

OFFICIAL MICROSOFT LEARNING PRODUCT

10266A

Programming in C# with
Microsoft® Visual Studio® 2010

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.

Product Number: 10266A

Part Number: 01918

Released: 09/2010

Module 1

Lab Instructions: Introducing C# and the .NET Framework

Contents:

Exercise 1: Building a Simple Console Application	5
Exercise 2: Building a WPF Application	10
Exercise 3: Verifying the Application	14
Exercise 4: Generating Documentation for an Application	17

Lab: Introducing C# and the .NET Framework

- Exercise 1: Building a Simple Console Application
- Exercise 2: Building a WPF Application
- Exercise 3: Verifying the Application
- Exercise 4: Generating Documentation for an Application

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

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

Introduction

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

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

Lab Setup

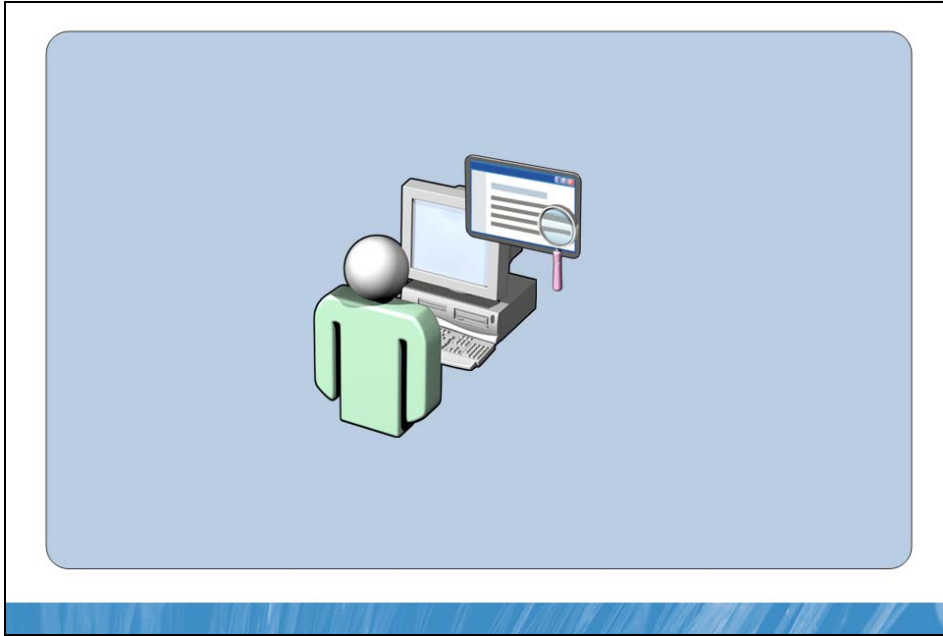
For this lab, you will use the available virtual machine environment. Before you begin the lab, you must:

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



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

Lab Scenario



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

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

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

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

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

Exercise 1: Building a Simple Console Application

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

Scenario

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

The main tasks for this exercise are as follows:

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

► Task 1: Create a new Console Application project

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

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

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

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

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

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

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

```
static void Main(string[] args)  
{  
    // Buffer to hold a line as it is read in  
    string line;  
  
    // Read a line of text from the keyboard  
    line = Console.ReadLine();  
  
    // Write the results out to the console window  
    Console.WriteLine(line);  
}
```

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

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

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

```
static void Main(string[] args)  
{  
    // Buffer to hold a line as it is read in  
    string line;  
  
    // Loop until no more input (Ctrl-Z in a console, or end-of-file)  
    while ((line = Console.ReadLine()) != null)  
    {
```



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

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

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

```
static void Main(string[] args)  
{  
    // Buffer to hold a line as it is read in  
    string line;  
  
    // Loop until no more input (Ctrl-Z in a console, or end-of-file)  
    while ((line = Console.ReadLine()) != null)  
    {  
        // Write the results out to the console window  
        Console.WriteLine(line);  
    }  
}
```

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

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

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

```
static void Main(string[] args)  
{  
    // Buffer to hold a line as it is read in  
    string line;  
  
    // Loop until no more input (Ctrl-Z in a console, or end-of-file)  
    while ((line = Console.ReadLine()) != null)  
    {  
        // Write the results out to the console window  
        Console.WriteLine(line);  
    }  
}
```

```

        // 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.

```

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

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

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

```

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

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

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

```
23.54367,25.6789
```

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

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

► **Task 5: Test the application by using a data file**

1. Perform the following steps to add the DataFile.txt file that contains the sample data to the project. This file is located in the E:\Labfiles\Lab 1\Ex1\Starter folder. These steps specify that the file should be copied to the folder that holds the compiled application when the project is built:
 - a. In Solution Explorer, right-click the **ConsoleApplication** project, point to **Add**, and then click **Existing Item**.
 - b. In the **Add Existing Item – ConsoleApplication** dialog box, move to the **E:\Labfiles\Lab 1\Ex1\Starter** folder, select **All Files (*.*)** in the drop-down list box adjacent to the **File name** text box, click **DataFile.txt**, and then click **Add**.
 - c. In Solution Explorer, select **DataFile.txt**. In the Properties window, change the **Build Action** property to **None**, and then change the **Copy to Output** property to **Copy Always**.
2. Rebuild the application.
3. Open a Visual Studio Command Prompt window, and then move to the **E:\Labfiles\Lab 1\Ex1\Starter\ConsoleApplication\bin\Debug** folder.
4. Run the ConsoleApplication application and redirect input to come from DataFile.txt.

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

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

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

```
ConsoleApplication < DataFile.txt
```

5. Close the Command Prompt window, and then return to Visual Studio.
6. Modify the project properties to redirect input from the DataFile.txt file when the project is run by using Visual Studio.
7. Run the application in Debug mode from Visual Studio.

The application will run, but the console window will close immediately after the output is generated. This is because Visual Studio only prompts the user to close the console window when a program is run without debugging. When a program is run in Debug mode, Visual Studio automatically closes the console window as soon as the program finishes.

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

Exercise 2: Building a WPF Application

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

Scenario

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

The main tasks for this exercise are as follows:

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

► Task 1: Create a new WPF Application project

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

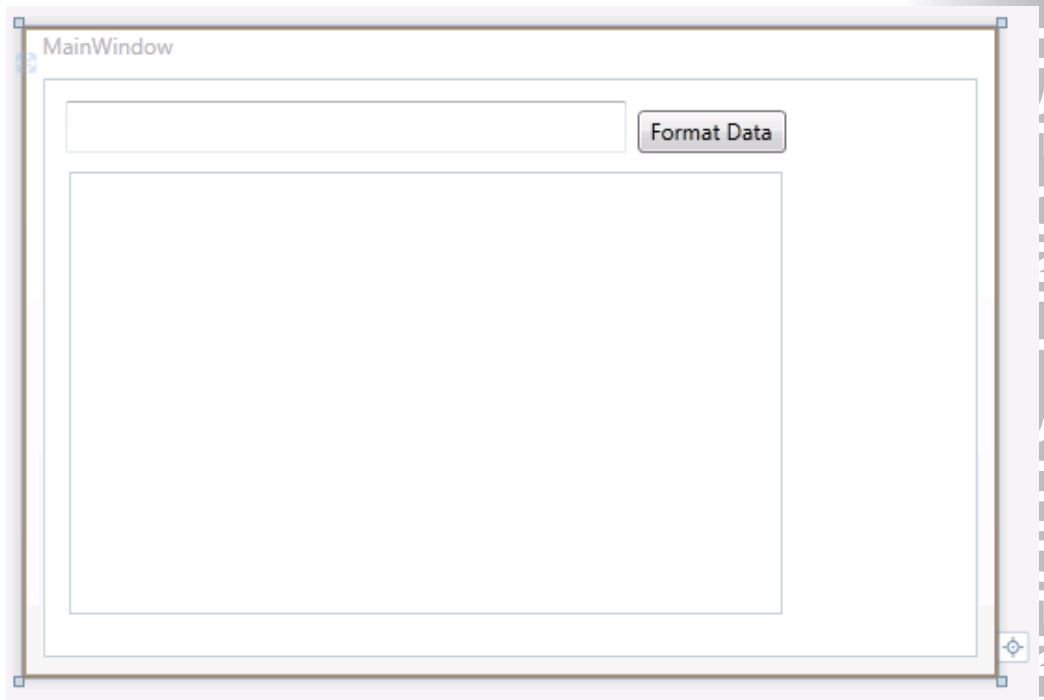
► Task 2: Create the user interface

1. Add **TextBox**, **Button**, and **TextBlock** controls to the `MainWindow` window. Place them anywhere in the window.

2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

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	<i>blank</i>
	VerticalAlignment	Top
	Width	384

The MainWindow window should look like the following screen shot.



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

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

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

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

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

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

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

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

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

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

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

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

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

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

3. Perform the following steps to modify the project settings to redirect standard input to come from the DataFile.txt file. A copy of this file is available in the E:\Labfiles\Lab 1\Ex2\Starter folder:
 - a. In Solution Explorer, right-click the **WpfApplication** project, point to **Add**, and then click **Existing Item**.
 - b. In the **Add Existing Item – WpfApplication** dialog box, move to the **E:\Labfiles\Lab 1\Ex2\Starter** folder, select **All Files (*.*)** in the drop-down list box adjacent to the **File name** text box, click **DataFile.txt**, and then click **Add**.
 - c. In Solution Explorer, select **DataFile.txt**. In the Properties window, change the **Build Action** property to **None**, and then change the **Copy to Output** property to **Copy Always**.
 - d. In Solution Explorer, right-click the **WpfApplication** project, and then click **Properties**.
 - e. On the **Debug** tab, in the **Command line arguments:** text box, type **< DataFile.txt**
 - f. On the **File** menu, click **Save All**.
 - g. Close the WpfApplication properties window.
4. Build and run the application in Debug mode. Verify that, when the application starts, it reads the data from DataFile.txt and displays in the **TextBlock** control the results in the following code example.

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

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

Exercise 3: Verifying the Application

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

Scenario

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

The main tasks for this exercise are as follows:

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

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

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

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



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

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

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

When the application runs the **Window_Loaded** event handler, it reaches the breakpoint and drops into Visual Studio. The opening brace of the method is highlighted.

3. Step into the first statement in the **Window_Loaded** method that contains executable code.

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

4. Examine the value of the line variable. It should be **null** because it has not yet been assigned a value.
5. Step into the next statement.
The cursor moves to the opening brace at the start of the body of the **while** loop.
6. Examine the value of the line variable. It should be **1.2543,0.342**. This is the text from the first line of the DataFile.txt file. The **Console.ReadLine** statement in the **while** statement reads this text from the file.
7. Step into the next statement.
The cursor moves to the line in the following code example.

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

8. Step into the next statement.
9. Examine the value of the line variable. It should now be **1.2543 y:0.342**. This is the result of calling the **Replace** method and assigning the result back to line.
10. Step into the next statement.
11. Examine the value of the line variable. It should now be **x:1.2543 y:0.342\n**. This is the result of prefixing the text "x:" to line and suffixing a newline character.
12. Step into the next statement.
The cursor moves to the closing brace at the end of the **while** loop.
13. In the Immediate window, examine the value of the **Text** property of the **formattedText TextBlock** control. It should contain the same text as the line variable.



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

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

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

Exercise 4: Generating Documentation for an Application

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

Scenario

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

The main tasks for this exercise are as follows:

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

► Task 1: Open the starter project

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

► Task 2: Add XML comments to the application

1. Display the MainWindow.xaml.cs file.

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

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

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

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

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

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

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

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

6. Save MainWindow.xaml.cs.

► Task 3: Generate an XML comments file

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

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

► **Task 4: Generate a .chm file**

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

Module 2

Lab Instructions: Using C# Programming Constructs

Contents:

Exercise 1: Calculating Square Roots with Improved Accuracy	4
Exercise 2: Converting Integer Numeric Data to Binary	12
Exercise 3: Multiplying Matrices	16

Lab: Using C# Programming Constructs

- Exercise 1: Calculating Square Roots with Improved Accuracy
- Exercise 2: Converting Integer Numeric Data to Binary
- Exercise 3: Multiplying Matrices

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 90 minutes

Objectives

After completing this lab, you will be able to:

- Use C# data types and expressions to help implement a numeric algorithm.
- Use C# programming constructs to perform common programming tasks.
- Use arrays to store and process data.

Introduction

In this lab, you will create several applications that implement some common algorithms. This will help you to become familiar with using the C# syntax and learn many of the core C# programming constructs.



Important: The purpose of these exercises, and the remaining exercises throughout this course, is not to make you familiar with mathematical algorithms or engineering

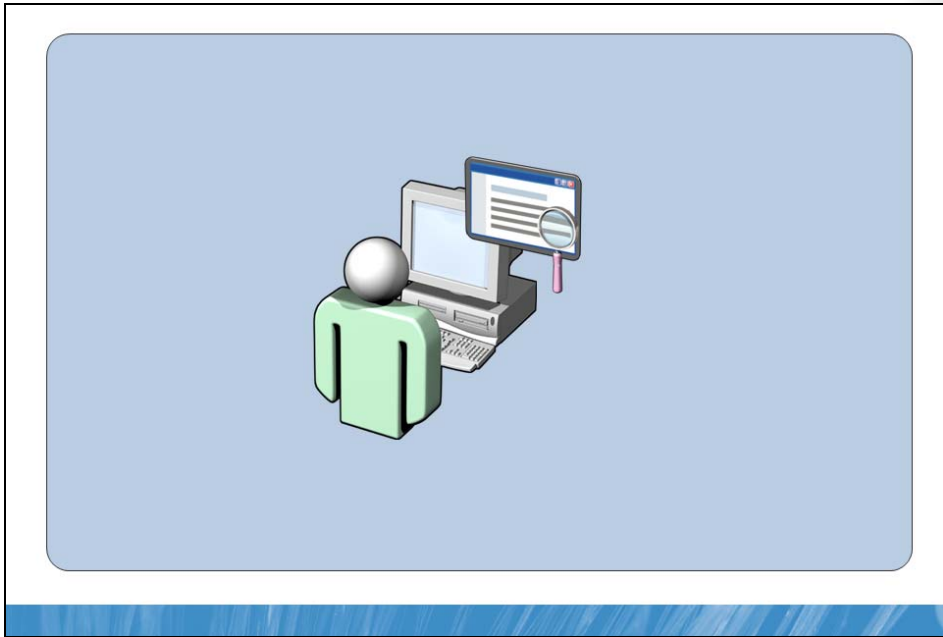
processes. Rather, the aim is to enable you to take a description of a problem or algorithm and use C# to implement a solution.

Lab Setup

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

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

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data. You have been asked to implement some embedded functionality that several scientific instruments require. You will write C# applications to build and test your implementations.

Exercise 1: Calculating Square Roots with Improved Accuracy

In this exercise, you will write a program that prompts the user for a numeric value and then uses Newton's method to calculate the square root of this number. You will display the result, and compare it to the **double** value that is calculated by using the **Math.Sqrt** method in the .NET Framework class library.

Scenario

Some of the software that is being developed to support devices that perform scientific analysis requires applications to perform calculations with a high degree of accuracy. The .NET Framework uses the **double** type to perform many of its calculations. The **double** type has a very large range, but the accuracy is not always sufficient. The **decimal** type provides a higher degree of accuracy at the cost of a

smaller range and increased memory requirements. However, this accuracy is important. One scientific calculation requires the ability to calculate square roots to a high degree of accuracy. You decide to implement Newton's algorithm for estimating and successively refining square roots, but generate the result by using the **decimal** type.

The process that Newton used for calculating the square root of 10 is as follows:

1. Start with an initial guess: use the value that you want to find the square root of and divide by 2.

In this case, $10 / 2$, has the value 5.

2. Refine the guess by dividing the original number by the previous guess, adding the value of the previous guess, and dividing the entire result by 2: calculate $((\text{number} / \text{guess}) + \text{guess}) / 2$.

In this example, calculate $((10 / 5) + 5) / 2 = 3.5$

The answer 3.5 then becomes the next guess.

3. Perform the calculation $((\text{number} / \text{guess}) + \text{guess}) / 2$ again, with the new guess

In this example, calculate $((10 / 3.5) + 3.5) / 2 = 3.17857$

3.17857 is then the next guess.

4. Repeat this process until the difference between subsequent guesses is less than some predetermined amount. The final guess is the square root of 10 to the accuracy that was specified by this predetermined amount.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Calculate square roots by using the **Math.Sqrt** method of the .NET Framework.
4. Calculate square roots by using Newton's method.
5. Test the application.

► **Task 1: Create a new WPF Application project**

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$word**.
2. Open Microsoft Visual Studio® 2010.
3. Create a new project called SquareRoots by using the Windows® Presentation Foundation (WPF) Application template in the E:\Labfiles\Lab 2\Ex1\Starter folder.

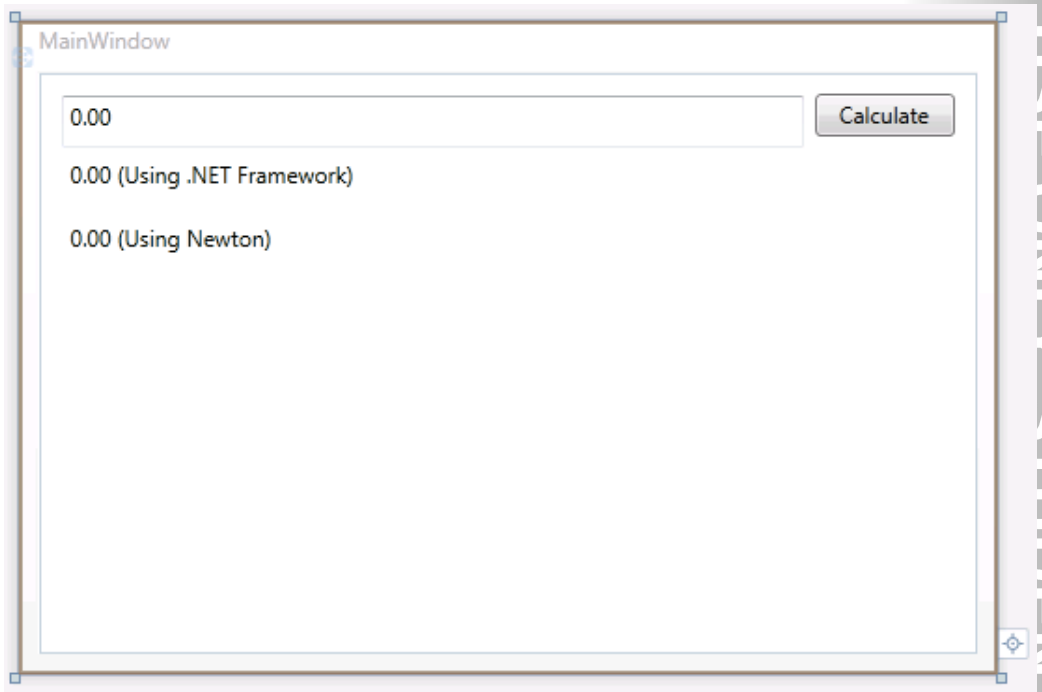
► **Task 2: Create the user interface**

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

Control	Property	Value
TextBox	Name	inputTextBox
	Height	28
	HorizontalAlignment	Left
	Margin	12,12,0,0
	Text	0.00
	VerticalAlignment	Top
	Width	398
Button	Name	calculateButton
	Content	Calculate
	Height	23
	HorizontalAlignment	Right
	Margin	0,11,12,0

Control	Property	Value
	VerticalAlignment	Top
	Width	75
Label	Name	frameworkLabel
	Content	0.00 (Using .NET Framework)
	Height	28
	HorizontalAlignment	Left
	Margin	12,41,0,0
	VerticalAlignment	Top
	Width	479
Label	Name	newtonLabel
	Content	0.00 (Using Newton)
	Height	28
	HorizontalAlignment	Left
	Margin	12,75,0,0
	VerticalAlignment	Top
	Width	479

The MainWindow window should look like the following screen shot.



► **Task 3: Calculate square roots by using the Math.Sqrt method of the .NET Framework**

1. Create an event handler for the **Click** event of the button.
2. In the **calculateButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into a **double** value. Store the **double** value in a variable called **numberDouble**. Use the **TryParse** method of the **double** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "Please enter a double," and then execute a **return** statement to quit the method.



Note: You can display a message in a message box by using the **MessageBox.Show** method.

3. Check that the value that the user enters is a positive number. If it is not, display a message box with the text "Please enter a positive number," and then return from the method.
4. Calculate the square root of the value in the `numberDouble` variable by using the **Math.Sqrt** method. Store the result in a double variable called `squareRoot`.
5. Format the value in the `squareRoot` variable by using the layout shown in the following code example, and then display it in the **frameworkLabel Label** control.

99.999 (Using the .NET Framework)

Use the **string.Format** method to format the result. Set the **Content** property of a **Label** control to display the formatted result.

6. Build and run the application to test your code. Use the test values that are shown in the following table, and then verify that the correct square roots are calculated and displayed (ignore the "Using Newton" label for the purposes of this test).

Test value	Expected result
25	5
625	25
0.00000001	0.0001
-10	Message box appears with the message "Please enter a positive number"
Fred	Message box appears with the message "Please enter a double"
10	3.16227766016838
8.8	2.96647939483827
2.0	1.4142135623731
2	1.4142135623731

7. Close the application and return to Visual Studio.

► **Task 4: Calculate square roots by using Newton's method**

1. In the **calculateButton_Click** method, after the code that you added in the previous task, create a decimal variable called **numberDecimal**. Initialize this variable with the data that the user enters in the **inputTextBox TextBox** control, but convert it into a **decimal** this time (previously, you read it as a **double**). If the text that the user enters is not valid, display a message box with the text "Please enter a decimal," and then execute a **return** statement to quit the method.



Note: This step is necessary because the **decimal** and **double** types have different ranges. A number that the user enters that is a valid **double** might be out of range for the **decimal** type.

2. Declare a decimal variable called **delta**, and initialize it to the value of the expression **Math.Pow(10, -28)**. This is the smallest value that the **decimal** type supports, and you will use this value to determine when the answer that is generated by using Newton's method is sufficiently accurate. When the difference between two successive estimates is less than this value, you will stop.



Note: The **Math.Pow** method returns a **double**. You will need to use the **Convert.ToDecimal** method to convert this value to a decimal before you assign it to the **delta** variable.

3. Declare another decimal variable called **guess**, and initialize it with the initial guess at the square root. This initial guess should be the result of dividing the value in **numberDecimal** by 2.
4. Declare another decimal variable called **result**. You will use this variable to generate values for each iteration of the algorithm, based on the value from the previous iteration. Initialize the result variable to the value for the first iteration by using the expression **((numberDecimal / guess) + guess) / 2**.
5. Add a **while** loop to generate further refined guesses. The body of the **while** loop should assign **result** to **guess**, and generate a new value for **result** by using the expression **((numberDecimal / guess) + guess) / 2**. The **while** loop should terminate when the difference between **result** and **guess** is less than or equal to **delta**.

Exercise 2: Converting Integer Numeric Data to Binary

In this exercise, you will create another application that enables the user to enter an integer value, generate a string that holds the binary representation of this value, and then display the result.

Scenario

Another device has the requirement to display decimal numeric data in a binary format. You have been asked to develop some code that can convert a non-negative decimal integer value into a string that contains the binary representation of this value.

The process for converting the decimal value 6 into its binary representation is as follows:

1. Divide the integer by 2, save the integer result, and use the remainder as the first binary digit.

In this example, $6 / 2$ is 3 remainder 0. Save the character "0" as the first character of the binary representation.

2. Divide the result of the previous division by 2, save the result, and use the remainder as the next binary digit.

In this example, $3 / 2$ is 1 remainder 1. Save the character "1" as the next character of the binary representation.

3. Repeat the process until the result of the division is zero.

In this example, $1 / 2$ is zero remainder 1. Save the character "1" as the final character of the binary representation.

4. Display the characters saved in reverse order.

In this example, the characters were generated in the sequence "0", "1", "1", so display them in the order "1", "1", "0". The value 110 is the binary representation of the decimal value 6.

The main tasks for this exercise are as follows:

1. Create a new WPF Application project.
2. Create the user interface.
3. Add code to generate the binary representation of an integer value.
4. Test the application.

► **Task 1: Create a new WPF Application project**

- Create a new project called IntegerToBinary by using the WPF Application template in the E:\Labfiles\Lab 2\Ex2\Starter folder.

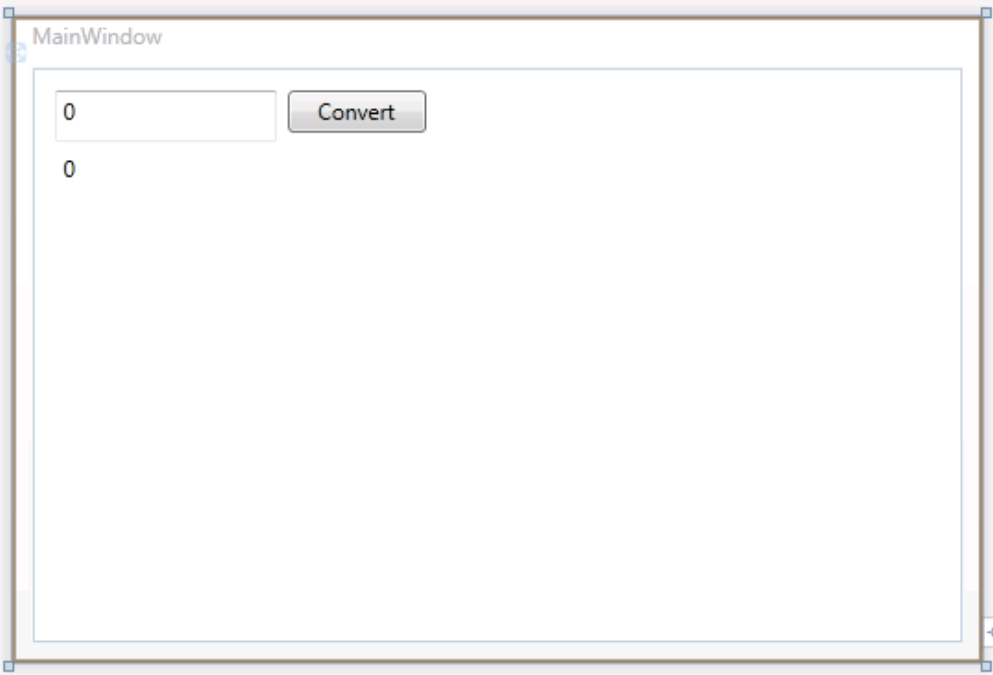
► **Task 2: Create the user interface**

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

Control	Property	Value
TextBox	Name	inputTextBox
	Height	28
	HorizontalAlignment	Left
	Margin	12,12,0,0
	Text	0
	VerticalAlignment	Top
	Width	120
Button	Name	convertButton
	Content	Convert
	Height	23
	HorizontalAlignment	Left
	Margin	138,12,0,0
	VerticalAlignment	Top
	Width	75

Control	Property	Value
Label	Name	binaryLabel
	Content	0
	Height	28
	HorizontalAlignment	Left
	Margin	12,41,0,0
	VerticalAlignment	Top
	Width	120

The MainWindow window should look like the following screen shot.



► **Task 3: Add code to generate the binary representation of an integer value**

1. Create an event handler for the **Click** event of the button.
2. In the **convertButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into an **int** type. Store the integer value in a variable called **i**. Use the **TryParse** method of the **int** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "TextBox does not contain an integer," and then execute a **return** statement to quit the method.
3. Check that the value that the user enters is not a negative number (the integer-to-binary conversion algorithm does not work for negative numbers). If it is negative, display a message box with the text "Please enter a positive number or zero," and then return from the method.
4. Declare an integer variable called **remainder** and initialize it to zero. You will use this variable to hold the remainder after dividing **i** by 2 during each iteration of the algorithm.
5. Declare a **StringBuilder** variable called **binary** and instantiate it. You will use this variable to construct the string of bits that represent **i** as a binary value.
6. Add a **do** loop that performs the following tasks:
 - a. Calculate the remainder after dividing **i** by 2, and then store this value in the **remainder** variable.
 - b. Divide **i** by 2.
 - c. Prefix the value of **remainder** to the start of the string being constructed by the **binary** variable.

Terminate the **do** loop when **i** is less than or equal to zero.



Note: To prefix data into a **StringBuilder** object, use the **Insert** method of the **StringBuilder** class, and then insert the value of the data at position 0.

7. Display the value in the **binary** variable in the **binaryLabel Label** control.



Note: Use the **ToString** method to retrieve the string that a **StringBuilder** object constructs. Set the **Content** property of the **Label** control to display this string.

► 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	1000000000000000

2. Close the application and return to Visual Studio.

Exercise 3: Multiplying Matrices

In this exercise, you will create another WPF application. This WPF application will provide a user interface that enables the user to provide the data for two matrices and store this data in rectangular arrays. The application will calculate the product of these two arrays and display them.

Scenario

Some of the devices that Fabrikam, Inc. has developed perform calculations that involve sets of data that are held as matrices. You have been asked to implement code that performs matrix multiplication. You decide to test your code by building

a WPF application that enables a user to specify the data for two matrices, calculate the product of these matrices, and then view the result.

Multiplying matrices is an iterative process that involves calculating the sum of the products of the values in each row in one matrix with the values in each column in the other, as the following screen shot shows.

$$\begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 5 & 4 & 2 & 3 \end{pmatrix} \times \begin{pmatrix} 3 & \cdot & \cdot & \cdot \\ 2 & \cdot & \cdot & \cdot \\ 6 & \cdot & \cdot & \cdot \\ 1 & \cdot & \cdot & \cdot \end{pmatrix} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & x_{3,2} & \cdot & \cdot \end{pmatrix}$$

This screen shot shows a 3×4 matrix multiplying a 4×5 matrix. This will result in a 3×5 matrix.



Note: The number of columns in the first matrix must match the number of rows in the second matrix. The starter code that is provided for you in this lab ensures that this is always the case.

To calculate each element $x_{a,b}$ 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:

$$(5 \times 3) + (4 \times 2) + (2 \times 6) + (3 \times 1) = 38$$

You perform this calculation for every element in the result matrix.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication project and examine the starter code.
2. Define the matrix arrays and populate them with the data in the **Grid** controls.
3. Multiply the two input matrices and calculate the result.
4. Display the results and test the application.

► Task 1: Open the MatrixMultiplication project and examine the starter code

1. Open the MatrixMultiplication project located in the
E:\Labfiles\Lab 2\Ex3\Starter folder.

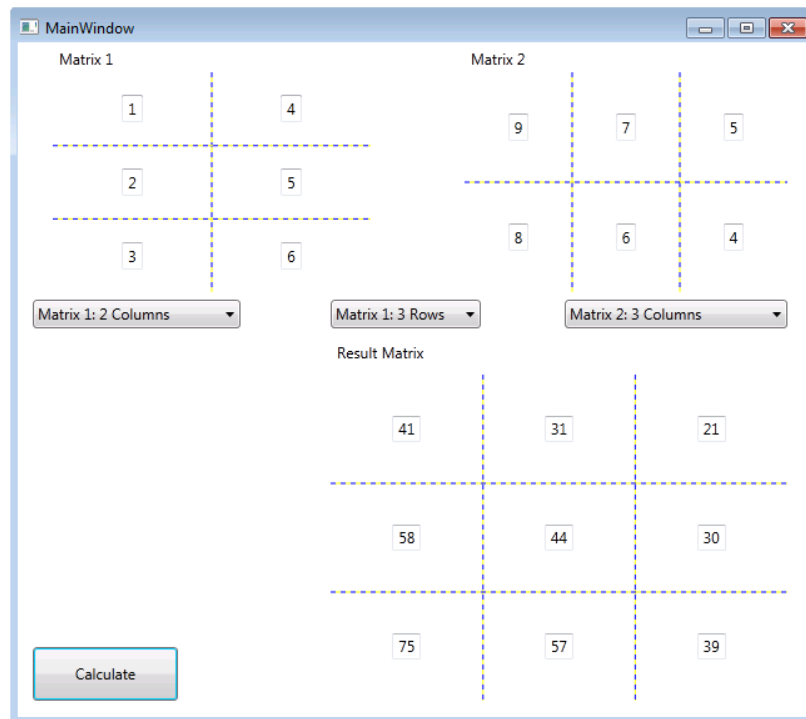
2. Examine the user interface that the MainWindow window defines.

The user interface contains three **Grid** controls, three **ComboBox** controls, and a **Button** control.

When the application runs, the first **Grid** control, labeled **Matrix 1**, represents the first matrix, and the second **Grid** control, labeled **Matrix 2**, represents the second matrix. The user can specify the dimensions of the matrices by using the **ComboBox** controls, and then enter data into each cell in them. There are several rules that govern the compatibility of matrices to be multiplied together, and Matrix 2 is automatically configured to have an appropriate number of rows based on the number of columns in Matrix 1.

When the user clicks the **Calculate** button, Matrix 1 and Matrix 2 are multiplied together, and the result is displayed in the **Grid** control labeled **Result Matrix**. The dimensions of the result are determined by the shapes of Matrix 1 and Matrix 2.

The following screen shot shows the completed application running. The user has multiplied a 2×3 matrix with a 3×2 matrix, and the result is a 3×3 matrix.



► **Task 2: Define the matrix arrays and populate them with the data in the Grid controls**

1. In Visual Studio, review the task list.
2. Open the `MainWindow.xaml.cs` file.
3. At the top of the **MainWindow** class, remove the comment **TODO Task 2 declare variables**, and then add statements that declare three two-dimensional arrays called **matrix1**, **matrix2**, and **result**. The type of the elements in these arrays should be **double**, but the size of each dimension should be omitted because the arrays will be dynamically sized based on the input that the user provides. The first dimension will be set to the number of columns, and the second dimension will be set to the number of rows.
4. In the task list, double-click the task **TODO Task 2 Copy data from input Grids**. This task is located in the **buttonCalculate_Click** method.
5. In the **buttonCalculate_Click** method, remove the comment **TODO Task 2 Copy data from input Grids**. Add two statements that call the **getValuesFromGrid** method. This method (provided in the starter code) expects the name of a **Grid** control and the name of an array to populate with data from that **Grid** control. In the first statement, specify that the method should use the data in **grid1** to populate **matrix1**. In the second statement, specify that the method should use the data from **grid2** to populate **matrix2**.
6. Remove the comment **TODO Task 2 Get the matrix dimensions**. Declare three integer variables called **m1columns**, **m2rows**, **m1rows**, and **m2columns**. Initialize **m1columns** with the number of columns in the **matrix1** array (this is also the same as the number of rows in the **matrix2** array) by using the **GetLength** method of the first dimension of the array. Initialize **m1rows** with the number of rows in the **matrix1** array by using the **GetLength** method of the second dimension of the array. Initialize **m2columns** with the number of columns in the **matrix2** array.

► **Task 3: Multiply the two input matrices and calculate the result**

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 3 Calculate the result**. Define a **for** loop that iterates through all of the rows in the **matrix1** array. The dimensions of an array are integers, so use an integer variable called **row** as the control variable in this **for** loop. Leave the body of the **for** loop blank; you will add code to this loop in the next step.

2. In the body of the **for** loop, add a nested **for** loop that iterates through all of the columns in the **matrix2** array. Use an integer variable called **column** as the control variable in this **for** loop. Leave the body of this **for** loop blank.
3. The contents of each cell in the **result** array are calculated by adding the product of each item in the row identified by the row variable in **matrix1** with each item in the column identified by the column variable in **matrix2**. You will require another loop to perform this calculation, and a variable to store the result as this loop calculates it.

In the inner **for** loop, declare a double variable called **accumulator**, and then initialize it to zero.

4. Add another nested **for** loop after the declaration of the **accumulator** variable. This loop should iterate through all of the columns in the current row in the **matrix1** array. Use an integer variable called **cell** as the control variable in this **for** loop. Leave the body of this **for** loop blank.
5. In the body of this **for** loop, multiply the value in **matrix1[cell, row]** with the value in **matrix2[column, cell]**, and then add the result to **accumulator**.
6. After the closing brace of the innermost **for** loop, store the value in **accumulator** in the **result** array. The value should be stored in the cell that the column and row variables have identified.

► Task 4: Display the results and test the application

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 4 Display the result**. The starter code contains a method called **initializeGrid** that displays the contents of an array in a **Grid** control in the WPF window. Add a statement that calls this method. Specify that the method should use the **grid3 Grid** control to display the contents of the **result** array.
2. Build the solution and correct any errors.
3. Run the application in Debug mode.
4. In the **MainWindow** window, define **Matrix 1** as a 3×2 matrix and define **Matrix 2** as a 3×3 matrix.



Note: The number of rows in the **Matrix 2** matrix is determined by the number of columns in the **Matrix 1** matrix.

5. Specify the values for the cells in the matrices as shown in the following tables.

Matrix 1		
1	5	-9
3	-7	11

Matrix 2		
2	-8	14
4	-10	16
6	-12	18

6. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

Result		
-32	50	-68
44	-86	128

7. Change the data in **Matrix 2** as shown in the following table.

Matrix 2		
1	0	0
0	1	0
0	0	1

8. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

Result		
1	5	-9
3	-7	11

Matrix 2 is an example of an identity matrix. When you multiply a matrix by an identity matrix, the result is the same data as defined by the original matrix (it is the matrix equivalent of multiplying a value by 1 in regular arithmetic). In this case, the values in the **Result** matrix are the same as those in **Matrix 1**.

9. Change the data in **Matrix 2** again, as shown in the following table.

Matrix 2		
-1	0	0
0	-1	0
0	0	-1

10. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

Result		
-1	-5	9
-3	7	-11

This time, the values in **Result** are the same as those in **Matrix 1** except that the sign of each element is inverted (**Matrix 2** is the matrix equivalent of -1 in regular arithmetic).

11. Close the MainWindow window.
12. Close Visual Studio.

Module 3

Lab Instructions: Declaring and Calling Methods

Contents:

Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid’s Algorithm	4
Exercise 2: Calculating the GCD of Three, Four, or Five Integers	8
Exercise 3: Comparing the Efficiency of Two Algorithms	12
Exercise 4: Displaying Results Graphically	18
Exercise 5: Solving Simultaneous Equations (optional)	21

Lab: Declaring and Calling Methods

- Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid's Algorithm
- Exercise 2: Calculating the GCD of Three, Four, or Five Integers
- Exercise 3: Comparing the Efficiency of Two Algorithms
- Exercise 4: Displaying Results Graphically
- Exercise 5: Solving Simultaneous Equations (optional)

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 75 minutes

Objectives

After completing this lab, you will be able to:

- Create and call methods.
- Define overloaded methods.
- Define methods that take output parameters.
- Define methods that take optional parameters and call them by using named arguments.

Introduction

In this lab, you will create methods to calculate the greatest common divisor (GCD) of a pair of positive integers. You will create an overloaded version of one of these methods that can take up to five integer parameters. You will modify the methods to take an output parameter that returns the time taken to perform the

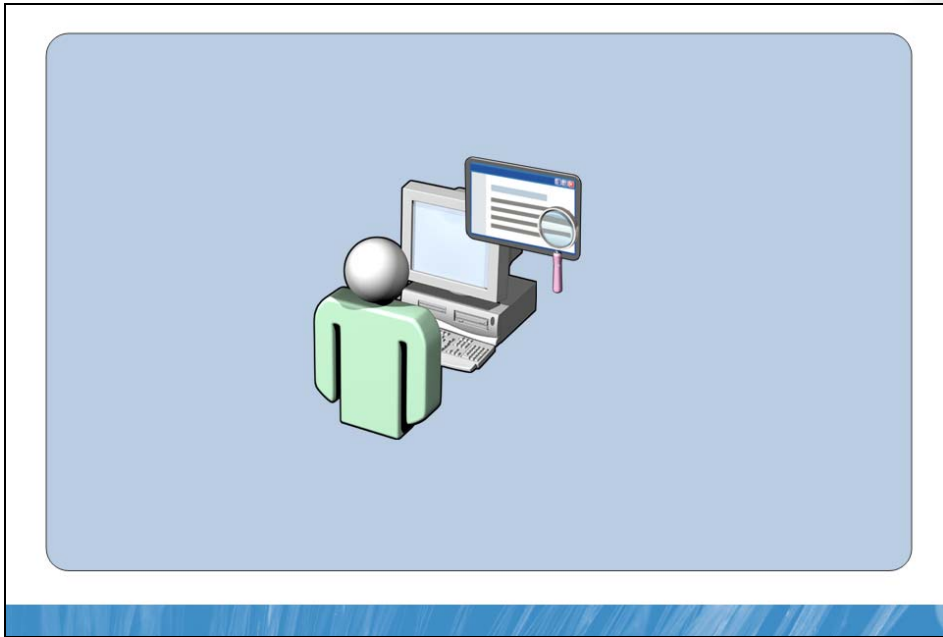
calculations. Finally, you will use a method that uses optional parameters to display the relative performance of the methods by displaying a simple graph.

Lab Setup

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

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

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Some of the calculations that various scientific instruments perform depend on statistical information that is generated by using prime numbers. One of your colleagues has implemented a method for generating prime numbers, but it does not have sufficient performance to meet the requirements of the devices that it will be used with. The software analysts have examined the code and have determined that it can be improved by using a faster algorithm for calculating the GCDs. You have been asked to implement a test application that can calculate the GCD of a set of numbers by using different well-known algorithms, and compare their relative performance.

Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid's Algorithm

In this exercise, you will write a method that implements Euclid's algorithm for calculating the GCD of two integers passed in as parameters. You will test this method by using a Windows® Presentation Foundation (WPF) application that

prompts the user for the parameter values, and displays the result. You will also generate a unit test project to enable you to automate testing this method.

Scenario

Some of the data that is collected by devices built by Fabrikam, Inc. must be encrypted for security purposes. Encryption algorithms often make use of prime numbers. A part of the algorithm that generates prime numbers needs to calculate the GCD of two numbers.

The GCD of two numbers is the largest number that can exactly divide into the two numbers. For example, the GCD of 15 and 12 is 3. Three is the largest whole number that divides exactly into 15 and 12.

The process for finding the GCD of 2806 and 345 by using Euclid's algorithm is as follows.

1. Keep taking 345 away from 2806 until less than 345 is left and store the remainder.
In this case, $2806 = (8 \times 345) + 46$, so the remainder is 46.
2. Keep taking the remainder (46) away from 345 until less than 46 is left, and store the remainder.
 $345 = (7 \times 46) + 23$, so the remainder is 23.
3. Keep taking 23 away from 46 until less than 23 is left, and store the remainder.
 $46 = (2 \times 23) + 0$
4. The remainder is 0, so the GCD of 2806 and 345 was the value of the previously stored remainder, which was 23 in this case.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Implement Euclid's algorithm.
3. Test the **FindGCDEuclid** method.
4. Create a unit test for the **FindGCDEuclid** method.

► Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$w0rd**.

2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 3\Snippets folder.
4. Open the Euclid solution in the E:\Labfiles\Lab 3\Ex1\Starter folder.

► **Task 2: Implement Euclid's algorithm**

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 1, Task 2** task.
This task is located in the `GCDAlgorithms.cs` file.
3. In the `GCDAlgorithms` class, remove the **TODO Exercise 1, Task 2** comment and declare a **public static** method called `FindGCDEuclid`. The method should accept two integer parameters called *a* and *b*, and return an integer value.
4. In the `FindGCDEuclid` method, add code that calculates and returns the GCD of the values specified by the parameters *a* and *b* by using Euclid's algorithm.
Euclid's algorithm works as follows:
 - a. If *a* is zero, the GCD of *a* and *b* is *b*.
 - b. Otherwise, repeatedly subtract *b* from *a* (when *a* is greater than *b*) or subtract *a* from *b* (when *b* is greater than *a*) until *b* is zero.
 - c. The GCD of the two original parameters is the new value in *a*.

► **Task 3: Test the FindGCDEuclid method**

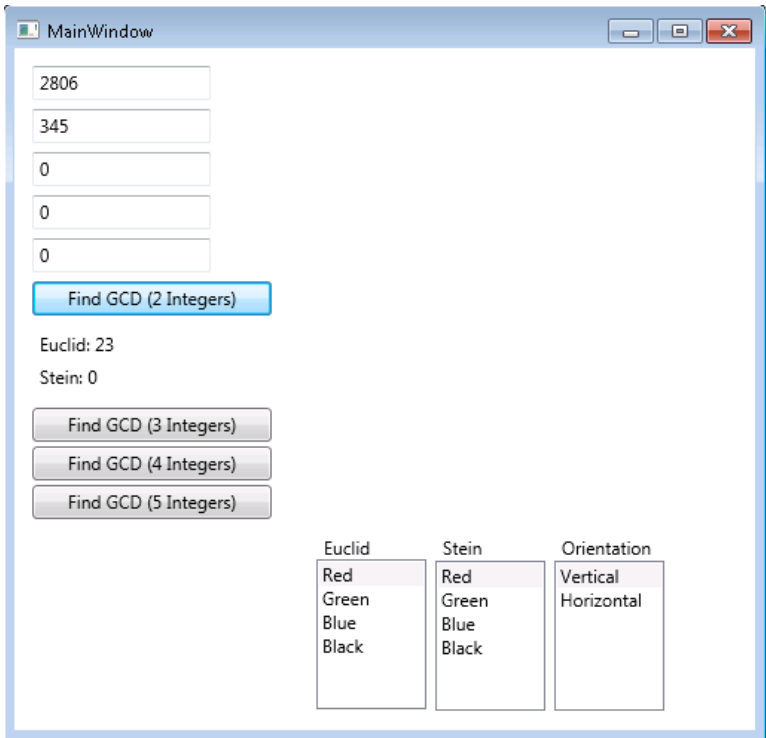
1. Use the Task List window to navigate to the **TODO Exercise 1, Task 3** task.
This task is located in the `MainWindow.xaml.cs` file. This is the code-behind file for a WPF window that you will use to test the `FindGCDEuclid` method and display the results.
2. Remove the **TODO Exercise 1, Task 3** comment, add code to call the static `FindGCDEuclid` method of the `GCDAlgorithms` class, and display the results in the `resultEuclid` label control. In the method call, use the `firstNumber` and `secondNumber` variables as arguments (these variables contain values that the user enters in the WPF window). Finally, the result should be formatted as the following code example shows.

Euclid: *result*



Hint: Set the **Content** property of a label control to display data in a label. Use the **String.Format** method to create a formatted string.

- 3. Build the solution and correct any errors.
- 4. Run the GreatestCommonDivisor application.
- 5. In the GreatestCommonDivisor application, in the MainWindow window, in the first text box, type **2806**
- 6. In the second text box, type **345** and then click **Find GCD (2 Integers)**. The result of **23** should be displayed, as the following screen shot shows.



- 7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

First number	Second number	Result
0	0	0
0	10	10
25	10	5
25	100	25
26	100	2
27	100	1

8. Close the GreatestCommonDivisor application.

► **Task 4: Create a unit test for the FindGCDEuclid method**

1. Open the GCDAlgorithms.cs file.
2. In the **GCDAlgorithms** class, create a unit test for the **FindGCDEuclid** method. Create a new Test Project called GCD Test Project to hold the unit test.
3. In the GCD Test Project project, in the GCDAlgorithmsTest.cs file, locate the **FindGCDEuclidTest** method.
4. In the **FindGCDEuclidTest** method, set the a variable to **2806**, set the b variable to **345**, set the expected variable to **23**, and then remove the **Assert.Inconclusive** method call.
5. Open the Test View window and refresh the display if the unit test is not listed.
6. Run the FindGCDEuclidTest test and verify that the test ran successfully.

Exercise 2: Calculating the GCD of Three, Four, or Five Integers

In this exercise, you will create overloaded versions of this method that can take three, four, or five integer parameters and calculate the GCD of all of these parameters.

Scenario

Some of the encryption algorithms used by devices that Fabrikam, Inc. builds require calculating the GCD of sets of numbers, not just pairs. You have been asked to provide implementations of the Euclid algorithm that can calculate the GCD of three, four, or five integers.

The process for finding the GCD of three numbers x , y , and z is straightforward:

1. Calculate the GCD of x and y by using the algorithm for two numbers, and store the result in a variable r .
2. Calculate the GCD of r and z . The result is the GCD of x , y , and z .

You can apply the same technique to calculate the GCD of four or five integers:

- $\text{GCD}(w, x, y, z) = \text{GCD}(w, \text{GCD}(x, y, z))$
- $\text{GCD}(v, w, x, y, z) = \text{GCD}(v, \text{GCD}(w, x, y, z))$

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add overloaded methods to the **GCDAlgorithms** class.
3. Test the overloaded methods.
4. Create unit tests for the overloaded methods.

► Task 1: Open the starter project

- Open the Euclid solution in the E:\Labfiles\Lab 3\Ex2\Starter folder.
This solution contains a completed copy of the code from Exercise 1.

► Task 2: Add overloaded methods to the **GCDAlgorithms** class

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 2** task.
3. In the **GCDAlgorithms** class, remove the **TODO Exercise 2, Task 2** comment, and then declare an overloaded version of the **FindGCDEuclid** method. The method should accept three integer parameters called a , b , and c , and return an integer value.

4. In the new method, add code that uses the original **FindGCDEuclid** method, to find the GCD for the parameters *a* and *b*. Store the result in a new variable called *d*.
5. Add a second call to the original **FindGCDEuclid** method to find the GCD for variable *d* and parameter *c*. Store the result in a new variable called *e*.
6. Add code to return the parameter *e* from the **FindGCDEuclid** method.
7. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept four integer parameters called *a*, *b*, *c*, and *d*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.
8. Declare another overloaded version of the **FindGCDEuclid** method. The method should accept five integer parameters called *a*, *b*, *c*, *d*, and *e*, and return an integer value. Use the other **FindGCDEuclid** method overloads to find the GCD of these parameters and return the result.

► Task 3: Test the overloaded methods

1. Use the Task List window to navigate to the **TODO Exercise 2, Task 3** task.
This task is located in the code for the WPF window that you can use to test your code.
2. Remove the **TODO Exercise 2, Task 3** comment, locate the **else if (sender == findGCD3)** block, and modify the statement that sets the **Content** property of the **resultEuclid** label to "N/A" as follows:
 - a. Call the **FindGCDEuclid** overload that accepts three parameters and pass the variables *firstNumber*, *secondNumber*, and *thirdNumber* as arguments.
 - b. Display the results in the **resultEuclid** label control. The result should be formatted as the following code example shows.

```
Euclid: result
```

3. Locate the **else if (sender == findGCD3)** block, the **else if (sender == findGCD4)** block, and the **else if (sender == findGCD5)** block, and modify the statements that set the **Content** property of the **resultEuclid** label to "N/A". Call the appropriate **FindGCDEuclid** overload by using the *firstNumber*, *secondNumber*, *thirdNumber*, *fourthNumber*, and *fifthNumber* variables as arguments. Display the results in the **resultEuclid** label control.

4. Build the solution and correct any errors.
5. Run the GreatestCommonDivisor application.
6. In the GreatestCommonDivisor application, in the MainWindow window, type the values **7396 1978 1204 430 258** and then click **Find GCD (5 Integers)**.
Verify that the result **86** is displayed.
7. Use the window to calculate the GCD for the values that are specified in the following table, and verify that the results that are displayed match those in the table.

First number	Second number	Third number	Fourth number	Fifth number	Result
2806	345	0	0	0	23
0	0	0	0	0	0
0	0	0	0	1	1
12	24	36	48	60	12
13	24	36	48	60	1
14	24	36	48	60	2
15	24	36	48	60	3
16	24	36	48	60	4
0	24	36	48	60	12

8. Close the GreatestCommonDivisor application.

► **Task 4: Create unit tests for the overloaded methods**

1. In Visual Studio, review the task list.
2. Use the Task List window to navigate to the **TODO Exercise 2, Task 4** task.
3. Remove the **TODO Exercise 2, Task 4** comment and add a test method called **FindGCDEuclidTest1**.
4. In the **FindGCDEuclidTest1** method, declare four variables called **a**, **b**, **c**, and **expected**, and assign them values **7396**, **1978**, **1204**, and **86** respectively.

5. Declare a variable called `actual`, and assign it the result of a call to the **FindGCDEuclid** method call. Use the variables `a`, `b`, and `c` as arguments.
6. Call the **AreEqual** static method of the **Assert** class, and pass the expected and actual variables as arguments.
7. Repeat steps 4–6 to create two more test methods to test the other **FindGCDEuclid** method overloads. Create test methods called **FindGCDEuclidTest2** and **FindGCDEuclidTest3**. Use the values **7396**, **1978**, **1204**, and **430** for the **FindGCDEuclidTest2** method, and the values **7396**, **1978**, **1204**, **430**, and **258** for the **FindGCDEuclidTest3** method. The result should be **86** in both cases.
8. Open the Test View window and refresh the display if the unit test is not listed.
9. Run the **FindGCDEuclidTest**, **FindGCDEuclidTest1**, **FindGCDEuclidTest2**, and **FindGCDEuclidTest3** tests and verify that the tests ran successfully.

Exercise 3: Comparing the Efficiency of Two Algorithms

In this exercise, you will write another method that implements Stein's algorithm for calculating the GCD of two integer parameters. The method will take an output parameter that contains the time taken to perform the calculation. You will also modify the method that implements Euclid's algorithm for calculating the GCD of two parameters to take an output parameter, also containing the time taken to perform the calculation. You will then modify the WPF application to test the relative performance of the methods and display the times taken.

Scenario

Stein's algorithm is an alternative algorithm for finding the GCD of two numbers. You have been told that it is more efficient than Euclid's algorithm. A colleague has previously implemented Stein's algorithm, but you decide to test this hypothesis by comparing the time taken to calculate the GCD of pairs of numbers with that taken by using Euclid's algorithm.

The following steps describe the process of calculating the GCD of two numbers, u and v , by following Stein's algorithm:

1. $\text{gcd}(0, v) = v$ because everything divides by zero, and v is the largest number that divides v .

Similarly, $\text{gcd}(u, 0) = u$. $\text{gcd}(0, 0)$ is not typically defined, but it is convenient to set $\text{gcd}(0, 0) = 0$.

2. If u and v are both even, $\text{gcd}(u, v) = 2\text{gcd}(u/2, v/2)$ because 2 is a common divisor.
3. If u is even and v is odd, $\text{gcd}(u, v) = \text{gcd}(u/2, v)$ because 2 is not a common divisor.

Similarly, if u is odd and v is even, $\text{gcd}(u, v) = \text{gcd}(u, v/2)$.

4. If u and v are both odd, and $u \geq v$, $\text{gcd}(u, v) = \text{gcd}((u - v)/2, v)$.

If both are odd and $u < v$, $\text{gcd}(u, v) = \text{gcd}((v - u)/2, u)$.

These are combinations of one step of the simple Euclidean algorithm, which uses subtraction at each step, and an application of step 4 above. The division by 2 results in an integer because the difference of two odd numbers is even.

5. Repeat steps 3–5 until $u = v$, or (one more step) until $u = 0$.

In either case, the result is $2kv$, where k is the number of common factors of 2 found in step 2.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Implement Stein's algorithm.
3. Test the **FindGCDStein** method.
4. Add code to test the performance of the algorithms.

► Task 1: Open the starter project

- Open the Stein solution in the E:\Labfiles\Lab 3\Ex3\Starter folder.
This solution contains a completed copy of the code from Exercise 2.

► Task 2: Implement Stein's algorithm

1. Open the GCDAlgorithms.cs file.
2. At the end of the **GCDAlgorithms** class, remove the TODO comment and declare a **public static** method called **FindGCDStein**. The method should accept two integer parameters called u and v , and return an integer value.
3. In the **FindGCDStein** method, add the code in the following code example, which calculates and returns the GCD of the values that are specified by the

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  $u \geq v$ ,
// then  $\text{gcd}(u, v) = \text{gcd}((u - v)/2, v)$ .
// If both are odd and  $u < v$ , then  $\text{gcd}(u, v) = \text{gcd}((v - u)/2, u)$ .
// 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  $\text{diff}(u, v)$  is even.
    // Let  $u = \min(u, v)$ ,  $v = \text{diff}(u, v)/2$ .
    if (u < v)
    {
        v -= u;
    }
    else
    {
        int diff = u - v;
        u = v;
        v = diff;
    }
    v >>= 1;
    // Step 5.
    // Repeat steps 3-4 until  $u = v$ , or (one more step)
    // until  $u = 0$ .
    // In either case, the result is  $(2^k) * v$ , where  $k$  is
    // the number of common factors of 2 found in step 2.
} while (v != 0);

u <= k;

return u;
}

```

► Task 3: Test the FindGCDStein method

1. Open the MainWindow.xaml.cs file.
2. In the **MainWindow** class, in the **FindGCD_Click** method, locate the **TODO Exercise 3, Task 2** comment. Remove this comment and replace the statement that sets the **Content** property of the **resultStein** label with code that calls the

FindGCDStein method by using the variables `firstNumber` and `secondNumber` as arguments. Display the results in the **resultStein** label control. The result should be formatted as the following code example shows.

```
Stein: result
```

3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**.
Verify that the value **4** is displayed in both labels.
6. Close the GreatestCommonDivisor application.
7. Open the GCDAlgorithmsTest.cs file.
8. At the end of the **GCDAlgorithmsTest** class, locate the **TODO Exercise 3, Task 2** comment, remove the comment, and then add a test method called **FindGCDSteinTest**.
9. In the **FindGCDSteinTest** method, declare three variables called `u`, `v`, and `expected`, and assign them values **298467352**, **569484**, and **4** respectively.
10. Declare a variable called `actual`, and assign it the result of a call to the **FindGCDStein** method call. Use the variables `u` and `v` as arguments.
11. Call the **static AreEqual** method of the **Assert** class, and pass the `expected` and `actual` variables as arguments.
12. Open the Test View window and refresh the display if the unit test is not listed.
13. Run the **FindGCDSteinTest** test, and verify that the test ran successfully.

► **Task 4: Add code to test the performance of the algorithms**

1. Open the GCDAlgorithms.cs file.
2. In the **GCDAlgorithms** class, locate the **FindGCDEuclid** method that accepts two parameters, and modify the method signature to take an *out* parameter called `time` of type **long**.

3. At the start of the **FindGCDEuclid** method, add code to initialize the time parameter to zero, create a new **Stopwatch** object called **sw**, and start the stop watch.

The **Stopwatch** class is useful for timing code. The **Start** method starts an internal timer running. You can subsequently use the **Stop** method to halt the timer, and establish how long the interval was between starting and stopping the timer by querying the **ElapsedMilliseconds** or **ElapsedTicks** properties.

4. At the end of the **FindGCDEuclid** method, before the **return** statement, add code to stop the **Stopwatch** object, and set the *time* parameter to the number of elapsed ticks of the **Stopwatch** object.
5. Comment out the other **FindGCDEuclid** method overloads.
6. Modify the **FindGCDStein** method to include the time output parameter, and add code to record the time each method takes to run. Note that the **FindGCDStein** method contains two **return** statements, and you should record the time before each one.
7. Open the `MainWindow.xaml.cs` file.
8. In the **FindGCD_Click** method, modify each of the calls to the **FindGCDEuclid** method and the **FindGCDStein** method to use the updated method signatures, as follows:
 - a. For calling the Euclid algorithm, create a long variable called `timeEuclid`.
 - b. For calling the Stein algorithm, create a long variable called `timeStein`.
 - c. Format the results displayed in the labels as the following code example shows.

```
[Euclid]
Euclid: result, Time (ticks): result
```

```
[Stein]
Stein: result, Time (ticks): result
```

9. Comment out the code that calls the overloaded versions of the **FindGCDEuclid** method.
10. Open the `GCDAlgorithmsTest.cs` file.
11. Modify the **FindGCDEuclidTest** and **FindGCDSteinTest** methods to use the new method signatures. Comment out the methods **FindGCDEuclidTest1**, **FindGCDEuclidTest2**, and **FindGCDEuclidTest3**.

12. Build the solution and correct any errors.
13. Run the GreatestCommonDivisor application.
14. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The result of **4** should be displayed. The time reported for Euclid's algorithm should be approximately three times more than that for Stein's algorithm.



Note: The bigger the difference between the two values, the more efficient Stein's algorithm becomes compared to Euclid's. If you have time, try experimenting with different values.

15. Close the GreatestCommonDivisor application.
16. Open the Test View window and refresh the display if the unit test is not listed.
17. Run the **FindGCDEuclidTest** and **FindGCDSteinTest** methods and verify that the tests ran successfully.

Exercise 4: Displaying Results Graphically

In this exercise, you will add a method to the application that displays the results graphically by using a bar graph. The parameters to the method are the two times taken, the orientation of the graph, and the colors to use to display the bars. The graph orientation and color parameters will be optional parameters. The default values will generate a vertical bar graph with a red bar for the first value and a blue bar for the second.

Scenario

You want to display the results of the timing comparisons graphically by using a simple, customizable bar graph.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Display the algorithm timings graphically.
3. Modify the **DrawGraph** method.
4. Modify the code that calls the **DrawGraph** method.

► **Task 1: Open the starter project**

- Open the Charting solution in the E:\Labfiles\Lab 3\Ex4\Starter folder.
This solution contains a completed copy of the code from Exercise 3.

► **Task 2: Display the algorithm timings graphically**

1. Open the MainWindow.xaml.cs file.
2. In the **FindGCD_Click** method, locate the Call **DrawGraph** comment, and add a call to the **DrawGraph** method, using the **timeEuclid** and **timeStein** variables as parameters.
3. Build the solution and correct any errors.
4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The result of **4** should be displayed. The time reported for both algorithms should be represented by a simple bar graph in the window.
6. Close the GreatestCommonDivisor application.

► **Task 3: Modify the DrawGraph method**

1. In the **MainWindow** class, locate the **DrawGraph** method and add the following three optional parameters:
 - a. A parameter called **orientation** of type **Orientation** with a default value of **Orientation.Horizontal**.
 - b. A parameter called **colorEuclid** of type **string** with a default value of **"Red"**.
 - c. A parameter called **colorStein** of type **string** with a default value of **"Blue"**.
2. In the **DrawGraph** method, locate the **Use optional orientation parameter** comment, and remove the existing declaration of the orientation variable.
3. Locate the **Use optional color parameters** comment, and modify the assignment of the **bEuclid** and **bStein** variables to use the optional parameters in the method signature. To do this, you will need to use the **BrushConverter** class and the **ConvertFromString** instance method as shown in the following code example.

```

...
private void DrawGraph(long euclidTime, long steinTime,
    Orientation orientation = Orientation.Horizontal,
    string colorEuclid = "Red",
    string colorStein = "Blue")
{
    ...
    BrushConverter bc = new BrushConverter();
    Brush bEuclid = (Brush)bc.ConvertFromString(colorEuclid);
    Brush bStein = (Brush)bc.ConvertFromString(colorStein);
    ...
}
...

```

4. Build the solution and correct any errors.
5. Run the GreatestCommonDivisor application.
6. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484** and then click **Find GCD (2 Integers)**. The graph should be displayed as before, except the **DrawGraph** method call is now using the default parameter values, and the graph is displayed as a pair of red and blue vertical bars.
7. Close the GreatestCommonDivisor application.

► Task 4: Modify the code that calls the DrawGraph method

1. Open the MainWindow.xaml.cs file.
2. In the **FindGCD_Click** method, locate the **Modify the call to Drawgraph to use the optional parameters** comment, and modify the **DrawGraph** method call to use the *orientation*, *colorEuclid*, and *colorStein* optional parameters as follows:
 - a. *orientation*—set to the selected value of the **chartOrientation** list box.
 - b. *colorEuclid*—set to the selected item of the **euclidColor** list box.
 - c. *colorStein*—set to the selected item of the **steinColor** list box.

These list boxes are already included in the user interface; they appear in the lower part of the window. The user can select the values in these list boxes to change the appearance of the graph that is displayed.

3. Build the solution and correct any errors.

4. Run the GreatestCommonDivisor application.
5. In the GreatestCommonDivisor application, in the MainWindow window, in the first two boxes, type the values **298467352** and **569484**
6. In the **Euclid** list box, select **Green**, in the **Stein** list box, select **Black**, in the **Orientation** box, select **Horizontal**, and then click **Find GCD (2 Integers)**. The graph should be displayed with the specified colors and direction.
7. Close the GreatestCommonDivisor application.

Exercise 5: Solving Simultaneous Equations (optional)

In this exercise, you will write a method that solves simultaneous linear equations with four variables (w , x , y , and z). You will use a WPF application to obtain input from the user (the coefficients of w , x , y , and z and the result for four equations) to simulate the data captured by a device, and call the method. The method will use Gaussian Elimination (a well-known algorithm for solving simultaneous linear equations) to generate solutions for w , x , y , and z , which will be returned as an array. The WPF application will then display these values.

Scenario

A key requirement of one of the engineering applications produced by Fabrikam, Inc. is the ability to solve simultaneous linear equations based on some of the data captured by various measuring devices.

Suppose you need to find the values of x , y , and z given the equations in the following code example.

$2x + y - z = 8$	(equation E1)
$-3x - y + 2z = -11$	(equation E2)
$-2x + y + 2z = -3$	(equation E3)

The method to solve these equations, known as Gaussian Elimination, proceeds as follows:

1. Eliminate x from equations E2 and E3:
 - To eliminate x from E2, calculate $(3 \div 2) \times E1 + E2$
The coefficient of x in E2 is $-(3 \div 2)$ times that of the coefficient of x in E1, so multiplying E1 by $(3 \div 2)$ and adding E2 removes x from E2.
 - To remove x from E3, calculate $E1 + E3$

The coefficient of x in $E3$ is -1 times that of the coefficient of x in $E1$, so adding $E1$ to $E3$ removes x from $E3$.

The result is shown in the following code example.

$$\begin{array}{ll} 2x + y - z = 8 & (E1) \\ (1/2)y + (1/2)z = 1 & (E2) \\ 2y + z = 5 & (E3) \end{array}$$

2. Next, eliminate y from $E3$:

- To eliminate y from $E3$, calculate $-4 \times E2 + E3$

The coefficient of y in $E3$ is four times that of the coefficient of y in $E2$, so multiplying $E2$ by -4 and adding $E3$ removes y from $E3$.

The result is shown in the following code example.

$$\begin{array}{ll} 2x + y - z = 8 & (E1) \\ (1/2)y + (1/2)z = 1 & (E2) \\ -z = 1 & (E3) \end{array}$$

The equations are now in triangular form—three unknowns in the first equation, two in the second equation, and one in the third equation.

3. Solve $E3$ and calculate the value for z , as the following code example shows.

$$z = -1 \quad (E3)$$

4. Substitute the value of z into $E2$ to calculate the value of y , as the following code example shows.

$$\begin{array}{ll} (1/2)y - 1/2 = 1 & \Rightarrow \\ (1/2)y = 3/2 & \Rightarrow \\ y = 3 & (E2) \end{array}$$

5. Substitute the values of z and y into $E1$ to calculate the value of x , as the following code example shows.

$$\begin{array}{ll} 2x + 3 + 1 = 8 & \Rightarrow \\ 2x = 4 & \Rightarrow \\ x = 2 & (E1) \end{array}$$

This process is known as back substitution.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create methods to copy arrays.
3. Convert the equations to triangular form.
4. Perform back substitution.
5. Test the solution.

► **Task 1: Open the starter project**

1. Open the SimultaneousEquations solution in the E:\Labfiles\Lab 3\Ex5\Starter folder.
2. Open the MainWindow.xaml file.

This is a different application from the one that the previous exercises have used. It is a WPF application that enables a user to enter the coefficients for four simultaneous equations that contain four variables (w, x, y, and z), and then uses Gaussian Elimination to find a solution for these equations. The results are displayed in the lower part of the screen.

► **Task 2: Create methods to copy arrays**

1. Open the Gauss.cs file.

This file contains a class called **Gauss** that provides a method called **SolveGaussian**. This method takes two arrays as parameters:

- A two-dimensional array of **double** values containing the coefficients for the variables w, x, y, and z specified by the user for each equation.
- An array of **double** values containing the result of each equation specified by the user (the value to the right of the equal sign).

The method returns an array of **double** values that will be populated with the values of w, x, y, and z that provide the solutions to these equations.

You will implement the body of this method in this exercise.

2. In the **Gauss** class, locate the **TODO Exercise 5, Task 2** comment. Remove this comment and declare a **private static** method called **DeepCopy1D**. The method should accept and return a **double** array.

The **SolveGaussian** method will make a copy of the arrays passed in as parameters to avoid changing the original data that the user provided.

3. In the **DeepCopy1D** method, add code to create a deep copy of the one-dimensional array that was passed into the method. Your code should perform the following tasks:
 - a. Create and initialize an array with the same number of columns as the array that was passed in.
 - b. Copy the values in the array that was passed as a parameter into the new array.
 - c. Return the new array.
4. In the **Gauss** class, declare another **private static** method called **DeepCopy2D**. The method should accept and return a two-dimensional **double** array.
5. In the **DeepCopy2D** method, add code to create a deep copy of the two-dimensional array that was passed into the method. Your code should do the following:
 - a. Create and initialize an array with the same number of columns and rows as the array that was passed in.
 - b. Copy the values in the array that was passed in as the parameter into the new array.
 - c. Return the new array.

► Task 3: Convert the equations to triangular form

1. In the **SolveGaussian** method, use the **DeepCopy1D** and **DeepCopy2D** methods to create deep copies of the **rhs** and **coefficients** arrays.
2. Locate the **Convert the equation to triangular form** comment, and add code to convert the equations represented by the copies of the **coefficients** and **rhs** arrays into triangular form.



Note: The **Gauss** class defines a constant integer called **numberOfEquations** that specifies the number of coefficients that the application can resolve.

► **Task 4: Perform back substitution**

- In the **Gauss** class, in the **SolveGaussian** method, locate the **Perform the back substitution and return the result** comment, and then add code to perform back substitution. To do this, you will need to work back from the equation with one unknown and substituting the values calculated at each stage to solve the remaining equations.

► **Task 5: Test the solution**

1. Open the `MainWindow.xaml.cs` file.
2. In the **MainWindow** class, locate the **TODO Exercise 5, Step 5** comment, and add code to call the **SolveGaussian** method. Use the coefficients and rhs variables as parameters and set the **answers** array to the result.
3. Run the GaussianElimination application.
4. In the GaussianElimination application, in the MainWindow window, enter the following equations, and then click **Solve**.



Note: Enter a value of zero in the corresponding text if no value is specified for w, x, y, or z in the equations below.

$$2w + x - y + z = 8$$

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

$$-2w + x - 2y = -3$$

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

Verify that the following results are displayed:

$$w = 4$$

$$x = -17$$

$$y = -11$$

$$z = 6$$

5. Experiment with other equations. Note that not all systems of equations have a solution. How does your code handle this situation?
6. Close the MainWindow window.
7. Close Visual Studio.

Module 4

Lab Instructions: Handling Exceptions

Contents:

Exercise 1: Making a Method Fail-Safe	4
Exercise 2: Detecting an Exceptional Condition	8
Exercise 3: Checking for Numeric Overflow	12

Lab: Handling Exceptions

- Exercise 1: Making a Method Fail-Safe
- Exercise 2: Detecting an Exceptional Condition
- Exercise 3: Checking for Numeric Overflow

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

- Add code to make a method fail-safe.
- Add code to a method to detect a condition and throw an exception if that condition is met.
- Add code to use the **checked** keyword to test for numeric overflow.

Introduction

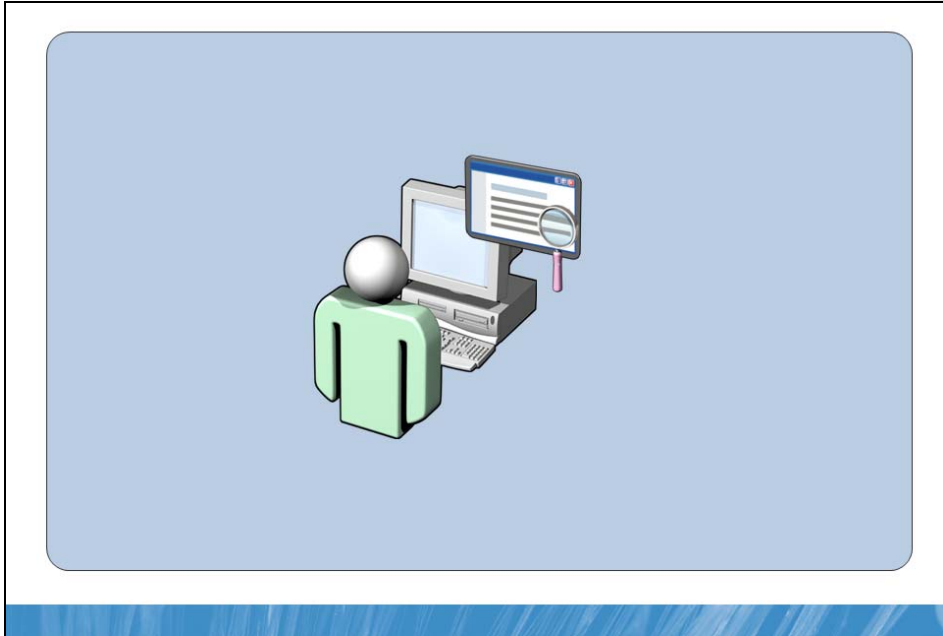
In this lab, you will catch and handle the possible exceptions that can occur in a method. You will also use the **finally** construct to implement code that runs even if an exception occurs. You will also add code that throws an exception if an error condition is detected in a method. Finally, you will enable integer overflow checking in an application.

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**

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Exception handling and resource management are a critical part of all of the applications that Fabrikam, Inc. develops. Failure to handle exceptions correctly in software that drives a large piece of machinery could result in life-threatening situations. Even in smaller, less critical scientific devices, an unhandled exception could result in lost data and the need to repeat experiments.

Exercise 1: Making a Method Fail-Safe

In this exercise, you will add fail-safe functionality to an application to ensure that it continues to function even if one or more exceptions occur. The code itself is located in a Windows Presentation Foundation (WPF) application that acts as a test harness.

Scenario

Fabrikam, Inc. provides intelligent switching devices that can monitor the environment for a critical condition (such as the temperature exceeding a specified

value), and trigger a shutdown operation. These switching devices are used in applications in the energy industry to initiate the shutdown of nuclear reactors. Needless to say, the correct operation of these devices is essential. Fabrikam, Inc. is developing a new model of switching device, and requires you to write part of the software that controls its operation. You have been provided with the code that performs the shutdown operation. This code contains a number of steps, and they must all be run. If any step fails, the code must report the failure, but continue with the next step.

The main tasks for this exercise are as follows:

1. Open the Failsafe solution and run the application.
2. Examine the **Switch** class.
3. Handle the exceptions that the **Switch** class throws.
4. Test the application.

► Task 1: Open the Failsafe solution and run the application

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$w0rd**.
2. Open Visual Studio 2010.
3. Open the Failsafe solution in the E:\Labfiles\Lab 4\Ex1\Starter folder.
4. Run the Failsafe project and repeatedly click **Shutdown** until an exception occurs.



Note: The **Switch** class is designed to randomly throw an exception, so you may not encounter an exception the first time that you click the button. Repeatedly click the **Shutdown** button until an exception occurs.

► Task 2: Examine the Switch class

1. If it is not already open, open the Switch.cs file in Visual Studio.
2. Examine the **Switch** class.

Note that the class contains several methods, each of which is capable of throwing at least one exception, dependent on the outcome of a random number generation. Toward the bottom of the file, note the definitions of each

of the custom exceptions that the **Switch** class can throw. These are very basic exception classes that simply encapsulate an error message.

► Task 3: Handle the exceptions that the Switch class throws

The SwitchTestHarness project contains a reference to the **SwitchDevice** class, and invokes each method in the **Switch** class to simulate polling multiple sensors and diagnostic devices. Currently, the project contains no exception handling, so when an exception occurs, the application will fail. You must add exception-handling code to the SwitchTestHarness project, to protect the application from exceptions that the **Switch** class throws.

1. Open the MainWindow.xaml.cs file in Visual Studio.
2. In the **MainWindow** class, locate the **Button1_Click** method. This method runs when the user clicks the **Shutdown** button.
3. Remove the comment **TODO - Add exception handling**, and then locate the **Step 1 - disconnect from the Power Generator** and **Step 2 - Verify the status of the Primary Coolant System** comments. Enclose the code between these comments in a **try/catch** block that catches the **SwitchDevices.PowerGeneratorCommsException** exception. This is the exception that the **DisconnectPowerGenerator** method can throw.
4. In the **catch** block, add code to append a new line of text to the **textBlock1** control with the message "*** Exception in step 1:" and then the contents of the **Message** property of the exception. The **Message** property contains the error message that the **Switch** object specified when it threw the exception.



Hint: To append a line of text to a **TextBlock** control, use the **+=** operator on the **Text** property of the control.

5. Enclose the code between the **Step 2 - Verify the status of the Primary Coolant System** and **Step 3 - Verify the status of the Backup Coolant System** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, following the same pattern as step 3, print a message on a new line in the **textBlock1** control (note that this is step 2, not step 1 of the shutdown process).
6. Enclose the code between the **Step 3 - Verify the status of the Backup Coolant System** and **Step 4 - Record the core temperature prior to shutting**

down the reactor comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 3).

7. Enclose the code between the **Step 4 - Record the core temperature prior to shutting down the reactor** and **Step 5 - Insert the control rods into the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 4).
8. Enclose the code between the **Step 5 - Insert the control rods into the reactor** and **Step 6 - Record the core temperature after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.RodClusterReleaseException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 5).
9. Enclose the code between the **Step 6 - Record the core temperature after shutting down the reactor** and **Step 7 - Record the core radiation levels after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 6).
10. Enclose the code between the **Step 7 - Record the core radiation levels after shutting down the reactor** and **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.CoreRadiationLevelReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 7).
11. Enclose the two statements after **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.SignallingException** exception. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 8).
12. Build the solution and correct any errors.

► Task 4: Test the application

- Run the application, and then click the **Shutdown** button. Examine the messages displayed in the MainWindow window, and verify that exceptions are now caught and reported.



Note: The **Switch** class randomly generates exceptions as before, so you may not see any exception messages the first time that you click the button. Repeat the process of clicking the button and examining the output until you see exception messages appear.

Exercise 2: Detecting an Exceptional Condition

In this exercise, you will modify a method so that it throws an **ArgumentException** exception if it is invoked with arguments that contain erroneous or invalid data.

Scenario

One of the engineering devices that Fabrikam, Inc. produces performs several calculations that involve matrices. These matrices represent the coordinates of sets of points within the bounds of a multidimensional mesh. The device itself collects the data for these points and constructs the matrices. Then, it uses a C# method to multiply them together to generate a new set of data points. Under normal operations, none of the data items in any of the matrices should be negative. However, sometimes the data that the device captures contains an error—if the device detects a value that is out of range, it generates the value -1 for a data point. Unfortunately, the code that multiplies matrices together fails to detect this condition, and calculates a result that is erroneous. You have been provided with a copy of this code as a method that is embedded in a WPF application.

The main tasks for this exercise are as follows:

1. Open the MatrixMultiplication solution.
2. Add code to throw exceptions in the **MatrixMultiply** method.
3. Handle the exceptions that the **MatrixMultiply** method throws.
4. Implement test cases and test the application.

► **Task 1: Open the MatrixMultiplication solution**

1. In Visual Studio, open the MatrixMultiplication solution in the E:\Labfiles\Lab 4\Ex2\Starter folder.
2. Open the Matrix.cs file, and then locate the **MatrixMultiply** method.

The **MatrixMultiply** method performs the arithmetic to multiply together the two matrices passed as parameters and return the result.

Currently, the method accepts matrices of any size, and performs no validation of data in the matrices before calculating the results. You will add checks to ensure that the two matrices are compatible (the number of columns in the first matrix is equal to the number of rows in the second matrix), and that no value in either matrix is a negative number.

If the matrices are not compatible, or either of them contain a negative value, the method must throw an exception.

► **Task 2: Add code to throw exceptions in the MatrixMultiply method**

1. In the **MatrixMultiply** method, locate and remove the comment **TODO – Evaluate input matrices for compatibility**. Below the comment block, add code to perform the following actions:
 - a. Compare the number of columns in **matrix1** to the number of rows in **matrix2**.
 - b. Throw an **ArgumentException** exception if the values are not equal. The exception message should specify that the number of columns and rows should match.



Hint: You can obtain the number of columns in a matrix by examining the length of the first dimension. You can obtain the number of rows in a matrix by examining the length of the second dimension.

2. Locate and remove the comment **TODO – Evaluate matrix data points for invalid data**. At this point, the method iterates through the data points in each matrix, multiplying the value in each cell in **matrix1** against the value in the corresponding cell in **matrix2**. Add code below the comment block to perform the following actions:

- a. Check that the value in the current column and row of **matrix1** is greater than zero. The cell and row variables contain the column and row that you should examine.
- b. Throw an **ArgumentException** exception if the value is not greater than zero. The exception should contain the message "Matrix1 contains an invalid entry in cell[x, y]." where x and y are the column and row values of the cell.



Hint: Use the **String.Format** method to construct the exception message.

3. Add another block of code to check that the value in the current column and row of **matrix2** is greater than zero. If it is not, throw an **ArgumentException** exception with the message "Matrix2 contains an invalid entry in cell[x, y]". The column and cell variables contain the column and row that you should examine.

► Task 3: Handle the exceptions that the **MatrixMultiply** method throws

1. Open the MainWindow WPF window in the Design View window and examine the window.

This window provides the user interface that enables the user to enter the data for the two matrices to be multiplied. The user clicks the **Calculate** button to calculate and display the result.
2. Open the code file for the MainWindow WPF window.
3. In the **MainWindow** class, locate the **ButtonCalculate_Click** method. This method runs when the user clicks the **Calculate** button.
4. In the **ButtonCalculate_Click** method, locate the line of code that invokes the **Matrix.MatrixMultiply** method, and enclose this line of code in a **try/catch** block that catches an **ArgumentException** exception named **ex**.
5. In the **catch** block, add a statement that displays a message box that contains the contents of the **Message** property of the exception object.



Hint: You can use the **MessageBox.Show** method to display a message box. Specify the message to display as a string passed in as a parameter to this method.

6. Build the solution and correct any errors.

7. Start the application without debugging.
8. In the MainWindow window, in the first drop-down list box, select **Matrix 1: 2 Columns**, in the second drop-down list box, select **Matrix 1: 2 Rows**, and then in the third drop-down list box, select **Matrix 2: 2 Columns**.

This creates a pair of 2×2 matrices initialized with zeroes.

9. Enter some non-negative values in the cells in both matrices, and then click **Calculate**.

Verify that the result is calculated and displayed, and that no exceptions occur.

10. Enter one or more negative values in the cells in either matrix, and then click **Calculate** again.

Verify that the appropriate exception message is displayed, and that it identifies the matrix and cell that is in error.

11. Close the MainWindow window and return to Visual Studio.

The application throws and catches exceptions, so you need to test that the application functions as expected. Although you can test for negative data points by using the application interface, the user interface does not let you create arrays of different dimensions. Therefore, you have been provided with unit test cases that will invoke the **MatrixMultiply** method with data that will cause exceptions. These tests have already been created; you will just run them to verify that your code works as expected.

► Task 4: Implement test cases and test the application

1. In the Matrix Unit Test Project, open the **MatrixTest** class, and then examine the **MatrixMultiplyTest1** method.

The **MatrixMultiplyTest1** method creates four matrices: **matrix1**, **matrix2**, **expected**, and **actual**. The **matrix1** and **matrix2** matrices are the input matrices that are passed to the **MatrixMultiply** method during the test. The **expected** matrix contains the expected result of the matrix multiplication, and the **actual** matrix stores the result of the **MatrixMultiply** method call. The method invokes the **MatrixMultiply** method before using a series of **Assert** statements to verify that the **expected** and **actual** matrices are identical.

This test method is complete and requires no further work.

2. Examine the **MatrixMultiplyTest2** method.

This method creates two compatible matrices, but **matrix2** contains a negative value. This should cause the **MatrixMultiply** method to throw an exception.

The **MatrixMultiplyTest2** method is prefixed with the **ExpectedException** attribute, indicating that the test method expects to cause an **ArgumentException** exception. If the test does not cause this exception, it will fail.

3. Examine the **MatrixMultiplyTest3** method.

This method creates two incompatible matrices and passes them to the **MatrixMultiply** method, which should throw an **ArgumentException** exception as a result. Again, the method is prefixed with the **ExpectedException** attribute, indicating that the test will fail if this exception is not thrown.

4. Run all tests in the solution, and verify that all tests execute correctly.

Exercise 3: Checking for Numeric Overflow

In this exercise, you will examine what happens by default if an integer calculation causes numeric overflow. You will then modify the application to check for numeric overflow exceptions and repeat the calculation.

Scenario

Part of the software for a measuring device performs integer multiplication, but the integer values used can be very large. You want to ensure that the software does not generate errors that are caused by numeric overflow.

The main tasks for this exercise are as follows:

1. Open the IntegerOverflow solution.
2. Add a **checked** block.
3. Test the application.

► Task 1: Open the IntegerOverflow solution

1. Open the IntegerOverflow solution in the E:\Labfiles\Lab 4\Ex3\Starter folder.
2. Run the application, and then click **Multiply**. Observe the result that is displayed and note that it is incorrect.

The application multiplies 2147483647 by 2, and displays the result -2. This is because the multiplication causes an integer numeric overflow. By default, overflow errors of this nature do not cause an exception. However, in many

situations, it is better to catch the overflow error than to let an application proceed with incorrect data.

3. In Visual Studio, on the **Debug** menu, click **Stop Debugging**.

► Task 2: Add a checked block

1. In Solution Explorer, open the MainWindow.xaml.cs file.
2. Locate the **DoMultiply_Click** method
This method runs when the user clicks the **Multiply** button.
3. Remove the **TODO - Place the multiplication in a checked block** comment. Add a **try/catch** block around the line of code that performs the multiplication operation, and then catch the **OverflowException** exception.
4. Inside the **try** block, add a **checked** block around the line of code that performs the multiplication arithmetic.
5. Build the solution and correct any errors.

► Task 3: Test the application

1. Start the application.
2. Click **Multiply**. Verify that the application now displays a message informing you that the arithmetic operation resulted in an overflow.
3. Click **OK**, close the MainWindow window, and then return to Visual Studio.
4. Close Visual Studio.

Module 5

Lab Instructions: Reading and Writing Files

Contents:

Exercise 1: Building a Simple File Editor

4

Exercise 2: Making the Editor XML Aware

10

Lab: Reading and Writing Files

- Exercise 1: Building a Simple File Editor
- Exercise 2: Making the Editor XML Aware

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 45 minutes

Objectives

After completing this lab, you will be able to:

- Read and write data by using the **File** class.
- Read and write data by using a **FileStream** class.

Introduction

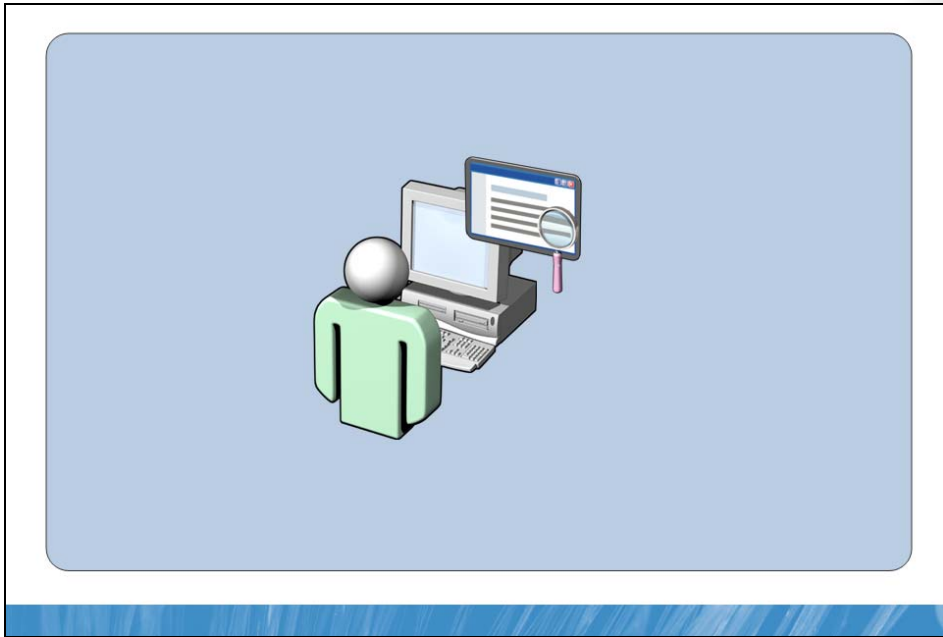
In this lab, you will use the **File** class in the **System.IO** namespace to read and write data to a file on the file system. You will then use a stream class to process this file.

Lab Setup

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

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

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

Many of the robotic devices that Fabrikam, Inc. builds are controlled by using instructions that are held in a text file that is stored on the device. You have been asked to write a simple application that a user can use to open, display, and edit one of these text files (the device will not have Notepad installed). The application will run on the device, and make use of a small screen and keypad that is built into the device. The application must be easy to use, and include full exception handling.

Exercise 1: Building a Simple File Editor

In this exercise, you will add functionality to a simple WPF application that can be used to edit text files. The WPF application expects the user to enter the name and path of a text file by using the **Open File** common dialog box. The application will then open this file and display its contents in a text box on the WPF form. The user can edit this text, and then save the amended text back to the file.

The user interface for this application has already been completed, but you will implement the logic to enable the user to specify the file to edit, and to load and save the file.

The main tasks for this exercise are as follows:

1. Open the SimpleEditor project.
2. Display a dialog box to accept a file name from the user.
3. Implement a new class to read and write text to a file.
4. Update the MainWindow event handlers to consume the **TextFileOperations** class.
5. Implement test cases.

► **Task 1: Open the SimpleEditor project**

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$w0rd**.
2. Open Microsoft Visual Studio® 2010.
3. Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex1\Starter folder.

► **Task 2: Display a dialog box to accept a file name from the user**

1. Display the MainWindow.xaml window.

The MainWindow window implements a very simple text editor. The main part of the window contains a text box that a user can use to display and edit text. The **Open** button enables the user to open a file, and the **Save** button enables the user to save the changes to the text back to a file. You will add the code that implements the logic for these two buttons.

2. Review the task list.
3. Locate the task **TODO - Implement a method to get the file name**. Double-click this task.

This task is located in the MainWindow.xaml.cs class file.

4. Delete the comment, and then define a new private method named **GetFileName** that accepts no parameters and returns a string value that holds the file name that the user specified.

5. In the method body, declare a new string member named **fname**, and then initialize it with the **String.Empty** value.
6. At the end of the collection of **using** statements at the top of the file, add a statement to bring the **Microsoft.Win32** namespace into scope.
7. In the **GetFileName** method, after the statement that declares the fname variable, add code to the method to perform the following actions:
 - a. Create a new instance of the **OpenFileDialog** dialog box, named **openFileDialog**.
 - b. Set the **InitialDirectory** property of **openFileDialog** to point to the `E:\Labfiles\Lab 5\Ex1\Starter` folder.



Note: When including file paths in code, you should prefix the string with the @ symbol. This symbol instructs the C# compiler to treat any backslash characters as literals rather than escape characters.

- c. Set the **DefaultExt** property of **openFileDialog** to `".txt"`;
 - d. Set the **Filter** property of **openFileDialog** to `"Text Documents (*.txt)|*.txt"`.
8. Add code to perform the following tasks:
 - a. Call the **ShowDialog** method of **openFileDialog**, 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 **openFileDialog** to the **fname** variable.
9. At the end of the method, return the value in the **fname** variable.

► Task 3: Implement a new class to read and write text to a file

1. Add a new class named **TextFileOperations** to the FileEditor project.

You will use this class to wrap some common file operations. This scheme enables you to change the way in which files are read from or written to without affecting the rest of the application.

2. At the top of the class file, add a statement to bring the **System.IO** namespace into scope.
3. In the **TextFileOperations** class, add a public static method named **ReadTextFileContents**. The method should accept a string parameter named *fileName*, and return a string object.
4. In the **ReadTextFileContents** method, add code to return the entire contents of the text file whose path is specified in the *fileName* parameter.



Hint: Use the static **ReadAllText** method of the **File** class.

5. Below the **ReadTextFileContents** method, add a public static method named **WriteTextFileContents**. The method should not return a value type, and should accept the following parameters:
 - a. A string parameter named *fileName*.
 - b. A string parameter named *text*.
6. In the **WriteTextFileContents** method, add code to write the text that is contained in the *text* parameter to the file that is specified in the *fileName* parameter.



Hint: Use the static **WriteAllText** method of the **File** class.

7. Build the solution and correct any errors.

► Task 4: Update the MainWindow event handlers to consume the TextFileOperations class

1. In the task list, locate the task **TODO - Update the OpenButton_Click method**. Double-click this task.

This task is located in the **OpenButton_Click** method of the **MainWindow** class.
2. Remove the comment, and then add code to perform the following tasks:
 - a. Invoke the **GetFileName** method. Store the result of the method in the **fileName** member.

- b. If **fileName** is not an empty string, call the static **ReadTextFileContents** method of the **TextFileOperations** class, and then pass *fileName* as the parameter. Store the result in the **Text** property of the editor **TextBox** control in the WPF window.
3. In the task list, locate the task **TODO - Update the SaveButton_Click method**. Double-click this task.

This task is located in the **SaveButton_Click** method of the **MainWindow** class.
4. In the **SaveButton_Click** method, remove the comment, and then add code to perform the following tasks:
 - a. Check that the **fileName** member is not an empty string.
 - b. If **fileName** is not an empty string, call the static **WriteTextFileContents** method of the **TextFileOperations** class. Pass **fileName** and the **Text** property of the editor **TextBox** control as the parameters.
5. Build the solution and correct any errors.
6. Start the application without debugging.
7. In the MainWindow window, click **Open**.
8. In the **Open** dialog box, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder, click **Commands.txt**, and then click **Open**.
9. In the MainWindow window, verify that the text in the following code example is displayed in the editor **TextBox** control.

```
Move x, 10
Move y, 20
If x < y Add x, y
If x > y & x < 20 Sub x, y
Store 30
```

This is the text from the **Commands.txt** file.

10. Change the **Store 30** line to **Save 50**, and then click **Save**.
11. Close the MainWindow window.
12. Using Windows Explorer, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder.
13. Open the **Commands.txt** file by using Notepad.
14. In Notepad, verify that the last line of the file contains the text **Save 50**.

15. Close Notepad and return to Visual Studio.

► **Task 5: Implement test cases**

1. In the task list, locate the task **TODO - Complete Unit Tests**. Double-click this task.

This task is located in the **TextFileOperationsTest** class.

2. Remove the comment.
3. Examine the **ReadTextFileContentsTest1** method, and then uncomment the commented line.

This method creates three strings:

- a. The **fileName** string contains the path of a prewritten file that contains specific content.
- b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.
- c. The **actual** string is initialized by calling the **ReadTextFileContents** method that you just implemented.

The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

4. Examine the **WriteTextFileContentsTest1** method, and then uncomment the commented line.

This method creates two strings:

- a. The **fileName** string contains the path of a nonexistent file, which the method will create when run.
- b. The **text** string contains some text that the method will write to the file.

The method calls the **WriteTextFileContents** method, passing the **fileName** and **text** strings as parameters. This creates the file at the specified location, and writes to the file. The method then creates a further string, **expected**, by calling the **File.ReadAllText** method and reading the text from the written file. The method then checks that the text string and the expected string are the same, before deleting the file that was created during the test.

5. Run all tests in the solution, and verify that all tests execute correctly.

Exercise 2: Making the Editor XML Aware

The applications that control a robotic device read the instructions from the file and then encode them as an XML document before passing them to the instruction execution module on the device. For example, imagine that a text file contains the instructions in the following code example.

```
Move x, 10
Move y, 20
If x < y Add x, y
If x > y Sub x, y
Store 30
```

The control applications will wrap them in a pair of XML tags, as the following code example shows.

```
<ControlApplication>
  <Instructions Code = "
    Move x, 10
    Move y, 20
    If x < y Add x, y
    If x > y Sub x, y
    Store 30"
  />
</ControlApplication>
```

However, some of the data in these instructions can contain characters such as ">" and "<" that might be misinterpreted as XML tags rather than data.

In this exercise, you will modify the WPF application to look for data that contains XML tags in the text file as it is read in and encode this data as XML escape sequences before displaying it. For example, the "<" character will be replaced with "<", the ">" symbol will be replaced with ">", and so on. The WPF application will use a file stream to read the data.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Add a new method to filter XML characters to the **TextFileOperations** class.
3. Update the user interface to invoke the new method.
4. Implement test cases.

► **Task 1: Open the starter project**

- Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex2\Starter folder.
This project is a completed version of the SimpleEditor project from Exercise 1.

► **Task 2: Add a new method to filter XML characters to the `TextFileOperations` class**

1. Review the task list.
2. In the task list, locate the **TODO - Implement a new method in the `TextFileOperations` class** task. Double-click this task.
This task is located in the **`TextFileOperations`** class.
3. Remove the comment, and then add a new public static method named **`ReadAndFilterTextFileContents`**. The method should accept a string parameter named *fileName*, and return a string.
4. In the **`ReadAndFilterTextFileContents`** method, add the following local variables:
 - a. A **`StringBuilder`** object named **`fileContents`**, initialized to a new instance of the **`StringBuilder`** class.
 - b. An integer variable called **`charCode`**.
5. Add a statement that instantiates a **`StreamReader`** object, named **`fileReader`**, by using the *fileName* parameter.
6. Add a **`while`** statement that reads each character in the **`StreamReader`** object until the end of the file is reached.



Hint: Use the **`Read`** method of the **`StreamReader`** class to read the next character from a stream. This method returns `-1` if there is no more data.

7. In the **`while`** block, add a **`switch`** statement that evaluates the **`charCode`** variable.

In the **`switch`** statement, add **`case`** statements for each of the characters in the following table. In each statement, append the **`fileContent`** **`StringBuilder`** object with the alternative representation shown in the table.

charCode	Standard representation	Alternative representation
34	" (straight quotation mark)	"
38	& (ampersand)	&
39	' (apostrophe)	'
60	< (less than)	<
62	> (greater than)	>

8. Add a default **case** statement that appends the actual character read from the stream to the **fileContent StringBuilder** object.



Note: The **Read** method returns the value read from the file as an integer and stores it in the **charCode** variable. You must cast this variable to a character before you append it to the end of the **StringBuilder** object.

9. At the end of the method, return the contents of the **fileContent StringBuilder** object as a string.
10. Build the solution and correct any errors.

► Task 3: Update the user interface to invoke the new method

1. In the task list, locate the **TODO - Update the UI to use the new method** task. Double-click this task.

This task is located in the **OpenButton_Click** method of the **MainWindow.xaml.cs** class.

2. Delete the comment, and then modify the line of code that calls the **TextFileOperations.ReadTextFileContents** method to call the **TextFileOperations.ReadAndFilterTextFileContents** method instead. Pass the **fileName** field as the parameter, and then save the result in the **Text** property of the editor **TextBox** control.
3. Build the solution and correct any errors.
4. Start the application without debugging.
5. In the **MainWindow** window, click **Open**.

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 < y Add x, y
If x > y & x < 20 Sub x, y
Store 30
```

This is the text from the Commands.txt file. Notice that the **<**, **>**, and **&** characters have been replaced with the text **<**, **>**, and **&**.

8. Close the MainWindow window and return to Visual Studio.

► Task 4: Implement test cases

1. In the task list, locate the **TODO - Complete Unit Tests** task. Double-click this task.

This task is located in the **TextFileOperationsTest** class.

2. Examine the **ReadAndFilterTextFileContentsTest** method, and then uncomment the commented line.

This method creates three strings:

- a. The **filename** string contains the path of a prewritten file that contains specific content.
- b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.
- c. The **actual** string is initialized by calling the **ReadAndFilterTextFileContents** method that you just implemented.

The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

This method is complete, and requires no further work.

3. Run all tests in the solution, and verify that all tests execute correctly.

Module 6

Lab Instructions: Creating New Types

Contents:

Exercise 1: Using Enumerations to Specify Domains	4
Exercise 2: Using a Struct to Model a Simple Type	8
Exercise 3: Using a Class to Model a More Complex Type	10
Exercise 4: Using a Nullable Struct	19

Lab: Creating New Types

- Exercise 1: Using Enumerations to Specify Domains
- Exercise 2: Using a Struct to Model a Simple Type
- Exercise 3: Using a Class to Model a More Complex Type
- Exercise 4: Using a Nullable Struct

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

- Use enumerations to specify domains.
- Use a struct to model a simple type.
- Use a class to model a more complex type.
- Use a nullable struct.

Introduction

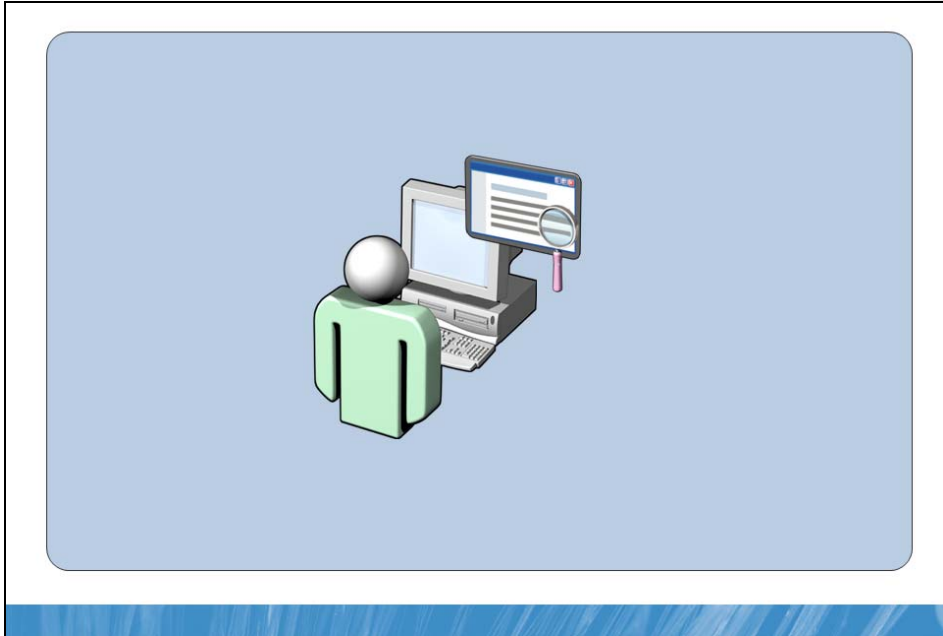
In this lab, you will define an enumeration and then use this type to create variables. You will also define a struct. Finally, you will define a class and use the struct as the type of a data member in the class.

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**

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

You are building an application that supports a machine that stress-tests girders for constructing high-rise buildings, bridges, and other critical structures.

Exercise 1: Using Enumerations to Specify Domains

In this exercise, you will define enumerations that represent different materials under stress (stainless steel, aluminum, reinforced concrete, and titanium) and the cross-section of the girders (I-Beam, Box, Z-Shaped, and C-Shaped). You will also define another enumeration called **TestResult** that represents the results of a stress test.

The main tasks for this exercise are as follows:

1. Open the Enumeration solution.
2. Add enumerations to the **StressTest** namespace.
3. Retrieve the enumeration values.

4. Display the selection results.
5. Test the solution.

► **Task 1: Open the Enumerations solution**

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$w0rd**.
2. Open Visual Studio 2010.
3. Open the Enumerations solution in the E:\Labfiles\Lab 6\Ex1\Starter folder.

► **Task 2: Add enumerations to the StressTest namespace**

1. Review the task list.
2. Locate the **TODO - Implement Material, CrossSection, and TestResult enumerations** task, and then double-click this task. This task is located in the StressTestType.cs file.
3. In the **StressTest** namespace, define a new enumeration named **Material**. The enumeration should have the following values:
 - a. **StainlessSteel**
 - b. **Aluminum**
 - c. **ReinforcedConcrete**
 - d. **Composite**
 - e. **Titanium**
4. Below the **Material** enumeration, define a new enumeration named **CrossSection**. The enumeration should have the following values:
 - a. **IBeam**
 - b. **Box**
 - c. **ZShaped**
 - d. **CShaped**
5. Below the **CrossSection** enumeration, define a new enumeration named **TestResult**. The enumeration should have the following values:
 - a. **Pass**

b. Fail

6. Build the solution and correct any errors.

► **Task 3: Retrieve the enumeration values**

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

The purpose of the TestHarness project is to enable you to display the values from each of the enumerations. When the application runs, the three lists are populated with the values that are defined for each of the enumerations. The user can select an item from each list, and the application will construct a string from the corresponding enumerations.

2. In the task list, locate the **TODO - Retrieve user selections from the UI** task, and then double-click this task. This task is located in the **MainWindow.xaml.cs** class.

3. Remove the comment, and add code to the **selectionChanged** method to perform the following tasks:

- a. Create a **Material** object called **selectedMaterial** and initialize it to the value of the **SelectedItem** property in the **materials** list box.
- b. Create a **CrossSection** object called **selectedCrossSection** and initialize it to the value of the **SelectedItem** property in the **crosssections** list box.
- c. Create a **TestResult** object called **selectedTestResult** and initialize it to the value of the **SelectedItem** property in the **testresults** list box.



Hint: The **SelectedItem** property of a **ListBox** control has the **object** type. You must cast this property to the appropriate type when you assign it to an enumeration variable.

► **Task 4: Display the selection results**

1. In the **selectionChanged** method, after the code that you added in the previous task, add a statement to create a new **StringBuilder** object named **selectionStringBuilder**.
2. Add a **switch** statement to evaluate the **selectedMaterial** variable. In the **switch** statement, add **case** statements for each potential value of the **Material** enumeration. In each **case** statement, add code to append the text "Material: *<selectedMaterial>*, " to the **selectionStringBuilder** object. Substitute the text

"<selectedMaterial>" in this string with the corresponding value for the selectedMaterial variable that is shown in the following table.

Material enumeration value	<selectedMaterial> string
Material.StainlessSteel	Stainless Steel
Material.Aluminum	Aluminum
Material.ReinforcedConcrete	Reinforced Concrete
Material.Composite	Composite
Material.Titanium	Titanium

3. Add another **switch** statement to evaluate the selectedCrossSection variable. In this **switch** statement, add **case** statements for each potential value of the **CrossSection** enumeration. In each **case** statement, add code to append the text "Cross-section: <selectedCrossSection>," to the **selectionStringBuilder** object. Substitute the text "<selectedCrossSection>" in this string with the corresponding value for the selectedCrossSection variable that is shown in the following table.

Material enumeration value	<selectedCrossSection> string
CrossSection.IBeam	I-Beam
CrossSection.Box	Box
CrossSection.ZShaped	Z-Shaped
CrossSection.CShaped	C-Shaped

4. Add a final **switch** statement to evaluate the **selectedTestResult** member. In the **switch** statement, add **case** statements for each potential value of the **TestResult** enumeration. In each **case** statement, add code to append the text "Result: <selectedTestResult>." to the **selectionStringBuilder** object. Substitute the text "<selectedTestResult>" in this string with the corresponding value for the selectedTestResult variable that is shown in the following table.

Material enumeration value	<selectedTestResult> string
TestResult.Pass	Pass

Material enumeration value	<selectedTestResult> string
TestResult.Fail	Fail

- At the end of the **selectionChanged** method, add code to display the string that is constructed by using the **selectionStringBuilder** object in the **Content** property of the **testDetails** label.

► Task 5: Test the solution

- Build the application and correct any errors.
- Run the application.
- 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.
- Experiment by selecting further values from all three lists, and verify that with each change, the label updates to reflect the changes.
- Close the application, and then return to Visual Studio.

Exercise 2: Using a Struct to Model a Simple Type

In this exercise, you will define a type called **TestCaseResult** that holds the result of a stress test. It will have the following public fields:

- Result : TestResult**
- ReasonForFailure: string**

This type is small, so it is best implemented as a struct. You will provide a constructor that initializes these fields.

The main tasks for this exercise are as follows:

- Open the Structures solution.
- Add the **TestCaseResult** structure.
- Add an array of **TestCaseResult** objects to the user interface project.
- Fill the **results** array with data.

5. Display the array contents.
6. Test the solution.

► **Task 1: Open the Structures solution**

- Open the Structures solution in the E:\Labfiles\Lab 6\Ex2\Starter folder.

► **Task 2: Add the TestCaseResult structure**

1. Review the task list:
2. In the task list, locate the **TODO - Declare a Structure** task, and then double-click this task. This task is located in the StressTestTypes.cs file.
3. Delete the comment, and then declare a new structure named **TestCaseResult**. In the **TestCaseResult** structure, add the following members:
 - a. A **TestResult** object named **Result**.
 - b. A **string** object named **ReasonForFailure**.

► **Task 3: Add an array of TestCaseResult objects to the user interface project**

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

This project simulates running stress tests and displays the results. It tracks the number of successful and failed tests, and for each failed test, it displays the reason for the failure.
2. In the task list, locate the **TODO - Declare a TestCaseResult array** task, and then double-click this task.
3. Remove the comment, and then declare a new array of **TestCaseResult** objects named **results**.

► **Task 4: Fill the results array with data**

1. In the **RunTests_Click** method, after the statement that clears the **reasonsList** list, add code to initialize the **results** array. Set the array length to **10**.
2. Below the statement that creates the array, add code that iterates through the items in the array and populates each one with the value that the static

GenerateResult method of the **TestManager** class returns. The **GenerateResult** method simulates running a stress test and returns a **TestCaseResult** object that contains the result of the test and the reason for any failure.

► **Task 5: Display the array contents**

- Locate the comment **TODO - Display the TestCaseResult data**. Delete the comment, and then add code that iterates through the **results** array. For each value in the array, perform the following tasks:
 - a. Evaluate the **result** value. If the **result** value is **TestResult.Pass**, increment the **passCount** value.
 - b. If the **result** value is **TestResult.Fail**, increment the **failCount** value, and add the **ReasonForFailure** string to the **reasonsList** list box that is displayed in the window.



Note: To add an item to a list box, you use the **ListBox.Items.Add** method and pass the item to add to the list as a parameter to the method.

► **Task 6: Test the solution**

1. Build the application and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Tests**.
Verify that the **Successes** and **Failures** messages are displayed. Also verify that a message appears in the **Failures** list if failures occur.
4. Click **Run Tests** again to simulate running another batch of tests and display the results of these tests.
5. Close the application, and then return to Visual Studio.

Exercise 3: Using a Class to Model a More Complex Type

In this exercise, you will define another type called **StressTestCase** that represents a stress test case for a girder. This type will be more complex than the

TestCaseResult struct and is best implemented as a class. The **StressTestCase** class will have the following public data members:

- **girderMaterial**: **MaterialType**
- **crossSection**: **CrossSection**
- **lengthInMm**: **int**
- **heightInMm**: **int**
- **widthInMm**: **int**
- **testCaseResult**: **TestCaseResult**

You will also define two constructors: a default constructor that initializes these fields (apart from **testCaseResult**) to default values and an overloaded constructor that enables a programmer to specify nondefault values. You will then add the following public methods to the class:

- **PerformStressTest**. This method will simulate performing a stress test and set the result to indicate whether the test passed or failed, together with a reason for failure.
- **GetStressTestResult**. This method will return the value of the **testCaseResult** field.
- **ToString**. This method will return a representation of the object as a string for display purposes.

The main tasks for this exercise are as follows:

1. Open the Classes solution.
2. Define the **StressTestCase** class.
3. Add a parameterized constructor and a default constructor to the class.
4. Add the **PerformStressTest** and **GetStressTestResult** methods to the class.
5. Override the **ToString** method to return a custom string representation
6. Create an array of **StressTestCase** objects.
7. Display the **StressTestCases** collection.
8. Test the solution.
9. Examine and run unit tests.

► **Task 1: Open the Classes solution**

- Open the Classes solution in the E:\Labfiles\Lab 6\Ex3\Starter folder.

► **Task 2: Define the StressTestCase class**

1. In the TestHarness project, display the MainWindow.xaml window.
This project is an extended version of the test harness from the previous two exercises. In addition to simulating stress-test results, it displays the details of the girder under test.
2. Review the task list.
3. In the task list, locate the **TODO - Add the StressTestCase class** task, and then double-click this task.
4. Remove the comment, and then add code to declare a public class named **StressTestCase** with the following public members:
 - a. A **Material** object named **GirderMaterial**.
 - b. A **CrossSection** object named **CrossSection**.
 - c. An integer named **LengthInMm**.
 - d. An integer named **HeightInMm**.
 - e. An integer named **WidthInMm**.
 - f. A **TestCaseResult** object named **TestCaseResult**.

► **Task 3: Add a parameterized constructor and a default constructor to the class**

1. Below the member declarations, add a constructor for the **StressTestCase** class that accepts the following parameters:
 - a. A **Material** object named **girderMaterial**.
 - b. A **CrossSection** object named **crossSection**.
 - c. An integer named **lengthInMm**.
 - d. An integer named **heightInMm**.
 - e. An integer named **widthInMm**.

In the constructor, add code to store the value for each parameter in the corresponding member.



Hint: In the constructor, to make it clear which items are member variables and which items are parameters, use the **this** keyword (which represents the current object) with all member variables.

2. Above the constructor, add a default constructor.



Hint: A default constructor is a constructor that accepts no parameters and implements functionality to create a default instance of a class.

In the default constructor, initialize the members of the **StressTestCase** object with default values by using the parameterized constructor and the data that are shown in the following table.

Parameter name	Value
<i>girderMaterial</i>	Material.StainlessSteel
<i>crossSection</i>	CrossSection.IBeam
<i>lengthInMm</i>	4000
<i>heightInMm</i>	20
<i>widthInMm</i>	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, ...)
{
    ...
}
```

► **Task 4: Add the `PerformStressTest` and `GetStressTestResult` methods to the class**

1. Below the class constructors, add code to declare a new method named **`PerformStressTest`**. The **`PerformStressTest`** method should take no parameters and should not return a value.

This method will simulate performing a stress test and then populate a **`StressTestCase`** object with the details of the test.

2. In the **`PerformStressTest`** method, create an array of strings called **`failureReasons`** that contains the following values:
 - a. **`"Fracture detected"`**
 - b. **`"Beam snapped"`**
 - c. **`"Beam dimensions wrong"`**
 - d. **`"Beam warped"`**
 - e. **`"Other"`**
3. Add a statement that invokes the **`Next`** method of the static **`Rand`** method of the **`Utility`** class. Pass the value **`10`** as a parameter.



Note: The **`Utility.Rand.Next`** method accepts an integer parameter and then returns a random integer value between zero and the value of the integer parameter. In this case, the method will return an integer between 0 and 9.

If the value that the **`Rand`** method returns is 9, add code to perform the following tasks:

- a. Set the **`TestCaseResult.Result`** member value to **`TestResult.Fail`**.
- b. Invoke the **`Utility.Rand.Next`** method with a parameter value of 5. Store the result in a new integer member named **`failureCode`**.
- c. Set the **`TestCaseResult.ReasonForFailure`** value to the value in the **`failureReasons`** array that the **`failureCode`** value indicates.



Note: This code simulates a 10 percent chance of a test case failing. The **`failureReasons`** array contains five possible causes of failure, and this code selects one of these causes at random.

4. If the **Rand** method returns a value other than 9, add code to set the **TestCaseResult.Result** member value to **TestResult.Pass**.
5. Below the **PerformStressTest** method, add a public method named **GetStressTestResult**, which accepts no parameters and returns a **TestCaseResult** object.
6. In the **GetStressTestResult** method, add code to return a reference to the **TestCaseResult** member.

► **Task 5: Override the ToString method to return a custom string representation**

1. Below the **GetStressTestResult** method, add the following public method named **ToString**.



Note: This overrides the **ToString** method that is inherited from the **object** type. You will see more about inheritance in a later module.

```
...
public class StressTestCase
{
    ...
    public override string ToString()
    {
    }
}
...
```

2. In the **ToString** method, add code to return a string with the format shown in the following code example, where each value in angle brackets is replaced with the corresponding member in the class.

```
Material: <girderMaterial>, CrossSection: <crossSection>, Length:
<lengthInMm>mm, Height: <heightInMm>mm, Width:<widthInMm>mm.
```



Hint: Use the **String.Format** method to build the string.

► **Task 6: Create an array of StressTestCase objects**

1. In the task list, locate the **TODO - Create an array of sample StressTestCase objects** task, and then double-click this task. This task is located in the **MainWindow.xaml.cs** class.
2. Remove the comment, and add a private method named **CreateTestCases**. The **CreateTestCases** method should accept no parameters and return an array of **StressTestCase** objects.
3. In the **CreateTestCases** method, add code to create an array of **StressTestCase** objects named **stressTestCases**. The array should be able to hold 10 objects.
4. Add code to generate 10 **StressTestCase** objects, and store each of them in the **stressTestCases** array. Use the following table to determine the parameters to pass to the constructor for each instance.

Array position	Material	CrossSection	Length	Height	Width
0	Use default constructor				
1	Material.Composite	CrossSection.CShaped	3500	100	20
2	Use default constructor				
3	Material.Aluminium	CrossSection.Box	3500	100	20
4	Use default constructor				
5	Material.Titanium	CrossSection.CShaped	3600	150	20
6	Material.Titanium	CrossSection.ZShaped	4000	80	20
7	Material.Titanium	CrossSection.Box	5000	90	20
8	Use default constructor				
9	Material.StainlessSteel	CrossSection.Box	3500	100	20

5. At the end of the method, return the **stressTestCases** array.

► **Task 7: Display the StressTestCases collection**

1. In the task list, locate the **TODO - Iterate through the StressTestCase samples displaying the results** task, and then double-click this task. This task is located in the **doTests_Click** method that runs when the user clicks **Run Stress Tests**.
2. Remove the comment, and then add code to invoke the **CreateTestCases** method. Store the result of the method call in a new array of **StressTestCase** objects named **stressTestCases**.
3. Add code to create a **StressTestCase** object named **currentTestCase** and a **TestCaseResult** object named **currentTestResult**. You will add code to instantiate these objects shortly.
4. Add code that iterates through the **StressTestCase** objects in the **stressTestCases** array. For each **StressTestCase** object, add code to perform the following tasks:
 - a. Set the **currentTestCase** object to refer to the **StressTestCase** object.
 - b. Invoke the **currentTestCase.PerformStressTest** method on the **currentTestCase** object.
 - c. Add the **currentTestCase** object to the **testList** list box that is displayed in the window.
 - d. Invoke the **currentTestCase.GetStressTestResult** method, and store the result in the **currentTestResult** object.
 - e. Add a string to the **resultList** list box that is displayed in the window. This string should consist of the **currentTestResult.Result** value and the **currentTestResult.ReasonForFailure** message.

► **Task 8: Test the solution**

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**.
Verify that the **Girder Tested** list contains a list of different girder compositions and the **Results** list contains a series of test results.
4. Click **Run Stress Tests** again. You should see a different set of results.
5. Close the application, and then return to Visual Studio

► **Task 9: Examine and run unit tests**

1. In the task list, locate the **TODO - Examine and Run Unit Tests** task, and then double-click this task. This task is located in the **StressTestCaseTest** class.
2. Examine the **StressTestCaseConstructorTest** method.

This method uses the parameterized constructor to create a new **StressTestCase** object the uses defined values. The method then uses a series of **Assert** statements to ensure that the properties of the created object match the values that are passed to the constructor.
3. Examine the **StressTestCaseConstructorTest1** method.

This method uses the default constructor to create a new **StressTestCase** object, passing no parameters. The method then uses a series of **Assert** statements to ensure that the properties of the created object match the intended default values.
4. Examine the **GetStressTestResultTest** method.

This method creates a new **StressTestCase** object and then retrieves a **TestCaseResult** object by calling the **StressTestCase.GetStressTestResult** method. The test method then uses **Assert** statements to ensure that the **TestCaseResult.Result** and **TestCaseResult.ReasonForFailure** properties contain the expected values.
5. Examine the **PerformStressTestTest** method.

This method creates a **StressTestCase** object, calls the **PerformStressTest** method, and then retrieves the **TestCaseResult** object. The method then checks that, if the test failed, the **TestCaseResult.ReasonForFailure** member contains some text. If the test passed, the method uses **Assert** statements to verify that the **ReasonForFailure** member contains no data. The method iterates 30 times.
6. Examine the **ToStringTest** method.

This method creates a default **StressTestCase** object, and then verifies that the object's **ToString** method returns a string that contains the correct details.
7. Run all of the tests in the solution, and verify that all of the tests execute successfully.

Exercise 4: Using a Nullable Struct

In this exercise, you will modify the constructor for the **StressTestCase** class to initialize the **testCaseResult** field to null (to indicate no result yet). However, you cannot set a value-type field or variable to a reference value such as null. Therefore, you will convert the **testCaseResult** field to a nullable field to support null values. You will then modify the methods in the **StressTestCase** class, and the test harness that invokes the **GetStressTestResult** method that displays the result of a test case, to dereference the value of this struct through the **Value** property.

The main tasks for this exercise are as follows:

1. Open the NullableStructs solution.
2. Modify the **TestCaseResult** field to make it nullable.
3. Modify the parameterized constructor to initialize the **TestCaseResult** member.
4. Modify the **PerformStressTest** method.
5. Modify the **GetStressTestResult** method.
6. Modify the **GetStressTestResult** method call.
7. Test the solution.
8. Update the unit tests.

► Task 1: Open the NullableStructs solution

- Open the NullableStructs solution in the E:\Labfiles\Lab 6\Ex4\Starter folder.

► Task 2: Modify the TestCaseResult field to make it nullable

1. Review the task list.
2. In the task list, locate the **TODO - Make TestCaseResult nullable** task, and then double-click this task. This task is located in the **StressTestTypes** class.
3. Remove the comment, and then modify the **TestCaseResult** member definition to allow it to store a null value.

► **Task 3: Modify the parameterized constructor to initialize the `TestCaseResult` member**

- In the `StressTestCase` parameterized constructor, remove the comment **TODO – Initialize `TestCaseResult` to null**, and then add code to initialize the `TestCaseResult` member to null.

► **Task 4: Modify the `PerformStressTest` method**

1. In the `PerformStressTest` method, remove the comment **TODO – Update the `PerformStressTest` method and work with the nullable type**, and then add code to declare a new `TestCaseResult` variable named `currentTestCase`.
2. Modify the `if` statement to perform the following tasks:
 - a. In all instances, modify the `currentTestCase` object rather than the `TestCaseResult` member.
 - b. At the end of the `if` block, assign the `currentTestCase` object to the `TestCaseResult` member.
3. Modify the `else` block to perform the following tasks:
 - a. Modify the `currentTestCase` object rather than the `TestCaseResult` member.
 - b. At the end of the `if` block, store the `currentTestCase` object in the `TestCaseResult` member.

► **Task 5: Modify the `GetStressTestResult` method**

- In the `GetStressTestResult` method, modify the method definition to return a nullable `TestCaseResult` value.

► **Task 6: Modify the `GetStressTestResult` method call**

1. In the task list, locate the **TODO - Modify call to `GetStressTestResult` method to handle nulls** task, and then double-click this task.
2. Remove the comment, and then modify the code to create a nullable `TestCaseResult` object named `currentTestResult`.
3. In the `for` block, after retrieving the value of the `currentTestResult` object from the `currentStressTest.GetStressTestResult` method, add code to check whether the `currentTestResult` object contains a value. If a value exists, add a

string that contains the **StressTestResult** **Result** and **ReasonForFailure** properties to the **resultList** list box.

► **Task 7: Test the solution**

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**.
Verify that the application functions in the same way as before.
4. Close the application, and then return to Visual Studio.

► **Task 8: Update the unit tests**

1. In the task list, locate the **TODO - Examine and run unit tests updated to deal with nullable type** task, and then double-click this task. This task is located in the **StressTestCaseTest** class.



Note: Most of the test cases are identical to those in Exercise 3. The only changes are in the **GetStressTestResult** and **PerformStressTestTest** methods.

2. Examine the **GetStressTestResult** method.

This method creates a new **StressTestCase** object. It then evaluates the **HasValue** property on the result of the **GetStressTestResult** method call to verify that property contains no value. The test then calls the **PerformStressTest** method, which generates a **TestCaseResult** value in the **StressTestCase** object. The test method again evaluates the **HasValue** property to verify that a value now exists.

3. Examine the changes to the **PerformStressTestTest** method.

This method creates a **StressTestCase** object and then calls the **PerformStressTest** method on that object. The method calls the **GetStressTestResult** method on the **StressTestCase** object and stores the result in a local nullable **TestCaseResult** object. The method then uses an **Assert** statement to evaluate the **HasValue** property of the **TestCaseResult** object to verify that the result is not null. The method then evaluates the **Value** property of the **TestCaseResult** object to determine whether the result indicates that the stress test failed or passed. If the stress test failed, an **Assert**

statement is used to verify that the **ReasonForFailure** string contains a value. If the stress test passed, an **Assert** statement is used to verify that the **ReasonForFailure** string is null. The method iterates 30 times.

4. Run all of the tests in the solution, and verify that all of the tests execute successfully.
5. Close Visual Studio.

Module 7

Lab Instructions: Encapsulating Data and Methods

Contents:

Exercise 1: Hiding Data Members	4
Exercise 2: Using Static Members to Share Data	7
Exercise 3: Implementing an Extension Method	11

Lab: Encapsulating Data and Methods

- Exercise 1: Hiding Data Members
- Exercise 2: Using Static Members to Share Data
- Exercise 3: Implementing an Extension Method

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

- Hide data members in a type by using access modifiers.
- Use static members to share data in types.
- Use extension methods to add functionality to the **System.Int64** struct.

Introduction

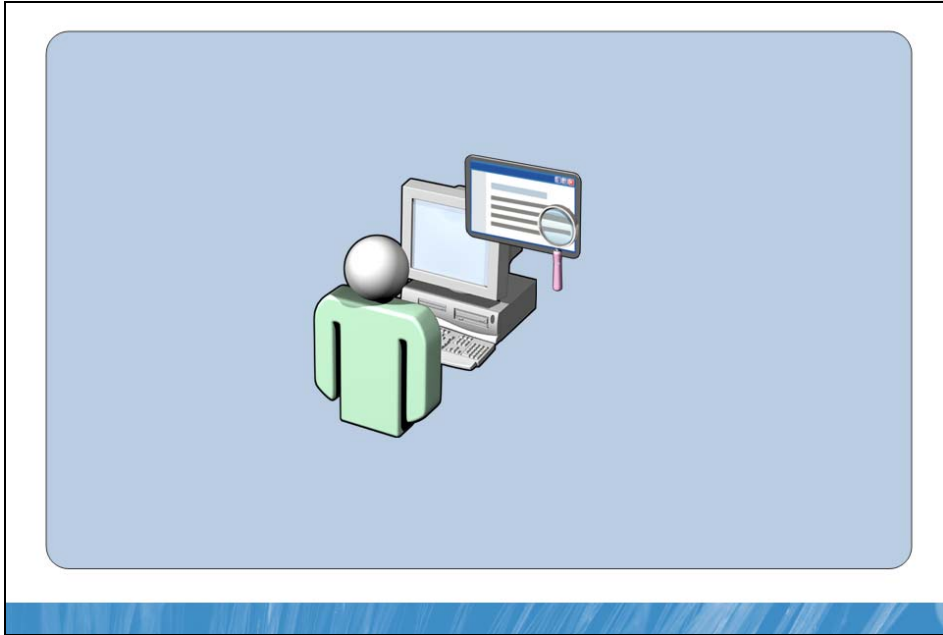
In this lab, you will use encapsulation to hide information in a class. You will add static members and methods to a type to share data between instances of the type. Finally, you will add an extension method to a built-in type in the .NET Framework.

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**

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data.

You are building an application that drives a machine that stress-tests girders for the construction of high-rise buildings, bridges, and other critical structures. You have defined types to support this application, but they currently expose all members publicly, which can cause problems. After they are created, the **girderMaterial**, **crossSection**, **lengthInMm**, **heightInMm**, and **widthInMm** members of a **StressTestCase** object should be immutable; this guarantees that the test case results that are reported in a test case object match the data for the test.

Exercise 1: Hiding Data Members

In this exercise, you will make the fields in the **StressTestCase** class private and verify that these fields are now inaccessible outside code in the class.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Declare fields in the **StressTestCase** class as private.

3. Build the project and correct errors.
4. Update unit tests to resolve errors.

► **Task 1: Open the StressTesting solution**

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$\$w0rd**.
2. Open Microsoft Visual Studio® 2010.
3. Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex1\Starter folder.

► **Task 2: Declare fields in the StressTestCase class as private**

1. Review the task list.
2. In the task list, locate the **TODO - Modify the StressTestCase class to make members private** task, and then double-click this task. This task is located in the **StressTestCase** class.
3. In the **StressTestCase** class, remove the **TODO - Modify the StressTestCase class to make members private** comment, and then modify each field definition to make all of the fields private.

► **Task 3: Build the project and correct errors**

1. Build the project, and then review the error list.
The project should fail to build because the code in the **doTests_Click** method in the test harness project attempts to access the fields in the **StressTestCase** class that are now private.
2. Comment out the code that caused the errors that are shown in the error list. These errors are caused by six statements in the **doTests_Click** method.

► **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.
4. Update the method to verify that the constructed object contains the correct member values by performing the following tasks:



Hint: You cannot access the member data directly because you have just declared private members. The **ToString** method returns a string representation of the object, including the member data.

- a. Before you instantiate the **target** object, declare a new string named **expected** and populate the string with the following data that represents the expected results of the test.

Material: Composite, CrossSection: CShaped, Length: 5000mm, Height: 32mm, Width: 18mm, No Stress Test Performed

- b. At the end of the method, add an **Assert** statement that checks whether the **expected** string matches the output of the **target.ToString** method.
5. Update the **StressTestCaseConstructorTest1** method and resolve the errors by performing the following tasks:
 - a. Comment out the five existing **Assert** statements.
 - b. Before the method creates the **target** object, create a new string that contains the expected result from a default **StressTestCase** class. This string is the same as the string that the previous test expects.
 - c. At the end of the method, add an **Assert** statement that checks whether the **expected** string matches the output of the **target.ToString** method.
 6. Rebuild the solution and correct any errors.
 7. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

Exercise 2: Using Static Members to Share Data

In this exercise, you will define a struct that holds a pair of private fields to record the total number of tests that are performed and the total number of failures. You will add a private static member to the **StressTestCase** class that is based on this struct. You will then modify the **PerformStressTest** method to increment the fields in this struct as appropriate. Finally, you will add a static method to the class that returns the value of this struct.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Create a struct to hold the number of successes and failures.
3. Modify the **StressTestCase** class to contain a **TestStatistics** object.
4. Display the statistics in the user interface.
5. Test the solution.
6. Examine and run unit tests for the **TestStatistics** class.

► Task 1: Open the StressTesting solution

- Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex2\Starter folder. This solution contains a copy of the **StressTestCase** class with the public properties made private.

► Task 2: Create a struct to hold the number of successes and failures

1. Review the task list.
2. In the task list, locate the **TODO - Create the TestStatistics struct** task, and then double-click this task. This task is located in the **StressTestCase** class.
3. Delete the **TODO - Create the TestStatistics struct** comment, and then define a new public struct named **TestStatistics**, which has the following private members:
 - a. An integer named **numberOfTestsPerformed**.
 - b. An integer named **numberOfFailures**.
4. Add a method to the **TestStatistics** struct named **IncrementTests**. The method should accept a Boolean parameter named *success*, but not return a value. Add code to the method to perform the following tasks:

- a. Increment the **numberOfTestsPerformed** member.
- b. If the *success* parameter is **false**, increment the **numberOfFailures** member.
5. Below the **IncrementTests** method, add a method named **GetNumberOfTestsPerformed**. This method should take no parameters and return an integer value. Add code to the method to return the value of the **numberOfTestsPerformed** member.
6. Below the **GetNumberOfTestsPerformed** method, add a method named **GetNumberOfFailures**. The method should take no parameters and return an integer value. Add code to the method to return the value of the **numberOfFailures** member.
7. Below the **GetNumberOfFailures** method, add an **internal** method named **ResetCounters**. The method should take no parameters and not return a value. Add code to the method to set both the **numberOfFailures** and the **numberOfTestsPerformed** members to zero.
8. Build the project and correct any errors.

► **Task 3: Modify the StressTestCase class to contain a TestStatistics object**

1. In the task list, locate the **TODO - Add a TestStatistics field and method to the StressTestCase class** task, and then double-click this task. This task is located in the **StressTestCase** class.
2. Delete the **TODO - Add a TestStatistics field and method the StressTestCase class** comment, and then declare a new private static member of type **TestStatistics** named **statistics**.
3. Below the **statistics** member declaration, add a public static method named **GetStatistics**. The method should take no parameters, but should return a **TestStatistics** object. Add code to the method to return the value of the **statistics** member.
4. Below the **GetStatistics** method, add a public static method named **ResetStatistics**. The method should take no parameters and should not return a value. Add code to the method to invoke the **ResetCounters** method on the **statistics** member.

5. In the task list, locate the **TODO - Update the PerformStressTest method to handle statistics** task, and then double-click this task. This method is located in the **StressTestCase** class.
6. Delete the **TODO - Update the PerformStressTest method to handle statistics** comment, and in the **PerformStressTest** method, add code to invoke the **IncrementTests** method on the **statistics** member when a test either passes or fails. If the test passes, specify the value **true** as the argument to the **IncrementTests** method. If the test fails, specify the value **false** as the argument to the **IncrementTests** method.

► **Task 4: Display the statistics in the user interface**

1. In the task list, locate the **TODO - Update the UI to display statistics** task, and then double-click this task. This task is located in the **MainWindow** class, at the end of the **doTests_Click** method.
2. At the end of the **doTests_Click** method, delete the comment and add code to perform the following tasks:
 - a. Create a new **TestStatistics** object named **statistics**. Initialize the object with the value that is returned by calling the **StressTestCase.GetStatistics** method.
 - b. In the **statisticsLabel1** label, display the message "Number of tests: <tests>, Failures: <failures>", where *tests* is the number of tests that were executed, and *failures* is the number of tests that failed.



Hint: Set the **Content** property of a **Label** control to display a message in that control.

- c. Invoke the **IncrementTests** method on the **statistics** object, and pass **true** as a parameter.
- d. Invoke the static **GetStatistics** method on the **StressTestCase** object, and store the result in the statistics variable.
- e. In the **statisticsLabel2** label, display the message "Number of tests: <tests>, Failures: <failures>", where *tests* is the number of tests that were executed, and *failures* is the number of tests that failed.



Note: This demonstrates the principle of passing or returning by value. When the code first calls the **GetStatistics** method, a copy of the value is returned from the **StressTestCase** object. Therefore, when the code calls the **IncrementTests** method, the

update is performed on the copied value and not the original value. When the **GetStatistics** method is called for the second time, another copy of the original value is retrieved; therefore, both labels will display the same value.

► **Task 5: Test the solution**

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**, and then examine the statistics labels, which should both display the same values.
4. Close the MainWindow window, and then return to Visual Studio.

► **Task 6: Examine and run unit tests for the TestStatistics class**

1. In the task list, locate the **TODO - Examine and run unit tests** task, and then double-click this task. This task is located in the `StressTestClass_TestStatisticsTest` file.
2. Examine the **GetNumberOfFailuresTest** method.

This method creates a new **TestStatistics** object named **target** and then invokes the **IncrementTests** method twice, passing **false** as the parameter. The method then retrieves the number of failures from the **TestStatistics** object and uses an **Assert** statement to verify that the value is correct.
3. Examine the **GetNumberOfTestsPerformed** method.

This method creates a new **TestStatistics** object named **target** and then invokes the **IncrementTests** method three times. The method then retrieves the number of tests that were performed from the **TestStatistics** object and uses an **Assert** statement to verify that the value is correct.
4. Examine the **IncrementTestsTest** method.

This method creates a **TestStatistics** object named **target** and then invokes the **IncrementTests** method on this object four times. The method then retrieves the number of tests that were performed from the **target** object and uses an **Assert** statement to verify that the value is correct.
5. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

Exercise 3: Implementing an Extension Method

In this exercise, you will add a long integer field to the **TestCaseResult** struct to hold information. You will update the **PerformStressTest** method in the **StressTestCase** class to populate this field with simulated results. To display the data in this field as a binary string, you will add an extension method called **ToBinaryString** to the **System.Int64** struct.

The main tasks for this exercise are as follows:

1. Open the StressTesting solution.
2. Define a new extension method.
3. Modify the **TestCaseResult** struct to include a long field.
4. Modify the **PerformStressTest** method.
5. Display the failure data.
6. Test the solution.
7. Examine and run unit tests.

► Task 1: Open the StressTesting solution

- Open the StressTesting solution in the E:\Labfiles\Lab 7\Ex3\Starter folder. This solution contains a copy of the solution from the previous exercise.

► Task 2: Define a new extension method

1. In the StressTest project, add a new public static class named **Extensions**, in a file named Extensions.cs:
 - a. In Solution Explorer, right-click the **StressTest** project, point to **Add**, and then click **Class**.
 - b. In the **Add New Item - StressTest** dialog box, in the **Name** box, type **Extensions** and then click **Add**.
 - c. Modify the **Extensions** class definition. This class should be a **public static** class.
2. In the **Extensions** class, add a new public static extension method named **ToBinaryString**. The method should take a 64-bit integer parameter named *i* and return a string value.



Hint: To indicate that a method is an extension method, prefix the parameter with the **this** keyword.



Hint: You can use **long** as an alias for the **System.Int64** type.

3. In the **ToBinaryString** method, add code to create a string that holds the binary representation of the 64-bit integer value that is passed in the **i** integer, and return this string.

► **Task 3: Modify the `TestCaseResult` struct to include a long field**

1. Review the task list.
2. In the task list, locate the **TODO - Modify the `TestCaseResult` struct** task, and then double-click this task. This task is located in the **`TestCaseResult`** struct.
3. In the **`TestCaseResult`** struct, delete the comment and add a public field of type **long** named **`failureData`**.

► **Task 4: Modify the `PerformStressTest` method**

1. In the task list, locate the **TODO - Update the `PerformStressTest` method** task, and then double-click this task. This task is located in the **`StressTestCase`** class, in the **`PerformStressTest`** method.
2. In the **`PerformStressTest`** method, delete the **TODO - Update the `PerformStressTest` method** comment, and then add code to update the **`failureData`** member of the **`TestCaseResult`** object with a random number to simulate the data that is retrieved from the stress-testing equipment.



Hint: Use the **`Rand`** member of the **`Utility`** static class to generate a random number. This method contains a method called **`Next`** that returns a random number in a specified range. Pass the value **`int.MaxValue`** as the parameter to the **`Next`** method to generate a random number between 0 and this value. The value **`int.MaxValue`** field specifies the maximum value that the integer type supports.

► **Task 5: Display the failure data**

1. In the task list, locate the **TODO - Update the UI to display the binary string** task, and then double-click this task. This task is located in the **MainWindow** class, in the **doTests_Click** method.
2. Modify the **doTests_Click** method to append the binary data that is contained in the **failureData** member to the failure information that is displayed in the user interface; append a space character followed by the result of the **ToBinaryString** method call to the end of the string that is added to the **resultList.Items** collection.

► **Task 6: Test the solution**

1. Build the solution and correct any errors.
2. Run the application.
3. In the MainWindow window, click **Run Stress Tests**, and then verify that when an error occurs, binary data is displayed after the reason for the failure.
4. Close the MainWindow window, and then return to Visual Studio.

► **Task 7: Examine and run unit tests**

1. In the task list, locate the **TODO - Review and run unit tests** task, and then double-click this task. This task is located in the **ExtensionsTest** class.
2. Examine the **ToBinaryStringTest** method.

This method creates a long variable, **i**, with the value 8 and then creates a string variable, **expected**, with the value "1000". The method then invokes the **ToBinaryString** extension method on the long variable **i** and stores the result in a string named **actual**. The method then uses an **Assert** statement to verify that the **expected** and **actual** values are the same. The method then updates the long variable **i** with the value 10266 and the **expected** variable with the binary representation "10100000011010". Next, it directly calls the **ToBinaryString** method, passes the long variable **i** as a parameter, and stores the result of the method call in the **actual** variable. The method uses a second **Assert** statement to verify that the **expected** and **actual** values are the same.
3. Run all of the tests in the solution, and then verify that all of the tests execute successfully.

Module 8

Lab Instructions: Inheriting from Classes and Implementing Interfaces

Contents:

Exercise 1: Defining an Interface	4
Exercise 2: Implementing an Interface	6
Exercise 3: Creating an Abstract Class	12

Lab: Inheriting from Classes and Implementing Interfaces

- Exercise 1: Defining an Interface
- Exercise 2: Implementing an Interface
- Exercise 3: Creating an Abstract Class

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

Objectives

After completing this lab, you will be able to:

- Define an interface.
- Implement an interface in a class.
- Create an abstract class and inherit from this abstract class.

Introduction

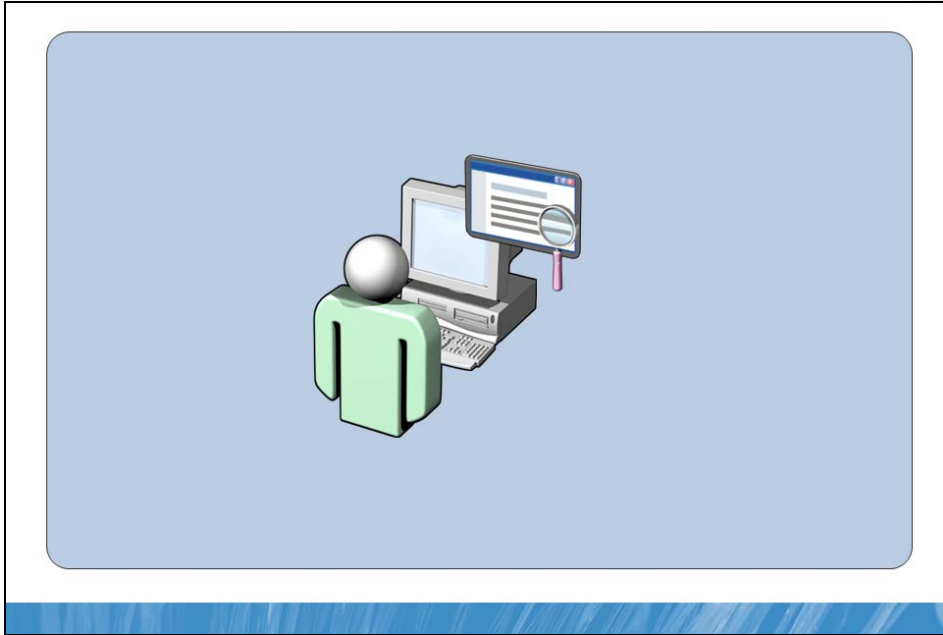
In this lab, you will define interfaces and create classes that implement them. You will then factor out common implementation code from the classes into methods in an abstract class and inherit from it.

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**

Lab Scenario



Fabrikam, Inc. produces a range of highly sensitive measuring devices that can repeatedly measure objects and capture data. These devices can be used to detect minuscule changes in objects over time. A measuring device monitors and measures one specific aspect of an object, such as its mass, its size in a given dimension (height, width, or length), or its distance from the measuring device. The data can be captured in metric or imperial units, and the device can convert the data that it has captured between the metric and imperial scales.

You have been asked to implement the software to drive these measuring devices.

Exercise 1: Defining an Interface

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

- **MetricValue.** This method will return a decimal that represents the metric value of the most recent measurement that was captured.
- **ImperialValue.** This method will return a decimal that represents the imperial value of the most recent measurement that was captured.

- **StartCollecting.** This method will start the device running. It will begin collecting measurements and record them.
- **StopCollecting.** This method will stop the device. It will cease collecting measurements.
- **GetRawData.** This method will retrieve a copy of all of the recent data that the measuring device has captured. The data will be returned as an array of integer values.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create the **IMeasuringDevice** interface.

► **Task 1: Open the starter project**

1. Log on to the 10266A-GEN-DEV machine as **Student** with the password **Pa\$\$w0rd**.
2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 8\Snippets folder
4. Open the Module8 solution in the E:\Labfiles\Lab 8\Ex1\Starter folder.

► **Task 2: Create the IMeasuringDevice interface**

1. Open the IMeasuringDevice code file.
2. In the **MeasuringDevice** namespace, declare the **IMeasuringDevice** interface. The **IMeasuringDevice** interface must be accessible to code in other assemblies.
3. Add a method named **MetricValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method.
4. Add a method named **ImperialValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method.
5. Add a method named **StartCollecting** with a no return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.

6. Add a method named **StopCollecting** with a no return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.
7. Add a method named **GetRawData** that returns an integer array return type to the interface. This method should take no parameters. Add a comment that describes the purpose of the method.
8. Build the solution and correct any errors.

Exercise 2: Implementing an Interface

In this exercise, you will define the following enumeration:

- Units: Metric, Imperial

You will then define a class called **MeasureLengthDevice** that implements the **IMeasuringDevice** interface and drives a device that measures the length of an object. This class will also include the following private fields:

- **unitsToUse**: Units
- **dataCaptured**: integer array
- **mostRecentMeasure**: integer

You will provide a constructor to initialize the fields in the class (the user will specify a parameter that populates **unitsToUse**).

When the device starts running (when the **StartCollecting** method is called), the device will capture data and store it in the **dataCaptured** array (you will simulate this in the lab by using the code that is provided). This array has a finite, fixed size; when the device is full, it will wrap around and start to overwrite the oldest data. Each time that it takes a new measurement, the device copies this measurement to the **mostRecentMeasure** field. The **GetRawData** method will return the contents of the array. The **MetricValue** and **ImperialValue** methods will return the value in this field, converted according to the units that are specified in the **unitsToUse** field. If **unitsToUse** is **Metric**, **MetricValue** simply returns the data and **ImperialValue** performs a calculation to convert the data to imperial units. Similarly, if **unitsToUse** is **Imperial**, **ImperialValue** simply returns the data and **MetricValue** performs a calculation to convert the data to metric units.

The main tasks for this exercise are as follows:

1. Open the starter project.

2. Create the **Units** enumeration.
3. Create the **MeasureLengthDevice** class.
4. Update the test harness.
5. Test the **MeasureLengthDevice** class by using the test harness.

► **Task 1: Open the starter project**

- Open the Module8 solution in the E:\Labfiles\Lab 8\Ex2\Starter folder. This solution contains the completed interface from Exercise 1 and skeleton code for Exercise 2.

► **Task 2: Create the Units enumeration**

1. In Visual Studio, review the task list.
2. In the task list, double-click the task **TODO: Implement the Units enumeration**. This task is located in the UnitsEnumeration.cs file.
3. Remove the TODO comment in the UnitsEnumeration file and declare an enumeration named **Units**. The enumeration must be accessible from code in different assemblies.
4. Add the values **Metric** and **Imperial** to the enumeration.
5. Comment your code to make it easier for developers who use the enumeration.
6. Build the solution and correct any errors.

► **Task 3: Create the MeasureLengthDevice class**

1. In the task list, double-click the task **TODO: Implement the MeasureLengthDevice class**. This task is located in the MeasureLengthDevice.cs file.
2. Remove the TODO comment and add a **public** class named **MeasureLengthDevice**.
3. Modify the **MeasureLengthDevice** class declaration to implement the **IMeasuringDevice** interface.
4. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.

5. Bring the **DeviceControl** namespace into scope.

The MeasuringDevice project already contains a reference to the DeviceController project. You are writing code to control a device. However, because the physical device is not available with this lab, the DeviceController project enables you to call methods that control an emulated device. The DeviceController project does not include a visual interface; to control the device, you must use the classes and methods that the project exposes. The DeviceController project is provided complete. You can review the code if you wish, but you do not need to modify it.

6. After the method stubs that the Implement Interface Wizard added in the **MeasureLengthDevice** class, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

DeviceType is an enumeration that contains the values **LENGTH** and **MASS**. It is used to specify the type of measurement that the device records. It is defined in the DeviceController project.

7. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.LENGTH**.
8. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to the **StartCollecting** method to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the value in the **measurementType** field as the parameter to the **StartCollecting** method.
9. In the **StartCollecting** method, call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.

10. Add the **GetMeasurements** method to the class, as shown in the following code example.



Note: A code snippet is available, called **Mod8GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();

        while (controller != null)
        {
            System.Threading.Thread.Sleep(timer.Next(1000, 5000));
            dataCaptured[x] = controller != null ?
                controller.TakeMeasurement() : dataCaptured[x];
            mostRecentMeasure = dataCaptured[x];

            x++;
            if (x == 10)
            {
                x = 0;
            }
        }
    });
}
```

The **GetMeasurements** method retrieves measurements from the emulated device. In this module, you will use the code in the **GetMeasurements** method to populate the **dataCaptured** array. This array acts as a fixed-length circular buffer, overwriting the oldest value each time a new measurement is taken. In a later module, you will modify this class to respond to events that the device raises whenever it detects a new measurement.

11. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.
12. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.

13. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
14. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 25.4.
15. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 0.03937.
16. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
17. Build the solution and correct any errors.

► Task 4: Update the test harness

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

1. In Visual Studio, review the task list.
2. Open the `MainWindow.xaml.cs` file by clicking the first **TODO: Add code to instantiate the device field** item in the task list. This task is located in the **createInstance_Click** method in the WPF window, and it runs when the user clicks the **Create Instance** button.
3. In the **createInstance_Click** method, replace both TODO comments with code to instantiate a field called **device** and set it to an instance of the **MeasureLengthDevice** class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureLengthDevice** constructor.
4. Build the solution and correct any errors.

► **Task 5: Test the MeasureLengthDevice class by using the test harness**

1. Set the Exercise2TestHarness project to be the default startup project.
2. Start the Exercise2TestHarness application.
3. Choose **Imperial**, and then click **Create MeasureLengthDevice Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.
4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **device** object that the **IMeasuringDevice** interface defines.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Raw Data**. You should see up to 10 values in the list box in the lower part of the window. This is the data that the device emulator has generated. It is stored in the **dataCaptured** array by the **GetMeasurements** method in the **MeasureLengthDevice** class. The **dataCaptured** array acts as a fixed-length circular buffer. Initially, it contains zero values, but as the device emulator reports measurements, they are added to this array. When the array is full, it wraps around and starts overwriting data, beginning with the oldest measurement.
7. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement. Note that a new measurement might have been taken since you clicked the **Get Raw Data** button.
8. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
9. Click **Stop Collecting**.
10. Choose **Metric**, and then click **Create MeasureLengthDevice Instance**. This action creates a new instance of the device emulator that uses metric measurements.
11. Click **Start Collecting**. This button starts the new **device** object.
12. Wait for 10 seconds.
13. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.

14. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
15. Click **Stop Collecting**.
16. Close the Exercise 2 Test Harness window.

Exercise 3: Creating an Abstract Class

In this exercise, you will define a class called **MeasureMassDevice**, which also implements the **IMeasuringDevice** interface. You will notice that, although the **MetricValue** and **ImperialValue** methods are implemented slightly differently from the **MeasureLength** class, the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods are identical. Code duplication is never a good thing, and can lead to maintenance difficulties. Consequently, you will create an abstract class called **MeasureDataDevice** that provides default implementations of the duplicated methods. Students will modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from this class.

The main tasks in this exercise are as follows:

1. Open the starter project.
2. Create the **MeasureMassDevice** class.
3. Update the test harness.
4. Test the **MeasureMassDevice** class by using the test harness.
5. Create the **MeasureDataDevice** abstract class.
6. Modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from the **MeasureDataDevice** abstract class.
7. Test the classes by using the test harness.

► Task 1: Open the starter project

- Open the Module8 solution in the E:\Labfiles\Lab 8\Ex3\Starter folder. This solution contains the completed interface from Exercise 2 and skeleton code for Exercise 3.

► **Task 2: Create the MeasureMassDevice class**

1. In Visual Studio, review the task list.
2. Open the MeasureMassDevice.cs file.
3. Replace the TODO comment with a **public** class named **MeasureMassDevice**.
4. Modify the **MeasureMassDevice** class declaration to implement the **IMeasuringDevice** interface.
5. Use the Implement Interface Wizard to generate method stubs for each of the methods in the **IMeasuringDevice** interface.
6. Bring the **DeviceControl** namespace into scope.

The MeasuringDevice project already contains a reference to the DeviceController project. This project implements the **DeviceController** type, which provides access to the measuring device emulator.

7. After the method stubs that Visual Studio added, add the fields shown in the following table.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

8. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.MASS**.
9. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the **measurementType** field as the parameter to the **StartDevice** method.
10. Add code to call the **GetMeasurements** method. This method takes no parameters and does not return a value. You will add the **GetMeasurements** method in the next step.

11. Add the **GetMeasurements** method to the class, as shown in the following code example.



Note: A code snippet is available, called **Mod8GetMeasurementsMethod**, that you can use to add this method.

```
private void GetMeasurements()
{
    dataCaptured = new int[10];
    System.Threading.ThreadPool.QueueUserWorkItem((dummy) =>
    {
        int x = 0;
        Random timer = new Random();

        while (controller != null)
        {
            System.Threading.Thread.Sleep(timer.Next(1000, 5000));
            dataCaptured[x] = controller != null ?
                controller.TakeMeasurement() : dataCaptured[x];
            mostRecentMeasure = dataCaptured[x];

            x++;
            if (x == 10)
            {
                x = 0;
            }
        }
    });
}
```

This is the same method that you defined for the **MeasureLengthDevice** class.

12. Locate the **StopCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add a conditional code block that only runs if the **controller** object is not **null**.
13. In the conditional code block, add code to call the **StopDevice** method of the **controller** object, and then set the **controller** field to **null**.
14. Locate the **GetRawData** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to return the **dataCaptured** array.
15. Locate the **MetricValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException**

exception. Add code to check the current units and, if they are metric, return the value from the **mostRecentMeasure** field. If the current units are imperial, return the result of multiplying the **mostRecentMeasure** field by 0.4536.

16. Locate the **ImperialValue** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to check the current units and, if they are imperial, return the value from the **mostRecentMeasure** field. If the current units are metric, return the result of multiplying the **mostRecentMeasure** field by 2.2046.
17. Add to the class a constructor that takes a *Units* parameter and sets the **unitsToUse** field to the value specified by this parameter.
18. Build the solution and correct any errors.

► Task 3: Update the test harness

The test harness application in this lab is a modified version of the WPF application that you used in Exercise 2. It is designed to test the functionality of the **MeasureLengthDevice** and **MeasureMassDevice** classes. It does not include any exception handling to ensure that it does not hide any exceptions thrown by the class that you have developed.

1. In Visual Studio, review the task list.
2. Open the `MainWindow.xaml.cs` file by using the first **TODO: Instantiate the device field by using the new MeasureMassDevice class** item in the task list.
3. In the **createInstance_Click** method, replace both **TODO** comments with code to instantiate the device field to an instance of the **MeasureMassDevice** class. You must use the appropriate member of the **Units** enumeration as the parameter for the **MeasureMassDevice** constructor.
4. Build the solution and correct any errors.

► Task 4: Test the MeasureMassDevice class by using the test harness

1. Set the `Exercise3TestHarness` project to be the default startup project.
2. Start the `Exercise3TestHarness` application.
3. Choose **Imperial**, choose **Mass Device**, and then click **Create Instance**. This button runs the code that you added to instantiate the **device** field that uses imperial measurements.

4. Click **Start Collecting**. This button runs the **StartCollecting** method of the **MeasureMassDevice** object.
5. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
6. Click **Get Metric Value** and **Get Imperial Value**. You should see the metric and imperial value of the most recently generated measurement.
7. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
8. Click **Stop Collecting**.
9. Choose **Metric**, and then click **Create Instance**. This action creates a new instance of the device emulator that uses metric measurements.
10. Click **Start Collecting**. This button starts the new **device** object.
11. Wait for 10 seconds.
12. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
13. Click **Get Raw Data**, and then verify that the metric value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
14. Click **Stop Collecting**.
15. Close the Exercise 3 Test Harness window.

► **Task 5: Create the MeasureDataDevice abstract class**

You have developed two classes, **MeasureLengthDevice** and **MeasureMassDevice**. Much of the functionality of these classes is common to both. This code duplication is unnecessary and risks introducing bugs. To reduce the code that is required and the risk of introducing bugs, you will create an abstract class that will contain the common functionality.

1. Open the **MeasureDataDevice.cs** file.
2. Remove the **TODO** comment and add an **abstract** class named **MeasureDataDevice**.
3. Modify the **MeasureDataDevice** class declaration to implement the **IMeasuringDevice** interface.

4. Bring the **DeviceControl** namespace into scope.
5. In the **MeasureDataDevice** class, add a **public abstract** method named **MetricValue**. This method should return a **decimal** value, but not take any parameters.

The implementation of the **MetricValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes. Declaring the **MetricValue** method as **abstract** forces child classes to implement this method.



Hint: Look at the code for the **MetricValue** method for the **MeasureLengthDevice** and **MeasureMassDevice** classes. You will observe that they are quite similar, apart from the conversion factors that are used, and you could factor this logic out into a method in the abstract **MeasureDataDevice** class. However, for the sake of this exercise, assume that these methods are totally different. The same note applies to the **ImperialValue** method that you will define in the next step.

6. In the **MeasureDataDevice** class, add a **public abstract** method with a **decimal** return type named **ImperialValue**.

Like the **MetricValue** method, the implementation of the **ImperialValue** method is specific to the type of device being controlled, so you must implement this functionality in the child classes.

7. In the **MeasureLengthDevice.cs** file, locate and copy the code for the **StartCollecting** method, and then add this method to the **MeasureDataDevice** class.

Visual Studio will warn you that the controller variable, the **measurementType** enumeration, and the **GetMeasurements** method are not defined. You will add these items to the **MeasureDataDevice** class in later steps in this task.

8. Copy the **StopCollecting** method from the **MeasureLengthDevice.cs** file to the **MeasureDataDevice** class.

Visual Studio will warn you that the controller variable is not defined.

9. Copy the **GetRawData** method from the **MeasureLengthDevice.cs** file to the **MeasureDataDevice** class.

Visual Studio will warn you that the **dataCaptured** variable is not defined.

10. Copy the **GetMeasurements** method from the **MeasureLengthDevice.cs** file to the **MeasureDataDevice** class.

Visual Studio will warn you that the `dataCaptured`, `controller`, and `mostRecentMeasure` variables are not defined.

- Copy the five fields in the following table from the `MeasureLengthDevice.cs` file to the **MeasureDataDevice** class.

Name	Type	Accessor
<code>unitsToUse</code>	<code>Units</code>	<code>private</code>
<code>dataCaptured</code>	<code>int[]</code>	<code>private</code>
<code>mostRecentMeasure</code>	<code>int</code>	<code>private</code>
<code>controller</code>	<code>DeviceController</code>	<code>private</code>
<code>measurementType</code>	<code>DeviceType</code>	<code>private</code>

The warnings in the **StartCollecting**, **StopCollecting**, **GetRawData**, and **GetMeasurements** methods should disappear.

- 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.
- Modify the declaration of the **measurementType** field so that it is no longer constant and not instantiated when it is declared.
- Build the solution and correct any errors.

► Task 6: Modify the **MeasureLengthDevice** and **MeasureMassDevice** classes to inherit from the **MeasureDataDevice** abstract class

In this task, you will remove the duplicated code from the **MeasureLengthDevice** and **MeasureMassDevice** classes by modifying them to inherit from the **MeasureDataDevice** abstract class that you created in the previous task.

- 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.
- Remove the **StartCollecting** method from the **MeasureLengthDevice** class.
- Remove the **StopCollecting** method from the **MeasureLengthDevice** class.

4. Remove the **GetRawData** method from the **MeasureLengthDevice** class.
5. Remove the **GetMeasurements** method from the **MeasureLengthDevice** class.
6. Remove the fields in the following table from the **MeasureLengthDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private
controller	DeviceController	private
measurementType	DeviceType	private

7. Modify the constructor to set the **measurementType** field to **DeviceType.LENGTH**.
8. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
9. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
10. In the **MeasureMassDevice.cs** file, modify the declaration of the **MeasureMassDevice** class so that it inherits from the **MeasureDataDevice** class.
11. Remove the **StartCollecting** method from the **MeasureMassDevice** class.
12. Remove the **StopCollecting** method from the **MeasureMassDevice** class.
13. Remove the **GetRawData** method from the **MeasureMassDevice** class.
14. Remove the **GetMeasurements** method from the **MeasureMassDevice** class.
15. Remove the fields in the following table from the **MeasureMassDevice** class.

Name	Type	Accessor
unitsToUse	Units	private
dataCaptured	int[]	private
mostRecentMeasure	int	private

Name	Type	Accessor
controller	DeviceController	private
measurementType	DeviceType	private

16. Modify the constructor to set the **measurementType** field to **DeviceType.MASS**.
17. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class.
18. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class.
19. Build the solution and correct any errors.

► **Task 7: Test the classes by using the test harness**

In this task, you will check that the **MeasureLengthDevice** and **MeasureMassDevice** classes still work as expected.

1. Start the Exercise3TestHarness application.
2. Choose **Imperial**, choose **Mass Device**, and then click **Create Instance**.
3. Click **Start Collecting**.
4. Wait for 10 seconds to ensure that the emulated device has generated some values before you perform the following steps.
5. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.
6. Click **Get Raw Data**, and then verify that the imperial value that the previous step displayed is listed in the raw data values. (The value can appear at any point in the list.)
7. Click **Stop Collecting**.
8. Choose **Metric**, choose **Length Device**, and then click **Create Instance**.
9. Click **Start Collecting**. This button starts the new **device** object.
10. Wait for 10 seconds.
11. Click **Get Metric Value** and **Get Imperial Value** to display the metric and imperial value of the latest measurement that the device has taken.

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.

Module 9

Lab Instructions: Managing the Lifetime of Objects and Controlling Resources

Contents:

Exercise 1: Implementing the IDisposable Interface

4

Exercise 2: Managing Resources Used by an Object

9

Lab: Managing the Lifetime of Objects and Controlling Resources

- Exercise 1: Implementing the `IDisposable` Interface
- Exercise 2: Managing Resources Used by an Object

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 45 minutes

Objectives

After completing this lab, you will be able to:

- Implement the **`IDisposable`** interface in a type.
- Ensure that resources associated with an object are reclaimed through a **`using`** statement.

Introduction

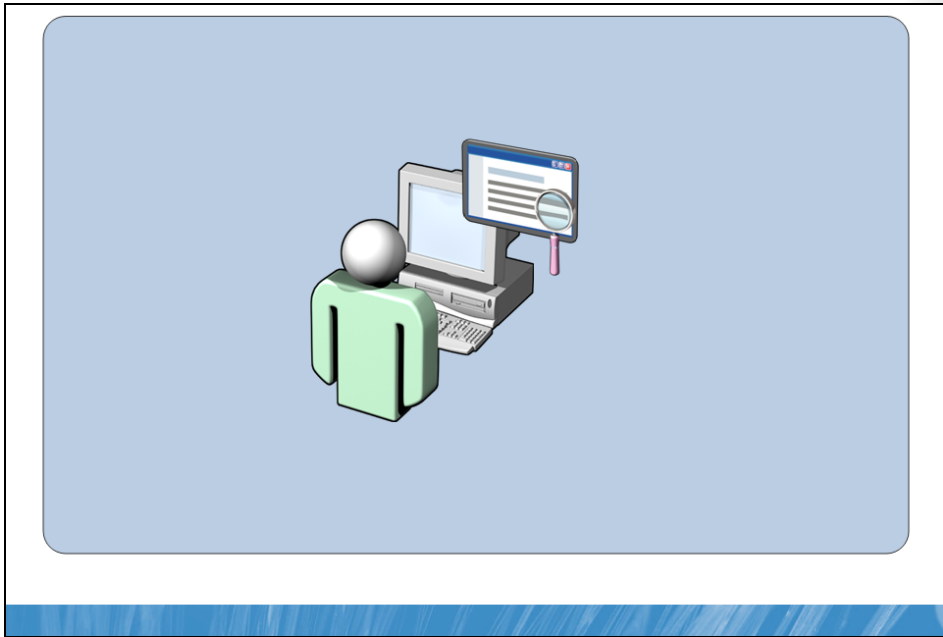
In this lab, you will define a type that implements the **`IDisposable`** interface and then reference objects of this type through a **`using`** statement to ensure that they are disposed of correctly.

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**

Lab Scenario



The first version of the family of measuring devices produced by Fabrikam, Inc. recorded data to a local circular buffer on the device, implemented by using an array. However, this array has a fixed, finite size. If the user does not retrieve the data from the device sufficiently often, measurements will be overwritten and lost. You have been asked to develop the software to drive an enhanced version of these devices. The new version supports logging to a file and to the buffer in memory. This should prevent data loss.

Exercise 1: Implementing the IDisposable Interface

Scenario

In this exercise, you will create a new interface called **ILoggingMeasuringDevice** that extends the **IMeasuringDevice** interface and adds the following method:

- **GetLogFile**. This method will return the name of the file that the device logs data to.

You will modify the **MeasureDataDevice** abstract class and add the following private field:

- **loggingFileName.** This field will contain the name of the file that the device will log data to.

You will implement the **GetLoggingFile** method in the abstract class to return the name of the file in the **loggingFileName** field.

In the **StartCollecting** method of the abstract class, you will add code to open the file and record measurements as they are written to the buffer. In the **StopCollecting** method, you will add code to close the file.

You will then extend the abstract class to implement the **IDisposable** interface. In the **Dispose** method, you will add code to ensure that the file is closed correctly and its contents are flushed to disk when the object is destroyed.

You will modify the constructor for the **MeasureMassDevice** class that inherits from the **MeasureDataDevice** abstract class and include a parameter that enables an application to specify a file name. The constructor will use this file name to populate the **loggingFileName** field.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Create the **ILoggingMeasuringDevice** interface.
3. Modify the **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface.
4. Modify the **MeasureDataDevice** class to implement the **IDisposable** interface.
5. Modify the **MeasureMassDevice** class to use logging.

► Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa\$Sw0rd**.
2. Open Visual Studio 2010.
3. Import the code snippets from the E:\Labfiles\Lab 9\Snippets folder.
4. Open the Module9 solution in the E:\Labfiles\Lab 9\Ex1\Starter folder.

► Task 2: Create the **ILoggingMeasuringDevice** interface

In this task, you will develop the **ILoggingMeasuringDevice** interface. You will develop this new interface, which inherits from the existing **IMeasuringDevice**

interface, rather than editing the existing interface to ensure compatibility with existing code.

1. In Visual Studio, review the task list.
2. Open the `ILoggingMeasuringDevice.cs` file.
3. Remove the `TODO` comment and declare an interface named **ILoggingMeasuringDevice**. The interface must be accessible from code in different assemblies.
4. Modify the interface to inherit from the **IMeasuringDevice** interface.
5. Add a method named **GetLoggingFile** that returns a **string** value to the interface. The method should take no parameters. The purpose of this method is to return the file name of the logging file used by the device. Add an XML comment that summarizes the purpose of the method.
6. Build the solution and correct any errors.

► **Task 3: Modify the MeasureDataDevice class to implement the ILoggingMeasuringDevice interface**

In this task, you will modify the existing **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface. You will add code to enable logging and modify existing methods to use the logging functionality.

1. Open the `MeasureDataDevice.cs` file.
2. Remove the comment **TODO: Modify this class to implement the ILoggingMeasuringDevice interface instead of the IMeasuringDevice interface** above the **MeasureDataDevice** class. Modify the **MeasureDataDevice** class to implement the **ILoggingMeasuringDevice** interface instead of the **IMeasuringDevice** interface.
3. In the task list, locate the comment **TODO: Add fields necessary to support logging**. Double-click this item to go to the relevant line in the `MeasureDataDevice.cs` file.
4. Remove the `TODO` comment and add a **string** field named **loggingFileName**. This field must be accessible to classes that inherit from this class. This field will store the file name and path for the log file.
5. Add a **TextWriter** field named **loggingFileWriter**. This field should only be accessible to code in this class. You will use this object to write to a file.

6. In the task list, locate the comment **TODO: Add methods to implement the `ILoggingMeasuringDevice` interface**. Double-click this comment to go to the relevant line in the `MeasureDataDevice.cs` file.
7. Remove the TODO comment and add the **`GetLoggingFile`** method defined in the **`ILoggingMeasuringDevice`** interface. The method should take no parameters and return the value in the **`loggingFileName`** field.
8. In the task list, locate the comment **TODO: Add code to open a logging file and write an initial entry**. Double-click this comment to go to the relevant line in the `MeasureDataDevice.cs` file.
9. Remove the TODO comment and add the following code to instantiate the **`loggingFileWriter`** field. You can either type this code manually, or you can use the `Mod9InstantiateLoggingFileWriter` code snippet.

```
// New code to check the logging file is not already open.
// If it is already open then write a log message.
// If not, open the logging file.
if (loggingFileWriter == null)
{
    // Check if the logging file exists - if not create it.
    if (!File.Exists(loggingFileName))
    {
        loggingFileWriter = File.CreateText(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Created");
        loggingFileWriter.WriteLine("Collecting Started");
    }
    else
    {
        loggingFileWriter = new StreamWriter(loggingFileName);
        loggingFileWriter.WriteLine
            ("Log file status checked - Opened");
        loggingFileWriter.WriteLine("Collecting Started");
    }
}
else
{
    loggingFileWriter.WriteLine
        ("Log file status checked - Already open");
    loggingFileWriter.WriteLine("Collecting Started");
}
```

The code checks whether the **`loggingFileWriter`** object has already been instantiated. If it has not, the code instantiates it by checking whether the file specified by the **`loggingFileName`** field already exists. If the file exists, the code opens the file; if it does not, the code creates a new file.

10. In the task list, locate the comment **TODO: Add code to write a message to the log file**. Double-click this comment to go to the relevant line in the `MeasureDataDevice.cs` file.
11. Remove the **TODO** comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.
12. In the task list, locate the comment **TODO: Add code to log each time a measurement is taken**. Double-click this comment to go to the relevant line in the `MeasureDataDevice.cs` file.
13. Remove the **TODO** comment and add code to write a message to the log file. Your code should check that the **loggingFileWriter** object is instantiated before writing the message.
14. Build the solution and correct any errors.

► **Task 4: Modify the `MeasureDataDevice` class to implement the `IDisposable` interface**

In this task, you will modify the existing `MeasureDataDevice` class to implement the `IDisposable` interface. You will add code to ensure that the `TextWriter` object that writes messages to the log file is properly closed when an instance of the `MeasureDataDevice` class is disposed of.

1. At the top of the `MeasureDataDevice` class, remove the comment **TODO: Modify this class to implement the `IDisposable` interface**, and then modify the `MeasureDataDevice` class to implement the `IDisposable` interface in addition to the `ILoggingMeasuringDevice` interface.
2. Use the Implement Interface Wizard to generate method stubs for each of the methods in the `IDisposable` interface.
3. Move to the end of the `MeasureDataDevice` class. After the **Dispose** method added by the Implement Interface Wizard, add an overloaded **virtual void Dispose** method that implements the dispose pattern. This method should take a Boolean parameter called **disposing** and perform the following tasks:
 - a. Check that the *disposing* parameter is set to **true**. If it is not, finish without disposing of anything.
 - b. If the **loggingFileWriter** object is not null, write the message "Object disposed" to the logging file, flush the contents of the **loggingFileWriter** object, close it, and set the `loggingFileWriter` variable to **null**.

4. Locate the **Dispose** method, which takes no parameters, and then remove the default method body inserted by Visual Studio, which throws a **NotImplementedException** exception. Add statements that call the overloaded **Dispose** method and specify true as the parameter, and then suppress finalization for the current object.
5. Build the solution and correct any errors.

► **Task 5: Modify the MeasureMassDevice class to use logging**

In this task, you will modify the existing **MeasureMassDevice** class to set the **loggingFileName** field when the class is instantiated.

1. Open the MeasureMassDevice.cs file.
2. In the **MeasureMassDevice** class, remove the comment **TODO: Modify the constructor to set the log filename based on a string parameter**, and then modify the constructor to take a *string* parameter called *logFileName*. In the body of the constructor, set the **loggingFileName** field to the *logFileName* parameter. You should also update the XML comments for the constructor to describe the new parameter.
3. Build the solution and correct any errors.

Exercise 2: Managing Resources Used by an Object

Scenario

In this exercise, you will use a test harness application to test the disposal functionality that you added to the classes in the previous exercise. The test harness is a simple Windows® Presentation Foundation (WPF) application. Note that this application does not include exception handling or necessarily follow best practices for implementing a graphical user interface.

The main tasks for this exercise are as follows:

1. Open the starter project.
2. Test the logging functionality by using the test harness.
3. Modify the test harness to dispose of objects correctly.
4. Verify that the object is disposed of correctly.

► **Task 1: Open the starter project**

- Open the Module9 solution from the E:\Labfiles\Lab 9\Ex2\Starter folder. This solution contains the completed code from Exercise 1 and skeleton code for Exercise 2.

► **Task 2: Test the logging functionality by using the test harness**

1. Run the Exercise2 Test Harness application.
2. Click **Get Measurements**. This action causes the application to pause for 20 seconds while some measurements data is generated and then display this data. This pause is necessary because the application waits for measurement data from the emulated device.

Note that the measurement data is logged to the E:\Labfiles\Lab 9\LogFile.txt file by default.

3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.
4. Using Notepad, open the LogFile.txt file in the E:\Labfiles\Lab 9 folder.
5. Review the contents of the LogFile.txt file.

The file is empty. Although the application has retrieved values from the emulated device and written them to the log file, the **TextWriter** object caches data in memory and writes to the underlying file system when it is either flushed or closed. When you closed the application, you disposed of the **TextWriter** object without flushing its in-memory cache to the log file, which is why the file is empty.

6. Close Notepad.
7. Run the Exercise2 Test Harness application again, click **Get Measurements**, and then wait for the data to appear.
8. After the application populates the text boxes with data from the emulated device, click **Get Measurements** again.

The application will throw an unhandled **IOException** exception. The exception is thrown because each time you click **Get Measurements**, you create a new instance of the **MeasureMassDevice** class. Each instance of the **MeasureMassDevice** class creates its own instance of the **TextWriter** class to log measurements. The test harness does not currently dispose of the **MeasureMassDevice** objects after the code run by the **Get Measurements** button completes. This means that the object is not closed and therefore

retains its lock on the log file. When you attempt to create a second instance of the **MeasureMassDevice** class that uses the same log file, this instance cannot access the file because it is still in use by the first instance.

9. Stop the Exercise 2 application.

► **Task 3: Modify the test harness to dispose of objects correctly**

1. In Visual Studio, open the MainWindow.xaml.cs file in the Exercise2 Test Harness project.
2. In the **createInstance_Click** method, remove the **TODO: Modify this method** comment in the MainWindow.xaml.cs file. Modify the **createInstance_Click** method to ensure that the **device** field is disposed of when the method completes by using a **using** block.
3. Build the solution and correct any errors.

► **Task 4: Verify that the object is disposed of correctly**

1. Run the Exercise2 Test Harness application.
2. Click **Get Measurements**, and then wait until the data appears.
3. After the application populates the text boxes with data from the emulated device, close the Exercise 2 window.
4. Open Notepad and examine the log file.
5. Review the contents of the log file.

The file now contains the values displayed on the form and status messages generated when the file is opened and closed. When the code for the **Get Measurements** button completes, it now disposes of the **MeasureMassDevice** instance, which forces the **TextWriter** object to flush its in-memory cache to the file, and then closes the **TextWriter** object.

6. Close Notepad
7. Run the Exercise2 Test Harness application again.
8. Click **Get Measurements**, and then wait for the data to appear.
9. In the Exercise 2 window, click **Get Measurements** again. The application will pause for another 20 seconds.

This time, the application does not throw an exception. This is because the resources are properly disposed of each time you click **Get Measurements**. When you close the **TextWriter** object, you release the lock on the file, and a new instance of the **TextWriter** class can now use the same log file without throwing an exception.

10. Open Notepad and examine the log file.
11. Review the contents of the log file.

The file contains the most recent values displayed on the form.

12. Close Notepad.
13. Close the Exercise 2 window.
14. Close Visual Studio.

Module 10

Lab Instructions: Encapsulating Data and Defining Overloaded Operators

Contents:

Lab A: Creating and Using Properties

Exercise 1: Defining Properties in an Interface 5

Exercise 2: Implementing Properties in a Class 6

Exercise 3: Using Properties Exposed by a Class 9

Lab B: Creating and Using Indexers

Exercise 1: Implementing an Indexer to Access Bits in a Control Register 16

Exercise 2: Using an Indexer Exposed by a Class 18

Lab C: Overloading Operators

Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types 25

Exercise 2: Implementing Operators for the Matrix Type 31

Exercise 3: Testing the Operators for the Matrix Type 34

Lab A: Creating and Using Properties

- Exercise 1: Defining Properties in an Interface
- Exercise 2: Implementing Properties in a Class
- Exercise 3: Using Properties Exposed by a Class

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 25 minutes

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

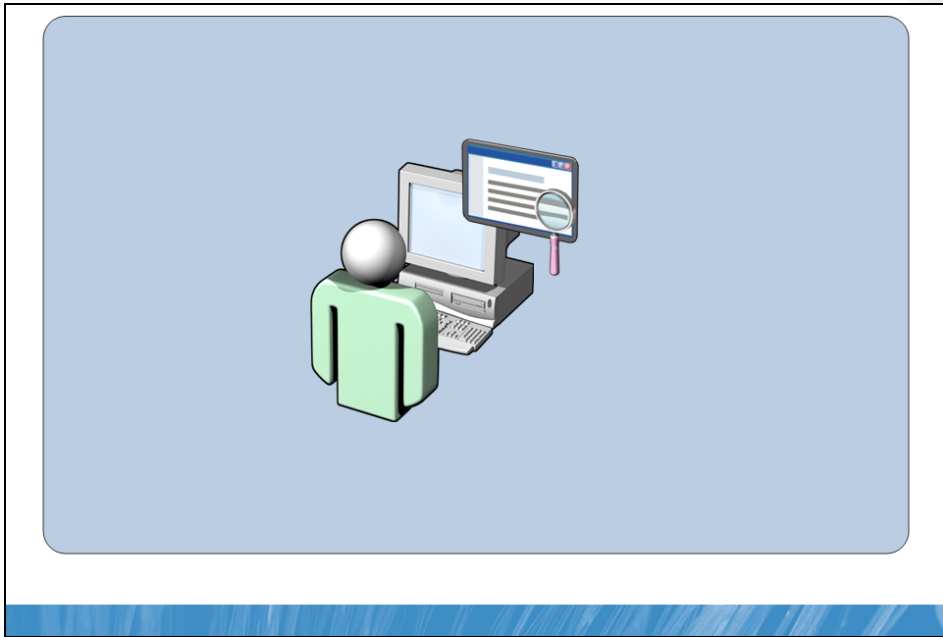
In this lab, you will define properties in an interface and then implement these properties in a class. You will also use a test application to verify that the properties behave as expected.

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**

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.

- **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.

► **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.

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);
```

```
        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.

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 = "";
```

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.

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.

Lab B: Creating and Using Indexers

- Exercise 1: Implementing an Indexer to Access Bits in a Control Register
- Exercise 2: Using an Indexer Exposed by a Class

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 25 minutes

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

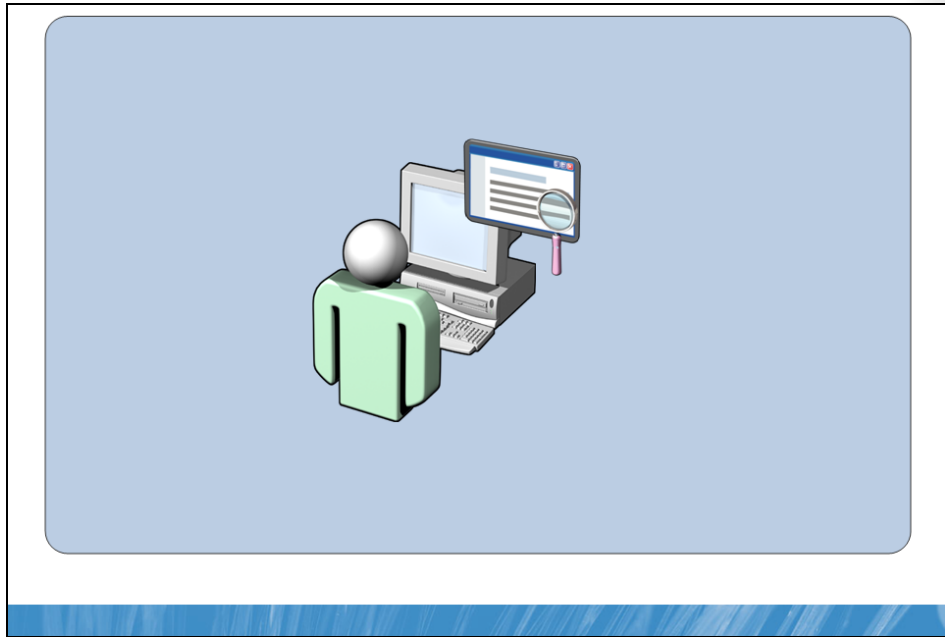
In this lab, you will add an indexer to a class. You will then use a test application to verify that the indexer functions correctly.

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**

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

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**.

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.

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.

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

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

Lab C: Overloading Operators

- Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types
- Exercise 2: Implementing Operators for the Matrix Type
- Exercise 3: Testing the Operators for the Matrix Type

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 40 minutes

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

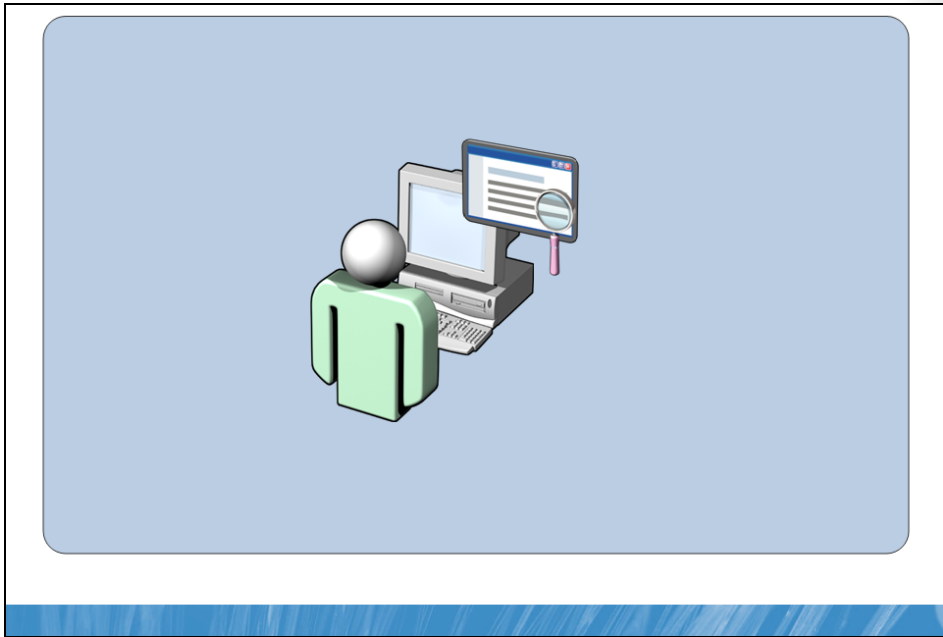
In this lab, you will create a new type that models square matrices. You will implement the addition, subtraction, and multiplication operators for this type and test that these operators function correctly.

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**

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.

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();
        }
    }
}
```

```
        }
        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.
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.
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;
        }
    }

    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)
    {
    }

    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.

► 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.

► **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.

► **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.

Module 11

Lab Instructions: Decoupling Methods and Handling Events

Contents:

Exercise 1: Raising and Handling Events

4

Exercise 2: Using Lambda Expressions to Specify Code

14

Lab: Decoupling Methods and Handling Events

- Exercise 1: Raising and Handling Events
- Exercise 2: Using Lambda Expressions to Specify Code

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

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

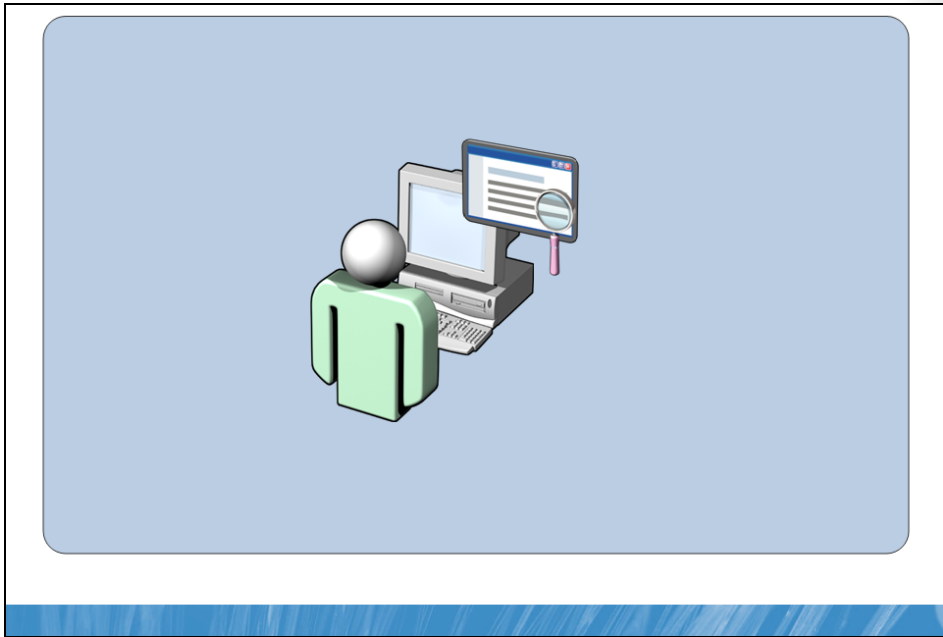
In this lab, you will define and raise events and handle them by using delegates. You will use lambda expressions to specify actions to perform and will run these actions by invoking the lambda expressions.

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**

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.

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.

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.

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.

► **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:

- 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.

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.

► **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.

► **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**.

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

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.

- 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.

► **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.

Module 12

Lab Instructions: Using Collections and Building Generic Types

Contents:

Lab A: Using Collections

Exercise 1: Optimizing a Method by Caching Data 3

Lab B: Building Generic Types

Exercise 1: Defining a Generic Interface 13

Exercise 2: Implementing a Generic Interface 14

Exercise 3: Implementing a Test Harness for the BinaryTree Project 20

Exercise 4: Implementing a Generic Method 22

Lab A: Using Collections

- Exercise 1: Optimizing a Method by Caching Data

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 20 minutes

Objectives

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

Introduction

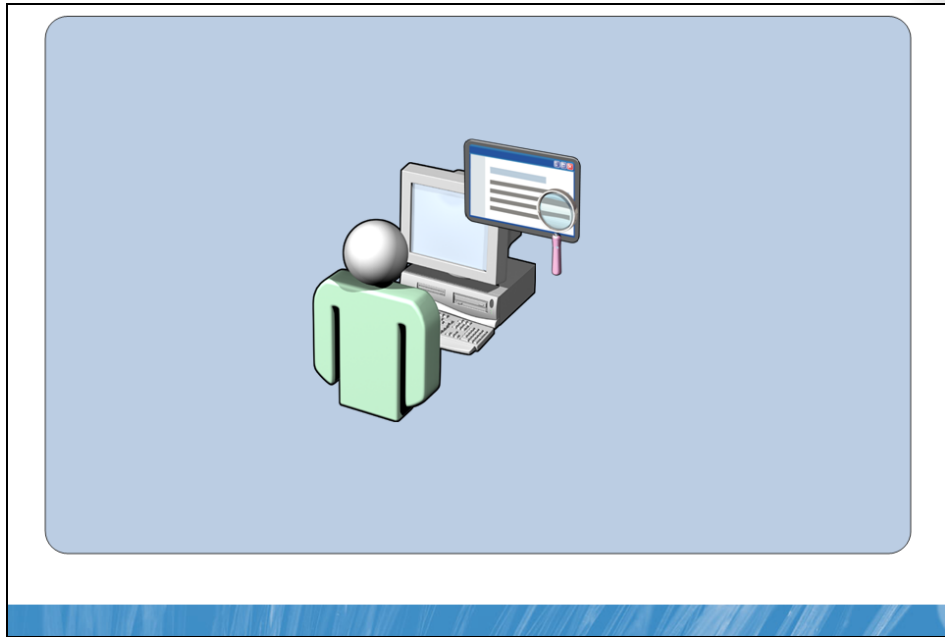
In this lab, you will use a collection to cache data and optimize an algorithm that a method implements.

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**

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:

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**.

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.

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.

Lab B: Building Generic Types

- Exercise 1: Defining a Generic Interface
- Exercise 2: Implementing a Generic Interface
- Exercise 3: Implementing a Test Harness for the BinaryTree Project
- Exercise 4: Implementing a Generic Method

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 40 minutes

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

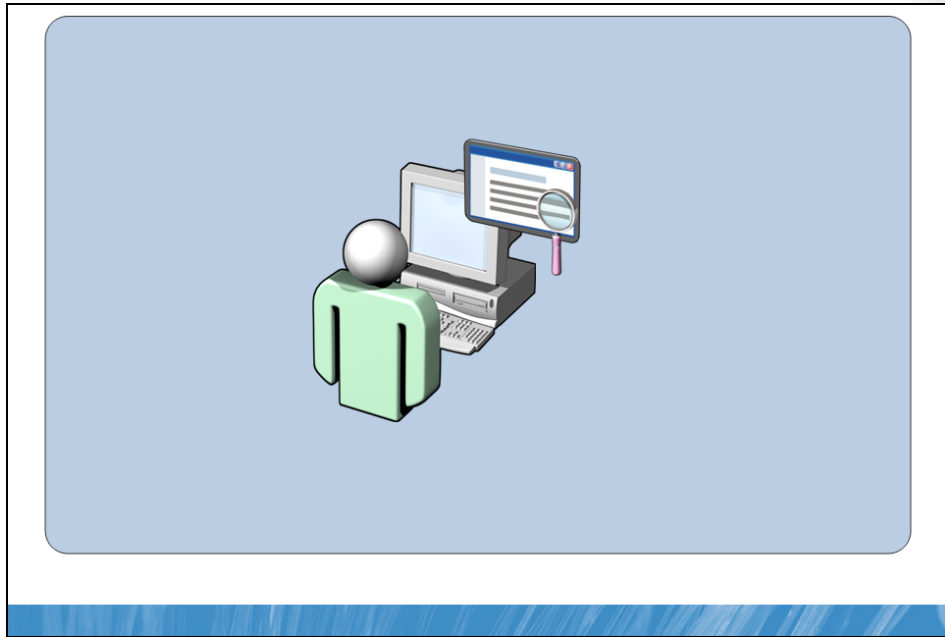
In this lab, you will build a generic collection type and write a generic method that can be used to populate an instance of this type.

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**

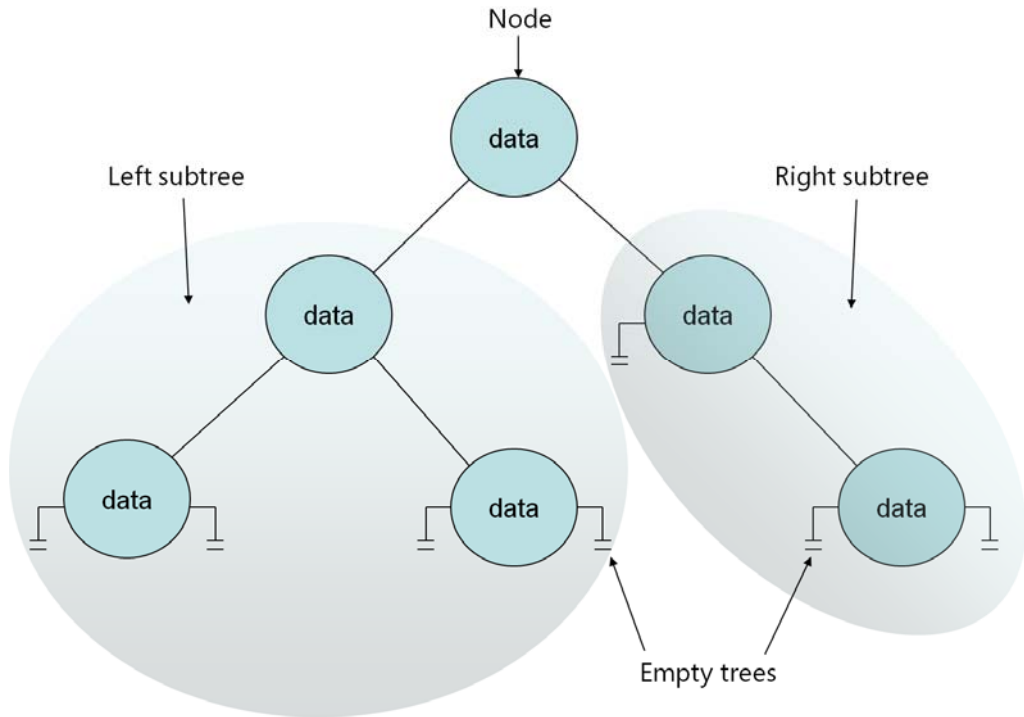
Lab Scenario



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.



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

```

```
        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.

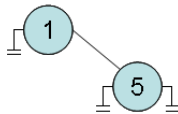


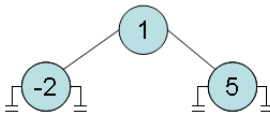
Note: The definition of the expression *greater than* depends on the type of data in the item and node. For numeric data, greater than can be a simple arithmetic comparison and for text data, it can be a string comparison. However, other forms of data must be given their own means of comparing values. In this exercise, you will use the **CompareTo** method of the **Comparable** interface to compare elements.

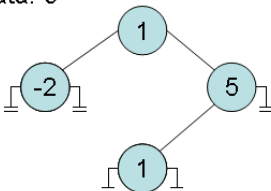
If you start with an empty binary tree and an unordered sequence of objects, you can iterate through the unordered sequence, inserting each object into the binary tree by using this algorithm, resulting in an ordered tree. The following image shows the steps in the process for constructing a tree from a set of five integers.

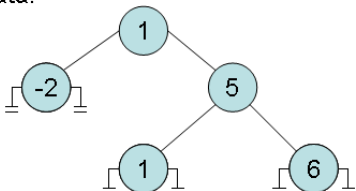
1 Data: 1, 5, -2, 1, 6
Tree: (Empty) 

2 Data: 5, -2, 1, 6


3 Data: -2, 1, 6


4 Data: 1, 6


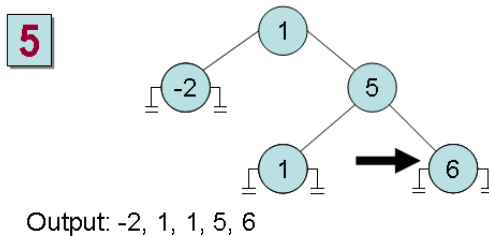
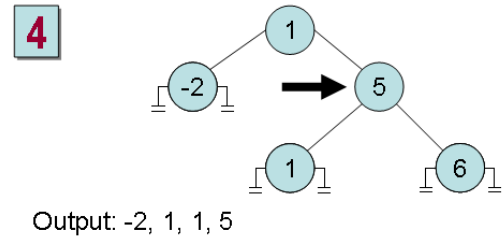
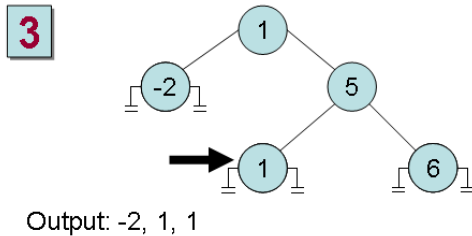
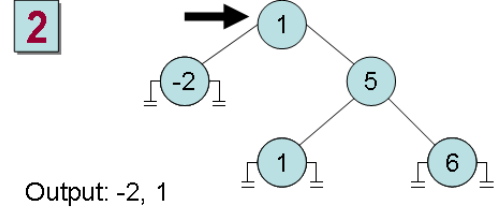
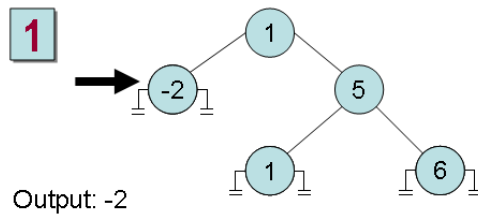
5 Data: 6


6 Data:


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.



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.

► **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 **Comparable** 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.

► 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 **Comparable** 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 **Comparable** 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;
    }
}
```

```
}

// 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.
        {

```

```

        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
    }
}

```

```

        {
            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

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.

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.

► 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 **TreelItem**, and the method should return a generic **Tree<TreelItem>** object. The **TreelItem** type parameter must represent a type that implements the generic **IComparable** interface.

The method should take two parameters: a **TreelItem** object called **nodeValue** and a **params** array of **TreelItem** 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 **TreelItem** 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.

► **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.

Module 13

Lab Instructions: Building and Enumerating Custom Collection Classes

Contents:

Exercise 1: Implementing the IList<TItem> Interface	4
Exercise 2: Implementing an Enumerator by Writing Code	18
Exercise 3: Implementing an Enumerator by Using an Iterator	28

Lab: Building and Enumerating Custom Collection Classes

- Exercise 1: Implementing the `IList<TItem>` Interface
- Exercise 2: Implementing an Enumerator by Writing Code
- Exercise 3: Implementing an Enumerator by Using an Iterator

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 90 minutes

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

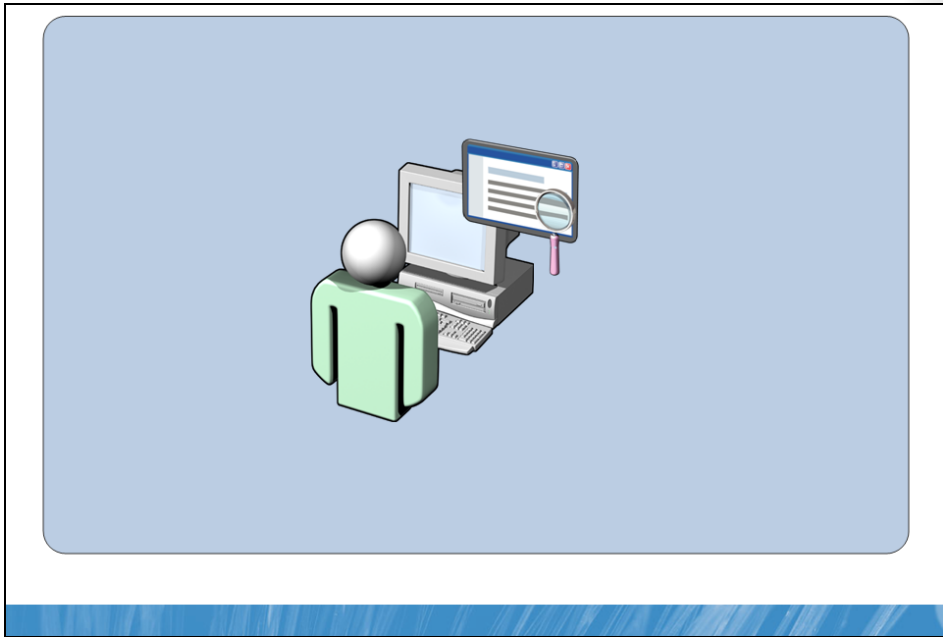
In this lab, you will implement the **`IList<T>`** interface in a custom collection class. You will then add enumerators to this collection class.

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**

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 **ICollection<T>** Interface

Collection classes that enable objects to be accessed by index should implement the **ICollection<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 **ICollection<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.

- **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 **NotSupportedException** 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\$Sw0rd**.
2. Open Microsoft Visual Studio® 2010.
3. Open the CustomCollections solution in the E:\Labfiles\Lab 13\Ex1\Starter folder.

► **Task 2: Modify the `Tree` class to implement the `ICollection<TItem>` interface**

1. Review the task list.
2. In the task list, locate the **TODO - Implement the generic `ICollection<TItem>` interface** task, and then double-click this task.

This task is located in the **`Tree`** class.
3. Remove the **TODO - Implement the generic `ICollection<TItem>` interface** comment, and then modify the class definition to implement the generic **`ICollection`** 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 **`ICollection`** 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.

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

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.

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

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

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.



Note: The **TestResult** class implements the **Comparable** 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
```



```
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
```

```
-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.

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
```

```
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.

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 **IEnumerable<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.

► **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.

► **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.

► **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
```

```
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
```

```
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
```

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
```

```
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
```

```
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.



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

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
```

```
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
```

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
```

```
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.

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
```

```
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**

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.

Module 14

Lab Instructions: Using LINQ to Query Data

Contents:

Exercise 1: Using the LINQ Query Operators

5

Exercise 2: Building Dynamic LINQ Queries

12

Lab: Using LINQ to Query Data

- Exercise 1: Using the LINQ Query Operators
- Exercise 2: Building Dynamic LINQ Queries

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

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

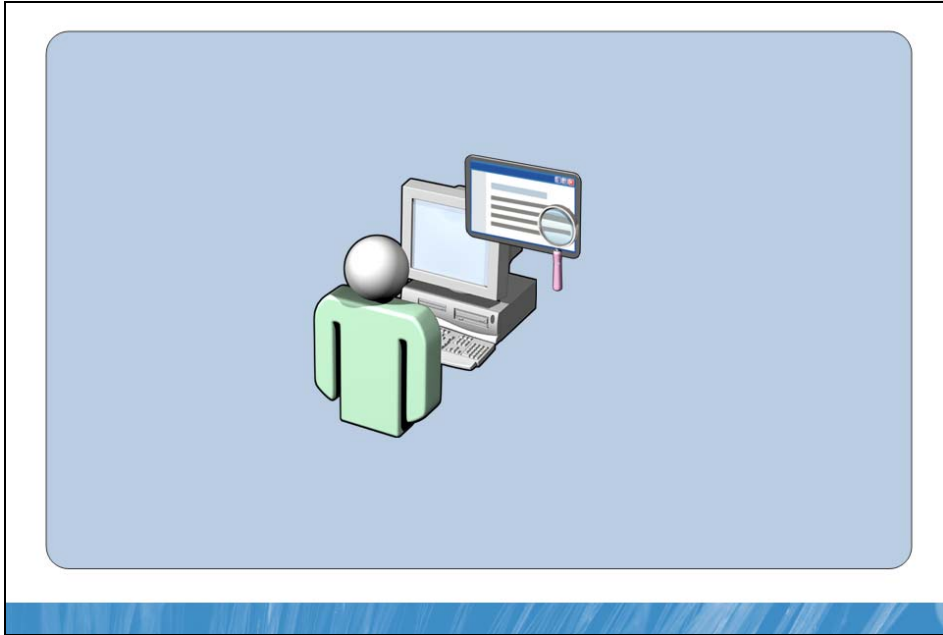
In this lab, you will use the LINQ query operators to retrieve data from a collection. You will then examine how to construct a LINQ query dynamically and optimize it for better performance.

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**

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.

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.

- **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 **Comparable** 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.
```

```
        while (readStream.Position < readStream.Length)
        {
            TResult data =
                (TResult)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.

► **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:

- 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.

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:

- 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.



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:

- 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`

- 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.

► **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.

► **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.

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:



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.

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.

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.

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.

Module 15

Lab Instructions: Integrating Visual C# Code with Dynamic Languages and COM Components

Contents:

Exercise 1: Integrating Code Written by Using a Dynamic Language into a Visual C# Application 4

Exercise 2: Using a COM Component from a Visual C# Application 13

Lab: 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
- Exercise 2: Using a COM Component from a Visual C# Application

Logon information

Virtual machine	10266A-GEN-DEV
User name	Student
Password	Pa\$\$w0rd

Estimated time: 60 minutes

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

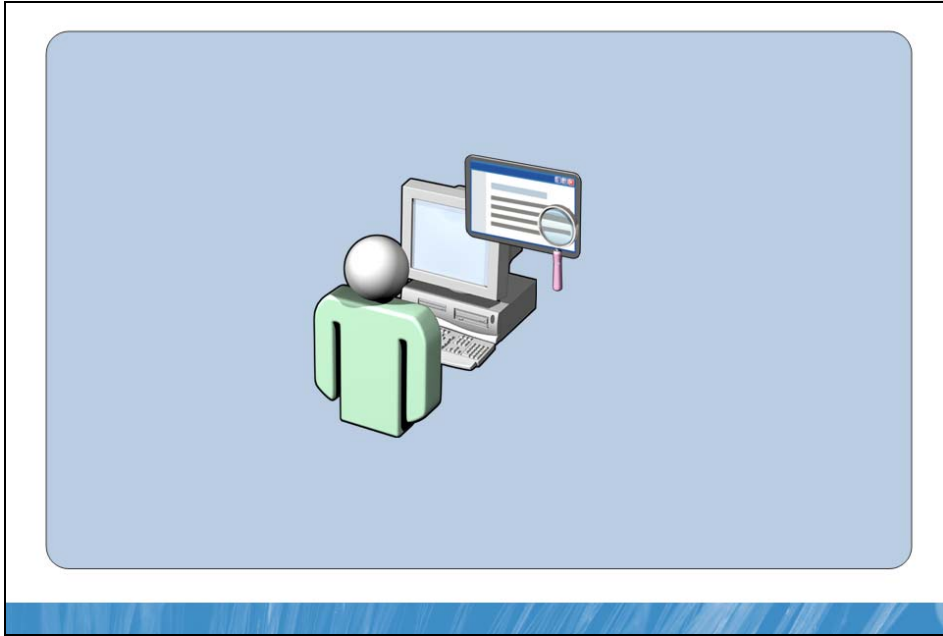
In this lab, you use the DLR to access objects defined in an IronRuby or IronPython script from a Visual C# application. You will also use COM interop to instantiate and use a COM component from a Visual C# application.

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**

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.

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.

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

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.

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.

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.

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.

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.

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.

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.

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.

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.



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.

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.

- 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
<i>Title</i>	"Applied Stress (kN) versus Deflection (mm)"
<i>Source</i>	dataRange
<i>Gallery</i>	Excel.XlChartType.xlLine
<i>PlotBy</i>	Excel.XlRowCol.xlColumns
<i>CategoryLabels</i>	1
<i>SeriesLabels</i>	1
<i>ValueTitle</i>	"Deflection"
<i>CategoryTitle</i>	"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.)

- 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.

► **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.

Module 1

Lab Answer Key: Introducing C# and the .NET Framework

Contents:

Exercise 1: Building a Simple Console Application	2
Exercise 2: Building a WPF Application	8
Exercise 3: Verifying the Application	14
Exercise 4: Generating Documentation for an Application	18

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.

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.

```
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.

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.

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.

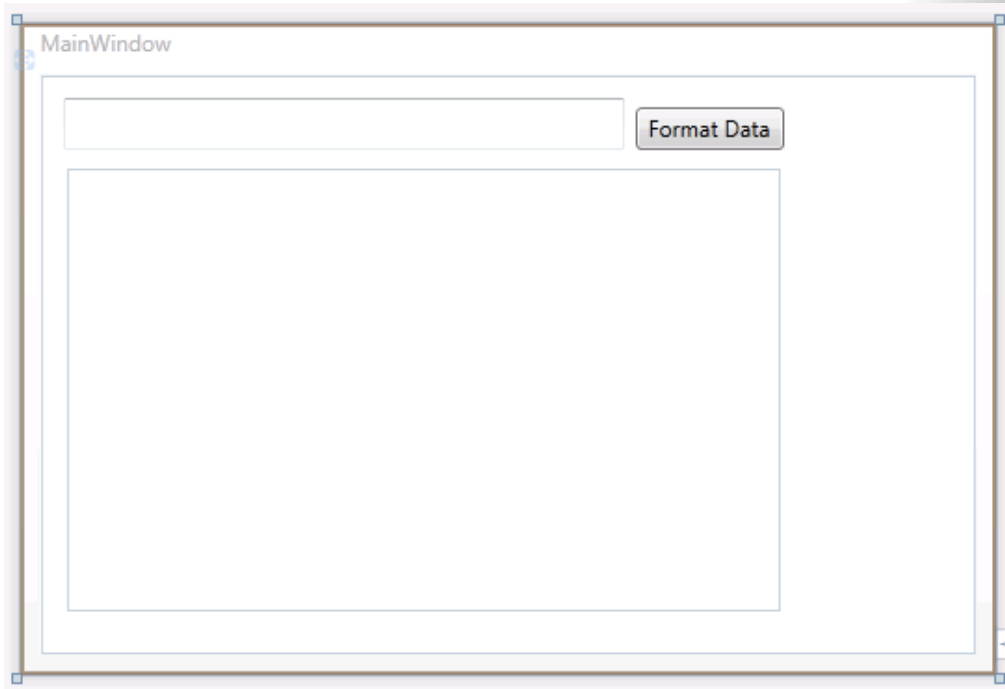
2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

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	<i>blank</i>
	VerticalAlignment	Top
	Width	384

- a. In the MainWindow window, click the **TextBox** control.

- 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.



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.

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**.

- 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.

- 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.

- 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.

- 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 WPF Application 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.


```

/// <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:

- 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
```

- b. Press ENTER when the script prompts you to do so.
5. Open the test.chm file that the bulddoc.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.

Module 2

Lab Answer Key: Using C# Programming Constructs

Contents:

Exercise 1: Calculating Square Roots with Improved Accuracy	2
Exercise 2: Converting Integer Numeric Data to Binary	14
Exercise 3: Multiplying Matrices	22

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.

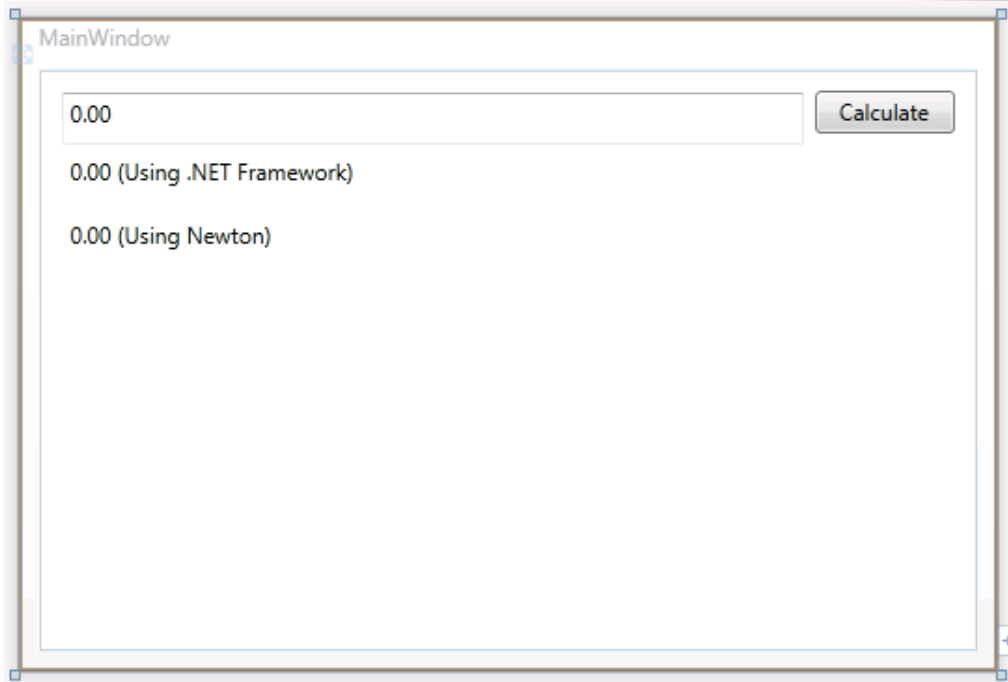
Control	Property	Value
TextBox	Name	inputTextBox
	Height	28
	HorizontalAlignment	Left
	Margin	12,12,0,0
	Text	0.00
	VerticalAlignment	Top
	Width	398
Button	Name	calculateButton
	Content	Calculate
	Height	23
	HorizontalAlignment	Right
	Margin	0,11,12,0
	VerticalAlignment	Top
	Width	75

Control	Property	Value
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

- In the MainWindow window, click the **TextBox** control.
- In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.
- In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.
- Repeat this process for the remaining properties of the **TextBox** control.
- In the MainWindow window, click the **Button** control.
- Follow the procedure described in steps b to d to set the specified properties for this control.
- 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.
 - 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;
    }
}
```

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;
    }

    // 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
frameworkLabel1.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

- On the **Debug** menu, click **Start Debugging**.
- Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.
- 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.



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 $((\text{numberDecimal} / \text{guess}) + \text{guess}) / 2$.

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 $((\text{numberDecimal} / \text{guess}) + \text{guess}) / 2$. The **while** loop

should terminate when the difference between result and guess is less than or equal to delta.



Note: Use the **Math.Abs** method to calculate the absolute value of the difference between result and guess. Using Newton's algorithm, it is possible for the difference between the two variables to alternate between positive and negative values as it diminishes. Consequently, if you do not use the **Math.Abs** method, the algorithm might terminate early with an inaccurate result.

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.

```
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
```

```
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	5.00000000000000000000000000000000
625	25	25.00000000000000000000000000000000
0.00000001	0.0001	0.0001000000000000000000000000000000
10	3.16227766016838	3.1622776601683793319988935444
8.8	2.96647939483827	2.9664793948382651794845589763
2.0	1.4142135623731	1.4142135623730950488016887242
2	1.4142135623731	1.4142135623730950488016887242

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.

Task 2: Create the user interface

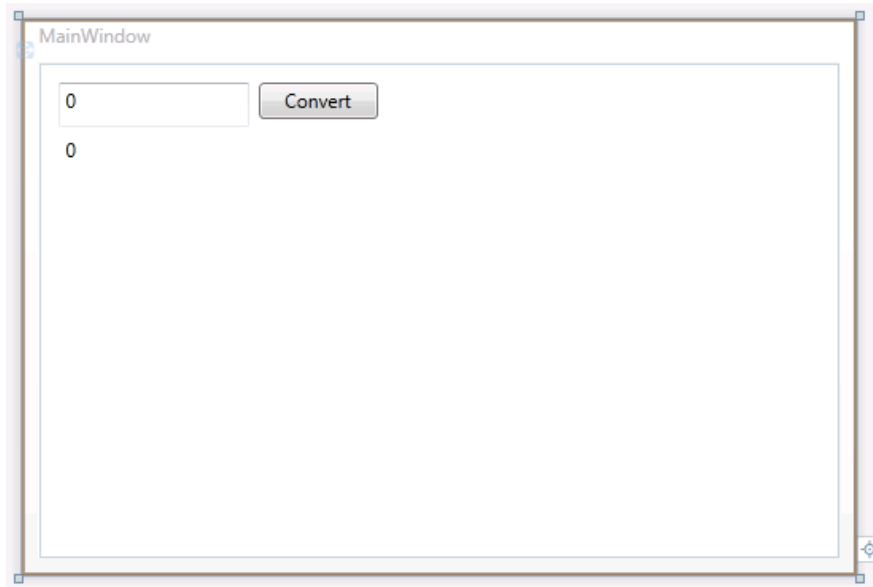
1. Add a **TextBox**, **Button**, and **Label** control to the MainWindow window. Place them anywhere in the window:
 - a. In Solution Explorer, double-click **MainWindow.xaml**.
 - b. Click the **Toolbox** tab.
 - c. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.
 - 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

- In the MainWindow window, click the **TextBox** control.
- In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.
- In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.
- Repeat this process for the remaining properties of the **TextBox** control.
- In the MainWindow window, click the **Button** control.
- Follow the procedure described in steps b to d to set the specified properties for this control.
- In the MainWindow window, click one of the **Label** controls.
- Follow the procedure described in steps b to d to set the specified properties for this control.
- In the MainWindow window, click the other **Label** control.
- 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: 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.

```
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.

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.

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();
}
```

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**.

Exercise 3: Multiplying Matrices

Task 1: Open the MatrixMultiplication project and examine the starter code

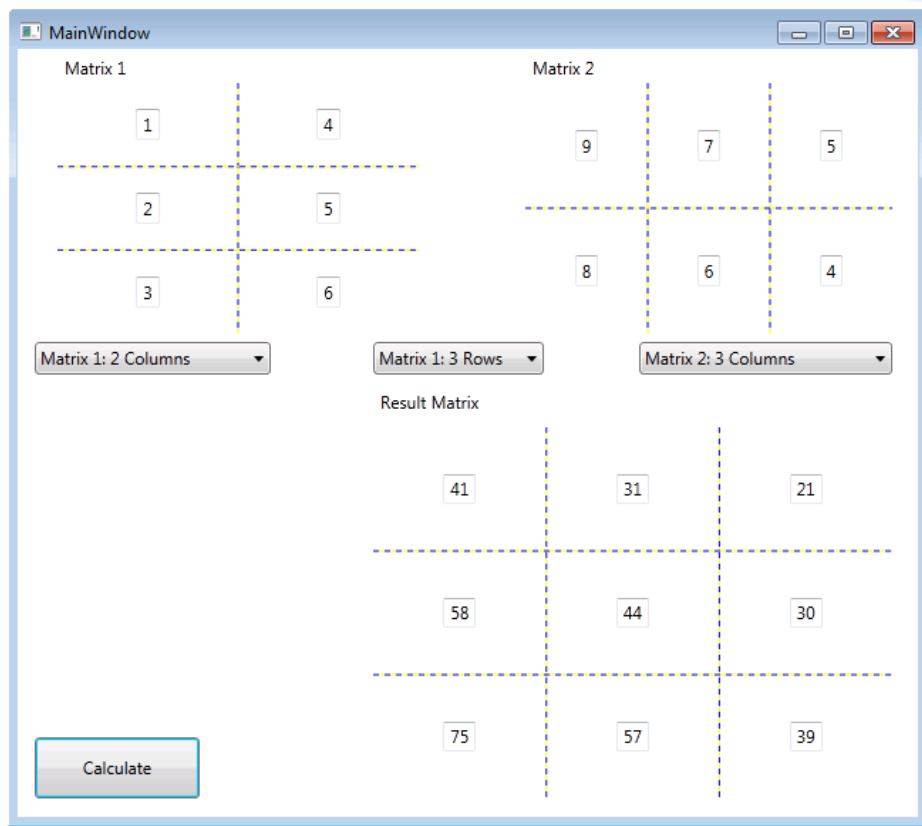
1. Open the MatrixMultiplication project located in the
E:\Labfiles\Lab 2\Ex3\Starter folder:
 - a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
 - b. In the **Open Project** dialog box, move to the
E:\Labfiles\Lab 2\Ex3\Starter folder.
 - c. Click **MatrixMultiplication.sln**, 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**.

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

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.

```
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.

```
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];
            }
        }
    }
    ...
}
```

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.

```
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.

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.

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**.

Module 3

Lab Answer Key: Declaring and Calling Methods

Contents:

Exercise 1: Calculating the Greatest Common Divisor of Two Integers by Using Euclid's Algorithm	2
Exercise 2: Calculating the GCD of Three, Four, or Five Integers	8
Exercise 3: Comparing the Efficiency of Two Algorithms	16
Exercise 4: Displaying Results Graphically	27
Exercise 5: Solving Simultaneous Equations (optional)	31

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 GCDAgorithms.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)
        {
            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.

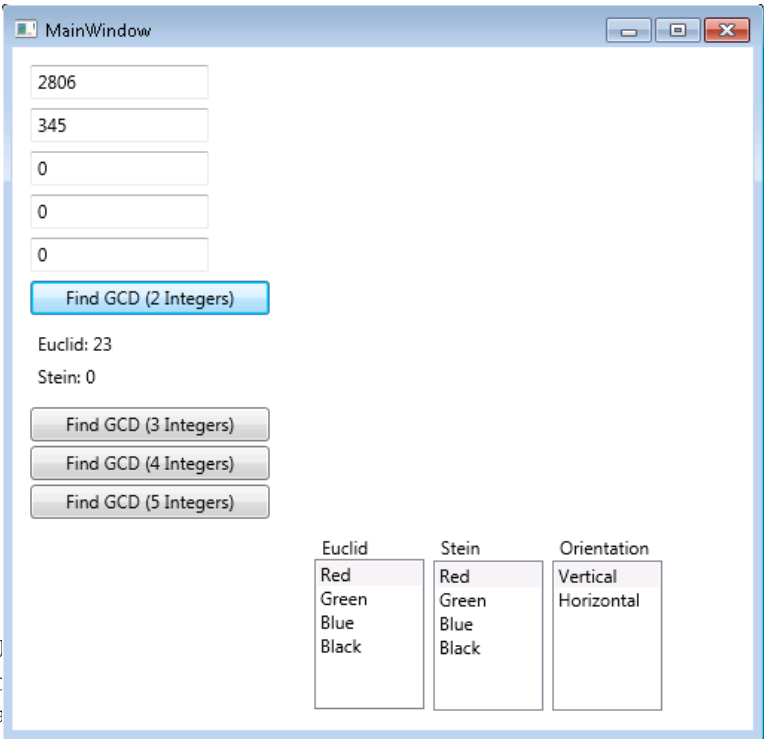
```

...
if (sender == findGCD) // Euclid for two integers
{

```

```
// Invoke the FindGCD method and display the result
this.resultEuclid.Content =
    String.Format("Euclid: {0}",
        GCDAgorithms.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.



7. Use the color selection options in the application to change the color of the text in the text boxes.

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**.
 - 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.

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 GCDAgorithms 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 **GCDAgorithms** 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 GCDAgorithms
{
    ...
    public static int FindGCDEuclid(int a, int b, int c)
    {
        ...
    }
}
...
```

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.

```
...
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)
    {
        int g = FindGCDEuclid(a, b, c, d);
        int h = FindGCDEuclid(g, e);
        return h;
    }
}
```

```

    {
        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 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}",
                GCDAgorithms.FindGCDEuclid(
                    firstNumber, secondNumber, thirdNumber));
        this.resultStein.Content = "N/A";
    }
    else if (sender == findGCD4) // Euclid for four integers
    {
        this.resultEuclid.Content =
            String.Format("Euclid: {0}",
                GCDAgorithms.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}",
                GCDAgorithms.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**.
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**.
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 = GCDAgorithms.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.

```
[TestMethod()]
public void FindGCDEuclidTest1()
{
    int a = 7396;
    int b = 1978;
    int c = 1204;
    int expected = 86;
    int actual = GCDAgorithms.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 GCDAgorithmsTest
{
    ...

    // 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 = GCDAgorithms.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;
        int actual = GCDAgorithms.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 = GCDAgorithms.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**.

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;
        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}",
            GCDAgorithms.FindGCDEuclid(firstNumber, secondNumber));
    // Call the method that implements Stein's algorithm and display
    // the results

    this.resultStein.Content =
        String.Format("Stein: {0}",

```

```

        GCDAgorithms.FindGCDStein(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 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;
}
```

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 = GCDAgorithms.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 = GCDAgorithms.FindGCDStein(u, v);
    Assert.AreEqual(expected, actual);
}
```

12. Open the Test View window and refresh the display if the unit test is not listed:
- On the **Test** menu, point to **Windows**, and then click **Test View**.
 - In the Test View window, click the **Refresh** button.
13. Run the **FindGCDSteinTest** test, and verify that the test ran successfully:
- In the Test View window, right-click the **FindGCDSteinTest** test, and then click **Run Selection**.
 - 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**.
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 stopwatch.

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();
    ...
    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**.
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}",
            GCDAgorithms.FindGCDEuclid(firstNumber, secondNumber,
                out timeEuclid), timeEuclid);
    this.resultStein.Content = String.Format("Stein: {0}, Time
        (ticks): {1}", GCDAgorithms.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}",
    //     GCDAgorithms.FindGCDEuclid(firstNumber,
```

```
//      secondNumber, thirdNumber));
    this.resultStein.Content = "N/A";
}
else if (sender == findGCD4) // Euclid for four integers
{
    //      this.resultEuclid.Content = String.Format("Euclid: {0}",
    //          GCDA1gorithms.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}",
    //          GCDA1gorithms.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 = GCDA1gorithms.FindGCDEuclid(a, b, out time);
        Assert.AreEqual(expected, actual);
    }
}

```

```
...  
[TestMethod]  
public void FindGCDSteinTest()  
{  
    int u = 298467352;  
    int v = 569484;  
    long time;  
    int expected = 4;  
    int actual = GCDAgorithms.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.

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")
{
    ...

    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.

```
...
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**.

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.

```
...  
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:
- Create and initialize an array with the same number of columns and rows as the array that was passed in.
 - Copy the values in the array that was passed in as the parameter into the new array.
 - 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;
}

...

```

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;

    ...

```

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 **SolveGaussian** 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.

Module 4

Lab Answer Key: Handling Exceptions

Contents:

Exercise 1: Making a Method Fail-Safe	2
Exercise 2: Detecting an Exceptional Condition	10
Exercise 3: Checking for Numeric Overflow	16

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() ==
        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 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).

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";
    }
}

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.

```
...
```

```
// 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 +=
    "\nText sequence complete: " +
    DateTime.Now.ToLongTimeString();
...
```

12. Build the solution and correct any errors:

- On the **Build** menu, click **Build Solution**.

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:

- 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];
...
```

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.

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.

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.

Module 5

Lab Answer Key: Reading and Writing Files

Contents:

Exercise 1: Building a Simple File Editor

2

Exercise 2: Making the Editor XML Aware

10

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 **openFileDialog**.
 - b. Set the **InitialDirectory** property of **openFileDialog** to point to the E:\Labfiles\Lab 5\Ex1\Starter folder.



Note: When including file paths in code, you should prefix the string with the @ symbol. This symbol instructs the C# compiler to treat any \" characters as literals rather than escape characters.

- c. Set the **DefaultExt** property of **openFileDialog** to ".txt";.
- d. Set the **Filter** property of **openFileDialog** to "Text Documents (*.txt)|*.txt".

Your code should resemble the following code example.

```
...
    string fname = string.Empty;

    OpenFileDialog openFileDialog = new OpenFileDialog();

    openFileDialog.InitialDirectory =
        @"E:\Labfiles\Lab 5\Ex1\Starter";

    openFileDialog.DefaultExt = ".txt";
    openFileDialog.Filter = "Text Documents (*.txt)|*.txt";
}
...
```

8. Add code to perform the following tasks:
 - a. Call the **ShowDialog** method of **openFileDialog**, 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 **openFileDialog** to the **fname** variable.

Your code should resemble the following code example.

```
...
    bool? result = openFileDialog.ShowDialog();

    if (result == true)
    {
        fname = openFileDialog.FileName;
    }
}
...
```

9. At the end of the method, return the value in the `fname` variable.

Your code should resemble the following code example.

```
...
    fname = openFileDialog.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.

```
...
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 string parameter named *fileName*.
 - 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)
    {
    }
}
...
```

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.

```
...
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:

- 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.

- 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.

```
...  
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.

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.



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(">");
        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.

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 &+ x &lt; 20 Sub x, y  
Store 30
```

This is the text from the Commands.txt file. Notice that the **<**, **>**, and **&** characters have been replaced with the text **<**, **>**, and **&+&**.

8. Close the MainWindow window and return to Visual Studio.

Task 4: Implement test cases

1. In the task list, locate the **TODO - Complete Unit Tests** task. Double-click this task.

This task is located in the **TextFileOperationsTest** class.

2. Examine the **ReadAndFilterTextFileContentsTest** method, and then uncomment the commented line.

This method creates three strings:

- a. The **filename** string contains the path of a prewritten file that contains specific content.
- b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.
- c. The **actual** string is initialized by calling the **ReadAndFilterTextFileContents** method that you just implemented.

The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

This method is complete, and requires no further work:

- 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.

Module 6

Lab Answer Key: Creating New Types

Contents:

Exercise 1: Using Enumerations to Specify Domains	2
Exercise 2: Using a Struct to Model a Simple Type	9
Exercise 3: Using a Class to Model a More Complex Type	12
Exercise 4: Using a Nullable Struct	25

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:
- IBeam**
 - Box**
 - ZShaped**
 - 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:
- 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.
 - 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.

Material enumeration value	<i><selectedMaterial></i> string
Material.StainlessSteel	Stainless Steel

Material enumeration value	<selectedMaterial> string
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.

Material enumeration value	<selectedCrossSection> string

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

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;
}
...
```

- 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

- Build the application and correct any errors:
 - On the **Build** menu, click **Build Solution**. Correct any errors.
- Run the application:
 - On the **Debug** menu, click **Start Without Debugging**.
- 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.
- Experiment by selecting further values from all three lists, and verify that with each change, the label updates to reflect the changes.
- Close the application, and then return to Visual Studio.

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;
}
...
```

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.

```
...
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**.

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

- Below the member declarations, add a constructor for the `StressTestCase` class that accepts the following parameters:
 - A **Material** object named `girderMaterial`.
 - A **CrossSection** object named `crossSection`.
 - An integer named `lengthInMm`.
 - An integer named `heightInMm`.
 - 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;  
}  
...
```

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
<i>girderMaterial</i>	Material.StainlessSteel
<i>crossSection</i>	CrossSection.IBeam
<i>lengthInMm</i>	4000
<i>heightInMm</i>	20
<i>widthInMm</i>	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)
{
}
...
```

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)
{
    ...
}
```

```
else
{
    TestCaseResult.Result = TestResult.Pass;
}

...
```

- 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()
    {
    }
}
...
```

- 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

- Below the **GetStressTestResult** method, add the following public method named **ToString**.



Note: This overrides the **ToString** method that is inherited from the **object** type. You will see more about inheritance in a later module.

```
...
public class StressTestCase
{
    ...
    public override string ToString()
    {
    }
}
...
```

2. In the **ToString** method, add code to return a string with the format shown in the following code example, where each value in angle brackets is replaced with the corresponding member in the class.

```
Material: <girderMaterial>, CrossSection: <crossSection>, Length:
<lengthInMm>mm, Height: <heightInMm>mm, Width:<widthInMm>mm.
```



Hint: Use the **String.Format** method to build the string.

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.

- 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

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()
    {
```

```

    ...
    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;
}
...

```

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.

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:

- 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;  
...
```

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

- 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",
    ...
}
...
```

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

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 **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**.

Module 7

Lab Answer Key: Encapsulating Data and Methods

Contents:

Exercise 1: Hiding Data Members	2
Exercise 2: Using Static Members to Share Data	7
Exercise 3: Implementing an Extension Method	14

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**.

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);
}
```

```
//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.

```

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**.

- 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;  
}  
}
```

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.

```
public struct TestStatistics
{
    ...

    public int GetNumberOfFailures()
    {
        return numberOfFailures;
    }
}
```

- 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;
    }
}
```

- 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

- 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.
- 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.

```
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();
    }
    ...
}
```


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.

- 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());
}
```

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.

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;
}
```

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

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.

Module 8

Lab Answer Key: Inheriting from Classes and Implementing Interfaces

Contents:

Exercise 1: Defining an Interface	2
Exercise 2: Implementing an Interface	5
Exercise 3: Creating an Abstract Class	18

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();
```

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
{
    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:

- 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
```

```
{  
    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:

- 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);  
}
```

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

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

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

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.

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;
            }
        }
    });
}
```

```
        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 **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 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**.

- 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.

```
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.

- 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;
```

- 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;
```

- 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.

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

- 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**

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);
    }
}
```

```

    }

    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
{
    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.
            }
        }
    }
}
```

```
        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)  
                return;  
        }  
    });  
}
```

```

        {
            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);
    }
    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.

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.

```
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:

- 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;
    }
}
```

- 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) =>
    {
```



```

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

- 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.

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.

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;
```

- 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;
}
```

- 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()
{
    ...
}
```

- 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()
{
    ...
}
```

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:

- 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.

```
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.

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**.

Module 9

Lab Answer Key: Managing the Lifetime of Objects and Controlling Resources

Contents:

Exercise 1: Implementing the IDisposable Interface

2

Exercise 2: Managing Resources Used by an Object

14

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();
    }
}
```

- -
 -
 -
 -
 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.

```
...
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.

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.

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.

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
{
```

```

public abstract class MeasureDataDevice
    : ILoggerMeasuringDevice, 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");
            }
        }
    }
}

```

```
        }
    }
    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)
```

```

        {
            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 IDisposable interface.

```

```
/// </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.

```

/// <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.

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.

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.

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.

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**.

Module 10

Lab Answer Key: Encapsulating Data and Defining Overloaded Operators

Contents:

Lab A: Creating and Using Properties

Exercise 1: Defining Properties in an Interface 2

Exercise 2: Implementing Properties in a Class 4

Exercise 3: Using Properties Exposed by a Class 8

Lab B: Creating and Using Indexers

Exercise 1: Implementing an Indexer to Access Bits in a Control Register 14

Exercise 2: Using an Indexer Exposed by a Class 16

Lab C: Overloading Operators

Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types 23

Exercise 2: Implementing Operators for the Matrix Type 32

Exercise 3: Testing the Operators for the Matrix Type 40

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**.

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**.
 - 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.

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.

```
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();
    // 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.

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.

```
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 **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.
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**.

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)  
        {  
            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.

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());
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.

```
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());
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
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**.

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];
    }
    ...
}
```

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.

```

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
    {
        ...
    }
}

```

```
    }  
    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  
    {  
    }
```

```

        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)
{
}

```

```

}
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

```



```
{
    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();
}
```

```
    }  
}  
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;  
    }  
}  
}
```

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.

```

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
    {

```

```
        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  
    {  
    }  
}
```

```

        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];
            }
        }
    }
}

```

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

```
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:

- 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.

```
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;
```

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.

4. Close Visual Studio:
 - In Visual Studio, on the **File** menu, click **Exit**.

Module 11

Lab Answer Key: Decoupling Methods and Handling Events

Contents:

Exercise 1: Raising and Handling Events

2

Exercise 2: Using Lambda Expressions to Specify Code

17

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.
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()
{
    }
} ...
```

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;

///  
...

```

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.

```
...  
  
    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)  
    {  
    }  
}  
  
...
```

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.


```
...
    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:

- Increment the integer **i**.
- 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.

```
...
    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.

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();
    }
}
...
```

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)
...
```

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();  
  
...
```

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;
}
...
```

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.

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);
...
```

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.
...
```

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;
```

```
// 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.

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

- 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

- 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.
- 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

- 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.

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;
}
...
```

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

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**.

Module 12

Lab Answer key: Using Collections and Building Generic Types

Contents:

Lab A: Using Collections

Exercise 1: Optimizing a Method by Caching Data 2

Lab B: Building Generic Types

Exercise 1: Defining a Generic Interface 8

Exercise 2: Implementing a Generic Interface 10

Exercise 3: Implementing a Test Harness for the BinaryTree Project 19

Exercise 4: Implementing a Generic Method 22

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.

- 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  
  
...
```

- 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>  
    ...  
}  
  
...
```

- 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)  
{  
    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)
    {
        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;
    }
}
...

```

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.

- 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

- Build the solution and correct any errors:
 - On the **Build** menu, click **Build Solution**. Correct any errors.
- Run the application:
 - On the **Debug** menu, click **Start Without Debugging**.
- 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.

- 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**.

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.

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.

```
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.

- 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)
```



```

    {
        ...
    }

    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.

```
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.
        }
    }
}
```

```

        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.

```
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.

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.

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);  
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.

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.

- 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.

```
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**.

Module 13

Lab Answer Key: Building and Enumerating Custom Collection Classes

Contents:

Exercise 1: Implementing the IList<TItem> Interface	2
Exercise 2: Implementing an Enumerator by Writing Code	19
Exercise 3: Implementing an Enumerator by Using an Iterator	32

Lab 13: Building and Enumerating Custom Collection Classes

Exercise 1: Implementing the `ICollection<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 `ICollection<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 `ICollection<TItem>` interface** task, and then double-click this task.

This task is located in the **Tree** class.
3. Remove the **TODO - Implement the generic `ICollection<TItem>` interface** comment, and then modify the class definition to implement the generic **ICollection** 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 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;
}
```

- 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;
    ...
}
```

- 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.

- 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;
    ...
}
```

- 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.

- 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
}
```

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:
 - 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 **ICollection<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 **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;
}
```

- 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.
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)
    {
        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.

```
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:
- If the tree contains a node that matches the value in the *item* parameter, call the local **Remove** method, and then return **true**.
 - 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**.

Task 5: Use the BinaryTreeTestHarness application to test the solution

- 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.



Note: The **TestResult** class implements the **Comparable** 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
```

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.

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
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
```

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.

- 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.
 - 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.
 - 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.

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();
        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.

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.
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.

```
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
```

```
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  
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
```

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
```

```
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
```

```

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.

- In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
- In the Open Project dialog box, move to the E:\Labfiles\Lab 13\Ex3\Starter folder, click CustomCollections.sln, and then click Open.

Task 2: Add an enumerator to return an enumerator that iterates through data in reverse order

- Review the task list:
 - 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
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  
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.

```
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
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.

```
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

```

```
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.

12. Press ENTER twice to return to Visual Studio.

Module 14

Lab Answer Key: Using LINQ to Query Data

Contents:

Exercise 1: Using the LINQ Query Operators

2

Exercise 2: Building Dynamic LINQ Queries

13

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.

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.

```
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.

```

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.

```

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.
 - 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)
{
    ...
    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",
            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.
        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;
            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 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.
 - 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:
 - 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.

```
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);
```

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);
```

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)
{
    ...
    // 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
    //
    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 =
        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**".

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.

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.

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

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)
    {

```



```
{
    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:



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.

```

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.

```

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.

```

```

        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:

- 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.

8. Close Visual Studio:
 - In Visual Studio, on the **File** menu, click **Exit**.

Module 15

Lab Answer Key: Integrating Visual C# Code with Dynamic Languages and COM Components

Contents:

Exercise 1: Integrating Code Written by Using a Dynamic Language into a Visual C# Application	2
Exercise 2: Using a COM Component from a Visual C# Application	16

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\$Sword**.
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.

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

- 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.

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);
}
```

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.

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.
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.

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.

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.

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.

```
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.
    // 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**.

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 catches 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.

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.

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.



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.
 - 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;
    }
}
```

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:
- 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
<i>Title</i>	"Applied Stress (kN) versus Deflection (mm)"
<i>Source</i>	dataRange
<i>Gallery</i>	Excel.XlChartType.xlLine
<i>PlotBy</i>	Excel.XlRowCol.xlColumns
<i>CategoryLabels</i>	1
<i>SeriesLabels</i>	1
<i>ValueTitle</i>	"Deflection"
<i>CategoryTitle</i>	"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");
    ...
}
```

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**.



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.

Your 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.

```
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);
    }
}
```

```
}  
}
```

- 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();  
    }  
}
```

- Build the solution and correct any errors:
 - On the **Build** menu, click **Build Solution**.

Task 7: Test the application

- Start the application in Debug mode:
 - On the **Debug** menu, click **Start Debugging**.
- In the Graphing Data window, click **Get Data**.
- In the **Graph Data** dialog box, click the **298K.txt** file, and then click **Open**.
- In the Graphing Data window, in the tree view, expand the **Temperature: 298K** node. Verify that the data has been correctly loaded.
- 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**.