Encapsulating Data and Defining Overloaded Operators

Lab A: Creating and Using Properties

Exercise 1: Defining Properties in an Interface

Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point **to All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 10\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** list, select **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 10\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab A\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab A\Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

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**.

Encapsulating Data and Defining Overloaded Operators

Lab B: Creating and Using Indexers

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

Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab B\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab B\Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Add an indexer to the ControlRegister class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the ControlRegister.cs file:
   - In Solution Explorer, double-click **ControlRegister.cs**.
3. Remove the comment **TODO: Add an indexer to enable access to individual bits in the control register** and add a **public** indexer to the class. The indexer should take an **int** called **index** as the parameter and return an **int**.

   Your code should resemble the following code example.

```
public int this[int index]
{
}
```

4. Add a **get** accessor to the indexer. In the **get** accessor, add code to determine whether the bit specified by the *index* parameter in the **registerData** object is set to 1 or 0 and return the value of this bit.

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

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

Your code should resemble the following code example.

```
public int this[int index]
{
    get
    {
        bool isSet = (registerData & (1 << index)) != 0;
        return isSet ? 1 : 0;
    }
}
```

5. Add a **set** accessor to the indexer. In the **set** accessor, add code to verify that the parameter specified is either 1 or 0. Throw an **ArgumentException** exception with the message "Argument must be 1 or 0" if it is not one of these values.

Your code should resemble the following code example.

```
public int this[int index]
{
    get
    {
        bool isSet = (registerData & (1 << index)) != 0;
        return isSet ? 1 : 0;
    }
    set
    {
        if (value != 0 && value !=1)
```

```
        {
            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**.

Encapsulating Data and Defining Overloaded Operators

Lab C: Overloading Operators

Exercise 1: Defining the Matrix and MatrixNotCompatibleException Types

Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Module10 solution in the E:\Labfiles\Lab 10\Lab C\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 10\Lab C \Ex1\Starter** folder, click **Module10.sln**, and then click **Open**.

Task 2: Create a Matrix class

1. Review the task list:
   a. If the task list is not already visible, on the **View** menu, click **Task List**.
   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.
2. Open the Matrix.cs file:
   - In Solution Explorer, double-click **Matrix.cs**.
3. Remove the comment **TODO: Add the Matrix class** and add a **public Matrix** class to the **MatrixOperators** namespace.

   Your code should resemble the following code example.

```
namespace MatrixOperators
{
    public class Matrix
    {
        // TODO Add an addition operator to the Matrix class.

        // TODO Add a subtraction operator to the Matrix class.

        // TODO Add a multiplication operator to the Matrix class.
    }

    // TODO: Add the MatrixNotCompatibleException exception class.

}
```

4. Add a two-dimensional array of integers named **data** to the **Matrix** class.

   Your code should resemble the following code example.

```
public class Matrix
{
    int[,] data;

    // TODO Add an addition operator to the Matrix class.

    // TODO Add a subtraction operator to the Matrix class.

    // TODO Add a multiplication operator to the Matrix class.
}
```

5. Add a **public** constructor to the **Matrix** class. The constructor should take a single integer parameter called *size* and initialize the **data** array to a square array by using the value passed to the constructor as the size of each dimension of the array.

   Your code should resemble the following code example.

```
public class Matrix
{
    int[,] data;

    public Matrix(int size)
    {
        data = new int[size, size];
    }
    ...
}
```

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**.