# Lab 2: Using C# Programming Constructs

## Exercise 1: Calculating Square Roots with Improved Accuracy

### Task 1: Create a new WPF Application project

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$word**.
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. Create a new project called SquareRoots by using the Windows Presentation Foundation (WPF) Application template in the E:\Labfiles\Lab 2\Ex1\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
   b. In the **New Project** dialog box, in the Project Types pane, expand **Visual C#**, and then click **Windows**.
   c. In the Templates pane, click **WPF Application**.
   d. Specify the following values for each of the properties in the dialog box, and then click **OK**:
      - Name: **SquareRoots**
      - Location: **E:\Labfiles\Lab 2\Ex1\Starter**
      - Solution name: **SquareRoots**
      - Create directory for solution: Select the check box.

### Task 2: Create the user interface

1. Add **TextBox**, **Button**, and two **Label** controls to the MainWindow window. Place them anywhere in the window:

    a. In Solution Explorer, double-click **MainWindow.xaml**.
    b. Click the **Toolbox** tab.
    c. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.
    d. Drag the **TextBox** control anywhere into the MainWindow window.
    e. Click the **Toolbox** tab.
    f. Drag the **Button** control anywhere into the MainWindow window.
    g. Click the **Toolbox** tab.
    h. Drag the **Label** control anywhere into the MainWindow window.
    i. Click the **Toolbox** tab.
    j. Drag the **Label** control anywhere into the MainWindow window.

2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

| Control | Property | Value |
|---|---|---|
| **TextBox** | **Name** | **inputTextBox** |
| | **Height** | **28** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **12,12,0,0** |
| | **Text** | **0.00** |
| | **VerticalAlignment** | **Top** |
| | **Width** | **398** |
| **Button** | **Name** | **calculateButton** |
| | **Content** | **Calculate** |
| | **Height** | **23** |
| | **HorizontalAlignment** | **Right** |
| | **Margin** | **0,11,12,0** |

| Control | Property | Value |
|---|---|---|
| | **VerticalAlignment** | **Top** |
| | **Width** | **75** |
| **Label** | **Name** | **frameworkLabel** |
| | **Content** | **0.00 (Using .NET Framework)** |
| | **Height** | **28** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **12,41,0,0** |
| | **VerticalAlignment** | **Top** |
| | **Width** | **479** |
| **Label** | **Name** | **newtonLabel** |
| | **Content** | **0.00 (Using Newton)** |
| | **Height** | **28** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **12,75,0,0** |
| | **VerticalAlignment** | **Top** |
| | **Width** | **479** |

a. In the MainWindow window, click the **TextBox** control.

b. In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.

c. In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.

d. Repeat this process for the remaining properties of the **TextBox** control.

e. In the MainWindow window, click the **Button** control.

f. Follow the procedure described in steps b to d to set the specified properties for this control.

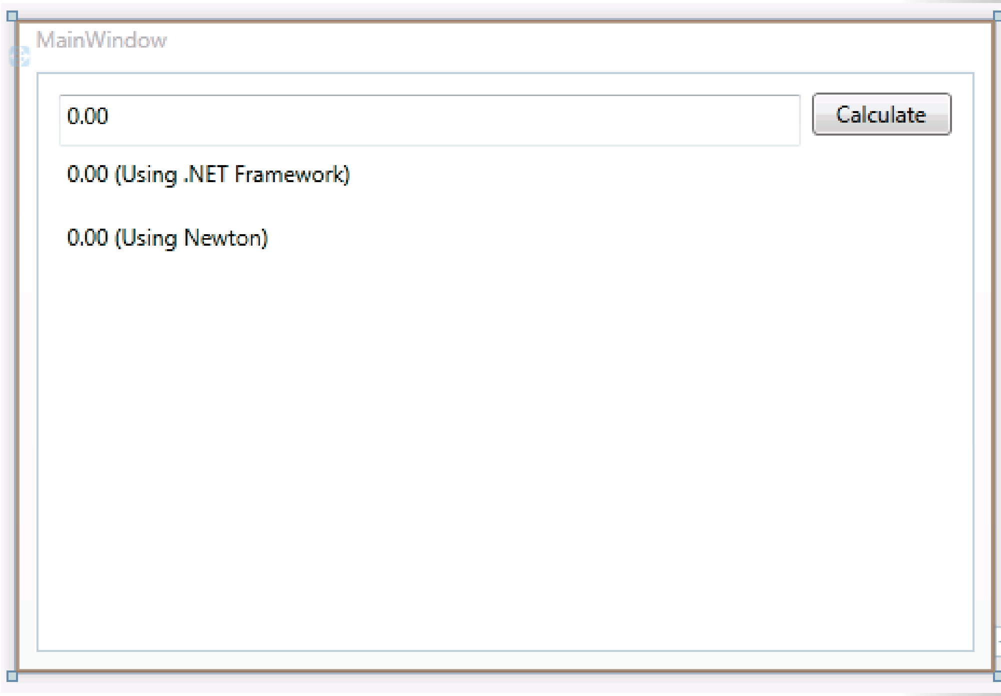
g. In the MainWindow window, click one of the **Label** controls.

h. Follow the procedure described in steps b to d to set the specified properties for this control.

i. In the MainWindow window, click the other **Label** control.

j. Follow the procedure described in steps b to d to set the specified properties for this control.

The MainWindow window should look like the following screen shot.



## Task 3: Calculate square roots by using the Math.Sqrt method of the .NET Framework

1. Create an event handler for the **Click** event of the button:
   - In the MainWindow window, click the **Button** control.
   - In the Properties window, click the **Events** tab.

- In the list of events, double-click the **Click** event.

2. In the **calculateButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into a **double**. Store the double value in a variable called numberDouble. Use the **TryParse** method of the **double** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "Please enter a double," and then execute a **return** statement to quit the method:

**Note**: You can display a message in a message box by using the **MessageBox.Show** method.

- Add the code in the following code example to the **calculateButton_Click** method.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox
    double numberDouble;
    if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }
}
```

3. Check that the value that the user enters is a positive number. If it is not, display a message box with the text "Please enter a positive number," and then return from the method:

- Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Check that the user has entered a positive number
    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }
}
```

4. Calculate the square root of the value in the numberDouble variable by using the **Math.Sqrt** method. Store the result in a double variable called squareRoot:

   - Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Use the .NET Framework Math.Sqrt method
    double squareRoot = Math.Sqrt(numberDouble);
}
```

5. Format the value in the squareRoot variable by using the layout shown in the following code example, and then display it in the **frameWorkLabel Label** control.

```
99.999 (Using the .NET Framework)
```

   Use the **string.Format** method to format the result. Set the **Content** property of a **Label** control to display the formatted result:

   - Add the statements in the following code example to the **calculateButton_Click** method, after the code that you added in the previous step.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Format the result and display it
    frameworkLabel.Content = string.Format("{0} (Using the .NET
Framework)", squareRoot);
}
```

   At this point, your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox
    double numberDouble;
    if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }
    // Check that the user has entered a positive number
    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }

    // Use the .NET Framework Math.Sqrt method
    double squareRoot = Math.Sqrt(numberDouble);

    // Format the result and display it
    frameworkLabel.Content = string.Format("{0} (Using the .NET
Framework)", squareRoot);
}
```

6. Build and run the application to test your code. Use the test values that are shown in the following table, and then verify that the correct square roots are calculated and displayed (ignore the "Using Newton" label for the purposes of this test).

| Test value | Expected result |
|---|---|
| 25 | 5 |
| 625 | 25 |
| 0.00000001 | 0.0001 |
| –10 | Message box appears with the message "Please enter a positive number" |
| Fred | Message box appears with the message "Please enter a double" |
| 10 | 3.16227766016838 |
| 8.8 | 2.96647939483827 |
| 2.0 | 1.4142135623731 |
| 2 | 1.4142135623731 |

   a. On the **Debug** menu, click **Start Debugging**.

   b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.

c. Verify that the result matches the text in the **Expected result** column.

d. Repeat steps b and c for each row in the table.

7. Close the application and return to Visual Studio.

## Task 4: Calculate square roots by using Newton's method

1. In the **calculateButton_Click** method, after the code that you added in the previous task, create a decimal variable called numberDecimal. Initialize this variable with the data that the user enters in the **inputTextBox TextBox** control, but convert it into a **decimal** this time (previously, you read it as a **double**). If the text that the user enters is not valid, display a message box with the text "Please enter a decimal," and then execute a **return** statement to quit the method:

   - Add the code in the following code example to the end of the **calculateButton_Click** method.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Newton's method for calculating square roots

    // Get user input as a decimal
    decimal numberDecimal;
    if (!decimal.TryParse(inputTextBox.Text, out numberDecimal))
    {
        MessageBox.Show("Please enter a decimal");
        return;
    }
}
```

**Note**: This step is necessary because the **decimal** and **double** types have different ranges. A number that the user enters that is a valid **double** might be out of range for the **decimal** type.

2. Declare a decimal variable called delta, and initialize it to the value of the expression **Math.Pow(10, –28)**. This is the smallest value that the **decimal** type supports, and you will use this value to determine when the answer that is generated by using Newton's method is sufficiently accurate. When the difference between two successive estimates is less than this value, you will stop.

**Note**: The **Math.Pow** method returns a double. You will need to use the **Convert.ToDecimal** method to convert this value to a decimal before you assign it to the delta variable.

Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Specify 10 to the power of -28 as the minimum delta between
    // estimates. This is the minimum range supported by the decimal
    // type. When the difference between 2 estimates is less than this
    // value, then stop.
    decimal delta = Convert.ToDecimal(Math.Pow(10, -28));
}
```

3. Declare another decimal variable called guess, and initialize it with the initial guess at the square root. This initial guess should be the result of dividing the value in **numberDecimal** by 2.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Take an initial guess at an answer to get started
    decimal guess = numberDecimal / 2;
}
```

4. Declare another decimal variable called result. You will use this variable to generate values for each iteration of the algorithm, based on the value from the previous iteration. Initialize the result variable to the value for the first iteration by using the expression **((numberDecimal** / **guess)** + **guess)** / **2**.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Estimate result for the first iteration
    decimal result = ((numberDecimal / guess) + guess / 2);
}
```

5. Add a **while** loop to generate further refined guesses. The body of the **while** loop should assign result to guess, and generate a new value for result by using the expression **((numberDecimal** / **guess)** + **guess)** / **2**. The **while** loop

should terminate when the difference between result and guess is less than or equal to delta.

**Note**: Use the **Math.Abs** method to calculate the absolute value of the difference between result and guess. Using Newton's algorithm, it is possible for the difference between the two variables to alternate between positive and negative values as it diminishes. Consequently, if you do not use the **Math.Abs** method, the algorithm might terminate early with an inaccurate result.

Your code should resemble the following code example.

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

    // While the difference between values for each current iteration
    // is not less than delta, then perform another iteration to
    // refine the answer.

    while (Math.Abs(result - guess) > delta)
    {
        // Use the result from the previous iteration
        // as the starting point
        guess = result;

        // Try again
        result = ((numberDecimal / guess) + guess) / 2;
    }
}
```

6. When the **while** loop has terminated, format and display the value in the result variable in the **newtonLabel Label** control. Format the data in a similar manner to the previous task.

   Your code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the result
    newtonLabel.Content = string.Format("{0} (Using Newton)", result);
}
```

Your completed code should resemble the following code example.

```
private void calculateButton_Click(object sender, RoutedEventArgs e)
{
    // Get a double from the TextBox

    double numberDouble;

  if (!double.TryParse(inputTextBox.Text, out numberDouble))
    {
        MessageBox.Show("Please enter a double");
        return;
    }

    // Check that the user has entered a positive number

    if (numberDouble <= 0)
    {
        MessageBox.Show("Please enter a positive number");
        return;
    }

    // Use the .NET Framework Math.Sqrt method

    double squareRoot = Math.Sqrt(numberDouble);

    // Format the result and display it

    frameworkLabel.Content = string.Format("{0} (Using .NET
Framework)", squareRoot);

    // Newton's method for calculating square roots

    // Get the user input as a decimal

    decimal numberDecimal;
    if (!decimal.TryParse(inputTextBox.Text, out numberDecimal))
    {
        MessageBox.Show("Please enter a decimal");
        return;
    }

    // Specify 10 to the power of -28 as the minimum delta between
    // estimates. This is the minimum range supported by the decimal
    // type. When the difference between 2 estimates is less than this
    // value, then stop.

    decimal delta = Convert.ToDecimal(Math.Pow(10, -28));

    // Take an initial guess at an answer to get started
    decimal guess = numberDecimal / 2;

    // Estimate result for the first iteration
    decimal result = ((numberDecimal / guess) + guess) / 2);

    // While the difference between values for each current iteration
    // is not less than delta, then perform another iteration to
    // refine the answer.
    while (Math.Abs(result - guess) > delta)
    {
        // Use the result from the previous iteration
        // as the starting point
        guess = result;

        // Try again
        result = ((numberDecimal / guess) + guess) / 2;
    }

    // Display the result
    newtonLabel.Content = string.Format("{0} (Using Newton)", result);
}
```

## Task 5: Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the correct square roots are calculated and displayed. Compare the value in the two labels, and then verify that the square roots that are calculated by using Newton's method are more accurate than those calculated by using the **Math.Sqrt** method.

| Test value | .NET Framework | Newton's algorithm |
|---|---|---|
| 25 | 5 | 5.0000000000000000000000000000 |
| 625 | 25 | 25.000000000000000000000000000 |
| 0.00000001 | 0.0001 | 0.0001000000000000000000000000 |
| 10 | 3.16227766016838 | 3.1622776601683793319988935444 |
| 8.8 | 2.96647939483827 | 2.9664793948382651794845589763 |
| 2.0 | 1.4142135623731 | 1.4142135623730950488016887242 |
| 2 | 1.4142135623731 | 1.4142135623730950488016887242 |

   a. On the **Debug** menu, click **Start Debugging**.
   b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.
   c. Repeat step b for each row in the table.

2. As a final test, try the value 0.0000000000000000000000000001 (27 zeroes after the decimal point). Can you explain the result?
   - The program halts and reports that **DivideByZeroException** was unhandled. This exception occurs because the value that is specified is smaller than the range of values that the **decimal** type allows, so the value in the guess variable is 0. You will see how to catch and handle exceptions in a later module.
3. Close the application and return to Visual Studio:
   - On the **Debug** menu, click **Stop Debugging**.

## Exercise 2: Converting Integer Numeric Data to Binary

### Task 1: Create a new WPF Application project

- Create a new project called IntegerToBinary by using the WPF Application template in the E:\Labfiles\Lab 2\Ex2\Starter folder:
   a. In Visual Studio, on the **File** menu, point to **New**, and then click **Project**.
   b. In the **New Project** dialog box, in the Project Types pane, expand **Visual C#**, and then click **Windows**.
   c. In the Templates pane, click **WPF Application**.
   d. Specify the following values for each of the properties in the dialog box, and then click **OK**:
      - Name: **IntegerToBinary**.
      - Location: **E:\Labfiles\Lab 2\Ex2\Starter**
      - Solution name: **IntegerToBinary**
      - Create directory for solution: Select the check box.

### Task 2: Create the user interface

1. Add a **TextBox**, **Button**, and **Label** control to the MainWindow window. Place them anywhere in the window:
   a. In Solution Explorer, double-click **MainWindow.xaml**.
   b. Click the **Toolbox** tab.
   c. Expand the **Common WPF Controls** section of the Toolbox if it is not already open.
   d. Drag the **TextBox** control anywhere into the MainWindow window.
   e. Click the **Toolbox** tab.
   f. Drag the **Button** control anywhere into the MainWindow window.
   g. Click the **Toolbox** tab.
   h. Drag the **Label** control anywhere into the MainWindow window.
2. Using the Properties window, set the properties of each control by using the values in the following table. Leave any other properties at their default values.

| Control | Property | Value |
|---|---|---|
| **TextBox** | **Name** | **inputTextBox** |
| | **Height** | **28** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **12,12,0,0** |
| | **Text** | **0** |
| | **VerticalAlignment** | **Top** |
| | **Width** | **120** |
| **Button** | **Name** | **convertButton** |
| | **Content** | **Convert** |
| | **Height** | **23** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **138,12,0,0** |

| Control | Property | Value |
|---|---|---|
| | **VerticalAlignment** | **Top** |
| | **Width** | **75** |
| **Label** | **Name** | **binaryLabel** |
| | **Content** | **0** |
| | **Height** | **28** |
| | **HorizontalAlignment** | **Left** |
| | **Margin** | **12,41,0,0** |
| | **VerticalAlignment** | **Top** |
| | **Width** | **120** |

a. In the MainWindow window, click the **TextBox** control.

b. In the Properties window, click the text **textBox1** adjacent to the **TextBox** prompt, and then change the name to **inputTextBox**.

c. In the list of properties in the Properties window, locate the **Height** property, and then change it to **28**.

d. Repeat this process for the remaining properties of the **TextBox** control.

e. In the MainWindow window, click the **Button** control.

f. Follow the procedure described in steps b to d to set the specified properties for this control.

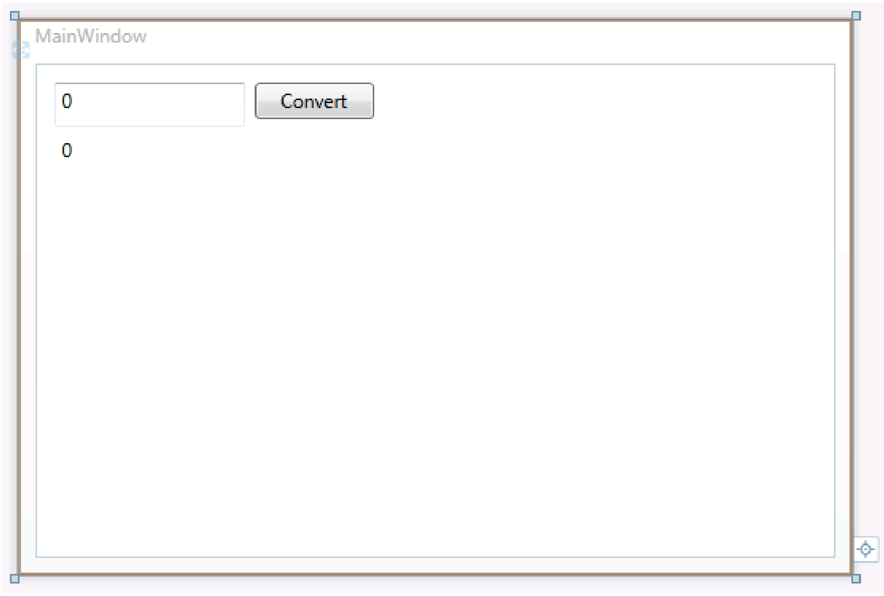
g. In the MainWindow window, click one of the **Label** controls.

h. Follow the procedure described in steps b to d to set the specified properties for this control.

i. In the MainWindow window, click the other **Label** control.

j. Follow the procedure described in steps b to d to set the specified properties for this control.

The MainWindow window should look like the following screen shot.



## Task 3: Add code to generate the binary representation of an integer value

1. Create an event handler for the **Click** event of the button:

   a. In the MainWindow window, click the **Button** control.

   b. In the Properties window, click the **Events** tab.

   c. In the list of events, double-click the **Click** event.

2. In the **convertButton_Click** method, add code to read the data that the user enters in the **inputTextBox TextBox** control, and then convert it into an **int** type. Store the integer value in a variable called i. Use the **TryParse** method of the **int** type to perform the conversion. If the text that the user enters is not valid, display a message box with the text "TextBox does not contain an integer," and then execute a **return** statement to quit the method:

   - Add the code in the following code example to the **convertButton_Click** method.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    // Get the integer entered by the user
    int i;
    if (!int.TryParse(inputTextBox.Text, out i))
    {
        MessageBox.Show("TextBox does not contain an integer");
        return;
    }
}
```

3. Check that the value that the user enters is not a negative number (the integer-to-binary conversion algorithm does not work for negative numbers). If it is negative, display a message box with the text "Please enter a positive number or zero," and then return from the method:

   - Add the statements in the following code example to the **convertButton_Click** method, after the code that you added in the previous step.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Check that the user has not entered a negative number
    if (i < 0)
    {
        MessageBox.Show("Please enter a positive number or zero");
        return;
    }
}
```

4. Declare an integer variable called remainder and initialize it to zero. You will use this variable to hold the remainder after dividing i by 2 during each iteration of the algorithm.

   Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Remainder will hold the remainder after dividing i by 2
    // after each iteration of the algorithm
    int remainder = 0;
}
```

5. Declare a StringBuilder variable called binary and instantiate it. You will use this variable to construct the string of bits that represent i as a binary value.

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Binary will be used to construct the string of bits
    // that represent i as a binary value
    StringBuilder binary = new StringBuilder();
}
```

6. Add a **do** loop that performs the following tasks:

   a. Calculate the remainder after dividing i by 2, and then store this value in the remainder variable.

   b. Divide i by 2.

   c. Prefix the value of remainder to the start of the string being constructed by the binary variable.

   Terminate the **do** loop when i is less than or equal to zero.

**Note**: To prefix data into a **StringBuilder** object, use the **Insert** method of the **StringBuilder** class, and then insert the value of the data at position 0.

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Generate the binary representation of i
    do
    {
        remainder = i % 2;
        i = i / 2;
        binary.Insert(0, remainder);
    }
    while (i > 0);
}
```

7. Display the value in the binary variable in the **binaryLabel Label** control.

**Note**: Use the **ToString** method to retrieve the string that a **StringBuilder** object constructs. Set the **Content** property of the **Label** control to display this string.

Your code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the result
    binaryLabel.Content = binary.ToString();
}
```

Your completed code should resemble the following code example.

```
private void convertButton_Click(object sender, RoutedEventArgs e)
{
    // Get the integer entered by the user
    int i;
    if (!int.TryParse(inputTextBox.Text, out i))
    {
        MessageBox.Show("TextBox does not contain an integer");
        return;
    }

    // Check that the user has not entered a negative number
    if (i < 0)
    {
        MessageBox.Show("Please enter a positive number or zero");
        return;
    }

    // Remainder will hold the remainder after dividing i by 2
    // after each iteration of the algorithm
    int remainder = 0;

    // Binary will be used to construct the string of bits
    // that represent i as a binary value
    StringBuilder binary = new StringBuilder();

    // Generate the binary representation of i
    do
    {
        remainder = i % 2;
        i = i / 2;
        binary.Insert(0, remainder);
    }
    while (i > 0);

    // Display the result
    binaryLabel.Content = binary.ToString();
}
```

## Task 4: Test the application

1. Build and run the application in Debug mode to test your code. Use the test values shown in the following table, and verify that the binary representations are generated and displayed.

| Test value | Expected result |
|---|---|
| 0 | 0 |
| 1 | 1 |
| –1 | Message box appears with the message "Please enter a positive number or zero" |
| 10.5 | Message box appears with the message "TextBox does not contain an integer" |
| Fred | Message box appears with the message "TextBox does not contain an integer" |
| 4 | 100 |
| 999 | 1111100111 |
| 65535 | 1111111111111111 |
| 65536 | 10000000000000000 |

   a. On the **Debug** menu, click **Start Debugging**.

   b. Enter the first value in the **Test value** column in the table in the **TextBox** control, and then click **Calculate**.

   c. Verify that the result matches the text in the **Expected result** column.

   d. Repeat steps b and c for each row in the table.

2. Close the application and return to Visual Studio:

   - On the **Debug** menu, click **Stop Debugging**.

## Exercise 3: Multiplying Matrices

### Task 1: Open the MatrixMultiplication project and examine the starter code

1. Open the MatrixMultiplication project located in the

   E:\Labfiles\Lab 2\Ex3\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 2\Ex3\Starter folder.**

   c. Click **MatrixMultiplication.sln**, and then click **Open**.

2. Examine the user interface that the MainWindow window defines:

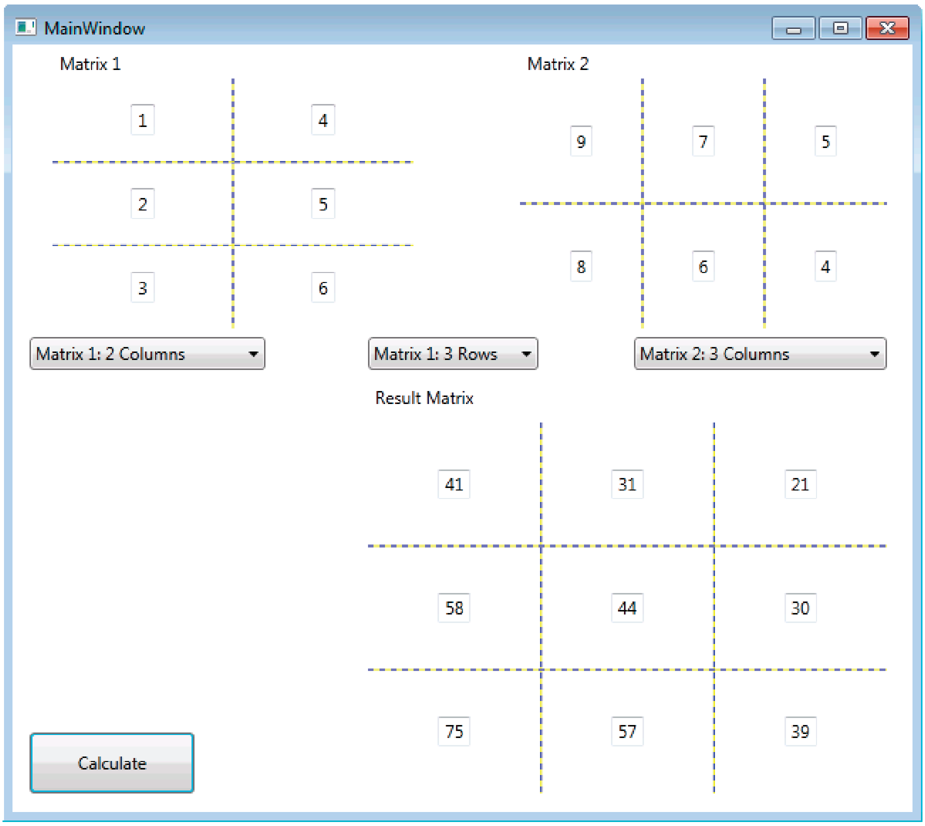
   - In Solution Explorer, double-click **MainWindow.xaml**.

   The user interface contains three **Grid** controls, three **ComboBox** controls, and a **Button** control.

   When the application runs, the first **Grid** control, labeled **Matrix 1**, represents the first matrix, and the second **Grid** control, labeled **Matrix 2**, represents the second matrix. The user can specify the dimensions of the matrices by using the **ComboBox** controls, and then enter data into each cell in them. There are several rules that govern the compatibility of matrices to be multiplied together, and Matrix 2 is automatically configured to have an appropriate number of rows based on the number of columns in Matrix 1.

   When the user clicks the **Calculate** button, Matrix 1 and Matrix 2 are multiplied together, and the result is displayed in the **Grid** control labeled **Result Matrix**. The dimensions of the result are determined by the shapes of Matrix 1 and Matrix 2.

   The following screen shot shows the completed application running. The user has multiplied a 2×3 matrix with a 3×2 matrix, and the result is a 3×3 matrix.



Matrix 1:

| | |
|---|---|
| 1 | 4 |
| 2 | 5 |
| 3 | 6 |

Matrix 2:

| | | |
|---|---|---|
| 9 | 7 | 5 |
| 8 | 6 | 4 |

Result Matrix:

| | | |
|---|---|---|
| 41 | 31 | 21 |
| 58 | 44 | 30 |
| 75 | 57 | 39 |

### Task 2: Define the matrix arrays and populate them with the data in the Grid controls

1. In Visual Studio, review the task list:

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

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

2. Open the MainWindow.xaml.cs file:

   - In Solution Explorer, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.

3. At the top of the **MainWindow** class, remove the comment **TODO Task 2 declare variables**, and then add statements that declare three two-dimensional arrays called **matrix1**, **matrix2**, and **result**. The type of the elements in these arrays should be **double**, but the size of each dimension should be omitted because the arrays will be dynamically sized based on the input that the user provides. The first dimension will be set to the number of columns, and the second dimension will be set to the number of rows.

   Your code should resemble the following code example.

```
public partial class MainWindow : Window
{
    // Declare three arrays of doubles to hold the 3 matrices:
    // The two input matrices and the result matrix
    double[,] matrix1;
    double[,] matrix2;
    double[,] result;
    ...
}
```

4. In the task list, double-click the task **TODO Task 2 Copy data from input Grids**. This task is located in the **buttonCalculate_Click** method.

5. In the **buttonCalculate_Click** method, remove the comment **TODO Task 2 Copy data from input Grids**. Add two statements that call the **getValuesFromGrid** method. This method (provided in the starter code) expects the name of a **Grid** control and the name of an array to populate with data from that **Grid** control. In the first statement, specify that the method should use the data in **grid1** to populate **matrix1**. In the second statement, specify that the method should use the data from **grid2** to populate **matrix2**.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    // Retrieve the contents of the first two grids
    // into the first two matrices
    getValuesFromGrid(grid1, matrix1);
    getValuesFromGrid(grid2, matrix2);
    ...
}
```

6. Remove the comment **TODO Task 2 Get the matrix dimensions**. Declare three integer variables called m1columns_m2rows, m1rows, and m2columns. Initialize m1columns_m2rows with the number of columns in the **matrix1** array (this is also the same as the number of rows in the **matrix2** array) by

using the **GetLength** method of the first dimension of the array. Initialize m1rows with the number of rows in the **matrix1** array by using the **GetLength** method of the second dimension of the array. Initialize m2columns with the number of columns in the **matrix2** array.

Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Discover the dimensions of the input matrices
    // (Remember that the number of columns in the first matrix will
    // always be the same as the number of rows in the second matrix)
    int m1columns_m2rows = matrix1.GetLength(0);
    int m1rows = matrix1.GetLength(1);
    int m2columns = matrix2.GetLength(0);
    ...
}
```

## Task 3: Multiply the two input matrices and calculate the result

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 3 Calculate the result**. Define a **for** loop that iterates through all of the rows in the **matrix1** array. The dimensions of an array are integers, so use an integer variable called row as the control variable in this **for** loop. Leave the body of the **for** loop blank; you will add code to this loop in the next step.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
    }
    ...
}
```

2. In the body of the **for** loop, add a nested **for** loop that iterates through all of the columns in the **matrix2** array. Use an integer variable called column as the control variable in this **for** loop. Leave the body of this **for** loop blank.

   Your code should resemble the following code example.

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

    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
        }
    }
    ...
}
```

3. The contents of each cell in the **result** array are calculated by adding the product of each item in the row identified by the row variable in **matrix1** with each item in the column identified by the column variable in **matrix2**. You will require another loop to perform this calculation, and a variable to store the result as this loop calculates it.

   In the inner **for** loop, declare a double variable called accumulator, and then initialize it to zero.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;
        }
    }
    ...
}
```

4. Add another nested **for** loop after the declaration of the accumulator variable. This loop should iterate through all of the columns in the current row in the **matrix1** array. Use an integer variable called cell as the control variable in this **for** loop. Leave the body of this **for** loop blank.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;
            // Iterate over the columns in the row in matrix1
            for (int cell = 0; cell < m1columns_m2rows; cell++)
            { }
        }
    }
    ...
}
```

5. In the body of this **for** loop, multiply the value in **matrix1[cell, row]** with the value in **matrix2[column, cell]**, and then add the result to accumulator.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Calculate the value for each cell in the result matrix
    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;

            // Iterate over the columns in the row in matrix1
            for (int cell = 0; cell < m1columns_m2rows; cell++)
            {
                // Multiply the value in the current column in the
                // current row in matrix1 with the value in the
                // current row in the current column in matrix2 and
                // add the result to accumulator
                accumulator +=
                    matrix1[cell, row] * matrix2[column, cell];
            }
        }
    }
    ...
}
```

6. After the closing brace of the innermost **for** loop, store the value in accumulator in the **result** array. The value should be stored in the cell that the column and row variables have identified.

   Your code should resemble the following code example.

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

    // Calculate the value for each cell in the result matrix

    for (int row = 0; row < m1rows; row++)
    {
        for (int column = 0; column < m2columns; column++)
        {
            // Initialize the value for the result cell
            double accumulator = 0;

            // Iterate over the columns in the row in matrix1

            for (int cell = 0; cell < m1columns_m2rows; cell++)
            {
                // Multiply the value in the current column in the
                // current row in matrix1 with the value in the
                // current row in the current column in matrix2 and
                // add the result to accumulator
                accumulator +=
                    matrix1[cell, row] * matrix2[column, cell];
            }

            result[column, row] = accumulator;
        }
    }
    ...
}
```

## Task 4: Display the results and test the application

1. In the **buttonCalculate_Click** method, delete the comment **TODO Task 4 Display the result**. The starter code contains a method called **initializeGrid** that displays the contents of an array in a **Grid** control in the WPF window. Add a statement that calls this method. Specify that the method should use the **grid3 Grid** control to display the contents of the **result** array.

   Your code should resemble the following code example.

```
private void buttonCalculate_Click(object sender, RoutedEventArgs e)
{
    ...
    // Display the results of your calculation in the third Grid
    initializeGrid(grid3, result);
}
```

2. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**. Correct any errors.
3. Run the application in Debug mode:
   - On the **Debug** menu, click **Start Debugging**.
4. In the MainWindow window, define **Matrix 1** as a 3×2 matrix and define **Matrix 2** as a 3×3 matrix.

**Note**: The number of rows in the **Matrix 2** matrix is determined by the number of columns in the **Matrix 1** matrix.

   a. In the first **ComboBox** control, select **Matrix 1: 3 Columns**.

   b. In the second **ComboBox** control, select **Matrix 1: 2 Rows**.

   c. In the third **ComboBox** control, select **Matrix 2: 3 Columns**.

5. Specify the values for the cells in the matrices as shown in the following tables.

| Matrix 1 | | |
|---|---|---|
| 1 | 5 | –9 |
| 3 | –7 | 11 |

| Matrix 2 | | |
|---|---|---|
| 2 | –8 | 14 |
| 4 | –10 | 16 |
| 6 | –12 | 18 |

6. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

| Result | | |
|---|---|---|
| –32 | 50 | –68 |
| 44 | –86 | 128 |

7. Change the data in **Matrix 2** as shown in the following table.

| Matrix 2 | | |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |

8. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

| Result | | |
|---|---|---|
| 1 | 5 | –9 |
| 3 | –7 | 11 |

**Matrix 2** is an example of an identity matrix. When you multiply a matrix by an identity matrix, the result is the same data as defined by the original matrix (it is the matrix equivalent of multiplying a value by 1 in regular arithmetic). In this case, the values in the **Result** matrix are the same as those in **Matrix 1**.

9. Change the data in **Matrix 2** again, as shown in the following table.

| Matrix 2 | | |
|---|---|---|
| –1 | 0 | 0 |
| 0 | –1 | 0 |
| 0 | 0 | –1 |

10. Click **Calculate**. Verify that the **Result** matrix displays the values in the following table.

| Result | | |
|---|---|---|
| –1 | –5 | 9 |
| –3 | 7 | –11 |

This time, the values in **Result** are the same as those in **Matrix 1** except that the sign of each element is inverted (**Matrix 2** is the matrix equivalent of −1 in regular arithmetic).

11. Close the MainWindow window.
12. Close Visual Studio:
    - On the **File** menu, click **Exit**.