Lab 5: Reading and Writing Files

Exercise 1: Building a Simple File Editor

Task 1: Open the SimpleEditor project

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 SimpleEditor solution in the E:\Labfiles\Lab 5\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 5\Ex1 \Starter** folder, click **SimpleEditor.sln**, and then click **Open**.

Task 2: Display a dialog box to accept a file name from the user

1. Display the MainWindow.xaml window:
   - In Solution Explorer, expand the **FileEditor** project, and then double-click **MainWindow.xaml**.

   The MainWindow window implements a very simple text editor. The main part of the window contains a text box that a user can use to display and edit text. The **Open** button enables the user to open a file, and the **Save** button enables the user to save the changes to the text back to a file. You will add the code that implements the logic for these two buttons.
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. Locate the task **TODO - Implement a method to get the file name.** Double-click this task.

   This task is located in the MainWindow.xaml.cs class file.

4. Delete the comment, and then define a new private method named **GetFileName** that accepts no parameters and returns a string value that holds the file name that the user specified.

   Your code should resemble the following code example.

```
...
private string GetFileName()
{

}
...
```

5. In the method body, declare a new string member named **fname**, and then initialize it with the **String.Empty** value.

   Your code should resemble the following code example.

```
...
private string GetFileName()
{
    string fname = String.Empty;
}
...
```

6. At the end of the collection of **using** statements at the top of the file, add a statement to bring the **Microsoft.Win32** namespace into scope.

   Your code should resemble the following code example.

```
...
using System.Windows.Shapes;
using Microsoft.Win32;
...
```

7. In the **GetFileName** method, after the statement that declares the fname variable, add code to the method to perform the following actions:

   a. Create a new instance of the **OpenFileDialog** dialog box, named **openFileDlg**.

   b. Set the **InitialDirectory** property of **openFileDlg** to point to the E:\Labfiles\Lab 5\Ex1\Starter folder.

Note: When including file paths in code, you should prefix the string with the @ symbol. This symbol instructs the C# compiler to treat any '\' characters as literals rather than escape characters.

c. Set the **DefaultExt** property of **openFileDlg** to "**.txt**";.

d. Set the **Filter** property of **openFileDlg** to "**Text Documents (.txt)|*.txt**".

Your code should resemble the following code example.

```
...
    string fname = string.Empty;

    OpenFileDialog openFileDlg = new OpenFileDialog();

    openFileDlg.InitialDirectory =
        @"E:\Labfiles\Lab 5\Ex1\Starter";

    openFileDlg.DefaultExt = ".txt";
    openFileDlg.Filter = "Text Documents (.txt)|*.txt";
}
...
```

8. Add code to perform the following tasks:

   a. Call the **ShowDialog** method of **openFileDlg**, and then save the result.

Note: The value that **ShowDialog** returns is a nullable Boolean value, so save the result in a nullable Boolean variable.

   b. If the result is **true**, assign the value of the **FileName** property of **openFileDlg** to the fname variable.

Your code should resemble the following code example.

```
...
    bool? result = openFileDlg.ShowDialog();

    if (result == true)

    {
        fname = openFileDlg.FileName;
    }
}
...
```

9. At the end of the method, return the value in the fname variable.

   Your code should resemble the following code example.

```
...
        fname = openFileDlg.FileName;
    }
    return fname;
}
...
```

Task 3: Implement a new class to read and write text to a file

1. Add a new class named **TextFileOperations** to the FileEditor project.

   You will use this class to wrap some common file operations. This scheme enables you to change the way in which files are read from or written to without affecting the rest of the application:

   a. In Solution Explorer, right-click the **FileEditor** project, point to **Add**, and then click **Class**.

   b. In the **Add New Item - FileEditor** dialog box, in the **Name** box, type **TextFileOperations.cs** and then click **Add**.

2. At the top of the class file, add a statement to bring the **System.IO** namespace into scope.

   Your code should resemble the following code example.

```
...

using System.Text;
using System.IO;

namespace FileEditor

{
...
}
```

3. In the **TextFileOperations** class, add a public static method named **ReadTextFileContents**. The method should accept a string parameter named *fileName*, and return a string object.

   Your code should resemble the following code example.

```
...
class TextFileOperations
{
    public static string ReadTextFileContents(string fileName)
    {

    }
}
...
```

4. In the **ReadTextFileContents** method, add code to return the entire contents of the text file whose path is specified in the *fileName* parameter.

Hint: Use the static **ReadAllText** method of the **File** class.

Your code should resemble the following code example.

```
...
public static string ReadTextFileContents(string fileName)
{
    return File.ReadAllText(fileName);
}
...
```

5. Below the **ReadTextFileContents** method, add a public static method named **WriteTextFileContents**. The method should not return a value type, and should accept the following parameters:

   a. A string parameter named *fileName*.

   b. A string parameter named *text*.

   Your code should resemble the following code example.

```
...
    return File.ReadAllText(fileName);
}

    public static void WriteTextFileContents
        (string fileName, string text)
    {
    }
}
...
```

6. In the **WriteTextFileContents** method, add code to write the text that is contained in the *text* parameter to the file that is specified in the *fileName* parameter.

Hint: Use the static **WriteAllText** method of the **File** class.

Your code should resemble the following code example.

```
...

public static void WriteTextFileContents
    (string filename, string text)

{
    File.WriteAllText(fileName, text);
}

...
```

7. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
   - Review the **Error** list and check for any errors.

Task 4: Update the MainWindow event handlers to consume the TextFileOperations class

1. In the task list, locate the task **TODO - Update the OpenButton_Click method**. Double-click this task.

   This task is located in the **OpenButton_Click** method of the **MainWindow** class.

2. Remove the comment, and then add code to perform the following tasks:

   a. Invoke the **GetFileName** method. Store the result of the method in the **fileName** member.

   b. If **fileName** is not an empty string, call the static **ReadTextFileContents** method of the **TextFileOperations** class, and then pass *fileName* as the parameter. Store the result in the **Text** property of the editor **TextBox** control in the Windows® Presentation Foundation (WPF) window.

   Your code should resemble the following code example.

```
...
private void OpenButton_Click(object sender, RoutedEventArgs e)
{
    // Call GetFileName to get the name of the file to load
    fileName = GetFileName();

    // Populate the editor text box with the file contents
    if (fileName != String.Empty)
    {
        editor.Text =
            TextFileOperations.ReadTextFileContents(fileName);
    }
}
...
```

3. In the task list, locate the task **TODO - Update the SaveButton_Click method**. Double-click this task.

   This task is located in the **SaveButton_Click** method of the **MainWindow** class.

4. In the **SaveButton_Click** method, remove the comment, and then add code to perform the following tasks:

   a. Check that the **fileName** member is not an empty string.

   b. If **fileName** is not an empty string, call the static **WriteTextFileContents** method of the **TextFileOperations** class. Pass **fileName** and the **Text** property of the editor **TextBox** control as the parameters.

   Your code should resemble the following code example.

```
...
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
    // Write the contents of the editor TextBox back to the file
    if (fileName != String.Empty)
    {
        TextFileOperations.WriteTextFileContents(fileName,
                                                  editor.Text);
    }
}
...
```

5. Build the solution and correct any errors:

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

6. Start the application without debugging:

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

7. In the MainWindow window, click **Open**.
8. In the **Open** dialog box, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder, click **Commands.txt**, and then click **Open**.
9. In the MainWindow window, verify that the text in the following code example is displayed in the editor **TextBox** control.

```
Move x, 10
Move y, 20
If x < y Add x, y
If x > y & x < 20 Sub x, y
Store 30
```

   This is the text from the Commands.txt file.

10. Change the **Store 30** line to **Save 50**, and then click **Save**.
11. Close the MainWindow window.
12. Using Windows Explorer, move to the **E:\Labfiles\Lab 5\Ex1\Starter** folder.
13. Open the Commands.txt file by using Notepad:
    - In Windows Explorer, right-click **Commands.txt**, point to **Open with**, and then click **Notepad**.
14. In Notepad, verify that the last line of the file contains the text **Save 50**.
15. Close Notepad and return to Visual Studio.

Task 5: Implement test cases

1. In the task list, locate the task **TODO - Complete Unit Tests**. Double-click this task.

   This task is located in the **TextFileOperationsTest** class.

2. Remove the comment.
3. Examine the **ReadTextFileContentsTest1** method, and then uncomment the commented line.

   This method creates three strings:

   a. The **fileName** string contains the path of a prewritten file that contains specific content.

b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.

c. The **actual** string is initialized by calling the **ReadTextFileContents** method that you just implemented.

The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same:

- Uncomment the fourth line of code, to enable the method to call the **FileEditor.TextFileOperations.ReadTextFileContents** method.

4. Examine the **WriteTextFileContentsTest1** method, and then uncomment the commented line.

   This method creates two strings:

   a. The **fileName** string contains the path of a nonexistent file, which the method will create when run.

   b. The **text** string contains some text that the method will write to the file.

   The method calls the **WriteTextFileContents** method, passing the **fileName** and **text** strings as parameters. This creates the file at the specified location, and writes to the file. The method then creates a further string, **expected**, by calling the **File**.**ReadAllText** method and reading the text from the written file. The method then checks that the text string and the expected string are the same, before deleting the file that was created during the test:

   - Uncomment the third line of code, to enable the method to call the **FileEditor.TextFileOperations.WriteTextFileContents** method.

5. 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 2: Making the Editor XML Aware

Task 1: Open the starter project

- Open the SimpleEditor solution in the E:\Labfiles\Lab 5\Ex2\Starter folder.

This project is a completed version of the SimpleEditor project from Exercise 1:

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 5\Ex2\Starter** folder, click **SimpleEditor.sln**, and then click **Open**.

Task 2: Add a new method to filter XML characters to the TextFileOperations class

1. Review the task list:

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

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

2. In the task list, locate the **TODO - Implement a new method in the TextFileOperations class** task. Double-click this task.

   This task is located in the **TextFileOperations** class.

3. Remove the comment, and then add a new public static method named **ReadAndFilterTextFileContents**. The method should accept a string parameter named *fileName*, and return a string.

   Your code should resemble the following code example.

```
...

public static string ReadAndFilterTextFileContents(string fileName)
{

}

...
```

4. In the **ReadAndFilterTextFileContents** method, add the following local variables:

   a. A **StringBuilder** object named **fileContents**, initialized to a new instance of the **StringBuilder** class.

   b. An integer variable called charCode.

   Your code should resemble the following code example.

```
...
public static string ReadAndFilterTextFileContents(string fileName)
{
    StringBuilder fileContents = new StringBuilder();
    int charCode;
}
...
```

5. Add a statement that instantiates a **StreamReader** object, named **fileReader**, by using the *fileName* parameter.

   Your code should resemble the following code example.

```
...
{
    StringBuilder fileContents = new StringBuilder();
    int charCode;

    StreamReader fileReader = new StreamReader(fileName);
}
...
```

6. Add a **while** statement that reads each character in the **StreamReader** object until the end of the file is reached.

Hint: Use the **Read** method of the **StreamReader** class to read the next character from a stream. This method returns –1 if there is no more data.

   Your code should resemble the following code example.

```
...
    StreamReader fileReader = new StreamReader(fileName);
    while ((charCode = fileReader.Read()) != -1)
    {

    }
...
```

7. In the **while** block, add a **switch** statement that evaluates the charCode variable.

   In the **switch** statement, add **case** statements for each of the characters in the following table. In each statement, append the **fileContent StringBuilder** object with the alternative representation shown in the table.

charCode	Standard representation	Alternative representation
34	" (straight quotation mark)	&quot;
38	& (ampersand)	&amp;
39	' (apostrophe)	&apos;
60	< (less than)	&lt;
62	> (greater than)	&gt;

Your code should resemble the following code example.

```
...
while ((charCode = fileReader.Read()) != -1)
{
    switch (charCode)
    {
        case 34: // "
            fileContents.Append("&quot;");
            break;

        case 38: // &
            fileContents.Append("&amp;");
            break;

        case 39: // '
            fileContents.Append("&apos;");
            break;

        case 60: // <
            fileContents.Append("&lt;");
            break;

        case 62: // >
            fileContents.Append("&gt;");
            break;
    }
}
...
```

8. Add a default **case** statement that appends the actual character read from the stream to the **fileContent StringBuilder** object.

Note: The **Read** method returns the value read from the file as an integer and stores it in the charCode variable. You must cast this variable to a character before you append it to the end of the **StringBuilder** object.

Your code should resemble the following code example.

```
...
            case 62: // >
                fileContents.Append("&gt;");
                break;

            default:
                fileContents.Append((char)charCode);
                break;
        }

}
...
```

9. At the end of the method, return the contents of the **fileContent StringBuilder** object as a string.

   Your code should resemble the following code example.

```
...

public static string ReadAndFilterTextFileContents(string fileName)
{
    ...
    return fileContents.ToString();
}
...
```

10. Build the solution and correct any errors:

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

Task 3: Update the user interface to invoke the new method

1. In the task list, locate the **TODO - Update the UI to use the new method** task. Double-click this task.

   This task is located in the **OpenButton_Click** method of the **MainWindow.xaml.cs** class.

2. Delete the comment, and then modify the line of code that calls the **TextFileOperations.ReadTextFileContents** method to call the **TextFileOperations.ReadAndFilterTextFileContents** method instead. Pass the **fileName** field as the parameter, and then save the result in the **Text** property of the editor **TextBox** control.

   Your code should resemble the following code example.

```
...
if (filename != string.Empty)
{
    // Call the new read file contents method
    editor.Text =
        TextFileOperations.ReadAndFilterTextFileContents(filename);
}
...
```

3. Build the solution and correct any errors:
   - On the **Build** menu, click **Build Solution**.
4. Start the application without debugging:
   - On the **Debug** menu, click **Start Without Debugging**.
5. In the MainWindow window, click **Open**.
6. In the **Open** dialog box, move to the **E:\Labfiles\Lab 5\Ex2\Starter** folder, click **Commands.txt**, and then click **Open**.
7. In the MainWindow window, verify that the text in the following code example is displayed in the editor **TextBox** control.

```
Move x, 10
Move y, 20
If x &lt; y Add x, y
If x &gt; y &amp; x &lt; 20 Sub x, y
Store 30
```

   This is the text from the Commands.txt file. Notice that the **<**, **>**, and **&** characters have been replaced with the text **&lt;**, **&gt;**, and **&amp;**.
8. Close the MainWindow window and return to Visual Studio.

Task 4: Implement test cases

1. In the task list, locate the **TODO - Complete Unit Tests** task. Double-click this task.

   This task is located in the **TextFileOperationsTest** class.

2. Examine the **ReadAndFilterTextFileContentsTest** method, and then uncomment the commented line.

   This method creates three strings:

   a. The **filename** string contains the path of a prewritten file that contains specific content.

   b. The **expected** string contains the contents of the prewritten file, including formatting and escape characters.

   c. The **actual** string is initialized by calling the **ReadAndFilterTextFileContents** method that you just implemented.

   The test method then uses an **Assert** statement to verify that the **expected** and **actual** strings are the same.

   This method is complete, and requires no further work:

   - Uncomment the fourth line of code, to enable the method to call the **FileEditor.TextFileOperations.ReadAndFilterTextFileContents** method.

3. 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.