Lab 4: Handling Exceptions

Exercise 1: Making a Method Fail-Safe

Task 1: Open the Failsafe solution and run the application

1. Log on to the 10266A-GEN-DEV virtual machine as **Student** with the password **Pa$$w0rd**.
2. Open Microsoft® Visual Studio® 2010:
   - Click **Start**, point to **All Programs**, click **Microsoft Visual Studio 2010**, and then click **Microsoft Visual Studio 2010**.
3. Open the Failsafe solution in the E:\Labfiles\Lab 4\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 4\Ex1\Starter** folder, click Failsafe.sln, and then click Open.
4. Run the Failsafe project and repeatedly click **Shutdown** until an exception occurs:

Note: The **Switch** class is designed to randomly throw an exception, so you may not encounter an exception the first time that you click the button. Repeatedly click the **Shutdown** button until an exception occurs.

   a. On the **Debug** menu, click **Start Debugging**.
   b. In the MainWindow window, click **Shutdown**, and then examine the unhandled exception message that appears in Visual Studio.
   c. In Visual Studio, on the **Debug** menu, click **Stop Debugging**.

Task 2: Examine the Switch class

1. If it is not already open, open the Switch.cs file in Visual Studio:

a. In Solution Explorer, in the Failsafe solution, expand the **SwitchDevice** project.

b. Right-click **Switch.cs**, and then click **View Code**.

2. Examine the **Switch** class.

   Note that the class contains several methods, each of which is capable of throwing at least one exception, dependent on the outcome of a random number generation. Toward the bottom of the file, note the definitions of each of the custom exceptions that the **Switch** class can throw. These are very basic exception classes that simply encapsulate an error message.

Task 3: Handle the exceptions that the Switch class throws

The SwitchTestHarness project contains a reference to the **SwitchDevice** class, and invokes each method in the **Switch** class to simulate polling multiple sensors and diagnostic devices. Currently, the project contains no exception handling, so when an exception occurs, the application will fail. You must add exception-handling code to the SwitchTestHarness project, to protect the application from exceptions that the **Switch** class throws.

1. Open the MainWindow.xaml.cs file in Visual Studio:

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

   b. Expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

2. In the **MainWindow** class, locate the **Button1_Click** method. This method runs when the user clicks the **Shutdown** button.

3. Remove the comment **TODO - Add exception handling**, and then locate the **Step 1 - disconnect from the Power Generator** and **Step 2 - Verify the status of the Primary Coolant System** comments. Enclose the code between these comments in a **try/catch** block that catches the **SwitchDevices.PowerGeneratorCommsException** exception. This is the exception that the **DisconnectPowerGenerator** method can throw.

   Your code should resemble the following code example.

```
...
// Step 1 - disconnect from the Power Generator
try
{

    if (sd.DisconnectPowerGenerator() ==
```

```
        SwitchDevices.SuccessFailureResult.Fail)
    {
        this.textBlock1.Text +=
            "\nStep 1: Failed to disconnect power generation system";
    }
    else
    {
        this.textBlock1.Text +=
        "\nStep 1: Successfully disconnected power generation system";
    }
}
catch (SwitchDevices.PowerGeneratorCommsException ex)
{

}

// Step 2 - Verify the status of the Primary Coolant System
...
```

4. In the **catch** block, add code to append a new line of text to the **textBlock1** control with the message "*** Exception in step 1:" and then the contents of the **Message** property of the exception. The **Message** property contains the error message that the **Switch** object specified when it threw the exception.

Hint: To append a line of text to a **TextBlock** control, use the **+=** operator on the **Text** property of the control.

Your code should resemble the following code example.

```
...
catch (SwitchDevices.PowerGeneratorCommsException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 1: " + ex.Message;
}
...
```

5. Enclose the code between the **Step 2 - Verify the status of the Primary Coolant System** and **Step 3 - Verify the status of the Backup Coolant System** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, following the same pattern as step 3, print a message on a

new line in the **textBlock1** control (note that this is step 2, not step 1 of the shutdown process).

Your code should resemble the following code example.

```
...
// Step 2 - Verify the status of the Primary Coolant System
try
{
    switch (sd.VerifyPrimaryCoolantSystem())
    {
        case SwitchDevices.CoolantSystemStatus.OK:
            this.textBlock1.Text +=
                "\nStep 2: Primary coolant system OK";
            break;
        case SwitchDevices.CoolantSystemStatus.Check:
            this.textBlock1.Text +=
                "\nStep 2: Primary coolant system requires manual
check";
            break;
        case SwitchDevices.CoolantSystemStatus.Fail:
            this.textBlock1.Text +=
                "\nStep 2: Problem reported with primary coolant
system";
            break;
    }
}
catch (SwitchDevices.CoolantPressureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 2: " + ex.Message;
}
catch (SwitchDevices.CoolantTemperatureReadException ex)
{
    this.textBlock1.Text
        += "\n*** Exception in step 2: " + ex.Message;
}

// Step 3 - Verify the status of the Backup Coolant System
...
```

6. Enclose the code between the **Step 3 - Verify the status of the Backup Coolant System** and **Step 4 - Record the core temperature prior to shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoolantPressureReadException** and **SwitchDevices.CoolantTemperatureReadException** exceptions. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 3).

Your code should resemble the following code example.

```
...
// Step 3 - Verify the status of the Backup Coolant System
try
{
    switch (sd.VerifyBackupCoolantSystem())
    {
        case SwitchDevices.CoolantSystemStatus.OK:
            this.textBlock1.Text +=
                "\nStep 3: Backup coolant system OK";
            break;
        case SwitchDevices.CoolantSystemStatus.Check:
            this.textBlock1.Text +=
                "\nStep 3: Backup coolant system requires manual
check";
            break;
        case SwitchDevices.CoolantSystemStatus.Fail:
            this.textBlock1.Text +=
                "\nStep 3: Backup reported with primary coolant
system";
            break;
    }
}
catch (SwitchDevices.CoolantPressureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 3: " + ex.Message;
}
catch (SwitchDevices.CoolantTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 3: " + ex.Message;
}

// Step 4 - Record the core temperature prior to shutting down the
reactor

...
```

7. Enclose the code between the **Step 4 - Record the core temperature prior to shutting down the reactor** and **Step 5 - Insert the control rods into the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 4).

   Your code should resemble the following code example.

```
...
// Step 4 - Record the core temperature prior to shutting down the
reactor

try
{

    this.textBlock1.Text +=
        "\nStep 4: Core temperature before shutdown: " +
        sd.GetCoreTemperature();
}

catch (SwitchDevices.CoreTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 4: " + ex.Message;
}

// Step 5 - Insert the control rods into the reactor
...
```

8. Enclose the code between the **Step 5 - Insert the control rods into the reactor** and **Step 6 - Record the core temperature after shutting down the reactor** comments in a **try**/**catch** block, which catches the **SwitchDevices.RodClusterReleaseException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 5).

   Your code should resemble the following code example.

```
...
// Step 5 - Insert the control rods into the reactor

try
{
    if (sd.InsertRodCluster() ==
        SwitchDevices.SuccessFailureResult.Success)
    {
        this.textBlock1.Text +=
            "\nStep 5: Control rods successfully inserted";
    }
    else
    {
        this.textBlock1.Text +=
            "\nStep 5: Control rod insertion failed";
    }
}
```

```
catch (SwitchDevices.RodClusterReleaseException ex)
{
    this.textBlock1.Text
        += "\n*** Exception in step 5: " + ex.Message;
}
// Step 6 - Record the core temperature after shutting down the
reactor
...
```

9. Enclose the code between the **Step 6 - Record the core temperature after shutting down the reactor** and **Step 7 - Record the core radiation levels after shutting down the reactor** comments in a **try/catch** block, which catches the **SwitchDevices.CoreTemperatureReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 6).

   Your code should resemble the following code example.

```
...
// Step 6 - Record the core temperature after shutting down the
reactor

try
{
    this.textBlock1.Text +=
        "\nStep 6: Core temperature after shutdown: " +
        sd.GetCoreTemperature();
}
catch (SwitchDevices.CoreTemperatureReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 6: " + ex.Message;
}

// Step 7 - Record the core radiation levels after shutting down the
reactor
...
```

10. Enclose the code between the **Step 7 - Record the core radiation levels after shutting down the reactor** and **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.CoreRadiationLevelReadException** exception. In the exception handler, print a message on a new line in the **textBlock1** control (this is step 7).

    Your code should resemble the following code example.

```
...
// Step 7 - Record the core radiation levels after shutting down the
reactor
try
{
    this.textBlock1.Text +=
        "\nStep 7: Core radiation level after shutdown: " +
        sd.GetRadiationLevel();
}
catch (SwitchDevices.CoreRadiationLevelReadException ex)
{
    this.textBlock1.Text +=
        "\n*** Exception in step 7: " + ex.Message;
}

// Step 8 - Broadcast "Shutdown Complete" message
...
```

11. Enclose the two statements after **Step 8 - Broadcast "Shutdown Complete" message** comments in a **try/catch** block, which catches the **SwitchDevices.SignallingException** exception. In each exception handler, print a message on a new line in the **textBlock1** control (this is step 8).

    Your code should resemble the following code example.

```
...
// Step 8 - Broadcast "Shutdown Complete" message
try
{
    sd.SignalShutdownComplete();
    this.textBlock1.Text +=
        "\nStep 8: Broadcasting shutdown complete message";
}
catch (SwitchDevices.SignallingException ex)
{
    this.textBlock1.Text +=
    "\n*** Exception in step 8: " + ex.Message;
}

this.textBlock1.Text +=
    "\nTexst sequence complete: " +
    DateTime.Now.ToLongTimeString();
...
```

12. Build the solution and correct any errors:

    - On the **Build** menu, click **Build Solution**.

Task 4: Test the application

- Run the application, and then click the **Shutdown** button. Examine the messages displayed in the MainWindow window, and verify that exceptions are now caught and reported:

Note: The **Switch** class randomly generates exceptions as before, so you may not see any exception messages the first time that you click the button. Repeat the process of clicking the button and examining the output until you see exception messages appear.

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

b. In the MainWindow window, click **Shutdown**.

c. Read through the messages that are displayed in the window, and verify that, where an exception occurred, a message appears that states "*** Exception in step *x* :*message*", where *x* is a step number, and *message* is an exception message.

d. Close the application and return to Visual Studio.

Exercise 2: Detecting an Exceptional Condition

Task 1: Open the MatrixMultiplication solution

1. In Visual Studio, open the MatrixMultiplication solution in the

   E:\Labfiles\Lab 4\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 4\Ex2\Starter folder, click **MatrixMultiplication.sln**, and then click **Open**.

2. Open the Matrix.cs file, and then locate the **MatrixMultiply** method:

   a. In Solution Explorer, in the MatrixMultiplication project, right-click **Matrix.cs**, and then click **View Code**.

   b. Examine the **MatrixMultiply** method.

      The **MatrixMultiply** method performs the arithmetic to multiply together the two matrices passed as parameters and return the result.

Currently, the method accepts matrices of any size, and performs no validation of data in the matrices before calculating the results. You will add checks to ensure that the two matrices are compatible (the number of columns in the first matrix is equal to the number of rows in the second matrix), and that no value in either matrix is a negative number.

If the matrices are not compatible, or either of them contain a negative value, the method must throw an exception.

Task 2: Add code to throw exceptions in the MatrixMultiply method

1. In the **MatrixMultiply** method, locate and remove the comment **TODO – Evaluate input matrices for compatibility**. Below the comment block, add code to perform the following actions:

   a. Compare the number of columns in **matrix1** to the number of rows in **matrix2**.

   b. Throw an **ArgumentException** exception if the values are not equal. The exception message should specify that the number of columns and rows should match.

Hint: You can obtain the number of columns in a matrix by examining the length of the first dimension. You can obtain the number of rows in a matrix by examining the length of the second dimension.

Your code should resemble the following code example.

```
...
// Check the matrices are compatible
if (matrix1.GetLength(0) != matrix2.GetLength(1))
    throw new ArgumentException(
        "The number of columns in the first matrix must be the same as
the number of rows in the second matrix");

// Get the dimensions
...
```

2. Locate and remove the comment **TODO – Evaluate matrix data points for invalid data**. At this point, the method iterates through the data points in each matrix, multiplying the value in each cell in **matrix1** against the value in the corresponding cell in **matrix2**. Add code below the comment block to perform the following actions:

a. Check that the value in the current column and row of **matrix1** is greater than zero. The cell and row variables contain the column and row that you should examine.

b. Throw an **ArgumentException** exception if the value is not greater than zero. The exception should contain the message "Matrix1 contains an invalid entry in cell[*x*, *y*]." where *x* and *y* are the column and row values of the cell.

Hint: Use the **String.Format** method to construct the exception message.

Your code should resemble the following code example.

```
...
// Throw exceptions if either matrix contains a negative entry
if (matrix1[cell, row] < 0d)
    throw new ArgumentException(String.Format(
        "Matrix1 contains an invalid entry in cell[{0}, {1}]",
        cell, row));

accumulator += matrix1[cell, row] * matrix2[column, cell];
...
```

3. Add another block of code to check that the value in the current column and row of **matrix2** is greater than zero. If it is not, throw an **ArgumentException** exception with the message "Matrix2 contains an invalid entry in cell[*x*, *y*].". The column and cell variables contain the column and row that you should examine.

   Your code should resemble the following code example.

```
...
    "Matrix1 contains an invalid entry"
        );

if (matrix2[column, cell] < 0d)
    throw new ArgumentException(String.Format(
        "Matrix2 contains an invalid entry in cell[{0}, {1}].",
        column, cell));

accumulator += matrix1[cell, row] * matrix2[column, cell];
...
```

Task 3: Handle the exceptions that the MatrixMultiply method throws

1. Open the MainWindow Windows® Presentation Foundation (WPF) window in the Design View window and examine the window.

   This window provides the user interface that enables the user to enter the data for the two matrices to be multiplied. The user clicks the **Calculate** button to calculate and display the result:

   - In Solution Explorer, in the MatrixMultiplication project, double-click **MainWindow.xaml**.

2. Open the code file for the MainWindow WPF window:

   - In Solution Explorer, in the MatrixMultiplication project, expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

3. In the **MainWindow** class, locate the **ButtonCalculate_Click** method. This method runs when the user clicks the **Calculate** button.

4. In the **ButtonCalculate_Click** method, locate the line of code that invokes the **Matrix.MatrixMultiply** method, and enclose this line of code in a **try/catch** block that catches an **ArgumentException** exception named **ex**.

   Your code should resemble the following code example.

```
...
// Do the multiplication - checking for exceptions
try
{
    result = Matrix.MatrixMultiply(matrix1, matrix2);
}
catch (ArgumentException ex)
{

}

// Show the results
...
```

5. In the **catch** block, add a statement that displays a message box that contains the contents of the **Message** property of the exception object.

Hint: You can use the **MessageBox.Show** method to display a message box. Specify the message to display as a string passed in as a parameter to this method.

Your code should resemble the following code example.

```
...
catch (ArgumentException ex)
{
    MessageBox.Show(ex.Message);
}

...
```

6. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
7. Start the application without debugging:
   - On the **Debug** menu, click **Start Without Debugging**.
8. In the MainWindow window, in the first drop-down list box, select **Matrix 1: 2 Columns**, in the second drop-down list box, select **Matrix 1: 2 Rows**, and then in the third drop-down list box, select **Matrix 2: 2 Columns**.

   This creates a pair of 2 × 2 matrices initialized with zeroes.
9. Enter some non-negative values in the cells in both matrices, and then click **Calculate**.

   Verify that the result is calculated and displayed, and that no exceptions occur.
10. Enter one or more negative values in the cells in either matrix, and then click **Calculate** again.

    Verify that the appropriate exception message is displayed, and that it identifies the matrix and cell that is in error.
11. Close the MainWindow window and return to Visual Studio.

The application throws and catches exceptions, so you need to test that the application functions as expected. Although you can test for negative data points by using the application interface, the user interface does not let you create arrays of different dimensions. Therefore, you have been provided with unit test cases that will invoke the **MatrixMultiply** method with data that will cause exceptions. These tests have already been created; you will just run them to verify that your code works as expected.

Task 4: Implement test cases and test the application

1. In the Matrix Unit Test Project, open the **MatrixTest** class, and then examine the **MatrixMultiplyTest1** method.

   The **MatrixMultiplyTest1** method creates four matrices: **matrix1**, **matrix2**, **expected**, and **actual**. The **matrix1** and **matrix2** matrices are the input matrices that are passed to the **MatrixMultiply** method during the test. The **expected** matrix contains the expected result of the matrix multiplication, and the **actual** matrix stores the result of the **MatrixMultiply** method call. The method invokes the **MatrixMultiply** method before using a series of **Assert** statements to verify that the **expected** and **actual** matrices are identical.

   This test method is complete and requires no further work:

   - In Solution Explorer, expand the **Solution Items** folder, expand **Matrix Unit Test Project**, expand **Test References**, and then double-click **MatrixText.cs**.

2. Examine the **MatrixMultiplyTest2** method.

   This method creates two compatible matrices, but **matrix2** contains a negative value. This should cause the **MatrixMultiply** method to throw an exception.

   The **MatrixMultiplyTest2** method is prefixed with the **ExpectedException** attribute, indicating that the test method expects to cause an **ArgumentException** exception. If the test does not cause this exception, it will fail.

3. Examine the **MatrixMultiplyTest3** method.

   This method creates two incompatible matrices and passes them to the **MatrixMultiply** method, which should throw an **ArgumentException** exception as a result. Again, the method is prefixed with the **ExpectedException** attribute, indicating that the test will fail if this exception is not thrown.

4. Run all tests in the solution, and verify that all tests execute correctly:

   a. On the **Build** menu, click **Build Solution**.

   b. On the **Test** menu, point to **Run**, and then click **All Tests in Solution**.

   c. Wait for the tests to run, and then in the Test Results window, verify that all tests passed.

Exercise 3: Checking for Numeric Overflow

Task 1: Open the IntegerOverflow solution

1. Open the IntegerOverflow solution in the E:\Labfiles\Lab 4\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 4\Ex3\Starter folder, click **IntegerOverflow.sln**, and then click **Open**.

2. Run the application, and then click **Multiply**. Observe the result that is displayed and note that it is incorrect.

    The application multiplies 2147483647 by 2, and displays the result **−2**. This is because the multiplication causes an integer numeric overflow. By default, overflow errors of this nature do not cause an exception. However, in many situations, it is better to catch the overflow error than to let an application proceed with incorrect data:

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

    b. In the MainWindow window, click **Multiply**.

3. In Visual Studio, on the **Debug** menu, click **Stop Debugging**.

Task 2: Add a checked block

1. In Solution Explorer, open the MainWindow.xaml.cs file:

    - In Solution Explorer, in the IntegerMultiplySolution solution, expand **MainWindow.xaml**, right-click **MainWindow.xaml.cs**, and then click **View Code**.

2. Locate the **DoMultiply_Click** method.

    This method runs when the user clicks the **Multiply** button.

3. Remove the **TODO - Place the multiplication in a checked block** comment. Add a **try/catch** block around the line of code that performs the multiplication operation, and then catch the **OverflowException** exception.

    Your code should resemble the following code example.

```
...
    return;
}

try
{
    labelAnswer.Content = (x * y).ToString();
}
catch (OverflowException ex)
{
    MessageBox.Show(ex.Message);
}
...
```

4. Inside the **try** block, add a **checked** block around the line of code that performs the multiplication arithmetic.

   Your code should resemble the following code example.

```
...
try
{
    checked
    {
        labelAnswer.Content = (x * y).ToString();
    }
}
catch (Exception ex)
...
```

5. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.

Task 3: Test the application

1. Start the application:
   - On the **Debug** menu, click **Start Debugging**.
2. Click **Multiply**. Verify that the application now displays a message informing you that the arithmetic operation resulted in an overflow.
3. Click **OK**, close the MainWindow window, and then return to Visual Studio.
4. Close Visual Studio.