. Call the **transferDataToExcelSheet** method, and pass the **excelWS** object the **dataRange** object, and the graphData variable as parameters. Note that the **dataRange** object should be an output parameter.
   c. If the value that the **transferDataToExcelSheet** method returns is **true**, call the **generateExcelChart** method. Pass the **FileName** property of the **SaveDialog** object, the **excelWB** object, and the **dataRange** object as parameters.

   Your code should resemble the following code example.

MCT USE ONLY. STUDENT USE PROHIBITED

```
if (saveDialog.ShowDialog().Value)
{
    ...
    // TODO: Copy the data from the graphData variable to the new
Worksheet and generate a graph
    // The dataRange variable specifies the cells on the worksheet
that hold the data
    Excel.Range dataRange = null;
    if (transferDataToExcelSheet(excelWS, out dataRange,
                                 this.graphData))
    {
        generateExcelChart(saveDialog.FileName, excelWB, dataRange);
    }
}
```

4. At the end of the **generateGraph_Click** method, in the **finally** block, after the comment **TODO: Close Excel and release any resources**, add code to check whether the excelApp variable is null; if it is not, close the Office Excel application.



Hint: Use the **Quit** method of an **Excel.Application** object to close Office Excel.

Your code should resemble the following code example.

```
finally
{
    // TODO: Close Excel and release any resources
    if (excelApp != null)
    {
        excelApp.Quit();
    }
}
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 7: Test the application

1. Start the application in Debug mode:
   - On the **Debug** menu, click **Start Debugging**.
2. In the Graphing Data window, click **Get Data**.

MCT USE ONLY. STUDENT USE PROHIBITED

3. In the **Graph Data** dialog box, click the **298K.txt** file, and then click **Open**.
4. In the Graphing Data window, in the tree view, expand the **Temperature: 298K** node. Verify that the data has been correctly loaded.
5. Repeat steps 2, 3, and 4 and load the data in the 318K.txt and 338K.txt files. Verify that the tree view lists the data from all three files.



Note: The **displayData** method displays the value –1 for any missing deflection data.

6. Click **Graph**.
7. In the **Graph Data** dialog box, accept the default file name, StressData.xlsx, for the name of the Office Excel workbook to be generated, and then click **Save**.

   You will see Office Excel start to run and your data copied across to a new worksheet. You will also briefly see the graph that is generated before the workbook is saved and Office Excel closes.
8. Using Windows Explorer, move to the **E:\Labfiles\Lab 15** folder. Verify that this folder contains the Office Excel workbook StressData.xlsx.
9. Double-click the **StressData.xlsx** file to start Office Excel and open the workbook.

   The workbook should contain a chart that displays the stress test results by using the data and settings that you specified.
10. In Office Excel, click the **Sheet1** tab.

    This is the worksheet that your code generated. The first column contains the applied stress values, and the remaining three columns contain the deflections recorded at each of the three temperatures.
11. Close Office Excel.
12. Close the Graphing Data window.
13. Close Visual Studio:
    - On the **File** menu, click **Exit**.

MCT USE ONLY. STUDENT USE PROHIBITED

MCT USE ONLY. STUDENT USE PROHIBITED