**Exercise 1: Raising and Handling Events**

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

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

measurement and logged it.

You will use a **BackgroundWorker** component to poll for new measurements.

The polling for new measurements will takeplace in the **DoWork** event, and the

**ProgressReported** event will raise the **NewMeasurementTaken** event to notify

the clientapplication that a new measurement has been taken.

You will start the background thread running by using the **RunWorkerAsync**

 method, and the device will support cancellation of the background thread by using the

**CancelWorkerAsync** method.

You will test the new functionality by using an existing WPF application that creates

an instance of the **MeasureMassDevice** class and trapping the events that it raises

by using a delegate. The WPF application should be able to pause and then restart

the the **MeasureMassDevice** class.

The main tasks for this exercise are as follows:

|  |  |
| --- | --- |
| 1. | Open the Events solution. |
| 2. | Create a new interface that extends the **IMeasuringDevice** interface. |
| 3. | Add the **NewMeasurementTaken** event to the **MeasureDataDevice** class. |
| 4. | Add a **BackgroundWorker** member to the **MeasureDataDevice** class. |
| 5. | Add the **GetMeasurements** method to the **MeasureDataDevice** class. |
| 6. | Implement the **dataCollector\_DoWork** method. |
| 7. | Implement the **dataCollector\_ProgressChanged** method. |
| 8. | Call the **GetMeasurements** method to start collecting measurements. |
| 9. | Call the **CancelAsync** method to stop collecting measurements. |
| 10. | Dispose of the **BackgroundWorker** object when the **MeasureDataDevice** object is destroyed. |
| 11. | Update the UI to handle measurement events. |
| 12. | Implement the **device\_NewMeasurementTaken** event-handling method. |
| 13. | Disconnect the event handler. |
| 14. | Test the solution. |

**Task 1: Open the Events solution**

|  |  |
| --- | --- |
|  | Open the Events solution in the Lab 11\Ex1\Starter folder. |

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

|  |  |
| --- | --- |
| 1. | In the MeasuringDevice project, add a new interface named **IEventEnabledMeasuringDevice**  in a file namedIEventEnabledMeasuringDevice.cs.  **Note:** Creating a new interface that extends an existing interface is good programming practice, because itpreserves the structure of the original interface for backward compatibility with preexisting code. All preexistingcode can reference the original interface, and new code can reference the new interface and take advantage ofany new functionality. |
| 2. | Modify the interface definition so that the **IEventEnabledMeasuringDevice** interface   extends the **IMeasuringDevice**interface. |
| 3. | In the **IEventEnabledMeasuringDevice** interface, add an event named  **NewMeasurementTaken** by using the base**EventHandler** delegate. |
| 4. | Build the application to enable Microsoft IntelliSense® to reflect your changes. |

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

|  |  |
| --- | --- |
| 1. | Review the task list. |
| 2. | Locate the **TODO - Modify the class definition to implement the extended interface** task, and then double-click thistask. 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 theclass definition to implement the **IEventEnabledMeasuringDevice**  interface instead of the **IMeasuringDevice**interface. |
| 4. | In the task list, locate the **TODO - Add the NewMeasurementTaken event** task, and then double-click this task. Thistask is located at the end of the **MeasureDataDevice** class. |
| 5. | Remove the **TODO - Add the NewMeasurementTaken event** comment, and then declare an event named  **NewMeasurementTaken** by using the same signature as the interface. |
| 6. | Below the event, remove the **TODO - Add an OnMeasurementTaken** **method** comment, and then add a protectedvirtual method  named **OnNewMeasurementTaken**. The method should accept no parameters and have  a void returntype. The **MeasureDataDevice** class will use this method to raise the **NewMeasurementTaken** event. |
| 7. | In the **OnNewMeasurementTaken** method, add code to check that there is a subscriber  for the**NewMeasurementTaken** event; if so, raise the event. The signature of the **EventHandler**  delegate defines twoparameters: an *object* parameter that indicates the object that raised the event   and an *EventArgs* parameter that providesany 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 anevent has no subscribers, the related delegate is null, and the .NET Framework runtime will throw an exceptionif the event is raised. |

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

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

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

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

|  |  |
| --- | --- |
| 1. | In the task list, locate the **TODO - Implement the GetMeasurements method**task, and then double-click this task. |
| 2. | Remove the **TODO - Implement the GetMeasurements method** comment, and then add a new private method named**GetMeasurements** to the class. This method should take no parameters and not return a value. |
| 3. | In the **GetMeasurements** method, add code to perform the following actions:   |  |  | | --- | --- | | a. | Instantiate the **dataCollector** **BackgroundWorker** member. | | b. | Specify that the **dataCollector** **BackgroundWorker** member supports cancellation. | | c. | Specify that the **dataCollector BackgroundWorker** member reports progress while running.  **Hint:** Set the **WorkerSupportsCancellation** and **WorkerReportsProgress** properties. | |
| 4. | Add the following code to instantiate a **DoWorkEventHandler** delegate that refers to a method called**dataCollector\_DoWork**. Attach the delegate to the **DoWork** event property of the **dataCollector** member. The**dataCollector** object will call the **dataCollector\_DoWork** method when the **DoWork** event is raised.  **Hint:** Use IntelliSense to generate a code stub for the **dataCollector\_DoWork** method. To do this, type the firstpart of the line of code, up to the **+=** operators, and then press the TAB key twice. Visual Studio uses a built-incode snippet to complete the line of code and then add a method stub. You can do this each time you hook up anevent 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 amethod called **dataCollector\_ProgressChanged**. Attach this delegate to the **ProgressChanged** event property of the**dataCollector** member. The **dataCollector** object will call the **dataCollector\_ProgressChanged** method when the**ProgressChanged** event is raised. |
| 6. | Add code to start the **dataCollector** **BackgroundWorker** object running asynchronously. |

**Task 6: Implement the dataCollector\_DoWork method**

|  |  |
| --- | --- |
| 1 | Underneath the **GetMeasurements** method, locate the **dataCollector\_DoWork** method.  This method was generated during the previous task. It runs on a background thread, and its purpose is to collect andstore measurement data. |
| 2 | In the **dataCollector\_DoWork** method, remove the statement that raises the **NotImplementedException** exception andadd code to perform the following actions:   |  |  | | --- | --- | | a. | Instantiate the **dataCaptured** array with a new integer array that contains 10 items. | | b. | Define an integer i with an initial value of zero. You will use this variable to track the current position in the**dataCaptured** array. | | c. | Add a **while** loop that runs until the **dataCollector**.**CancellationPending** property is **false**. | |
| 3 | In the **while** loop, add code to perform the following actions:   |  |  | | --- | --- | | a | Invoke the **controller**.**TakeMeasurement** method, and store the result in the **dataCaptured** array at the positionthat the integer **i** indicates. The **TakeMeasurement** method of the **controller** object blocks until a newmeasurement is available. | | b | Update the **mostRecentCapture** property to contain the value in the **dataCaptured** array at the position that theinteger **i** indicates. | | c | If the value of the disposed variable is **true**, terminate the **while** loop. This step ensures that the measurementcollection stops when the **MeasureDataDevice** object is destroyed. | |
| 4 | Add code to the **while** loop after the statements that you added in the previous step to perform the following actions:   |  |  | | --- | --- | | a | Check whether the **loggingFileWriter** property is null. | | b | If the **loggingFileWriter** property is not null, call the **loggingFileWriter**.**Writeline** method, passing a stringparameter of the format "Measurement - *mostRecentMeasure*" where *mostRecentMeasure* is the value of themostRecentMeasure variable.  **Note:** The **loggingFileWriter** property is a simple **StreamWriter** object that writes to a text file. Thisproperty is initialized in the **StartCollecting** method. You can use the **WriteLine** method to write to a**StreamWriter** object. | |
| 5 | Add a line of code to the end of the **while** loop to invoke the **dataCollector**.**ReportProgress** method, passing zero asthe parameter.  The **ReportProgress** method raises the **ReportProgress** event and is normally used to return the percentage completionof the tasks assigned to the **BackgroundWorker** object. You can use the **ReportProgress** event to update progress barsor time estimates in the UI. In this case, because the task will run indefinitely until canceled, you will use the**ReportProgress** event as a mechanism to prompt the UI to refresh the display with the new measurement. |
| 6 | Add code to the end of the **while** loop to perform the following actions:   |  |  | | --- | --- | | a | Increment the integer **i**. | | b | If the value of the integer is greater than nine, reset **i** to zero.  You are using the integer i as a pointer to the next position to write to in the **dataCaptured** array. This array hasspace for 10 measurements. When element 9 is filled, the device will start to overwrite data beginning at element0. | |

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

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

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

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

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

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

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

**Task 11: Update the UI to handle measurement events**

|  |  |
| --- | --- |
| 1 | In the task list, locate the **TODO - Declare a delegate to reference** **NewMeasurementEvent** task, and then double-click this task. This task is located in the code behind the MainWindow.xaml window. |
| 2 | Remove the comment and add code to define a delegate of type **EventHandler** named **newMeasurementTaken**. |
| 3 | In the **startCollecting**\_**Click** method, remove the comment **TODO - use a delegate to refer to the event handler**, andadd code to initialize the **newMeasurementTaken** delegate with a new **EventHandler** delegate that is based on amethod named **device\_NewMeasurementTaken**. You will create the **device\_NewMeasurementTaken** method in thenext task.  **Note:** You cannot use IntelliSense to automatically generate the stub for the **device\_NewMeasurementTaken**method, as you did in earlier tasks. |
| 4 | In the **startCollecting**\_**Click** method, remove the **TODO - Hook up the event handler to the event** comment, and addcode 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** abstractclass.  **Hint:** To connect a delegate to an event, use the **+=** compound assignment operator on the event. |

**Task 12: Implement the device\_NewMeasurementTaken event-handling method**

|  |  |
| --- | --- |
| 1 | In the task list, locate the **TODO - Add the device\_NewMeasurementTaken event handler method to update the UIwith 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 thenew measurement** comment, and add a private event-handler method named **device\_NewMeasurementTaken**. Themethod should not return a value, but should take the following parameters:   |  |  | | --- | --- | | a. | An **object**object named **sender**. | | b. | An **EventArgs**object named**e**. | |
| 3 | In the **device\_NewMeasurementTaken** method, add code to check that the **device** member is not null. If the **device**member is not null, perform the following tasks:   |  |  | | --- | --- | | a | Update the **Text** property of the **mostRecentMeasureBox** text box with the value of the**device.MostRecentMeasure** property.  **Hint:** Use the **ToString** method to convert the value that the **device.MostRecentMeasure** property returnsfrom an integer to a string. | | b | Update the **Text** property of the **metricValueBox** text box with the value that the **device.MetricValue** methodreturns. | | c | Update the **Text** property of the **imperialValueBox** text box with the value that the **device.ImperialValue** methodreturns. | | d | Reset the **rawDataValues.ItemsSource** property to **null**. | | e | Set the **rawDataValues.ItemsSource** property to the value that the **device.GetRawData** method returns.  **Note:** The final two steps are both necessary to ensure that the data-binding mechanism that the **Raw Data**box uses on the WPF window updates the display correctly. | |

**Task 13: Disconnect the event handler**

|  |  |
| --- | --- |
| 1. | In the task list, locate the **TODO - Disconnect the event handler** task, and then double-click this task. This task islocated in the **stopCollecting\_Click** method, which runs when the user clicks the **Stop Collecting** button. |
| 2. | Remove the **TODO - Disconnect** **the event handler** comment, and add code to disconnect the **newMeasurementTaken**delegate from the **device.NewMeasurementTaken** event.  **Hint:** To disconnect a delegate from an event, use the **-=** compound assignment operator on the event. |

**Task 14: Test the solution**

|  |  |
| --- | --- |
| 1 | Build the project and correct any errors. |
| 2 | Start the application. |
| 3 | Click **Start Collecting**, and verify that measurement values begin to appear in the **Raw Data** box.  The **MeasureMassDevice** object used by the application takes metric measurements and stores them, before raising the**NewMeasurementTaken** event. The event calls code that updates the UI with the latest information. Continue to watchthe **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 capturedand displayed. |
| 6 | Click **Stop** **Collecting**. |
| 7 | Close the application, and then return to Visual Studio. |

**Exercise 2: Using Lambda Expressions to Specify Code**

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

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

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

The main tasks for this exercise are as follows:

|  |  |
| --- | --- |
| 1. | Open the Events solution. |
| 2. | Define a new **EventArgs** class to support heartbeat events. |
| 3. | Declare a new delegate type. |
| 4. | Update the **IEventEnabledMeasuringDevice** interface. |
| 5. | Add the **HeartBeat** event and **HeartBeatInterval** property to the **MeasureDataDevice** class. |
| 6. | Use a **BackgroundWorker** object to generate the heartbeat. |
| 7. | Call the **StartHeartBeat** method when the **MeasureDataDevice** object starts running. |
| 8. | Dispose of the **heartBeatTimer** **BackgroundWorker** object when the **MeasureDataDevice** object is destroyed. |
| 9. | Update the constructor for the **MeasureMassDevice**class. |
| 10. | Handle the **HeartBeat** event in the UI. |
| 11. | Test the solution. |

**Task 1: Open the Events solution**

|  |  |
| --- | --- |
| • | Open the Events solution in the E:\Labfiles\Lab 11\Ex2\Starter folder.  **Note:** The Events solution in the Ex2 folder is functionally the same as the code that you completed in Exercise 1;however, it includes an updated task list to enable you to complete this exercise. |

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

|  |  |
| --- | --- |
| 1 | In the MeasuringDevice project, add a new code file named HeartBeatEvent.cs. |
| 2 | In the code file, add a **using** directive to bring the **System** namespace into scope. |
| 3 | Define a new class named **HeartBeatEventArgs** in the**MeasuringDevice**namespace. The class should extend the**EventArgs** class.  **Note:** A custom event arguments class can contain any number of properties; these properties store informationwhen the event is raised, enabling an event handler to receive event-specific information when the event ishandled. |
| 4 | In the **HeartBeatEventArgs** class, add a read-only automatic **DateTime** property named **TimeStamp**. |
| 5 | Add a constructor to the **HeartBeatEventArgs** class. The constructor should accept no arguments, and initialize the**TimeStamp** property to the date and time when the class is constructed. The constructor should also extend the baseclass constructor. |

**Task 3: Declare a new delegate type**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| • | Below the **HeartBeatEventArgs** class, declare a **public** **delegate** type named **HeartBeatEventHandler**. The delegateshould refer to a method that does not return a value, but that has the following parameters:   |  |  | | --- | --- | | a. | An **object** parameter named *sender*. | | b. | A **HeartBeatEventArgs** parameter named *args*. | |

**Task 4: Update the IEventEnabledMeasuringDevice interface**

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

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

|  |  |
| --- | --- |
| 1 | In the task list, locate the **TODO - Add the HeartBeatInterval property** task, and then double-click this task. This taskis located in the **MeasureDataDevice** class. |
| 2 | Remove the **TODO - Add the HeartBeatInterval property** comment, and add a protected integer member named**heartBeatIntervalTime**. |
| 3 | Add code to implement the public integer property **HeartBeatInterval** that the **IEventEnabledMeasuringDevice**interface defines. The property should return the value of the **heartBeatInterval** member when the **get** accessor methodis called. The property should have a **private** **set** accessor method to enable the constructor to set the property. |
| 4 | Remove the **TODO - Add the HeartBeat event** comment, and add the **HeartBeat** event that the**IEventEnabledMeasuringDevice** interface defines. |
| 5 | Remove the **TODO - add the OnHeartBeat method** **to fire the event** comment, and add a protected virtual voidmethod named **OnHeartBeat** that takes no parameters. |
| 6 | In the **OnHeartBeat** method, add code to perform the following actions:   |  |  | | --- | --- | | a | Check whether the **HeartBeat** event has any subscribers. | | b | If the event has subscribers, raise the event, passing the current object and a new instance of the**HeartBeatEventArgs** object as parameters. | |

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

|  |  |
| --- | --- |
| 1 | Remove the **TODO - Declare the BackgroundWorker to generate the heartbeat** comment, and then define a private**BackgroundWorker** object named **heartBeatTimer**. |
| 2 | Remove the **TODO - Create a method to configure the BackgroundWorker using Lambda Expressions** comment,and declare a private method named **StartHeartBeat** that accepts no parameters and does not return a value. |
| 3 | In the **StartHeartBeat** method, add code to perform the following actions:   |  |  | | --- | --- | | a. | Instantiate the **heartBeatTimer** **BackgroundWorker** object. | | b. | Configure the **heartBeatTimer** object to support cancellation. | | c. | Configure the **heartBeatTimer** object to support progress notification. | |
| 4 | Add a handler for the **heartBeatTimer** **DoWork** event by using a lambda expression to define the actions to beperformed. The lambda expression should take two parameters (use the names *o* and *args*). In the lambda expressionbody, 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 thebody of the lambda expression. | |
| 5 | Add a handler for the **heartBeatTimer**.**ReportProgress** event by using another lambda expression to create the methodbody. In the lambda expression body, add code to call the **OnHeartBeat** method, which raises the **HeartBeat** event. |
| 6 | At the end of the **StartHeartBeat** method, add a line of code to start the **heartBeatTimer** **BackgroundWorker** objectrunning asynchronously. |

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

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

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

|  |  |
| --- | --- |
| 1 | In the task list, locate the **TODO - dispose of the heartBeatTimer BackgroundWorker** task, and then double-clickthis task. This task is located in the **Dispose** method. |
| 2 | Remove the comment and add code to check that the **heartBeatTimer** **BackgroundWorker** object is not null. If the**heartBeatTimer** object is not null, call the **Dispose** method of the **BackgroundWorker**object.  You have now updated the **MeasureDataDevice** abstract class to implement event handlers by using lambdaexpressions. To enable the application to benefit from these changes, you must modify the **MeasureMassDevice** class,which extends the **MeasureDataDevice** class. |

**Task 9: Update the constructor for the MeasureMassDevice class**

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

**Task 10: Handle the HeartBeat event in the UI**

|  |  |
| --- | --- |
| 1 | In the task list, locate the **TODO - Use a lambda expression to handle the HeartBeat event in the UI** task, and thendouble-click this task. This task is located in the **startCollecting\_Click** method in the code behind the MainWindowwindow in the Monitor project. |
| 2 | Remove the comment, and add a lambda expression to handle the **device**.**HeartBeat** event. The lambda expressionshould take two parameters (name them *o* and *args*). In the body of the lambda expression, add code to update the**heartBeatTimeStamp** label with the text "HeartBeat Timestamp: *timestamp"* where timestamp is the value of the**args.TimeStamp** property.  **Hint:** Set the **Content** property of a label to modify the text that the label displays. |

**Task 11: Test the solution**

|  |  |
| --- | --- |
| 1 | Build the project and correct any errors. |
| 2 | Start the application. |
| 3 | Click **Start** **Collecting**, and verify that values begin to appear as before. Also note that the **HeartBeat** **Timestamp** valuenow updates once per second. |
| 4 | Click **Stop** **Collecting**, and verify that the **RawData** list box no longer updates. Note that the timestamp continues toupdate, 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. |