Lab 14: Using LINQ to Query Data

Exercise 1: Using the LINQ Query Operators

Task 1: Open the starter solution

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft Visual Studio 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Import the code snippets from the E:\Labfiles\Lab 14\Snippets folder:
   a. In Visual Studio, on the **Tools** menu, click **Code Snippets Manager**.
   b. In the **Code Snippets Manager** dialog box, in the **Language** list, click **Visual C#**.
   c. Click **Add**.
   d. In the **Code Snippets Directory** dialog box, move to the **E:\Labfiles\Lab 14\Snippets** folder, and then click **Select Folder**.
   e. In the **Code Snippets Manager** dialog box, click **OK**.
4. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 14\Ex1\Starter** folder, click **StressDataAnalyzer.sln**, and then click **Open**.
5. Examine the user interface (UI) for the StressDataAnalyzer application. Note the following features of the application:
   - The stress test data is generated by a stress test device. The data is stored in a binary data file, and this application reads the data from this file when the application starts to run. The application holds the data in memory by using a **Tree** object.

- The UI contains two main areas. The upper area enables the user to specify criteria to match stress data. The lower area displays the data.
- The stress test data criteria are:

  i. The date that the test was performed.

  ii. The temperature at which the test was performed.

  iii. The stress that was applied during the test.

  iv. The deflection that resulted from applying the stress.

  Each criterion is specified as a range by using the slider controls.
- After selecting the criteria to match, the user clicks **Display** to generate a Language-Integrated Query (LINQ) query that fetches the matching data from the **Tree** object in memory and shows the results.

a. In Solution Explorer, expand the **StressDataAnalyzer** project.

b. Double-click the **DataAnalyzer.xaml** file.

Task 2: Declare variables to specify the stress data file name and the Tree object

1. Review the task list:

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

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

2. In the task list, locate the **TODO - Declare filename and tree variables** task, and then double-click this task. This task is located in the **DataAnalyzer**.**xaml**.**cs** class.

3. Delete the **TODO - Declare filename and tree variables** comment, and then add code to declare the following variables:

   a. A private constant **string** object named stressDataFilename. Initialize the object with the string "E:\Labfiles\Lab 14\StressData.dat". This is the name of the data file that holds the stress data.

   b. A private **Tree** object named stressData that is based on the **TestResult** type. This **Tree** object will hold the data that is read from the stress data file. Initialize this object to null.

The **TestResult** type is a struct that contains the following four fields, corresponding to the data for each stress test record:

- **TestDate**. This is a **DateTime** field that contains the date on which the stress test was performed.
- **Temperature**. This is a **short** field that contains the temperature, in Kelvin, at which the test was performed.
- **AppliedStress**. This is another **short** field that specifies the stress, in kiloNewtons (kN), that was applied during the test.
- **Deflection**. This is another **short** field that specifies the deflection of the girder, in millimeters (mm), when the stress was applied.

The **TestResult** type implements the **IComparable** interface. The comparison of test data is based on the value of the **Deflection** field.

Your code should resemble the following code example.

```
public partial class DataAnalyzer : Window
{

    // Declare a string variable to hold the name of the file
    // that contains the stress test data.
    private const string stressDataFilename =
        @"E:\Labfiles\Lab 14\StressData.dat";

    // Declare a Tree variable to hold the loaded data.
    private Tree<TestResult> stressData = null;

    public DataAnalyzer()
    ...
}
```

Task 3: Add a method to read the test data

1. In the task list, locate the **TODO - Add a method to read the contents of the StressData file** task, and then double-click this task.
2. Delete the **TODO - Add a method to read the contents of the StressData file** comment, and then add the method in the following code example, which is named **ReadTestData**. This method reads the stress data from the file and populates the **Tree** object. It is not necessary for you to fully understand how this method works, so you can either type this code manually, or you can use the Mod14ReadTestData code snippet.

```
private void ReadTestData()
{
    // Open a stream over the file that holds the test data.
    using (FileStream readStream =
        File.Open(stressDataFilename, FileMode.Open))
    {
        // The data is serialized as TestResult instances.
        // Use a BinaryFormatter object to read the stream and
        // deserialize the data.
        BinaryFormatter formatter = new BinaryFormatter();
        TestResult initialNode =
            (TestResult)formatter.Deserialize(readStream);

        // Create the binary tree and use the first item retrieved
        // as the root node. (Note: The tree will likely be
        // unbalanced, because it is probable that most nodes will
        // have a value that is greater than or equal to the value in
        // this root node - this is because of the way in which the
        // test results are generated and the fact that the TestResult
        // class uses the deflection as the discriminator when it
        // compares instances.)
        stressData = new Tree<TestResult>(initialNode);

        // Read the TestResult instances from the rest of the file
        // and add them into the binary tree.
        while (readStream.Position < readStream.Length)
        {
            TestResult data =
                (TestResult)formatter.Deserialize(readStream);
            stressData.Insert(data);
        }
    }
}
```

- To use the code snippet, type **Mod14ReadTestData** and then press the TAB key twice.

Task 4: Read the test data by using a BackgroundWorker object

1. In the **Window_Loaded** method, add code to perform the following tasks:

   a. Create a **BackgroundWorker** object named **workerThread**.

   b. Configure the **workerThread** object; the object should not report progress or support cancellation.

   Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{

    // Read the test data and populate the binary tree.
    // Use a BackgroundWorker object to avoid tying up the UI.

    BackgroundWorker workerThread = new BackgroundWorker();
    workerThread.WorkerReportsProgress = false;
    workerThread.WorkerSupportsCancellation = false;

}
```

2. In the **Window_Loaded** method, add an event handler for the **workerThread.DoWork** event. When the event is raised, the event handler should invoke the **ReadTestData** method.

   Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...

    workerThread.DoWork += (o, args) =>
        {
            this.ReadTestData();
        };

}
```

3. Add an event handler for the **workerThread**.**RunWorkerComplete** event. When the event is raised, the event handler should perform the following tasks:

   a. Enable the **displayResults** button.

   b. Display the message 'Ready' in the **statusMessage** StatusBarItem in the status bar at the bottom of the Windows Presentation Foundation (WPF) window.

Hint: Set the **Content** property of a status bar item to display a message in that item.

Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...
    workerThread.RunWorkerCompleted += (o, args) =>
        {
            this.displayResults.IsEnabled = true;
            this.statusMessage.Content = "Ready";
        };
}
```

4. At the end of the **Window_Loaded** method, add code to perform the following tasks:

    a. Start the **workerThread BackgroundWorker** object running asynchronously.

    b. Display the message "Reading Test Data..." in the **statusMessage** item in the status bar at the bottom of the WPF window.

    Your code should resemble the following code example.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    ...
    workerThread.RunWorkerAsync();
    this.statusMessage.Content = "Reading test data ...";
}
```

Task 5: Define the LINQ query

1. In the task list, locate the **TODO - Define the LINQ query** task, and then double-click this task. This task is located in the **CreateQuery** method.

2. Replace the existing code in the method with code that defines an **IEnumerable<TestResult>** object called **query**. Initialize the query variable with a LINQ query that retrieves all of the **TestResult** objects in the **stressData** tree that meet the following criteria. The query should order returned values by the **TestDate** property. The query should evaluate each object by using the following criteria:

    a. The value of the **TestDate** property is greater than or equal to the *dateStart* parameter value.

    b. The value of the **TestDate** property is less than or equal to the *dateEnd* parameter value.

c. The value of the **Temperature** property is greater than or equal to the **temperatureStart** parameter value.

d. The value of the **Temperature** property is less than or equal to the *temperatureEnd* parameter value.

e. The value of the **AppliedStress** property is greater than or equal to the *appliedStressStart* parameter value.

f. The value of the **AppliedStress** property is less than or equal to the *appliedStressEnd* parameter value.

g. The value of the **Deflection** property is greater than or equal to the *deflectionStart* parameter value.

h. The value of the **Deflection** property is less than or equal to the *deflectionEnd* parameter value.

Your code should resemble the following code example.

```
private IEnumerable<TestResult> CreateQuery
    (DateTime dateStart, DateTime dateEnd, short temperatureStart,
     short temperatureEnd, short appliedStressStart, short
     appliedStressEnd, short deflectionStart, short deflectionEnd)
{

    IEnumerable<TestResult> query =
        from result in stressData
        where result.TestDate >= dateStart &&
              result.TestDate <= dateEnd &&
              result.Temperature >= temperatureStart &&
              result.Temperature <= temperatureEnd &&
              result.AppliedStress >= appliedStressStart &&
              result.AppliedStress <= appliedStressEnd &&
              result.Deflection >= deflectionStart &&
              result.Deflection <= deflectionEnd
        orderby result.TestDate
        select result;
}
```

3. At the end of the method, return the **query** object.

Your code should resemble the following code example.

```
private IEnumerable<TestResult> CreateQuery
    (DateTime dateStart, DateTime dateEnd, short temperatureStart,
     short temperatureEnd, short appliedStressStart, short
     appliedStressEnd, short deflectionStart, short deflectionEnd)
{
    ...
        select result;

    return query;
}
```

4. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 6: Execute the query

1. In the task list, locate the **TODO - Execute the LINQ query** task, and then double-click this task. This task is located in the **FormatResults** method. This method takes an enumerable collection of **TestResult** objects as a parameter and generates a string that contains a formatted list of **TestResult** objects. The parameter is the item that the **CreateQuery** method returns. Iterating through this list runs the LINQ query.

2. Delete the **TODO - Execute the LINQ query** comment, and then add code to the **FormatResults** method to perform the following task:

   - For each item that the query returns, format and append the details of each item to the **builder** StringBuilder object. Each item should be formatted to display the following properties in a double-tab delimited format:

     i. **TestDate**

     ii. **Temperature**

     iii. **AppliedStress**

     iv. **Deflection**

   Your code should resemble the following code example.

```
private string FormatResults(IEnumerable<TestResult> query)
{
    ...
    builder.Append
        ("Test Date\t\tTemperature\tApplied Stress\tDeflection\n");

    // Iterate through the results and format each item found.
    foreach (var item in query)
    {
        builder.Append(String.Format("{0:d}\t\t{1}\t\t{2}\t\t{3}\n",
            item.TestDate,
            item.Temperature,
            item.AppliedStress,
            item.Deflection));
    }

    // Return the string that is constructed by using the
    // StringBuilder object.

    return builder.ToString();

}
```

3. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 7: Run the query by using a BackgroundWorker object

1. In the task list, locate the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** task, and then double-click this task. This task is located in the **DisplayResults_Click** method. This method calls the **CreateQuery** method to generate the LINQ query that matches the criteria that the user specifies, and it then runs the query to generate and format the results by using a **BackgroundWorker** object called **workerThread**.

2. Delete the **TODO - Add a BackgroundWorker DoWork event handler to invoke the query operation** comment, and then define an event handler for the **workerThread.DoWork** event. Add code to the event handler to invoke the **FormatResults** method, passing the **query** object as the parameter to the method. Store the value that the method returns in the *Result* parameter of the **DoWork** event handler.

   Your code should resemble the following code example.

```
private void DisplayResults_Click(object sender, RoutedEventArgs e)
{

    try
    {
        ...

        workerThread.WorkerSupportsCancellation = false;

        // Return the formatted string as the result of the background
        // operation.
        workerThread.DoWork += (o, args) =>
            {
                args.Result = FormatResults(query);
            };

        ...
    }

    ...
}
```

3. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 8: Display the results

1. Below the event handler for the **DoWork** event, add an event handler for the **workerThread**.**RunWorkerComplete** event. Add code to the event handler to perform the following tasks:

   a. Update the **results**.**Text** property with the value of the *Result* parameter of the **RunWorkerComplete** event handler.

   b. Enable the **displayResults** button.

   c. Update the **statusMessage** status bar item to "Ready".

   Your code should resemble the following code example.

```
private void DisplayResults_Click(object sender, RoutedEventArgs e)
{
    try
    {
        ...

        workerThread.DoWork += (o, args) =>
        {
            args.Result = FormatResults(query);
        };

        // When the BackgroundWorker object has completed reading
        // the test data, display the results, set the status bar
        // to "Ready", and enable the displayResults button.

        workerThread.RunWorkerCompleted += (o, args) =>
        {
            this.results.Text = args.Result as string;
```

```
                this.displayResults.IsEnabled = true;
                this.statusMessage.Content = "Ready";
            };
            ...
        }
        ...
    }
```

2. Build the solution and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 9: Test the solution

1. Run the application:

   - On the **Debug** menu, click **Start Without Debugging**.

2. Click **Display**, and make a note of the **Time (ms)** value that is displayed next to the **Display** button.

3. Click **Display** two more times. The times for these operations will probably be lower than the time that the initial query took because the various internal data structures have already been initialized. Make a note of these times.

Note: The time that is displayed is the time that is required to fetch the data by using the LINQ query, but not the time that is taken to format and display this data. This is why the "Fetching results" message appears for several seconds after the data has been retrieved.

4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 40,641 values.

5. Use the **DatePicker** and slider controls to modify the search criteria to the values in the following table, and then click **Display** again.

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

6. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values. Note the time that it

took to complete the search—the time should be less than the times that you recorded in Step 3. Keep a note of these values for comparison in Exercise 2.

7. Close the Stress Data Analyzer window, and then return to Visual Studio.

   Currently, any search through the data uses all four criteria—date, temperature, applied stress, and deflection—regardless of the values that are specified in the UI. If the user does not change the default values for any criteria, the LINQ query that the application generates still contains criteria for each field. This is rather inefficient. However, you can construct dynamic LINQ queries to enable you to generate a custom query that is based only on the criteria that are specified at run time. You will implement this functionality in the next exercise.

Exercise 2: Building Dynamic LINQ Queries

Task 1: Open the StressDataAnalyzer solution

1. Open the StressDataAnalyzer solution in the E:\Labfiles\Lab 14\Ex2\Starter folder:

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

   b. In the **Open Project** dialog box, move to the **E:\Labfiles\Lab 14\Ex2 \Starter** folder, click **StressDataAnalyzer.sln**, and then click **Open**.

2. Review the task list:

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

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

3. Examine the modified UI for the StressDataAnalyzer application. Note the following features of the application:

   - The UI is an extended version of that used in Exercise 1. The user can specify which criteria to apply by using check boxes. Any criteria that are not selected are not included in the LINQ query.

   - The user can change the order in which the data is displayed by selecting the appropriate option button in the **Order By** section of the window.

- The user can limit the number of items that a query returns by selecting the **Limit** check box and by using the slider control to specify the number of items.

a. In Solution Explorer, expand the **StressDataAnalyzer** project.

b. Double-click the **DataAnalyzer.xaml** file.

Task 2: Dynamically build a lambda expression for the query criteria

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

   The **BuildLambdaExpressionForQueryCriteria** method dynamically constructs a lambda expression from the values that are passed in as parameters. There are 12 parameters, which are divided into four groups. The *dateRangeSpecified* parameter is a Boolean value that indicates whether the user has selected the date criteria in the window, and the *startDate* and *endDate* parameters contain the start date and end date values that the user specifies. If the *dateRangeSpecified* parameter is **false**, the date is not included in the criteria for matching stress data. The same logic applies to the remaining parameters.

   The value that the **BuildLambdaExpressionForQueryCriteria** method returns is an **Expression** object. The **Expression** type represents a strongly typed lambda expression as a data structure in the form of an expression tree. The type parameter is a delegate that indicates the form of the lambda expression. In the **BuildLambdaExpressionForQueryCriteria** method, the lambda expression takes a **TestResult** object and returns a Boolean value that indicates whether this object should be included in the results that are generated by running the lambda expression.

   The existing code in this method creates a reference to an **Expression** object named **lambda**. You will add code to populate this object with an expression tree that represents a lambda expression that matches the query criteria that the 12 parameters specify. If the user does not specify any query criteria, this method returns a null value.

Note: The **Expression** type is located in the **System.Linq.Expressions** namespace. The application creates an alias for this namespace called **Expressions**. You cannot refer to the **Expression** type without the qualifying namespace in a WPF application because the WPF assemblies also contain a type called **Expression**.

2. Delete the **TODO - Complete the BuildLambdaExpressionForQueryCriteria method** comment, and then add code to perform the following tasks:

   a. Create a **Type** reference for the **TestResult** type named **testResultType**.

Hint: Creating a type reference in this way enables you to repeatedly refer to an object type without repeatedly calling the **typeof** method. The **typeof** method is a relatively costly method compared to retrieving an object reference.

   b. Create an **Expressions.ParameterExpression** object named **itemBeingQueried** by using the **Expressions.Expression.Parameter** static method. Specify the **testResultType** type reference as the type of the parameter, and use the string "**item**" as the name of the parameter.

Hint: The string that is passed as the second parameter to the method call defines how your lambda expression will refer to the object that is being queried. In this example, one part of the resultant expression will resemble **"item.TestDate >= startDate"**.

Your code should resemble the following code example.

```
private Expressions.Expression<Func<TestResult, bool>>
    BuildLambdaExpressionForQueryCriteria
    (...)
{
    ...
    if (dateRangeSpecified || temperatureRangeSpecified ||
        appliedStressRangeSpecified || deflectionRangeSpecified)
    {
        // Create the expression that defines the parameter for the
        // lambda expression.
        // The type is TestResult, and the lambda expression refers to
        // it with the name "item".
        Type testResultType = typeof(TestResult);
        Expressions.ParameterExpression itemBeingQueried =
            Expressions.Expression.Parameter(testResultType, "item");
        ...
    }
    ...
}
```

3. Add code to the method to create the following **Expressions.BinaryExpression** objects; each object should have an initial value of null:

a. **dateCondition**

b. **temperatureCondition**

c. **appliedStressCondition**

d. **deflectionCondition**

You will populate these expression objects with query criteria that match the parameters that are passed in to the method. You will then combine these expression objects together to form the complete lambda expression tree.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...
    // Create expressions for each of the possible conditions.
    Expressions.BinaryExpression dateCondition = null;
    Expressions.BinaryExpression temperatureCondition = null;
    Expressions.BinaryExpression appliedStressCondition = null;
    Expressions.BinaryExpression deflectionCondition = null;
    ...
}
...
```

4. Add code to the method to invoke the **BuildDateExpressionBody** method, and store the result in the **dateCondition** object. Pass the following values as parameters to the method call:

   a. **dateRangeSpecified**

   b. **startDate**

   c. **endDate**

   d. **testResultType**

   e. **itemBeingQueried**

Note: The **BuildDateExpressionBody** method returns a **BinaryExpression** object that checks the stress test data against the **startDate** and **endDate** values. You will update the **BuildDateExpressionBody** method in the following task.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...

    // Build Boolean expressions for each of the possible criteria
    // that the user specifies.
    // These method calls may return null if the user did not
    // specify criteria for a property.
    dateCondition = BuildDateExpressionBody(
        dateRangeSpecified, startDate, endDate,
        testResultType, itemBeingQueried);
    ...
}
```

5. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **temperatureCondition** object. Pass the following values as parameters to the method call:
   a. **temperatureRangeSpecified**
   b. **fromTemperature**
   c. **toTemperature**
   d. **testResultType**
   e. A string that contains the value "**Temperature**"
   f. **itemBeingQueried**

Note: The **BuildNumericExpressionBody** method also returns a **BinaryExpression** object that will form part of the dynamic LINQ query. In this case, the data that this part of the query checks will contain numeric data rather than a **DateTime** value, and the name of the field that is being checked is **Temperature**. You will update the **BuildNumericExpressionBody** method later in the lab.

Your code should resemble the following code example.

```
dateCondition = BuildDateExpressionBody(
     dateRangeSpecified, startDate, endDate,
     testResultType, itemBeingQueried);

temperatureCondition = BuildNumericExpressionBody(
    temperatureRangeSpecified, fromTemperature, toTemperature,
    testResultType, "Temperature", itemBeingQueried);
```

6. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **appliedStressCondition** object. Pass the following values as parameters to the method call:

    a. **appliedStressRangeSpecified**

    b. **fromStressRange**

    c. **toStressRange**

    d. **testResultType**

    e. A string that contains the value "**AppliedStress**"

    f. **itemBeingQueried**

    Your code should resemble the following code example.

```
temperatureCondition = BuildNumericExpressionBody(
    temperatureRangeSpecified, fromTemperature, toTemperature,
    testResultType, "Temperature", itemBeingQueried);

appliedStressCondition = BuildNumericExpressionBody(
    appliedStressRangeSpecified, fromStressRange, toStressRange,
    testResultType, "AppliedStress", itemBeingQueried);
```

7. Add code to the method to invoke the **BuildNumericExpressionBody** method, and store the result in the **deflectionCondition** object. Pass the following values as parameters to the method call:

    a. **deflectionRangeSpecified**

    b. **fromDeflection**

    c. **toDeflection**

    d. **testResultType**

    e. A string that contains the value "**Deflection**"

    f. **itemBeingQueried**

    Your code should resemble the following code example.

```
appliedStressCondition = BuildNumericExpressionBody(
    appliedStressRangeSpecified, fromStressRange, toStressRange,
    testResultType, "AppliedStress", itemBeingQueried);

deflectionCondition = BuildNumericExpressionBody(
    deflectionRangeSpecified, fromDeflection, toDeflection,
    testResultType, "Deflection", itemBeingQueried);
```

8. Add code to the method to invoke the **BuildLambdaExpressionBody** method, and store the result in a new **Expressions**.**Expression** object named **body**. Pass the **dateCondition**, **temperatureCondition**, **appliedStressCondition**, and **deflectionCondition** objects as parameters to the method.

Note: The **BuildLambdaExpressionBody** method takes the four expression objects, each of which evaluate a single property in a **TestResult** object, and combines them into a complete lambda expression that evaluates all of the properties that the user specifies criteria for. You will complete the **BuildLambdaExpressionBody** method later in the lab.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...

    // Combine the Boolean expressions together into a single body.
    Expressions.Expression body = BuildLambdaExpressionBody(
        dateCondition, temperatureCondition,
        appliedStressCondition, deflectionCondition);
    ...

}
```

9. Add code to the method to invoke the **Expression.Lambda** generic method, and store the response in the **lambda** object. The **Expression.Lambda** method should construct a lambda expression from the body of the lambda expressions in the **body Expression** object and the **itemBeingQueried ParameterExpression** object. Specify the delegate type **Func<TestResult, bool>** as the type parameter of the method.

Hint: The static **Expression.Lambda** method constructs an expression tree that represents a completed lambda expression, including the data that is being queried by the expression.

Your code should resemble the following code example.

```
if (dateRangeSpecified || temperatureRangeSpecified ||
    appliedStressRangeSpecified || deflectionRangeSpecified)
{
    ...
    // Build the lambda expression by using the parameter and the
    // body expressions.

    lambda = Expressions.Expression.Lambda<Func<TestResult, bool>>(
        body, itemBeingQueried);

}
```

10. Build the project and correct any errors:
    - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 3: Dynamically build the date expression tree

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

   The existing code in this method defines a **BinaryExpression** object named **dateCondition**. This object will be used to return the expression tree that evaluates date values. The method then checks that the *dateRangeSpecified* parameter is **true**. You will add code to this conditional statement to build an expression tree that is equivalent to the condition in the following code example.

```
item.TestDate >= startDate && item.TestDate <= endDate
```

   If the user did not specify any date criteria, this method returns a null expression tree.

2. Delete the **TODO - Complete the BuildDateExpressionBody method** comment, and then add code to create a new **MemberInfo** object named **testDateProperty**. Call the **GetProperty** method to generate code that retrieves the **TestDate** property from the *testResultType* type parameter. Pass the string "TestDate" as the parameter to the **GetProperty** method.

   Your code should resemble the following code example.

```
if (dateRangeSpecified)
{

    // Generate the expression:
    //
    //    item.TestDate >= startDate
    //
    MemberInfo testDateProperty =
        testResultType.GetProperty("TestDate");

    ...
}
```

3. Add code to the method to create a **MemberExpression** object named **testDateMember**. Populate the object with the value that is returned by calling the **Expression**.**MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testDateProperty** value as parameters to the method.

Note: A **MemberExpression** object is an expression that represents access to a property of the object that is being queried. In this case, the object represents the ***item*.TestDate** property.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    MemberInfo testDateProperty =
        testResultType.GetProperty("TestDate");

    Expressions.MemberExpression testDateMember =
        Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testDateProperty);
}
```

4. Add code to create an **Expressions.ConstantExpression** object named **lowerDate**, and populate the object with the result of calling the **Expression.Expressions.Constant** method. Pass the *startDate* parameter as a parameter to the method call.

Note: A **ConstantExpression** object is an expression that represents the results of evaluating a constant value. In this case, the object represents the value in the startDate variable.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.MemberExpression testDateMember =
        Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testDateProperty);

    Expressions.ConstantExpression lowerDate =
        Expressions.Expression.Constant(startDate);
}
```

5. Add code to create an **Expressions.BinaryExpression** object named **lowerDateCondition**, and populate the object with the result of calling the **Expressions.Expression.GreaterThanOrEqual** method. Pass the **testDateMember** and **lowerDate** objects as parameters to the method call.

Note: The **GreaterThanOrEqual** method generates a binary expression that combines the **testDateMember** object (representing the "this.startDate" portion of the expression) and the **lowerDate** object (representing the "startDate" portion of the expression) to generate a tree for the expression **"this.startDate >= startDate"**.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.ConstantExpression lowerDate =
        Expressions.Expression.Constant(startDate);
    Expressions.BinaryExpression lowerDateCondition =
        Expressions.Expression.GreaterThanOrEqual(
            testDateMember, lowerDate);
}
```

6. By using the same principles that you saw in Steps 4 and 5, add code to perform the following tasks:

    a. Create a **ConstantExpression** object named **upperDate** by passing the *endDate* parameter as a parameter to the method call.

    b. Create a **BinaryExpression** object named **upperDateCondition** by invoking the **Expression**.**LessThanOrEqual** method. Pass the **testDateMember** and **upperDate** objects as parameters to the method call.

Note: This code should build the second part of the date evaluation expression, which represents **"endDate <= testDateMember"**.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    Expressions.BinaryExpression lowerDateCondition =
        Expressions.Expression.GreaterThanOrEqual
        (testDateMember, lowerDate);
    // Generate the expression:
    //
    //    item.Testdate <= endDate
    //
    Expressions.ConstantExpression upperDate =
        Expressions.Expression.Constant(endDate);
    Expressions.BinaryExpression upperDateCondition =
        Expressions.Expression.LessThanOrEqual(
            testDateMember, upperDate);
}
```

7. Add code to combine the expressions in the **lowerDate** and **upperDate ExpressionTree** objects into a single Boolean expression tree that returns **true** if both conditions are true or **false** otherwise; call the **Expressions**.**Expression.AndAlso** static method to combine the expressions together, and store the result in the **dateCondition** object.

Note: The **Expressions.Expression.AndAlso** method combines the two discrete expressions that you just created, **"item.TestDate >= startDate"** and **"Item.TestDate <= endDate"** into a **BinaryExpression** object that represents the expression **"item.TestDate >= startDate && Item.TestDate <= endDate"**.

Your code should resemble the following code example.

```
if (dateRangeSpecified)
{
    ...
    // Combine the expressions with the && operator.
    dateCondition = Expressions.Expression.AndAlso(
        lowerDateCondition, upperDateCondition);
}
```

8. Build the project and correct any errors:

- On the **Build** menu, click **Build Solution**. Correct any errors.

Task 4: Dynamically build numeric expression trees

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

   The existing code in this method defines a **BinaryExpression** object named **booleanCondition**. This object will be used to return the expression tree that evaluates conditions based on **short** integer values. You will add code to this conditional statement to build an expression tree that is equivalent to the expression in the following code example, where *propertyName* represents the value of the *propertyName* parameter.

```
item.PropertyName >= lowerRange && item.PropertyName <= upperRange
```

2. Delete the **TODO - Complete the BuildNumericExpressionBody method** comment, and then add code to generate the first half of the expression by performing the following tasks:

   a. Create a new **MemberInfo** object named **testProperty**. Call the **GetProperty** method to generate code that retrieves the property that the *propertyName* parameter specifies from the *testResultType* type parameter.

   b. Create a **MemberExpression** object named **testMember** that represents access to the property that the **testProperty** object specifies; call the static **Expression.MakeMemberAccess** method, passing the *itemBeingQueried* parameter and the **testProperty** object as parameters.

   c. Create a **ConstantExpression** object named **lowerValue** by invoking the **Expression**.**Constant** method. Pass the *lowerRange* parameter as a parameter to the method call.

   d. Create a **BinaryExpression** object named **lowerValueCondition**, which combines the **testMember** and **lowerValue** expression objects into a **GreaterThanOrEqual** binary expression.

Hint: Your code should build the first half of the target expression, which represents **"item.***PropertyName* **>= lowerRange"**, where *PropertyName* represents the value of the *propertyName* parameter. Your code should use similar syntax to that used to generate the expression in Task 3

Your code should resemble the following code example.

```
private Expressions.BinaryExpression BuildNumericExpressionBody(bool
rangeSpecified, short lowerRange, short upperRange, Type
testResultType, string propertyName, Expressions.ParameterExpression
itemBeingQueried)
{
    ...
    if (rangeSpecified)
    {
        // Generate the expression:
        //
        //    item.<Property> >= lowerRange
        //
        MemberInfo testProperty =
            testResultType.GetProperty(propertyName);

        Expressions.MemberExpression testMember =
            Expressions.Expression.MakeMemberAccess(
            itemBeingQueried, testProperty);

        Expressions.ConstantExpression lowerValue =
            Expressions.Expression.Constant(lowerRange);

        Expressions.BinaryExpression lowerValueCondition =
            Expressions.Expression.GreaterThanOrEqual(
                testMember, lowerValue);
        ...
    }
    ...
}
```

3. Add code to generate the second half of the target expression by performing the following tasks:

   a. Create a **ConstantExpression** object named **upperValue** by invoking the static **Expression**.**Constant** method. Pass the *upperRange* parameter as a parameter to the method call.

   b. Create a **BinaryExpression** object named **upperValueCondition**, which combines the **testMember** and **upperValue** expression objects into a **LessThanOrEqual** binary expression.

Hint: Your code should build the second half of the target expression, which represents **"item**.*PropertyName* **<= upperRange"**, where *PropertyName* represents the value of the *propertyName* parameter. Your code should again use similar syntax to that used to generate the expression in Task 3.

Your code should resemble the following code example.

```
if (rangeSpecified)
{
    ...
    // Generate the expression:
    //
    //     item.<Property> <= upperRange
    //
    Expressions.ConstantExpression upperValue =
        Expressions.Expression.Constant(upperRange);

    Expressions.BinaryExpression upperValueCondition =
        Expressions.Expression.LessThanOrEqual(
            testMember, upperValue);
}
```

4. At the end of the method, add code to set the **booleanCondition** object to an expression that combines the **lowerValueCondition** and **upperValueCondition** expressions by using the static **Expressions.Expression.AndAlso** method.

   Your code should resemble the following code example.

```
if (rangeSpecified)
{
    ...

    // Combine the expressions with &&
    booleanCondition =
        Expressions.Expression.AndAlso(
            lowerValueCondition, upperValueCondition);
}
```

5. Build the project and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 5: Combine the expression trees

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

   This method takes four parameters that define the expression trees for each of the possible criteria that the user can enter. If any criteria are missing, the corresponding expression tree is null. The purpose of this method is to combine these expression trees together into an overall expression tree that includes all of the criteria that the user enters.

The existing code in this method creates an **Expression** object named **body**. This object will contain the combined binary expressions that form the body of the LINQ query. If the user did not specify any criteria, the **body** object is assigned an expression tree that contains the Boolean constant **true**. This expression tree causes all items to be retrieved.

2. Delete the **TODO - Complete the BuildLambdaExpressionBody method** comment, and then add code to check whether the *dateCondition* parameter is null. If not, set the **body** object to the **dateCondition** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...
    Expressions.Expression body = null;
    if (dateCondition != null)
    {
        body = dateCondition;
    }
    ...
}
```

3. Add code to check whether the *temperatureCondition* parameter is null, and if not, perform the following tasks:

   a. If the **body** object is null, set the **body** object to the **temperatureCondition** expression tree.

   b. If the **body** object is not null, combine the expression tree in the body object with the expression tree in the *temperatureCondition* parameter by using the static **Expressions.Expression**.**AndAlso** method. Assign the result to the **body** object.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...
    // Add the temperatureCondition expression.
    if (temperatureCondition != null)
    {
        if (body == null)
        {
            body = temperatureCondition;
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                       body, temperatureCondition);
        }
    }
    ...
}
```

4. Repeat the logic in Step 3 and add the expression tree that the *appliedStressCondition* parameter defines to the **body** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...

    // Repeat the same logic for the remaining condition expressions.
    if (appliedStressCondition != null)
    {
        if (body == null)
        {
            body = appliedStressCondition;
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                       body, appliedStressCondition);
        }
    }
    ...
}
```

5. Repeat the logic in Step 3 and add the expression tree that the *deflectionCondition* parameter defines to the **body** expression tree.

   Your code should resemble the following code example.

```
private Expressions.Expression BuildLambdaExpressionBody(...)
{
    ...

    // Repeat the same logic for the remaining condition expressions.
    ...

    if (deflectionCondition != null)
    {
        if (body == null)
        {
            body = deflectionCondition;
        }
        else
        {
            body = Expressions.Expression.AndAlso(
                       body, deflectionCondition);
        }
    }
    ...
}
```

6. Build the project and correct any errors:

   - On the **Build** menu, click **Build Solution**. Correct any errors.

Task 6: Build a lambda expression for the OrderBy statement

1. In the task list, locate the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** task, and then double-click this task. This task is located in the **BuildLambdaExpressionForOrderBy** method.

   The purpose of this method is to construct an expression that specifies the order in which the data should be retrieved. The parameter to this method is a value from the **OrderByKey** enumeration. This enumeration is defined as part of the application and contains the following values: **ByDate**, **ByTemperature**, **ByAppliedStress**, **ByDeflection**, and **None**. The following code example shows the form of the lambda expression that this method generates.

```
item => item.Property
```

   In this example, Property references the property from the **TestResult** type that corresponds to the parameter that is passed into the method. If the user does not specify a sort key, this method returns a null value.

2. Delete the **TODO - Create the type reference and ParameterExpression in the BuildLambdaExpressionForOrderBy method** comment, and then add code to the method to create the **ParameterExpression** object that defines the parameter for the lambda expression by performing the following tasks:

Note: You will need to create a **Type** reference to the **TestResult** object, and the lambda expression should refer to the object **item**.

a. Create a **Type** reference named **testResultType** by using the **typeOf** operator and passing a **TestResult** object as a parameter.

b. Create a **ParameterExpression** object named **itemBeingQueried** by using the static **Expressions.Expression.Parameter** method. Specify the **testResultType** object and a string that contains the text "**item**" as parameters to the method.

Your code should resemble the following code example.

```
private Expressions.Expression<Func<TestResult, ValueType>>
    BuildLambdaExpressionForOrderBy(OrderByKey orderByKey)
{
    ...

    if (orderByKey != OrderByKey.None)
    {
        // Create the expression that defines the parameter for the
        // lambda expression.
        // The type is TestResult, and the lambda expression refers to
        // it with the name "item".

        Type testResultType = typeof(TestResult);
        Expressions.ParameterExpression itemBeingQueried =
            Expressions.Expression.Parameter(testResultType, "item");
    ...

    }

    ...
}
```

3. In the **BuildLambdaExpressionForOrderBy** method, replace the **TODO - Create a MemberExpression and MemberInfo object** comment with code to perform the following tasks:

   a. Create a **MemberExpression** object named **sortKey**, and initialize this object to null.

   b. Create a **MemberInfo** object named **property**, and initialize this object to null.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...

    // Create the expression that will define the sort key that
    // the lambda expression returns.

    // This expression will reference one of the properties in the
    // TestResult structure depending on the key that the user
    // specifies.

    Expressions.MemberExpression sortKey = null;
    MemberInfo property = null;

    ...
}
```

4. Replace the **TODO - Evaluate the orderByKey parameter to determine the property to sort by** comment with code to evaluate the *orderByKey* parameter. Use the **GetProperty** method of the testResultType variable to generate code that retrieves the corresponding property value from the item that is specified as the parameter to the lambda expression. Store the result in the property variable. The following table lists the name of each property to use, depending on the value of the *orderByKey* parameter.

orderByKey value	testResultType property to use
ByDate	**"TestDate"**
ByTemperature	**"Temperature"**
ByAppliedStress	**"AppliedStress"**
ByDeflection	**"Deflection"**

Note: Near the beginning of the **BuildLambdaExpressionForOrderBy** method, a conditional statement prevents the method from performing this code if the *orderByKey* parameter has the value **OrderByKey**.**None**; therefore, you do not need to check for this value.

Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...

    MemberInfo property = null;

    switch(orderByKey)
    {
        case OrderByKey.ByDate:
            // If the user selected the date column, set the property
            // object to TestDate.
            property = testResultType.GetProperty("TestDate");
            break;
        case OrderByKey.ByTemperature:
            // If the user selected the temperature column, set the
            // property object to Temperature.
            property = testResultType.GetProperty("Temperature");
            break;
        case OrderByKey.ByAppliedStress:
            // If the user selected the applied stress column, set the
            // property object to AppliedStress.
            property = testResultType.GetProperty("AppliedStress");
            break;
        case OrderByKey.ByDeflection:
            // If the user selected the deflection column, set the
            // property object to Deflection.
            property = testResultType.GetProperty("Deflection");
            break;
    }

...

}
```

5. Replace the **TODO - Construct the expression that specifies the OrderBy field** comment with code that retrieves the value that the property variable specifies from the item that the itemBeingQueried variable specifies. To do this, call the static **Expressions.Expression.MakeMemberAccess** method, and pass the **itemBeingQueried** expression tree and the **property** object as parameters to this method.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Construct an expression that specifies the value in the field
    // that the property object references in the TestResult object.
    sortKey = Expressions.Expression.MakeMemberAccess(
        itemBeingQueried, property);
    ...
}
```

6. Replace the **TODO - Create a UnaryExpression object to convert the sortKey object to a ValueType** comment with code to create a new **UnaryExpression** object named **convert** by invoking the static **Expressions.Expression.Convert** method. Pass the **sortKey** object and the type of the **ValueType** type as parameters to the method call. This step is necessary because the possible sort keys are all value types, and they must be converted to **ValueType** objects for the ordering to function correctly.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Cast the sortKey object to a ValueType object (ValueType is the
    // ancestor of all value types, including DateTime and short).
    Expressions.UnaryExpression convert =
        Expressions.Expression.Convert(sortKey, typeof(ValueType));
    ...
}
```

7. Replace the **TODO - Create the OrderBy lambda expression** comment with code to combine the converted unary expression that contains the sort key and the itemBeingQueried variable into a lambda expression by using the static **Expression.Lambda** generic method. Specify the type **Func<TestResult, ValueType>** as the type parameter to the **Lambda** method; the resulting lambda expression takes a **TestResult** object as the parameter and returns a **ValueType** object.

   Your code should resemble the following code example.

```
if (orderByKey != OrderByKey.None)
{
    ...
    // Build the lambda expression by using the parameter and the
    // expression that contains the sort key.
    lambda = Expressions.Expression.Lambda
        <Func<TestResult, ValueType>>(convert, itemBeingQueried);
}
```

8. Build the project and correct any errors:

- On the **Build** menu, click **Build Solution**. Correct any errors.

Task 7: Examine the CreateQuery method

- In the task list, locate the **TODO - Examine the CreateQuery method** task, and then double-click this task. This task is located in the **CreateQuery** method.

  This method is the starting point for the lambda expression generation. The method accepts parameters that indicate which query criteria the lambda expression should include and the upper and lower ranges for each of these criteria.

  The method first calls the **BuildLambdaExpressionForQueryCriteria** method to construct a lambda expression that incorporates the query criteria. It then calls the **BuildLambdaExpressionForOrderBy** method to construct the lambda expression that defines the sort order for retrieving the data. Note that, at this point, it is possible that either of these expressions may still be null if the user either did not specify any criteria or did not specify a sort key.

  After the method creates the expression objects, it creates an **IEnumerable** generic collection named **query** that is based on the **TestResult** type, and it initializes the object with the data in the **stressData** parameter.

  If the lambda expression that specifies the query criteria is not null, the method then filters the data in the **IEnumerable** collection by invoking the **Where** LINQ extension method on the collection. The parameter to the **Where** method is the lambda expression that contains the query criteria. Note that the **Compile** method of an **Expression<TDelegate>** object converts the expression tree into a compiled lambda expression that the common language runtime (CLR) can execute.

  If the lambda expression that defines the sort order is not null, this method then applies this lambda expression to the **IEnumerable** collection by using the **OrderBy** LINQ extension method. As before, the **Compile** method converts the expression tree that defines the sort key into code that can be executed by using the CLR.

  If the user specifies that the query should return a limited number of rows, the **Take** LINQ extension method is applied to the **IEnumerable** collection with the limit that the user specifies.

  Finally, the **IEnumerable** collection is returned to the caller. Note that this method does not run the LINQ query. This action occurs in the

DisplayResults_Click method, when the code calls the **Count** method of the **IEnumerable** collection.

Task 8: Test the solution

1. Run the application:
   - On the **Debug** menu, click **Start Without Debugging**.
2. In the Stress Data Analyzer window, click **Display** to display all results with no query criteria, sort key, or limit to the number of items that are returned. Note the time that it takes to execute the query.

Note: This test is different from the test that you performed at the end of the first exercise. In the original application, the LINQ query used a lambda expression that contained criteria for all properties, whereas this test does not use any criteria. Therefore, the operation should be faster.

3. Select the **Test Date** and **Temperature** check boxes, modify the search criteria to the values in the following table, and then click **Display** again.

Criteria	Value
Test Date	From 02/01/2009 To 02/28/2009
Temperature	From 250 to 450

4. When the query is complete, examine the contents of the box in the lower part of the window. The search should return 1,676 values, as in the test in Exercise 1. However, the time it takes to execute the query should again be less than the time that you recorded in Exercise 1.
5. Clear the **Test Date** and **Temperature** check boxes, and then select the **Limit?** check box. Set the limit value to 2,000, and then click **Display**.

   Note that when the number of rows is reduced, the time it takes to execute the query is substantially reduced.
6. In the **Order By** section, select **Temperature**, and then click **Display** again.

   Note that the expression takes substantially longer to execute when a sort key is included in the expression.
7. Close the Stress Data Analyzer window, and then return to Visual Studio.

8. Close Visual Studio:
    - In Visual Studio, on the **File** menu, click **Exit**.