Inheriting from Classes and Implementing Interfaces

Lab 8: Inheriting from Classes and Implementing Interfaces

Exercise 1: Defining an Interface

Task 1: Open the starter project

1. Log on to the 10266A-GEN-DEV machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 8\Snippets folder.
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** drop-down, click **Visual C#**.
   c. In the **Code Snippets Manager** dialog box, click **Add**.
   d. In the Code Snippets Directory dialog box, browse to the E:\Labfiles\Lab 8\Snippets folder, and then click Select Folder.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the Module8 solution in the E:\Labfiles\Lab 8\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, in the **File name** box, move to the **E:\Labfiles\Lab 8\Ex1\Starter** folder, click **Module8.sln**, and then click **Open**.

Task 2: Create the IMeasuringDevice interface

1. Open the IMeasuringDevice code file:

- In Solution Explorer, double-click **IMeasuringDevice.cs**.

2. In the **MeasuringDevice** namespace, declare the **IMeasuringDevice** interface. The **IMeasuringDevice** interface must be accessible to code in other assemblies.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
    }
}
```

3. Add a method named **MetricValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method.

   Your code should resemble the following code example.

```
namespace MeasuringDevice
{
    public interface IMeasuringDevice
    {
        /// <summary>
        /// Converts the raw data collected by the measuring device
        /// into a metric value.
        /// </summary>
        /// <returns>The latest measurement from the device converted
        /// to metric units.</returns>
        decimal MetricValue();
    }
}
```

4. Add a method named **ImperialValue** that returns a **decimal** value to the interface. The method should take no parameters. Add a comment that describes the purpose of the method:

   - Add the code in the following code example to the interface.

```
/// <summary>
/// Converts the raw data collected by the measuring device into an
/// imperial value.
/// </summary>
/// <returns>The latest measurement from the device converted to
/// imperial units.</returns>
decimal ImperialValue();
```

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.

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

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

Your code should resemble the following code example.

```
private Units unitsToUse;
private int[] dataCaptured;
private int mostRecentMeasure;
private DeviceController controller;
private DeviceType measurementType;
```

8. Modify the **measurementType** field to make it constant and initialize it to **DeviceType.MASS**.

Your modified code should resemble the following code example.

```
private const DeviceType measurementType = DeviceType.MASS;
```

9. Locate the **StartCollecting** method, and then remove the default method body that Visual Studio inserts, which throws a **NotImplementedException** exception. Add code to instantiate the **controller** field by using the static **StartDevice** method of the **DeviceController** class. Pass the **measurementType** field as the parameter to the **StartDevice** method.

Your code should resemble the following code example.

```
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)
```

```
                            {
                                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;
```

7. Modify the constructor to set the **measurementType** field to **DeviceType.LENGTH**:
   - Modify the code to resemble the following code example.

```
public MeasureLengthDevice(Units deviceUnits)
{
    unitsToUse = deviceUnits;
    measurementType = DeviceType.LENGTH;
}
```

8. Modify the **MetricValue** method signature to indicate that it overrides the abstract method in the base class:
   - Modify the code to resemble the following code example.

```
public override decimal MetricValue()
{
   ...
}
```

9. Modify the **ImperialValue** method signature to indicate that it overrides the abstract method in the base class:
   - Modify the code to resemble the following code example.

```
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**.