Lab 11: Decoupling Methods and Handling Events

Exercise 1: Raising and Handling Events

Task 1: Open the Events solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Events solution in the E:\Labfiles\Lab 11\Ex1\Starter folder:

   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 11\Ex1 \Starter** folder, click **Events.sln**, and then click **Open**.

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

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

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

   b. In the **Add New Item - MeasuringDevice** dialog box, under **Installed Templates**, click **Code**, and in the template list, click **Interface**.

   c. In the **Name** box, type **IEventEnabledMeasuringDevice.cs** and then click **Add**.

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

2. Modify the interface definition so that the **IEventEnabledMeasuringDevice** interface extends the **IMeasuringDevice** interface.

   Your code should resemble the following code example.

```
...
interface IEventEnabledMeasuringDevice : IMeasuringDevice
{
}
...
```

3. In the **IEventEnabledMeasuringDevice** interface, add an event named **NewMeasurementTaken** by using the base **EventHandler** delegate.

   Your code should resemble the following code example.

```
...
interface IEventEnabledMeasuringDevice : IMeasuringDevice
{
    event EventHandler NewMeasurementTaken;
}
...
```

4. Build the application to enable Microsoft IntelliSense to reflect your changes:

   - On the **Build** menu, click **Build Solution**, and then correct any errors.

Task 3: Add the NewMeasurementTaken event to the MeasureDataDevice class

1. Review the task list:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

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

3. Remove the **TODO - Modify the class definition to implement the extended interface** comment, and then modify the class definition to implement the **IEventEnabledMeasuringDevice** interface instead of the **IMeasuringDevice** interface:

   - At the top of the code file, in the class definition, replace **IMeasuringDevice** with **IEventEnabledMeasuringDevice**.

   Your code should resemble the following code example.

```
...
public abstract class MeasureDataDevice :
    IEventEnabledMeasuringDevice, IDisposable
{ ...
```

4. In the task list, locate the **TODO - Add the NewMeasurementTaken event** task, and then double-click this task. This task is located at the end of the **MeasureDataDevice** class.

5. Remove the **TODO - Add the NewMeasurementTaken event** comment, and then declare an event named **NewMeasurementTaken** by using the same signature as the interface.

   Your code should resemble the following code example.

```
...
// class implementation of the NewMeasurementTaken event.
public event EventHandler NewMeasurementTaken;

// TODO - Add an OnMeasurementTaken method.
...
```

6. Below the event, remove the **TODO - Add an OnMeasurementTaken method** comment, and then add a protected virtual method named **OnNewMeasurementTaken**. The method should accept no parameters and have a void return type. The **MeasureDataDevice** class will use this method to raise the **NewMeasurementTaken** event.

   Your code should resemble the following code example.

```
...
    // Method to raise the NewMeasurementTaken event.
    protected virtual void OnNewMeasurementTaken()
    {
    }
} ...
```

7. In the **OnNewMeasurementTaken** method, add code to check that there is a subscriber for the **NewMeasurementTaken** event; if so, raise the event. The signature of the **EventHandler** delegate defines two parameters: an *object* parameter that indicates the object that raised the event and an *EventArgs* parameter that provides any additional data that is passed to the event handler. Set the *object* parameter to **this** and the *EventArgs* parameter to **null**.

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

Your code should resemble the following code example.

```
...
protected virtual void OnNewMeasurementTaken()
{
    if (NewMeasurementTaken != null)
    {
        NewMeasurementTaken(this, null);
    }
}
...
```

Task 4: Add a BackgroundWorker member to the MeasureDataDevice class

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

   Your code should resemble the following code example.

```
...
// BackgroundWorker member to generate measurements.
private BackgroundWorker dataCollector;

/// <summary>
...
```

Task 5: Add the GetMeasurements method to the MeasureDataDevice class

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

1. In the task list, locate the **TODO - Implement the GetMeasurements method** task, and then double-click this task.

2. Remove the **TODO - Implement the GetMeasurements method** comment, and then add a new private method named **GetMeasurements** to the class. This method should take no parameters and not return a value.

   Your code should resemble the following code example.

```
...
// Add a GetMeasurements method to configure and start the
// BackgroundWorker.

private void GetMeasurements()
{
}
...
```

3. In the **GetMeasurements** method, add code to perform the following actions:

   a. Instantiate the **dataCollector BackgroundWorker** member.

   b. Specify that the **dataCollector BackgroundWorker** member supports cancellation.

   c. Specify that the **dataCollector BackgroundWorker** member reports progress while running.

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

   Your code should resemble the following code example.

```
...
private void GetMeasurements()
{
    dataCollector = new BackgroundWorker();
    dataCollector.WorkerSupportsCancellation = true;
    dataCollector.WorkerReportsProgress = true;
} ...
```

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

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

```
...
    dataCollector.WorkerReportsProgress = true;

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

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

   Your code should resemble the following code example.

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

    dataCollector.ProgressChanged += new
        ProgressChangedEventHandler(dataCollector_ProgressChanged);
}

...
```

6. Add code to start the **dataCollector BackgroundWorker** object running asynchronously.

   Your code should resemble the following code example.

```
...

    dataCollector.ProgressChanged += new
        ProgressChangedEventHandler(dataCollector_ProgressChanged);

    dataCollector.RunWorkerAsync();

}

...
```

Task 6: Implement the dataCollector_DoWork method

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

   This method was generated during the previous task. It runs on a background thread, and its purpose is to collect and store measurement data.

2. In the **dataCollector_DoWork** method, remove the statement that raises the **NotImplementedException** exception and add code to perform the following actions:

   a. Instantiate the **dataCaptured** array with a new integer array that contains 10 items.

   b. Define an integer i with an initial value of zero. You will use this variable to track the current position in the **dataCaptured** array.

   c. Add a **while** loop that runs until the **dataCollector**.**CancellationPending** property is **false**.

   Your code should resemble the following code example.

```
...
void dataCollector_DoWork(object sender, DoWorkEventArgs e)
{

    dataCaptured = new int[10];

    int i = 0;
    while (!dataCollector.CancellationPending)
    {
    }
}
...
```

3. In the **while** loop, add code to perform the following actions:

   a. Invoke the **controller**.**TakeMeasurement** method, and store the result in the **dataCaptured** array at the position that the integer **i** indicates. The **TakeMeasurement** method of the **controller** object blocks until a new measurement is available.

   b. Update the **mostRecentCapture** property to contain the value in the **dataCaptured** array at the position that the integer **i** indicates.

   c. If the value of the disposed variable is **true**, terminate the **while** loop. This step ensures that the measurement collection stops when the **MeasureDataDevice** object is destroyed.

   Your code should resemble the following code example.

```
...

while (!dataCollector.CancellationPending)
{
    dataCaptured[i] = controller.TakeMeasurement();
    mostRecentMeasure = dataCaptured[i];

    if (disposed)
    {
        break;
    }
}
...
```

4. Add code to the **while** loop after the statements that you added in the previous step to perform the following actions:

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

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

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

Your code should resemble the following code example.

```
...
        break;
    }

    if (loggingFileWriter != null)
    {
        loggingFileWriter.WriteLine
            ("Measurement - {0}", mostRecentMeasure.ToString());
    }
}
...
```

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

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

   Your code should resemble the following code example.

```
...
        loggingFileWriter.WriteLine
            ("Measurement - {0}", mostRecentMeasure.ToString());
    }

    dataCollector.ReportProgress(0);
}

...
```

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

   a. Increment the integer **i**.

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

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

   Your code should resemble the following code example.

```
...
    dataCollector.ReportProgress(0);

    i++;
    if (i > 9)
    {
        i = 0;
    }
}
...
```

Task 7: Implement the dataCollector_ProgressChanged method

1. Locate the **dataCollector_ProgressChanged** method.

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

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

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

   Your code should resemble the following code example.

```
...
void dataCollector_ProgressChanged(object sender,
ProgressChangedEventArgs e)
{
    OnNewMeasurementTaken();
}
...
```

Task 8: Call the GetMeasurements method to start collecting measurements

1. In the task list, locate the **TODO - Call the GetMeasurements method** task, and then double-click this task. This task is located in the **StartCollecting** method.

2. Remove the **TODO - Call the GetMeasurements method** comment, and add a line of code to invoke the **GetMeasurements** method.

   Your code should resemble the following code example.

```
    ...
    else
    {
        loggingFileWriter.WriteLine("Log file status checked - Already
open");
        loggingFileWriter.WriteLine("Collecting Started");
    }

    GetMeasurements();
}
...
```

Task 9: Call the CancelAsync method to stop collecting measurements

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

   Your code should resemble the following code example.

```
...
    }

    // Stop the data collection BackgroundWorker.

    if (dataCollector != null)
    {
        dataCollector.CancelAsync();
    }
}
...
```

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

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

   Your code should resemble the following code example.

```
...
    }
    // Dispose of the dataCollector BackgroundWorker object.
    if (dataCollector != null)
    {
        dataCollector.Dispose();
    }
}
...
```

Task 11: Update the UI to handle measurement events

1. In the task list, locate the **TODO - Declare a delegate to reference NewMeasurementEvent** task, and then double-click this task. This task is located in the code behind the MainWindow.xaml window.
2. Remove the comment and add code to define a delegate of type **EventHandler** named **newMeasurementTaken**.

   Your code should resemble the following code example.

```
...
MeasureMassDevice device;

EventHandler newMeasurementTaken;

private void startCollecting_Click(object sender, RoutedEventArgs e)
...
```

3. In the **startCollecting_Click** method, remove the comment **TODO - use a delegate to refer to the event handler**, and add code to initialize the **newMeasurementTaken** delegate with a new **EventHandler** delegate that is based on a method named **device_NewMeasurementTaken**. You will create the **device_NewMeasurementTaken** method in the next task.

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

Your code should resemble the following code example.

```
...

// Hook up the delegate to an event handler method.

newMeasurementTaken = new EventHandler(device_NewMeasurementTaken);

// TODO - Hook up the event handler to the event.

...
```

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

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

Your code should resemble the following code example.

```
...

newMeasurementTaken = new EventHandler(device_NewMeasurementTaken);
device.NewMeasurementTaken += newMeasurementTaken;
loggingFileNameBox.Text = device.GetLoggingFile();

...
```

Task 12: Implement the device_NewMeasurementTaken event-handling method

1. In the task list, locate the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** task, and then double-click this task.

2. Remove the **TODO - Add the device_NewMeasurementTaken event handler method to update the UI with the new measurement** comment, and add a private event-handler method named **device_NewMeasurementTaken**. The method should not return a value, but should take the following parameters:

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

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

   Your code should resemble the following code example.

```
...
private void device_NewMeasurementTaken(object sender, EventArgs e)
{
}
private void updateButton_Click(object sender, RoutedEventArgs e)
...
```

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

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

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

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

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

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

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

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

Your code should resemble the following code example.

```
...
void device_NewMeasurementTaken(object sender, EventArgs e)
{
    if (device != null)
    {
        mostRecentMeasureBox.Text =
            device.MostRecentMeasure.ToString();
        metricValueBox.Text = device.MetricValue().ToString();
        imperialValueBox.Text = device.ImperialValue().ToString();
        rawDataValues.ItemsSource = null;
        rawDataValues.ItemsSource = device.GetRawData();
    }
}
...
```

Task 13: Disconnect the event handler

1. In the task list, locate the **TODO - Disconnect the event handler** task, and then double-click this task. This task is located in the **stopCollecting_Click** method, which runs when the user clicks the **Stop Collecting** button.
2. Remove the **TODO - Disconnect the event handler** comment, and add code to disconnect the **newMeasurementTaken** delegate from the **device.NewMeasurementTaken** event.

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

Your code should resemble the following code example.

```
...
    device.StopCollecting();
    device.NewMeasurementTaken -= newMeasurementTaken;
}
...
```

Task 14: Test the solution

1. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**.
2. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
3. Click **Start Collecting**, and verify that measurement values begin to appear in the **Raw Data** box.

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

Exercise 2: Using Lambda Expressions to Specify Code

Task 1: Open the Events solution

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

  a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.

  b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 11\Ex2\Starter** folder, click **Events.sln**, and then click **Open**.

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

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

1. In the MeasuringDevice project, add a new code file named HeartBeatEvent.cs:
    a. In Solution Explorer, right-click the **MeasuringDevice** project, point to **Add**, and then click **New Item**.
    b. In the **Add New Item - MeasuringDevice** dialog box, in the template list click **Code File**.
    c. In the **Name** box, type **HeartBeatEvent** and then click **Add**.
2. In the code file, add a **using** directive to bring the **System** namespace into scope.

   Your code should resemble the following code example.

```
using System;
```

3. Define a new class named **HeartBeatEventArgs** in the **MeasuringDevice** namespace. The class should extend the **EventArgs** class.

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

Your code should resemble the following code example.

```
...

namespace MeasuringDevice
{

    public class HeartBeatEventArgs : EventArgs
    {

    }

}
```

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

   Your code should resemble the following code example.

```
...
public class HeartBeatEventArgs : EventArgs
{
    public DateTime TimeStamp { get; private set; }
}
...
```

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

   Your code should resemble the following code example.

```
public class HeartBeatEventArgs : EventArgs
{
    ...

    public HeartBeatEventArgs()
        : base()
    {
        this.TimeStamp = DateTime.Now;
    }
}
```

Task 3: Declare a new delegate type

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

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

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

  Your code should resemble the following code example.

```
...
// Delegate defining the HeartBeat event signature.
public delegate void HeartBeatEventHandler
    (object sender, HeartBeatEventArgs args);
...
```

Task 4: Update the IEventEnabledMeasuringDevice interface

1. In the task list, locate the **TODO - Define the new event in the interface** task and then double-click this task. This task is located in the **IEventEnabledMeasuringDevice** interface:

   a. If the task list is not already visible, on the **View** menu, click **Task List**.

   b. If the **Task List** is displaying **User Tasks**, in the drop-down list box click **Comments**.

   c. Double-click the **TODO - Define the new event in the interface** task.

2. Remove this comment and add an event called **HeartBeat** to the interface. The event should specify that subscribers use the **HeartBeatEventHandler** delegate type to specify the method to run when the event is raised.

   Your code should resemble the following code example.

```
...
// Event that fires every heartbeat.
event HeartBeatEventHandler HeartBeat;

// TODO - Define the HeartBeatInterval property in the interface.
...
```

3. Remove the **TODO - Define the HeartBeatInterval property in the interface** comment, and then add a read-only integer property called **HeartBeatInterval** to the interface.

   Your code should resemble the following code example.

```
...
    // Read-only heartbeat interval - set in constructor.
    int HeartBeatInterval { get; }
}
...
```

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

1. In the task list, locate the **TODO - Add the HeartBeatInterval property** task, and then double-click this task. This task is located in the **MeasureDataDevice** class.

2. Remove the **TODO - Add the HeartBeatInterval property** comment, and add a protected integer member named **heartBeatIntervalTime**.

   Your code should resemble the following code example.

```
...
// Heartbeat interval in milliseconds.
protected int heartBeatIntervalTime;
// TODO - Add the HeartBeat event.
...
```

3. Add code to implement the public integer property **HeartBeatInterval** that the **IEventEnabledMeasuringDevice** interface defines. The property should return the value of the **heartBeatInterval** member when the **get** accessor method is called. The property should have a **private set** accessor method to enable the constructor to set the property.

   Your code should resemble the following code example.

```
...
protected int heartBeatIntervalTime;
public int HeartBeatInterval
{
    get
    {
        return heartBeatIntervalTime;
    }
}


// TODO - Add the HeartBeat event.
...
```

4. Remove the **TODO - Add the HeartBeat event** comment, and add the **HeartBeat** event that the **IEventEnabledMeasuringDevice** interface defines.

   Your code should resemble the following code example.

```
...
// Event that fires every heartbeat
public event HeartBeatEventHandler HeartBeat;

// TODO - Add the OnHeartBeat method to fire the event.
...
```

5. Remove the **TODO - Add the OnHeartBeat method to fire the event** comment, and add a protected virtual void method named **OnHeartBeat** that takes no parameters.

   Your code should resemble the following code example.

```
...
// Overrideable method to fire the OnHeartBeat event.
protected virtual void OnHeartBeat()
{
}

// TODO - Declare the BackgroundWorker to generate the heartbeat.
...
```

6. In the **OnHeartBeat** method, add code to perform the following actions:

   a. Check whether the **HeartBeat** event has any subscribers.

   b. If the event has subscribers, raise the event, passing the current object and a new instance of the **HeartBeatEventArgs** object as parameters.

   Your code should resemble the following code example.

```
...
protected virtual void OnHeartBeat()
{
    if (HeartBeat != null)
    {
        HeartBeat(this, new HeartBeatEventArgs());
    }
}
...
```

Task 6: Use a BackgroundWorker object to generate the heartbeat

1. Remove the **TODO - Declare the BackgroundWorker to generate the heartbeat** comment, and then define a private **BackgroundWorker** object named **heartBeatTimer**.

   Your code should resemble the following code example.

```
...

// Background worker object to host the heartbeat thread.
private BackgroundWorker heartBeatTimer;
```

```
// TODO - Create a method to configure the background Worker by using
// Lambda Expressions.

...
```

2. Remove the **TODO - Create a method to configure the BackgroundWorker using Lambda Expressions** comment, and declare a private method named **StartHeartBeat** that accepts no parameters and does not return a value.

   Your code should resemble the following code example.

```
...
    // Start the BackgroundWorker that fires the heartbeat.
    private void StartHeartBeat()
    {

    }
}
...
```

3. In the **StartHeartBeat** method, add code to perform the following actions:

   a. Instantiate the **heartBeatTimer BackgroundWorker** object.

   b. Configure the **heartBeatTimer** object to support cancellation.

   c. Configure the **heartBeatTimer** object to support progress notification.

   Your code should resemble the following code example.

```
...
private void StartHeartBeat()
{
    heartBeatTimer = new BackgroundWorker();
    heartBeatTimer.WorkerSupportsCancellation = true;
    heartBeatTimer.WorkerReportsProgress = true;
}
...
```

4. Add a handler for the **heartBeatTimer DoWork** event by using a lambda expression to define the actions to be performed. The lambda expression should take two parameters (use the names *o* and *args*). In the lambda expression body, add a **while** loop that continually iterates and contains code to perform the following actions:

   a. Use the static **Thread.Sleep** method to put the current thread to sleep for the length of time that the **HeartBeatInterval** property indicates.

b. Check the value of the **disposed** property. If the value is **true**, terminate the loop.

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

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

Your code should resemble the following code example.

```
...
heartBeatTimer.WorkerReportsProgress = true;
heartBeatTimer.DoWork += (o, args) =>
    {
        while (true)
        {
            Thread.Sleep(HeartBeatInterval);

            if (disposed)
            {
                break;
            }
            heartBeatTimer.ReportProgress(0);
        }
    };
...
```

5. Add a handler for the **heartBeatTimer**.**ReportProgress** event by using another lambda expression to create the method body. In the lambda expression body, add code to call the **OnHeartBeat** method, which raises the **HeartBeat** event.

   Your code should resemble the following code example.

```
...
    heartBeatTimer.ReportProgress(0);
    }
};
heartBeatTimer.ProgressChanged += (o, args) =>
    {
        OnHeartBeat();
    };
...
```

6. At the end of the **StartHeartBeat** method, add a line of code to start the **heartBeatTimer BackgroundWorker** object running asynchronously.

   Your code should resemble the following code example.

```
...
    heartBeatTimer.ProgressChanged += (o, args) =>
        {
            OnHeartBeat();
        };
    heartBeatTimer.RunWorkerAsync();
}
...
```

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

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

   Your code should resemble the following code example.

```
...
    loggingFileWriter.WriteLine("Collecting Started");
}

StartHeartBeat();

GetMeasurements();
...
```

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

1. In the task list, locate the **TODO - dispose of the heartBeatTimer BackgroundWorker** task, and then double-click this task. This task is located in the **Dispose** method.

2. Remove the comment and add code to check that the **heartBeatTimer BackgroundWorker** object is not null. If the **heartBeatTimer** object is not null, call the **Dispose** method of the **BackgroundWorker** object.

   Your code should resemble the following code example.

```
...
    if (dataCollector != null)
    {
        dataCollector.Dispose();
    }

    if (heartBeatTimer != null)
    {
        heartBeatTimer.Dispose();
    }
}
...
```

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

Task 9: Update the constructor for the MeasureMassDevice class

1. Open the MeasureMassDevice class file:

   - In Solution Explorer, in the MeasuringDevice project, double-click **MeasureMassDevice.cs**.

2. At the start of the class, modify the signature of the constructor to take an additional **integer** value named **heartBeatInterval**.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice
    (Units deviceUnits, string logFileName, int heartBeatInterval)
{
    unitsToUse = DeviceUnits;
    measurementType = DeviceType.MASS;
    loggingFileName = LogFileName;
}
...
```

3. Modify the body of the constructor to store the value of the **HeartBeatInterval** member in the **heartBeatInterval** member.

   Your code should resemble the following code example.

```
...
    loggingFileName = LogFileName;
    heartBeatIntervalTime = heartBeatInterval;
}
...
```

4. Below the existing constructor, remove the **TODO – Add a chained constructor that calls the previous constructor** comment, and add a second constructor that accepts the following parameters:

   a. A **Units** instance named **deviceUnits**.

   b. A **string** instance named **logFileName**.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice(Units deviceUnits, string logFileName) {}
...
```

5. Modify the new constructor to implicitly call the existing constructor. Pass a value of **1000** as the *heartBeatInterval* parameter value.

   Your code should resemble the following code example.

```
...
public MeasureMassDevice(Units deviceUnits, string logFileName)
    : this(deviceUnits, logFileName, 1000) { }
...
```

Task 10: Handle the HeartBeat event in the UI

1. In the task list, locate the **TODO - Use a lambda expression to handle the HeartBeat event in the UI** task, and then double-click the task. This task is located in the **startCollecting_Click** method in the code behind the MainWindow window in the Monitor project.

2. Remove the comment, and add a lambda expression to handle the **device**.**HeartBeat** event. The lambda expression should take two parameters (name them *o* and *args*). In the body of the lambda expression, add code to

update the **heartBeatTimeStamp** label with the text "HeartBeat Timestamp: *timestamp*" where timestamp is the value of the **args.TimeStamp** property.

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

Your code should resemble the following code example.

```
...
device.HeartBeat += (o, args) =>
    {
        heartBeatTimeStamp.Content =
            string.Format("HeartBeat Timestamp: {0}", args.TimeStamp);
    };
...
```

Task 11: Test the solution

1. Build the project and correct any errors:
   - On the **Build** menu, click **Build Solution**.
2. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
3. Click **Start Collecting**, and verify that values begin to appear as before. Also note that the **HeartBeat Timestamp** value now updates once per second.
4. Click **Stop Collecting**, and verify that the **RawData** list box no longer updates. Note that the timestamp continues to update, because your code does not terminate the timestamp heartbeat when you stop collecting.
5. Click **Dispose Object**, and verify that the timestamp no longer updates.
6. Close the application, and then return to Visual Studio.
7. Close Visual Studio:
   - In Visual Studio, on the **File** menu, click **Exit**.