Module 1: Review of Visual C# Syntax

Lab: Developing the Class Enrollment Application

Exercise 1: Implementing Edit Functionality for the Students List

▶ Task 1: Detect whether the user has pressed the Enter key

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 Start window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod01\Labfiles\Databases** folder, and then double-click **SetupSchoolDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod01\Labfiles\Starter\Exercise 1**, click **School.sln**, and then click **Open**.
12. In Solution Explorer, expand **School**, and then expand **MainWindow.xaml**.
13. Double-click **MainWindow.xaml.cs**.
14. In Visual Studio, on the **View** menu, click **Task List**.
15. In the **Task List** window, in the **Categories** list, click **Comments**.
16. Double-click the **TODO: Exercise 1: Task 1a: If the user pressed Enter, edit the details for the currently selected student** task.
17. In the code editor, click at the beginning of the comment line, press Enter, and in the blank space above the comment, type the following code:

```
switch (e.Key)
{
```

18. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
    case Key.Enter: Student student = this.studentsList.SelectedItem as Student;
```

19. After all the comments in this method, type the following code:

```
        break;
}
```

▶ Task 2: Initialize the StudentForm window and populate it with the details of the currently selected student

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2a: Use the StudentsForm to display and edit the details of the student** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
StudentForm sf = new StudentForm();
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Set the title of the form and populate the fields on the form with the details of the student** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
sf.Title = "Edit Student Details";
sf.firstName.Text = student.FirstName;
sf.lastName.Text = student.LastName;
sf.dateOfBirth.Text = student.DateOfBirth.ToString("d");
```

▶ Task 3: Display the StudentForm window and copy the updated student details entered back to the Student object

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Display the form** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (sf.ShowDialog().Value)
{
```

3. After all the comments in this method, add the following code:

```
}
```

4. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: When the user closes the form, copy the details back to the student** task.
5. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
    student.FirstName = sf.firstName.Text;
    student.LastName = sf.lastName.Text;
    student.DateOfBirth =
    DateTime.Parse(sf.dateOfBirth.Text);
```

6. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3c: Enable saving (changes are not made permanent until they are written back to the database)** task.
7. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
    saveChanges.IsEnabled = true;
```

▶ Task 4: Run the application and verify that the edit functionality works as expected

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.

3. Verify that the application starts and displays the initial list of students.

The initial students list should look like this:

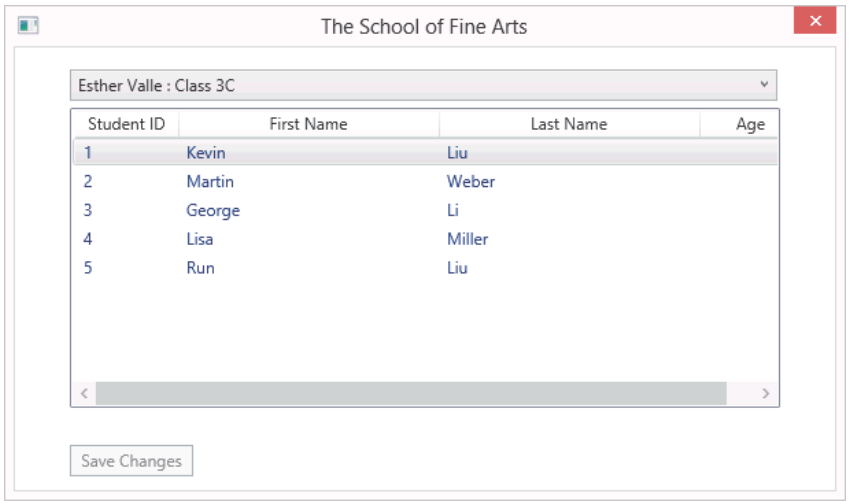

FIGURE 01.1:THE INITIAL STUDENTS LIST

4. Click the row containing the name **Kevin Liu**.
5. Press Enter and verify that the **Edit Student Details** window appears and displays the correct details:

The **Edit Student Details** window should look similar to the following:

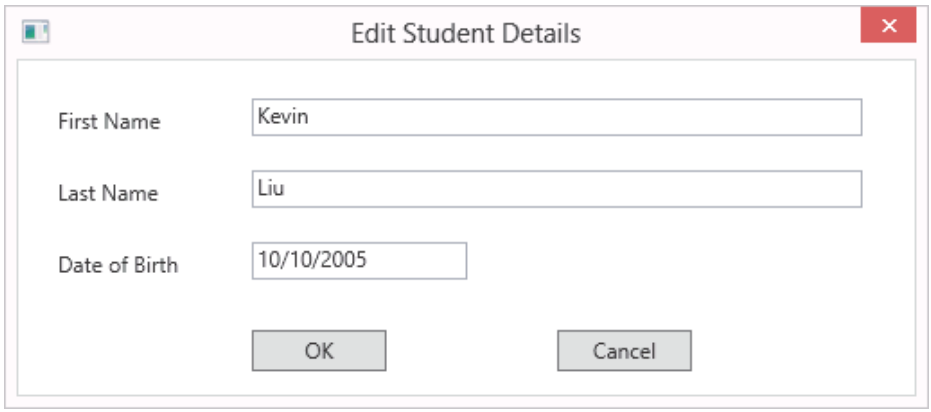

FIGURE 01.2:EDIT STUDENT DETAILS FORM

6. In the **Last Name** text box, delete the existing contents, type **Cook**, and then click **OK**.
7. Verify that Liu has changed to Cook in the students list, and that the **Save Changes** button is now enabled.
8. Close the application.

▶ Task 5: Use the Visual Studio Debugger to step through the code.

1. In Visual Studio, in the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Set the title of the form and populate the fields on the form with the details of the student** task.
2. In the following line of code, right-click the word **Title** in **sf.Title = "Edit Student Details";**, point to **Breakpoint**, and then click **Insert Breakpoint**.
3. On the **Debug** menu, click **Start Debugging**.
4. Click the row containing the name **George Li**, and then press Enter.

5. When Visual Studio enters break mode, in the bottom left window, click the **Watch 1** tab.
6. In the **Watch 1** window, click below **Name** to create a blank row.
7. In the **Name** column, type **sf.Title**, and then press Enter.
8. In the **Watch 1** window, click below **sf.Title** to create a blank row.
9. Type **sf.firstName.Text**,and then press Enter.
10. In the **Watch 1** window, click below **sf.firstName.Text** to create a blank row.
11. Type **sf.lastName.Text**, and then press Enter.
12. In the **Watch 1** window, click below **sf.lastName.Text** to create a blank row.
13. Type **sf.dateOfBirth.Text**, and then press Enter.
14. On the **Debug** menu, click **Step Over**.
15. Repeat step 14 three times.
16. In the bottom middle window, click the **Immediate Window** tab.
17. In the **Immediate Window**, type **sf.firstName.Text**, and then press Enter.
18. Verify that **"George"** is displayed.
19. In the **Watch 1** window, in the **sf.firstName.Text** row, right-click the **Value** field, and then click **Edit Value**.
20. Type **"Dominik"** and press Enter.
21. In the **Immediate Window**, type **sf.lastName.Text**, and then press Enter.
22. Verify that **"Li"** is displayed
23. Type **sf.lastName.Text = "Dubicki";**, and then press Enter.
24. In the **Watch 1** window, in the **sf.lastName.Text** row, verify that the **Value** column has changed to **"Dubicki"**.
25. On the **Debug** menu, click **Continue**.
26. Verify that the Edit Student Details form contains the information in the following table:

Field	Value
First Name	Dominik
Last Name	Dubicki
Date of Birth	8/10/2005

27. Close the application.
28. In Visual Studio, on the **Debug** menu, click **Delete All Breakpoints**.
29. In the **Microsoft Visual Studio** dialog box, click **Yes**.
30. On the **File** menu, click **Close Solution**.
31. In the **Microsoft Visual Studio** dialog box, click **Yes**.

Results: After completing this exercise, users will be able to edit the details of a student.

Exercise 2: Implementing Insert Functionality for the Students List

▶ Task 1: Add logic to the key down method to detect if the Insert key has been pressed.

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod01\Labfiles\Starter\Exercise 2**, click **School.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: If the user pressed Insert, add a new student** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
case Key.Insert:
```

▶ Task 2: Initialize the student form

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Use the StudentsForm to get the details of the student from the user** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
sf = new StudentForm();
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Set the title of the form to indicate which class the student will be added to (the class for the currently selected teacher)** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
sf.Title = "New Student for Class " + teacher.Class;
```

▶ Task 3: Display the StudentForm window and enable the user to provide the details of the new student

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Display the form and get the details of the new student** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (sf.ShowDialog().Value)
{
```

3. After all the comments in this method, add the following code:

```
}
break;
```

4. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3b: When the user closes the form, retrieve the details of the student from the form and use them to create a new Student object** task.
5. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
Student newStudent = new Student();
newStudent.FirstName = sf.firstName.Text;
newStudent.LastName = sf.lastName.Text;
newStudent.DateOfBirth = DateTime.Parse(sf.dateOfBirth.Text);
```

▶ Task 4: Assign the new student to a class and enable the user to save the details of the new student

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4a: Assign the new student to the current teacher** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
this.teacher.Students.Add(newStudent);
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4b: Add the student to the list displayed on the form** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
this.studentsInfo.Add(newStudent);
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4c: Enable saving (changes are not made permanent until they are written back to the database)** task.
6. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
saveChanges.IsEnabled = true;
```

▶ Task 5: Run the application and verify that the insert functionality works as expected

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Verify that the application starts and displays the initial list of students.
4. Click the row containing the name **Kevin Liu**.
5. Press Insert and verify that the new student window appears:
6. In the **First Name** text box, type **Darren**.
7. In the **Last Name** text box, type **Parker**.
8. In the **Date of Birth** text box, type **02/03/2006**, and then click **OK**.
9. Verify that Darren Parker has been added to the students list, and that the **Save Changes** button is now enabled. The ID of a new student will be 0 until they are saved to the database in the next lab.
10. Close the application.
11. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be able to add new students to a class.

Exercise 3: Implementing Delete Functionality for the Students List

▶ Task 1: Add logic to the key down method to detect if the Delete key has been pressed.

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod01\Labfiles\Starter\Exercise 3**, click **School.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO Exercise: 3: Task 1a: If the user pressed Delete, remove the currently selected student** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
case Key.Delete: student = this.studentsList.SelectedItem as Student;
```

▶ Task 2: Prompt the user to confirm that they want to remove the selected student from the class

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Prompt the user to confirm that the student should be removed** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
MessageBoxResult response = MessageBox.Show(
string.Format("Remove {0}", student.FirstName + " " + student.LastName),
"Confirm", MessageBoxButton.YesNo, MessageBoxImage.Question,
MessageBoxResult.No);
```

▶ Task 3: Remove the student and enable the user to save the changes

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3a: If the user clicked Yes, remove the student from the database** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (response == MessageBoxResult.Yes)
{
    this.schoolContext.Students.DeleteObject(student);
```

3. After the final comment in this method, type the following code:

```
}
break;
```

4. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3b: Enable saving (changes are not made permanent until they are written back to the database)** task.
5. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
saveChanges.IsEnabled = true;
```

▶ Task 4: Run the application and verify that the delete functionality works as expected

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

2. On the **Debug** menu, click **Start Without Debugging**.
3. Verify that the application starts and displays the initial list of students.
4. Click on the drop-down menu containing the text **Esther Valle: Class 3C**.
5. Click the list item containing the text **David Waite : Class 4B**.
6. Click the row containing the name **Jon Orton**.
7. Press Delete and verify that the confirmation prompt appears.
8. In the **Confirm** dialog box, click **Yes**, verify that Jon Orton is removed from the students list, and then verify that the **Save Changes** button is enabled.
9. Close the application.
10. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be able to remove students from classes.

Exercise 4: Displaying a Student's Age

▶ Task 1: Examine the MainWindow XAML

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod01\Labfiles\Starter\Exercise 4**, click **School.sln**, and then click **Open**.
3. On the **Build** menu, click **Build Solution**.
4. In Solution Explorer, expand the **School**, and then double-click the **MainWindow.xaml** and view the XAML markup.
5. Take note of the following lines of markup:

```
<app:AgeConverter x:key="ageConverter"/>
. . .
<GridViewColumn Width="75" Header="Age"
DisplayMemberBinding="{Binding Path=DateOfBirth, Converter={StaticResource
ageConverter}}" />
```

▶ Task 2: Add logic to the AgeConverter class to calculate a student's age from their date of birth

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2a: Check that the value provided is not null. If it is, return an empty string** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (value != null)
{
```

3. In the code editor, after all the comments in this method, delete the following line of code:

```
return "";
```

4. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2b: Convert the value provided into a DateTime value** task.
5. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
DateTime studentDateOfBirth = (DateTime)value;
```

6. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2c: Work out the difference between the current date and the value provided** task.
7. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
TimeSpan difference = DateTime.Now.Subtract(studentDateOfBirth);
```

8. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2d: Convert this result into a number of years** task.
9. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
int ageInYears = (int)(difference.Days / 365.25);
```

10. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2e: Convert the number of years into a string and return it** task.
11. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
        return ageInYears.ToString();
}
else
{
        return "";
}
```

▶ Task 3: Run the application and verify that the student's age now appears correctly

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Verify that the application starts and displays the initial list of students, with their ages.
4. Click the row containing the name **Kevin Liu**.
5. Press Insert.
6. In the new student window, enter your first name in the **First Name** box, your last name in the **Last Name** box and your date of birth in the **Date of Birth** box.
7. Click **OK** and verify that your name and age display correctly in the student list.
8. Close the application.
9. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the application will display a student's age in years.

Module 2: Creating Methods, Handling Exceptions, and Monitoring Applications

Lab: Extending the Class Enrollment Application Functionality

Exercise 1: Refactoring the Enrollment Code

▶ Task 1: Copy the code for editing a student into the studentsList_MouseDoubleClick event handler

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 Start window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod02\Labfiles\Databases** folder, and then double-click **SetupSchoolDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod02\Labfiles\Starter\Exercise 1**, click **School.sln**, and then click **Open**.
12. In Solution Explorer, expand **School**, expand **MainWindow.xaml**, and then double-click **MainWindow.xaml.cs**.
13. On the **View** menu, click **Task List**.
14. In the **Task List** window, in the **Categories** list, click **Comments**.
15. Double-click the **TODO: Exercise 1: Task 1a: If the user double-clicks a student, edit the details for that student** task.
16. Above the comment, in the **studentsList_Keydown** method, locate the **case Key.Enter:** block, and copy the following code to the clipboard:

```
Student student = this.studentsList.SelectedItem as Student;
// TODO: Exercise 1: Task 3a: Refactor as the editStudent method
// Use the StudentsForm to display and edit the details of the student
StudentForm sf = new StudentForm();
// Set the title of the form and populate the fields on the form with the details of
the student
sf.Title = "Edit Student Details";
sf.firstName.Text = student.FirstName;
sf.lastName.Text = student.LastName;
                    sf.dateOfBirth.Text = student.DateOfBirth.ToString("d"); //
Format the date to omit the time element
// Display the form
if (sf.ShowDialog().Value)
{
```

```
    // When the user closes the form, copy the details back to the student
                        student.FirstName = sf.firstName.Text;
                        student.LastName = sf.lastName.Text;
                        student.DateOfBirth = DateTime.Parse(sf.dateOfBirth.Text);
     // Enable saving (changes are not made permanent until they are written back to
the database)
     saveChanges.IsEnabled = true;
}
```

17. Double-click the **TODO: Exercise 1: Task 1a: If the user double-clicks a student, edit the details for the student** task.
18. Paste the code from the clipboard into **studentsList_MouseDoubleClick** method.

▶ Task 2: Run the application and verify that the user can now double-click a student to edit their details

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Click the row containing the name **Kevin Liu** and then press Enter.
4. Verify that the **Edit Student Details** window appears, displaying the correct details.
5. In the **Last Name** box, delete the existing contents, type **Cook**, and then click **OK**.
6. Verify that Liu has changed to Cook in the students list, and that the **Save Changes** button is now enabled.
7. Double-click the row containing the name **George Li.**
8. Verify that the **Edit Student Details** window appears, displaying the correct details.
9. In the **First Name** box, delete the existing contents, and then type **Darren**.
10. In the **Last Name** box, delete the existing contents, type **Parker**, and then click **OK**.
11. Verify that **George Li** has changed to **Darren Parker**.
12. Close the application.

▶ Task 3: Use the Analyze Solution for Code Clones wizard to detect the duplicated code

1. On the **Analyze** menu, click **Analyze Solution for Code Clones**.
2. In the **Code Clone Analysis Results** window, expand **Exact Match**.
3. Double-click the second row containing the text **MainWindow:studentsList_MouseDoubleClick**.
4. In the code editor, in the **studentsList_MouseDoubleClick** method, delete the following line of code:

```
Student student = this.studentsList.SelectedItem as Student;
```

5. In the **studentsList_MouseDoubleClick** method, highlight all of the code:

```
// TODO: Exercise 1: Task 3a: Refactor as the editStudent method
// Use the StudentsForm to display and edit the details of the student
StudentForm sf = new StudentForm();
// Set the title of the form and populate the fields on the form with the details of
the student
sf.Title = "Edit Student Details";
sf.firstName.Text = student.FirstName;
```

```
sf.lastName.Text = student.LastName;
sf.dateOfBirth.Text = student.DateOfBirth.ToString("d"); // Format the date to omit
the time element
// Display the form
if (sf.ShowDialog().Value)
{
      // When the user closes the form, copy the details back to the student
      student.FirstName = sf.firstName.Text;
       student.LastName = sf.lastName.Text;
       student.DateOfBirth = DateTime.Parse(sf.dateOfBirth.Text);
       // Enable saving (changes are not made permanent until they are written back
to the database)
       saveChanges.IsEnabled = true;
}
```

6. On the **Edit** menu, point to **Refactor**, and then click **Extract Method**.
7. In the **Extract Method** dialog box, in the **New method name** box, delete the existing contents, type **editStudent**, and then click **OK**.
8. In the **studentsList_MouseDoubleClick** method, modify the call to the **editStudent** method to look like the following code:

```
editStudent(this.studentsList.SelectedItem as Student);
```

9. Locate the **editStudent** method below the **studentsList_MouseDoubleClick** method, and modify the method parameters to look like the following code:

```
private void editStudent(Student student)
```

10. In the **Code Clone Analysis Results** window, double-click the row containing the text **MainWindow:studentsList_KeyDown**
11. In the code editor, in the **studentsList_KeyDown** method, in the **case Key.Enter:** block, delete the code shown in step 5.
12. Click at the end of the **Student student = this.studentsList.SelectedItem as Student;** code line, press Enter, and then type the following code:

```
editStudent(student);
```

13. Below the **Code Clone Analysis Results** window, click **Task List**.

▶ Task 4: Refactor the logic that adds and deletes a student into the addNewStudent and deleteStudent methods

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4a: Refactor as the addNewStudent method** task.
2. In the code editor, locate the **case Key.Insert:** block, and then highlight the following code:

```
// TODO: Exercise 1: Task 4a: Refactor as the addNewStudent method
// Use the StudentsForm to get the details of the student from the user
sf = new StudentForm();
// Set the title of the form to indicate which class the student will be added to
(the class for the currently selected teacher)
sf.Title = "New Student for Class " + teacher.Class;
// Display the form and get the details of the new student
if (sf.ShowDialog().Value)
{
       // When the user closes the form, retrieve the details of the student from the
form
       // and use them to create a new Student object
```

```
        Student newStudent = new Student();
        newStudent.FirstName = sf.firstName.Text;
        newStudent.LastName = sf.lastName.Text;
        newStudent.DateOfBirth = DateTime.Parse(sf.dateOfBirth.Text);
        // Assign the new student to the current teacher
                        this.teacher.Students.Add(newStudent);
          // Add the student to the list displayed on the form
                        this.studentsInfo.Add(newStudent);
          // Enable saving (changes are not made permanent until they are written back
to the database)
          saveChanges.IsEnabled = true;
}
```

3. On the **Edit** menu, point to **Refactor**, and then click **Extract Method**.
4. In the **Extract Method** dialog box, in the **New method name** box, type **addNewStudent**, and then click **OK**.
5. Locate the **addnewStudent** method and in the method, modify the **sf = new StudentForm();** code to look like the following code:

```
StudentForm sf = new StudentForm();
```

6. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4b: Refactor as the removeStudent method** task.
7. In the code editor, locate the **case Key.Delete** block, and cut the following code to the clipboard:

```
// TODO: Exercise 1: Task 4b: Refactor as the removeStudent method
// Prompt the user to confirm that the student should be removed
MessageBoxResult response = MessageBox.Show(
String.Format("Remove {0}", student.FirstName + " " + student.LastName), "Confirm",
MessageBoxButton.YesNo, MessageBoxImage.Question,
MessageBoxResult.No);
// If the user clicked Yes, remove the student from the database
if (response == MessageBoxResult.Yes)
{
                        this.schoolContext.Students.DeleteObject(student);
        // Enable saving (changes are not made permanent until they are written back
to the database)
        saveChanges.IsEnabled = true;
}
```

8. In the code editor, in the **case Key.Delete:** block, click at the end of **student = this.studentsList.SelectedItem as Student;** code line, press Enter, then type the following code

```
removeStudent(student);
```

9. Right-click the **removeStudent(student);** method call, point to **Generate**, and then click **Method Stub**.
10. Locate the **removeStudent** method below the **studentsList_KeyDown** method, delete all the generated code in this method, and then paste the code from the clipboard.

▶ Task 5: Verify that students can still be added and removed from the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** Menu, click **Start Without Debugging**.
3. Click the row containing the name **Kevin Liu**, and then press Insert.
4. Verify that the **New Student for Class 3C** window appears.

5. In the **First Name** box, type **Dominik**.
6. In the **Last Name** box, type **Dubicki**.
7. In the **Date of Birth** box, type **02/03/2006** and then click **OK**.
8. Verify that Dominik Dubicki has been added to the students list.
9. Click the row containing the name **Run Liu**, and then press Delete.
10. Verify that the confirmation prompt appears.
11. Click **Yes**, and then verify that Run Liu is removed from the students list.
12. Close the application.

▶ Task 6: Debug the application and step into the new method calls

1. In the code editor, locate the **studentsList_KeyDown** method, right-click on the **switch (e.key)** statement, point to **Breakpoint**, and then click **Insert Breakpoint**.
2. On the **Debug** menu, click **Start Debugging**.
3. Click the row containing the name **Kevin Liu** and press Enter.
4. In the **Immediate** window, click the **Call Stack** tab.
5. Note that the current method name is displayed in the **Call Stack** window.
6. In the **Watch 1** window, click the **Locals** tab.
7. Note the local variables **this**, **sender**, **e**, and **student** are displayed in the **Locals** window.
8. On the **Debug** menu, click **Step Over**.
9. Repeat step 8.
10. Look at the **Locals** window, and note after stepping over the **Student student = this.studentsList.SelectedItem as Student;** code, the value for the **student** variable has changed from **null** to **School.Data.Student**.
11. In the **Locals** window, expand **student** and note the values for **_FirstName** and **_LastName**.
12. On the **Debug** menu, click **Step Into**.
13. Note that execution steps into the **editStudent** method and that this method name has been added to the Call Stack.
14. Look at the **Locals** window and note that the local variables have changed to **this**, **student**, and **sf**.
15. On the **Debug** menu, click **Step Over**.
16. Repeat step 15 five times
17. In the **Locals** window, expand **sf** and note the values for **dateOfBirth**, **firstName**, and **lastName**.
18. On the **Debug** menu, click **Step Out** to run the remaining code in the **editStudent** method and step out again to the calling method.
19. In the **Edit Student Details** window, click **Cancel**.
20. Note that execution returns to the **studentsList_KeyDown** method.
21. On the **Debug** menu, click **Step Over**.
22. On the **Debug** menu, click **Continue**.
23. Click the row containing the name **Kevin Liu** and press Insert.

24. On the **Debug** menu, click **Step Over**.
25. On the **Debug** menu, click **Step Into**.
26. Note that execution steps into the **addNewStudent** method.
27. On the **Debug** menu, click **Step Out** to run the remaining code in the **addNewStudent** method and step out again to the calling method.
28. In the **New Student for Class 3C** window, click **Cancel**.
29. Note that execution returns to the **studentsList_KeyDown** method.
30. On the **Debug** menu, click **Step Over**.
31. On the **Debug** menu, click **Continue**.
32. Click the row containing the name **George Li** and press Delete.
33. On the **Debug** menu, click **Step Over**.
34. Repeat step 33.
35. On the **Debug** menu, click **Step Into**.
36. Note that execution steps into the **removeStudent** method.
37. On the **Debug** menu, click **Step Out** to run the remaining code in the **removeStudent** method and step out again to the calling method.
38. In the **Confirm** message box, click **No**.
39. Note that execution returns to the **studentList_KeyDown** method.
40. On the **Debug** menu, click **Step Over**.
41. On the **Debug** menu, click **Continue**.
42. Close the application
43. In Visual Studio, on the **Debug** menu, click **Delete All Breakpoints**.
44. In the **Microsoft Visual Studio** message box, click **Yes**.
45. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have updated the application to refactor duplicate code into reusable methods.

Exercise 2: Validating Student Information

▶ Task 1: Run the application and observe that student details that are not valid can be entered

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod02\Labfiles\Starter\Exercise 2**, click **School.sln**, and then click **Open**.
3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** menu, click **Start Without Debugging**.
5. Click on the row containing the name **Kevin Liu**, and then press Insert.
6. Leave the **First Name** and **Last Name** boxes empty.
7. In the **Date of Birth** box, type **10/06/3012**, and then click **OK**.
8. Verify that a new row has been added to the student list, containing a blank first name, blank last name, and a negative age.
9. Close the application.

▶ Task 2: Add code to validate the first name and last name fields

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Check that the user has provided a first name** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (String.IsNullOrEmpty(this.firstName.Text))
{
    MessageBox.Show("The student must have a first name", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
return;
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Check that the user has provided a last name** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
if (String.IsNullOrEmpty(this.lastName.Text))
{
    MessageBox.Show("The student must have a last name", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
return;
}
```

▶ Task 3: Add code to validate the date of birth

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Check that the user has entered a valid date for the date of birth** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
DateTime result;
if (!DateTime.TryParse(this.dateOfBirth.Text, out result))
{
```

```
      MessageBox.Show("The date of birth must be a valid date", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
       return;
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3b: Verify that the student is at least 5 years old** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
TimeSpan age = DateTime.Now.Subtract(result);
if (age.Days / 365.25 < 5)
{
    MessageBox.Show("The student must be at least 5 years old", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
    return;
}
```

▶ Task 4: Run the application and verify that student information is now validated correctly

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. Click on the row containing the name **Kevin Liu**, and then press Insert.
4. Leave the **First Name**, **Last Name**, and **Date of Birth** boxes empty and click **OK**.
5. Verify that an error message appears containing the text **The student must have a first name**.
6. In the **Error** message box, click **OK**.
7. In the new student window, in the **First Name** box, type **Darren**, and then click **OK**.
8. Verify that an error message appears containing the text **The student must have a last name**.
9. In the **Error** message box, click **OK**.
10. In the new student window, in the **Last Name** box, type **Parker**, and then click **OK**.
11. Verify that an error message appears containing the text **The date of birth must be a valid date**.
12. In the **Error** message box, click **OK**.
13. In the new student window, in the **Date of Birth** box, type **10/06/3012**, and then click **OK**.
14. Verify that an error message appears containing the text **The student must be at least 5 years old**.
15. In the **Error** message box, click **OK**.
16. In the new student window, in the **Date of Birth** box, delete the existing date, type **10/06/2006**, and then click **OK**.
17. Verify that Darren Parker is added to the student list with an age appropriate to the current date.
18. Close the application.
19. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, student data will be validated before it is saved.

Exercise 3: Saving Changes to the Class List

▶ Task 1: Verify that data changes are not persisted to the database

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod02\Labfiles\Starter\Exercise 3**, click **School.sln**, and then click **Open**.
3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** menu, click **Start Without Debugging**.
5. Click the row containing the name **Kevin Liu**.
6. Press Enter and verify that the **Edit Student Details** window appears displaying the correct details.
7. In the **Last Name** box, delete the existing contents, type **Cook**, and then click **OK**.
8. Verify that **Liu** has changed to **Cook** in the students list, and that the **Save Changes** button is now enabled.
9. Click **Save Changes**.
10. Click the row containing the student **George Li**, and then press Delete.
11. Verify that the confirmation prompt appears, and then click **Yes**.
12. Verify that **George Li** is removed from the student list, and then click **Save Changes**.
13. Close the application
14. On the **Debug** menu, click **Start Without Debugging**.
15. Verify that the application displays the original list of students.
16. Verify that **Kevin Liu** appears in the list instead of **Kevin Cook** and **George Li** is back in the student list.
17. Close the application.

▶ Task 2: Add code to save changes back to the database

1. In Visual Studio, in the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Bring the System.Data and System.Data.Objects namespace into scope** task.
2. In the code editor, click in the blank line above the comment, and then type the following code:

```
using System.Data;
using System.Data.Objects;
```

3. In Visual Studio, in the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: Save the changes by calling the SaveChanges method of the schoolContext object** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Save the changes
this.schoolContext.SaveChanges();
// Disable the Save button (it will be enabled if the user makes more changes)
saveChanges.IsEnabled = false;
```

▶ Task 3: Add exception handling to the code to catch concurrency, update, and general exceptions

1. In the code editor, enclose the code that you wrote in the previous task in a **try** block. Your code should look like the following:

```
try
{
     // Save the changes
     this.schoolContext.SaveChanges();
     // Disable the Save button (it will be enabled if the user makes more changes)
     saveChanges.IsEnabled = false;
}
```

2. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3a: If an OptimisticConcurrencyException occurs then another user has changed the same students earlier then overwrite their changes with the new data (see the lab instructions for details)** task.

3. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
catch (OptimisticConcurrencyException)
{
     // If the user has changed the same students earlier, then overwrite their
changes with the new data
     this.schoolContext.Refresh(RefreshMode.StoreWins, schoolContext.Students);
     this.schoolContext.SaveChanges();
}
```

4. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3b: If an UpdateException occurs then report the error to the user and rollback (see the lab instructions for details)** task.

5. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
catch (UpdateException uEx)
{
     // If some sort of database exception has occurred, then display the reason for
the exception and rollback
    MessageBox.Show(uEx.InnerException.Message, "Error saving changes");
    this.schoolContext.Refresh(RefreshMode.StoreWins, schoolContext.Students);
}
```

6. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3c: If some other sort of error has occurs, report the error to the user and retain the data so the user can try again - the error may be transitory (see the lab instructions for details)** task.

7. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
catch (Exception ex)
{
      // If some other exception occurs, report it to the user
      MessageBox.Show(ex.Message, "Error saving changes");
      this.schoolContext.Refresh(RefreshMode.ClientWins, schoolContext.Students);
}
```

▶ Task 4: Run the application and verify that data changes are persisted to the database

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

2. On the **Debug** menu, click **Start Without Debugging**.
3. Click the row containing the student **Kevin Liu**.
4. Press Enter, in the **Last Name** box delete the existing contents, type **Cook**, and then click **OK**.
5. Verify that **Liu** has changed to **Cook** in the students list, and that the **Save Changes** button is now enabled.
6. Click **Save Changes** and verify that the **Save Changes** button is now disabled.
7. Click the row containing the student **George Li** and press Delete.
8. Verify that the confirmation prompt appears, and then click **Yes**.
9. Verify that the **Save Changes** button is now enabled.
10. Click **Save Changes** and verify that the button is now disabled.
11. Close the application.
12. On the **Debug** menu, click **Start Without Debugging**.
13. Verify that the changes you made to the student data have been saved to the database and are reflected in the student list.
14. Close the application.
15. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, modified student data will be saved to the database

Module 3: Developing the Code for a Graphical Application

Lab: Writing the Code for the Grades Prototype Application

Exercise 1: Adding Navigation Logic to the Grades Prototype Application

▶ Task 1: Examine the window and views in the application

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window.
5. Click **Visual Studio 2012**.
6. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, browse to **E:\Mod03\Labfiles\Starter\Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
8. On the **Build** menu, click **Build Solution**.
9. In Solution Explorer, expand **GradesPrototype**, and then double-click **MainWindow.xaml**.
10. Note that this is the main window for the application that will host the following views:
    - LogonPage.xaml
    - StudentProfile.xaml
    - StudentsPage.xaml
11. In Solution Explorer, expand **Views**, and then double-click **LogonPage.xaml**.
12. Notice that this view contains text boxes for the username and password, a check box to identify the user as a teacher, and a button to log on to the application.
13. In Solution Explorer, double-click **StudentProfile.xaml**.
14. Notice that this view contains a Report Card that currently displays a list of dummy grades. The view also contains a **Back** button and a blank space that will display the student's name. This view is displayed when a student logs on or when a teacher views a student's profile.
15. In Solution Explorer, double-click **StudentsPage.xaml**.
16. Notice that this view contains the list of students in a particular class. This view is displayed when a teacher logs on. A teacher can click a student's name and the Students Profile view will be displayed, containing the selected student's data.

▶ Task 2: Define the LogonSuccess event and add dummy code for the Logon_Click event

1. On the **View** menu, click **Task List**.
2. In the **Task List** window, in the **Categories** list, click **Comments**.
3. Double-click the **TODO: Exercise 1: Task 2a: Define the LogonSuccess event handler** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
public event EventHandler LogonSuccess;
```

5. In the **Task List** window double-click the **TODO: Exercise 1: Task 2b: Implement the Logon_Click event handler for the Logon button** task.
6. In the code editor, click in the blank line below the comments, and then type the following code:

```
private void Logon_Click(object sender, RoutedEventArgs e)
{
    // Save the username and role (type of user) specified on the form in the global
context
    SessionContext.UserName = username.Text;
    SessionContext.UserRole = (bool)userrole.IsChecked ? Role.Teacher : Role.Student;
    // If the role is Student, set the CurrentStudent property in the global context
to a dummy student; Eric Gruber
    if (SessionContext.UserRole == Role.Student)
    {
        SessionContext.CurrentStudent = "Eric Gruber";
    }
    // Raise the LogonSuccess event
    if (LogonSuccess != null)
    {
        LogonSuccess(this, null);
    }
}
```

7. In Solution Explorer, double-click **LogonPage.xaml**.
8. In the XAML editor, locate the task **TODO: Exercise 1: Task 2c: Specify that the Logon button should raise the Logon_Click event handler in this view** task.
9. In the line below the comment, modify the XAML markup **<Button Grid.Row="3" Grid.ColumnSpan="2" VerticalAlignment="Center" HorizontalAlignment="Center" Content="Log on" FontSize="24" />** to look like the following markup:

```
<Button Grid.Row="3" Grid.ColumnSpan="2" VerticalAlignment="Center"
HorizontalAlignment="Center" Content="Log on" FontSize="24" Click="Logon_Click" />
```

▶ Task 3: Add code to display the Log On view

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Display the logon view and hide the list of students and single student view** task.
2. In the code editor, click in the blank line in the **GotoLogon** method, and then type the following code:

```
// Display the logon view and hide the list of students and single student view
logonPage.Visibility = Visibility.Visible;
studentsPage.Visibility = Visibility.Collapsed;
```

3. studentProfile.Visibility = Visibility.Collapsed;
4. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Handle successful logon** task.
5. In the code editor, click in the blank line below the comments, and then type the following code:

```
// Handle successful logon
private void Logon_Success(object sender, EventArgs e)
{
    // Update the display and show the data for the logged on user
    logonPage.Visibility = Visibility.Collapsed;
    gridLoggedIn.Visibility = Visibility.Visible;
```

```
    Refresh();
}
```

6. In Solution Explorer, double-click **MainWindow.xaml**.
7. In the XAML editor, locate the task **TODO: Exercise 1: Task 3c: Catch the LogonSuccess event and call the Logon_Success event handler (to be created)** task.
8. In the line below the comment, modify the XAML markup **<y:LogonPage x:Name="logonPage" Visibility="Collapsed" />** to look like the following markup:

```
<y:LogonPage x:Name="logonPage" LogonSuccess="Logon_Success" Visibility="Collapsed"
/>
```

▶ Task 4: Add code to determine the type of user

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4a: Update the display for the logged on user (student or teacher)** task.
2. In the code editor, click in the blank line in the **Refresh** method, and then type the following code:

```
switch (SessionContext.UserRole)
{
    case Role.Student:
        // Display the student name in the banner at the top of the page
        txtName.Text = string.Format("Welcome {0}", SessionContext.UserName);
        // Display the details for the current student
        GotoStudentProfile();
        break;
    case Role.Teacher:
        // Display the teacher name in the banner at the top of the page
        txtName.Text = string.Format("Welcome {0}", SessionContext.UserName);
        // Display the list of students for the teacher
        GotoStudentsPage();
        break;
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4b: Display the details for a single student** task.
4. In the code editor, click in the blank line in the **GotoStudentProfile** method, and then type the following code:

```
// Hide the list of students
studentsPage.Visibility = Visibility.Collapsed;
// Display the view for a single student
studentProfile.Visibility = Visibility.Visible;
studentProfile.Refresh();
```

5. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4c: Display the list of students** task.
6. In the code editor, click in the blank line in the **GotoStudentsPage** method, and then type the following code:

```
// Hide the view for a single student (if it is visible)
studentProfile.Visibility = Visibility.Collapsed;
// Display the list of students
studentsPage.Visibility = Visibility.Visible;
studentsPage.Refresh();
```

7. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4d: Display the details for the current student including the grades for the student** task.

8. In the code editor, click in the blank line in the **Refresh** method, and then type the following code:

```
// Parse the student name into the first name and last name by using a regular
expression
// The firstname is the initial string up to the first space character.
// The lastname is the string after the space character
Match matchNames = Regex.Match(SessionContext.CurrentStudent, @"([^ ]+) ([^ ]+)");
if (matchNames.Success)
{
    string firstName = matchNames.Groups[1].Value; // Indexing in the Groups
collection starts at 1, not 0
    string lastName = matchNames.Groups[2].Value;
    // Display the first name and last name in the TextBlock controls in the
studentName StackPanel
          ((TextBlock)studentName.Children[0]).Text = firstName;
                ((TextBlock)studentName.Children[1]).Text = lastName;
}
// If the current user is a student, hide the Back button
// (only applicable to teachers who can use the Back button to return to the list of
students)
if (SessionContext.UserRole == Role.Student)
{
    btnBack.Visibility = Visibility.Hidden;
}
else
{
    btnBack.Visibility = Visibility.Visible;
}
```

▶ Task 5: Handle the Student_Click event

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 5a: Handle the click event for a student** task.
2. In the code editor, click in the blank line in the **Student_Click** method, and then type the following code:

```
Button itemClicked = sender as Button;
if (itemClicked != null)
{
    // Find out which student was clicked - the Tag property of the button contains
the name
    string studentName = (string)itemClicked.Tag;
    if (StudentSelected != null)
    {
        // Raise the StudentSelected event (handled by MainWindow) to display the
details for this student
        StudentSelected(sender, new StudentEventArgs(studentName));
    }
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 5b: Handle the StudentSelected event when the user clicks a student on the Students page** task.
4. In the code editor, click in the blank line in the **studentsPage_StudentSelected** method, and then type the following code:

```
SessionContext.CurrentStudent = e.Child;
GotoStudentProfile();
```

5. In Solution Explorer, double-click **MainWindow.xaml**.
6. In the XAML editor, locate the task **TODO: Exercise 1: Task 5c: Catch the StudentSelected event and call the studentsPage_StudentSelected event handler** task.

7. In the line below the comment, modify the XAML markup **<y:StudentsPage x:Name="studentsPage" Visibility="Collapsed" />** to look like the following markup:

```
<y:StudentsPage x:Name="studentsPage" StudentSelected="studentsPage_StudentSelected"
Visibility="Collapsed" />
```

▶ Task 6: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the Debug menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password**.
4. Select the **Teacher** check box, and then click **Log on**.
5. Verify that the application displays the **StudentPage** view.

The Students page should look like this:

FIGURE 3.1:THE STUDENTS PAGE

6. Click the student **Kevin Liu** and verify that the application displays the **StudentProfile** view.

The Student Profile page should look like this:

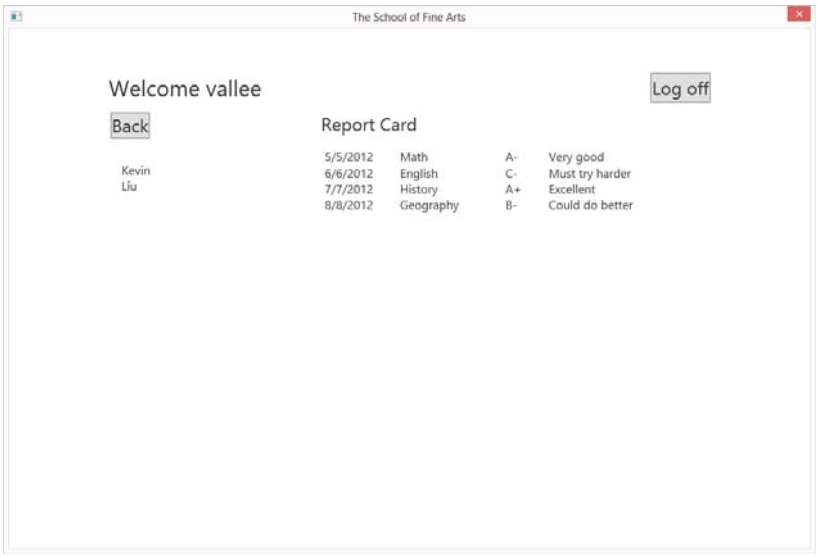

FIGURE 3.2:THE STUDENT PROFILE PAGE

7. Click **Log off**.
8. In the **Username** box, delete the existing contents, and then type **grubere**.
9. Clear the **Teacher** check box, and then click **Log on**.
10. Verify that the application displays the student profile page for Eric Gruber.
11. Close the application.
12. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have updated the Grades Prototype application to respond to user events and move among the application views appropriately.

Exercise 2: Creating Data Types to Store User and Grade Information

▶ Task 1: Define basic structs for holding Grade, Student, and Teacher information

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod03\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. On the **View** menu, click **Task List**.
4. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: Create the Grade struct** task.
5. In the code editor, click in the blank line below the comment, and then type the following code:

```
public struct Grade
{
    public int StudentID { get; set; }
    public string AssessmentDate { get; set; }
    public string SubjectName { get; set; }
    public string Assessment { get; set; }
    public string Comments { get; set; }
}
```

6. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Create the Student struct** task.
7. In the code editor, click in the blank line below the comment, and then type the following code:

```
public struct Student
{
    public int StudentID { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public int TeacherID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

8. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1c: Create the Teacher struct** task.
9. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
public struct Teacher
{
    public int TeacherID { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Class { get; set; }
}
```

▶ Task 2: Examine the dummy data source used to populate the collections

1. In Solution Explorer, expand **GradesPrototype**, expand **Data**, and then double-click **DataSource.cs**.
2. In the code editor, expand the region **Sample Data**, and then locate the method **CreateData**.
3. Note how the **Teachers ArrayList** is populated with **Teacher** data, each containing **TeacherID**, **UserName**, **Password**, **FirstName**, **LastName**, and **Class** fields.

4. Note how the **Students ArrayList** is populated with **Student** data, each containing a **StudentID**, **UserName**, **Password**, **TeacherID**, **FirstName**, and **LastName** fields.
5. Note how the **Grades ArrayList** is populated with **Grade** data, each containing a **StudentID**, **AssessmentDate**, **SubjectName**, **Assessment**, and **Comments** fields.

Results: After completing this exercise, the application will contain structs for the teacher, student, and grade types.

Exercise 3: Displaying User and Grade Information

▶ Task 1: Add the LogonFailed event

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod03\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1a: Define LogonFailed event** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
public event EventHandler LogonFailed;
```

5. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1b: Validate the username and password against the Users collection in the MainWindow window** task.
6. In the code editor, in the **Logon_Click** method, click in the blank line, and then type the following code:

```
// Find the user in the list of possible users - first check whether the user is a
Teacher
var teacher = (from Teacher t in DataSource.Teachers
               where String.Compare(t.UserName, username.Text) == 0
               && String.Compare(t.Password, password.Password) == 0
               select t).FirstOrDefault();
// If the UserName of the user retrieved by using LINQ is non-empty then the user is
a teacher
if (!String.IsNullOrEmpty(teacher.UserName))
{
    // Save the UserID and Role (teacher or student) and UserName in the global
context
    SessionContext.UserID = teacher.TeacherID;
    SessionContext.UserRole = Role.Teacher;
    SessionContext.UserName = teacher.UserName;
    SessionContext.CurrentTeacher = teacher;
    // Raise the LogonSuccess event and finish
    LogonSuccess(this, null);
    return;
}
// If the user is not a teacher, check whether the username and password match those
of a student
else
{
    var student = (from Student s in DataSource.Students
                   where String.Compare(s.UserName, username.Text) == 0
                    && String.Compare(s.Password, password.Password) == 0
                    select s).FirstOrDefault();
    // If the UserName of the user retrieved by using LINQ is non-empty then the user
is a student
    if (!String.IsNullOrEmpty(student.UserName))
    {
        // Save the details of the student in the global context
        SessionContext.UserID = student.StudentID;
        SessionContext.UserRole = Role.Student;
        SessionContext.UserName = student.UserName;
        SessionContext.CurrentStudent = student;
        // Raise the LogonSuccess event and finish
        LogonSuccess(this, null);
        return;
    }
}
// If the credentials do not match those for a Teacher or for a Student then they
must be invalid
```

```
// Raise the LogonFailed event
LogonFailed(this, null);
```

▶ Task 2: Add the Logon_Failed event handler

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Handle logon failure** task.
2. In the code editor, click in the blank line below the comments, and then type the following code:

```
private void Logon_Failed(object sender, EventArgs e)
{
    // Display an error message. The user must try again
    MessageBox.Show("Invalid Username or Password", "Logon Failed",
MessageBoxButton.OK, MessageBoxImage.Error);
}
```

3. In Solution Explorer, double-click **MainWindow.xaml**.
4. In the XAML editor, locate the task **TODO: Exercise 3: Task 2b: Connect the LogonFailed event of the logonPage view to the Logon_Failed method in MainWindow.xaml.cs** task.
5. In the line below the comment, modify the XAML markup **<y:LogonPage x:Name="logonPage" LogonSuccess="Logon_Success" Visibility="Collapsed" />** to look like the following markup:

```
<y:LogonPage x:Name="logonPage" LogonSuccess="Logon_Success"
LogonFailed="Logon_Failed" Visibility="Collapsed" />
```

6. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2c: Display the student name in the banner at the top of the page** task.
7. In the code editor, click in the blank line below the comment, and then type the following code:

```
// Display the student name in the banner at the top of the page
txtName.Text = string.Format("Welcome {0} {1}",
SessionContext.CurrentStudent.FirstName, SessionContext.CurrentStudent.LastName);
```

8. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2d: Display the teacher name in the banner at the top of the page** task.
9. In the code editor, click in the blank line below the comment, and then type the following code:

```
// Display the teacher name in the banner at the top of the page
txtName.Text = string.Format("Welcome {0} {1}",
SessionContext.CurrentTeacher.FirstName, SessionContext.CurrentTeacher.LastName);
```

▶ Task 3: Display the students for the current teacher

1. In Solution Explorer, expand **Views**, and then double-click **StudentsPage.xaml**.
2. In the XAML editor, locate the **ItemsControl** named **list** and note how data binding is used to display the name of each student.

Note: DataBinding is also used to retrieve the StudentID of a student. This binding is used when a user clicks on a Student on the Student Page list to identify which student's data to display in the Student Profile page.

3. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3a: Display students for the current teacher (held in SessionContext.CurrentTeacher)** task.
4. In the code editor, in the **Refresh** method, click in the blank line, and then type the following code:

```
// Find students for the current teacher
ArrayList students = new ArrayList();
foreach (Student student in DataSource.Students)
{
    if (student.TeacherID == SessionContext.CurrentTeacher.TeacherID)
    {
        students.Add(student);
    }
}
// Bind the collection to the list item template
list.ItemsSource = students;
// Display the class name
txtClass.Text = String.Format("Class {0}", SessionContext.CurrentTeacher.Class);
```

5. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3b: If the user clicks on a student, display the details for that student** task.

6. In the code editor, in the **Student_Click** method, click in the blank line, and then type the following code:

```
Button itemClicked = sender as Button;
if (itemClicked != null)
{
    // Find out which student was clicked
    int studentID = (int)itemClicked.Tag;
    if (StudentSelected !=null)
    {
        // Find the details of the student by examining the DataContext of the Button
        Student student = (Student)itemClicked.DataContext;
        // Raise the StudentSelected event (handled by MainWindow to display the
details for this student
        StudentSelected(sender, new StudentEventArgs(student));
    }
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3c: Set the current student in the global context to the student specified in the StudentEventArgs parameter** task.

8. In the code editor, click in the blank line below the comment, and then type the following code:

```
SessionContext.CurrentStudent = e.Child;
```

▶ Task 4: Set the DataContext for the page

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 4a: Display the details for the current student (held in SessionContext.CurrentStudent)** task.

2. In the code editor, click in the blank line below the comment, and then type the following code:

```
// Bind the studentName StackPanel to display the details of the student in the
TextBlocks in this panel
studentName.DataContext = SessionContext.CurrentStudent;
// If the current user is a student, hide the Back button
// (only applicable to teachers who can use the Back button to return to the list of
students)
if (SessionContext.UserRole == Role.Student)
{
    btnBack.Visibility = Visibility.Hidden;
}
else
{
    btnBack.Visibility = Visibility.Visible;
}
```

3. In Solution Explorer, expand **Views** and then double-click **StudentProfile.xaml**.
4. In the XAML editor, locate the task **TODO: Exercise 3: Task 4b: Bind the firstName TextBlock to the FirstName property of the DataContext for this control** task.
5. In the line below the comment, modify the XAML markup **<TextBlock x:Name="firstName" FontSize="16" />** to look like the following markup:

```
<TextBlock x:Name="firstName" Text="{Binding FirstName}" FontSize="16" />
```

6. In the XAML editor, locate the task **TODO: Exercise 3: Task 4c: Bind the lastName TextBlock to the LastName property of the DataContext for this control** task.
7. In the line below the comment, modify the XAML markup **<TextBlock x:Name="lastName" FontSize="16" />** to look like the following markup:

```
<TextBlock x:Name="lastName" Text="{Binding LastName}" FontSize="16"  />
```

8. In the **Task List** window, double-click the **TODO: Exercise 3: Task 4d: Create a list of the grades for the student and display this list on the page** task.
9. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Find all the grades for the student
ArrayList grades = new ArrayList();
foreach (Grade grade in DataSource.Grades)
{
    if (grade.StudentID == SessionContext.CurrentStudent.StudentID)
    {
        grades.Add(grade);
    }
}
// Display the grades in the studentGrades ItemsControl by using databinding
studentGrades.ItemsSource = grades;
```

▶ Task 5: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **parkerd**, in the **Password** box, type **password**, and then click **Log on**.
4. Verify that the **Logon Failed** dialog box appears, and then click **OK**.
5. In the **Username** box, delete the existing contents, type **vallee**, and then click **Log on**.
6. Verify that the Students page appears, displaying a list of students.
7. Click the student **Kevin Liu** and verify the Student Profile page for Kevin Liu is displayed.
8. Click **Log off**.
9. In the **Username** box, delete the existing contents, type **grubere**, and then click **Log on**.
10. Verify that the Student Profile page for Eric Gruber is displayed.
11. Close the application.
12. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, only valid users will be able to log on to the application and they will see only data appropriate to their role.

Module 4: Creating Classes and Implementing Type-Safe Collections

Lab: Adding Data Validation and Type-Safety to the Application

Exercise 1: Implementing the Teacher, Student, and Grade Structs as Classes

▶ Task 1: Convert the Grades struct into a class

1. Start the MSL-TNG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window.
5. Click **Visual Studio 2012**.
6. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Starter\Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
8. On the **View** menu, click **Task List**.
9. In the **Task List** window, in the **Categories** list, click **Comments**.
10. Double-click the **TODO: Exercise 1: Task 1a: Convert Grade into a class and define constructors** task.
11. In the code editor, below the comment, modify the **public struct Grade** declaration, replacing **struct** with **class**.

```
public class Grade
```

12. Click at the end of the code **public string Comments { get; set; }**, press Enter twice, and then type the following code:

```
// Constructor to initialize the properties of a new Grade
public Grade(int studentID, string assessmentDate, string subject, string assessment,
string comments)
{
    StudentID = studentID;
    AssessmentDate = assessmentDate;
    SubjectName = subject;
    Assessment = assessment;
    Comments = comments;
}
// Default constructor
public Grade()
{
    StudentID = 0;
    AssessmentDate = DateTime.Now.ToString("d");
    SubjectName = "Math";
    Assessment = "A";
    Comments = String.Empty;
}
```

▶ Task 2: Convert the Students and Teachers structs into classes

1. In the **Task List** window, in the **Categories** list, click **Comments**.
2. Double-click the **TODO: Exercise 1: Task 2a: Convert Student into a class, make the password property write-only, add the VerifyPassword method, and define constructors** task.
3. In the code editor, below the comment, modify the **public struct Student** declaration, replacing **struct** with **class**.

```
public class Student
```

4. Delete the following line of code from the **Student** class.

```
public string Password {get; set;}
```

5. Press Enter, and then type the following code:

```
private string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
public string Password {
    set
   {
      _password = value;
   }
}
public bool VerifyPassword(string pass)
{
     return (String.Compare(pass, _password) == 0);
}
```

Note: An application should not be able to read passwords; only set them and verify that a password is correct.

6. Click at the end of the code **public string LastName { get; set; }**, press Enter twice, and then type the following code:

```
// Constructor to initialize the properties of a new Student
public Student(int studentID, string userName, string password, string firstName,
string lastName, int teacherID)
{
   StudentID = studentID;
   UserName = userName;
   Password = password;
   FirstName = firstName;
   LastName = lastName;
   TeacherID = teacherID;
}
// Default constructor
public Student()
{
   StudentID = 0;
   UserName = String.Empty;
   Password = String.Empty;
   FirstName = String.Empty;
   LastName = String.Empty;
   TeacherID = 0;
}
```

7. In the **Task List** window, in the **Categories** list, click **Comments**.

8. Double-click the **TODO: Exercise 1: Task 2b: Convert Teacher into a class, make the password property write-only, add the VerifyPassword method, and define constructors** task.
9. In the code editor, below the comment, modify the **public struct Teacher** declaration, replacing **struct** with **class**.

```
public class Teacher
```

10. Delete the following line of code:

```
public string Password {get; set;},
```

11. Press Enter and then type the following code:

```
private string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
public string Password {
    set
   {
      _password = value;
   }
}
public bool VerifyPassword(string pass)
{
     return (String.Compare(pass, _password) == 0);
}
```

12. Click at the end of the code **public string Class {get; set;}**, press Enter twice, and then type the following code:

```
// Constructor to initialize the properties of a new Teacher
public Teacher(int teacherID, string userName, string password, string firstName,
string lastName, string className)
{
   TeacherID = teacherID;
   UserName = userName;
   Password = password;
   FirstName = firstName;
   LastName = lastName;
   Class = className;
}
// Default constructor
public Teacher()
{
   TeacherID = 0;
   UserName = String.Empty;
   Password = String.Empty;
   FirstName = String.Empty;
   LastName = String.Empty;
   Class = String.Empty;
}
```

▶ Task 3: Use the VerifyPassword method to verify the password when a user logs in

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Use the VerifyPassword method of the Teacher class to verify the teacher's password** task.
2. In the code editor, below the comment, in the code for the teacher variable, modify the **String.Compare(t.Password, password.Password) == 0** code to look like the following code:

```
t.VerifyPassword(password.Password)
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Check whether teacher is null before examining the UserName property** task.
4. In the code editor, in the line below the comment, modify the **if** statement condition from **!String.IsNullOrEmpty(teacher.UserName)** to look like the following code:

```
teacher != null && !String.IsNullOrEmpty(teacher.UserName)
```

5. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3c: Use the VerifyPassword method of the Student class to verify the student's password** task.
6. In the code editor, below the comment, in the code for the student variable, modify the **String.Compare(s.Password, password.Password) == 0** code to look like the following code:
7. s.VerifyPassword(password.Password)
8. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3d: Check whether student is null before examining the UserName property** task.
9. In the code editor, in the line below the comment, modify the **if** statement condition from **!String.IsNullOrEmpty(student.UserName)** to look like the following code:

```
student != null && !String.IsNullOrEmpty(student.UserName)
```

▶ Task 4: Build and run the application, and verify that a teacher or student can still log on

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** Menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password**, and then click **Log on**.
5. Verify that the welcome screen appears, displaying the list of students
6. Click **Log off**.
7. In the **Username** box, delete the existing contents, type **grubere**, and then click **Log on**.
8. Verify that the welcome screen appears, displaying the list of subjects and grades.
9. Click **Log off**.
10. Close the application.
11. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the Teacher, Student, and Grade structs will be implemented as classes and the **VerifyPassword** method will be called when a user logs on.

Exercise 2: Adding Data Validation to the Grade Class

▶ Task 1: Create a list of valid subject names

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: Define a List collection for holding the names of valid subjects** task.
4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
public static List<string> Subjects;
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Populate the list of valid subjects with sample data** task.
6. In the code editor, in the blank line below the comment, type the following code:

```
Subjects = new List<string>() { "Math", "English", "History", "Geography", "Science"
};
```

▶ Task 2: Add validation logic to the Grade class to check the data entered by the user

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Add validation to the AssessmentDate property** task.
2. In the code editor under comment, delete the **public string AssessmentDate { get; set; }** code, and then type the following code:

```
private string _assessmentDate;
public string AssessmentDate
{
   get
   {
       return _assessmentDate;
   }
   set
   {
       DateTime assessmentDate;
       // Verify that the user has provided a valid date
       if (DateTime.TryParse(value, out assessmentDate))
       {
           // Check that the date is no later than the current date
           if (assessmentDate > DateTime.Now)
           {
              // Throw an ArgumentOutOfRangeException if the date is after the
current date
               throw new ArgumentOutOfRangeException("AssessmentDate", "Assessment
date must be on or before the current date");
           }
           // If the date is valid, then save it in the appropriate format
           _assessmentDate = assessmentDate.ToString("d");
        }
        else
        {
             // If the date is not in a valid format then throw an ArgumentException
             throw new ArgumentException("AssessmentDate", "Assessment date is not
recognized");
        }
    }
```

```
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Add validation to the SubjectName property** task.

4. In the code editor, below the comment, delete the **public string SubjectName { get; set; }** code, and then type the following code:

```
private string _subjectName;
public string SubjectName
{
    get
    {
        return _subjectName;
    }
    set
    {
        // Check that the specified subject is valid
        if (DataSource.Subjects.Contains(value))
        {
             // If the subject is valid store the subject name
              _subjectName = value;
        }
        else
        {
             // If the subject is not valid then throw an ArgumentException
             throw new ArgumentException("SubjectName", "Subject is not recognized");
        }
    }
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Add validation to the Assessment property** task.

6. In the code editor, delete the **public string Assessment { get; set; }** code, and then type the following code:

```
private string _assessment;
public string Assessment
{
    get
    {
        return _assessment;
    }
    set
    {
        // Verify that the grade is in the range A+ to E-
        // Use a regular expression: a single character in the range A-E at the start
of the string followed by an optional + or - at the end of the string
        Match matchGrade = Regex.Match(value, @"[A-E][+-]?$");
        if (matchGrade.Success)
        {
            _assessment = value;
        }
        else
        {
            // If the grade is not valid then throw an ArgumentOutOfRangeException
            throw new ArgumentOutOfRangeException("Assessment", "Assessment grade
must be in the range of A+ to E-");
        }
    }
}
```

▶ Task 3: Add a unit test to verify that the validations defined for the Grade class functions as expected.

1. On the **File** menu, point to **Add**, and then click **New Project**.
2. In the **Add New Project** dialog box, in the **Installed** templates list, expand **Visual C#**, click **Test**, and then in the Templates list, click **Unit Test Project**.
3. In the **Name** box, type **GradesTest**, and then click **OK**.
4. In Solution Explorer, right-click **GradesTest**, and then click **Add Reference**.
5. In the **Reference Manager – GradesTest** dialog box, expand **Solution**.
6. Select the **GradesPrototype** check box, and then click **OK**.
7. In the code editor, in the **UnitTest1** class, delete all of the existing code, and then type the following code:

```
[TestInitialize]
public void Init()
{
   // Create the data source (needed to populate the Subjects collection)
   GradesPrototype.Data.DataSource.CreateData();
}
 [TestMethod]
public void TestValidGrade()
{
     GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1, "1/1/2012",
"Math", "A-", "Very good");
     Assert.AreEqual(grade.AssessmentDate, "1/1/2012");
     Assert.AreEqual(grade.SubjectName, "Math");
     Assert.AreEqual(grade.Assessment, "A-");
}
 [TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void TestBadDate()
{
    // Attempt to create a grade with a date in the future
    GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1, "1/1/2023",
"Math", "A-", "Very good");
}
 [TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void TestDateNotRecognized ()
{
    // Attempt to create a grade with an unrecognized date
    GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1,
"13/13/2012", "Math", "A-", "Very good");
}
 [TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void TestBadAssessment()
{
    // Attempt to create a grade with an assessment outside the range A+ to E-
    GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1, "1/1/2012",
"Math", "F-", "Terrible");
}
 [TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void TestBadSubject()
{
    // Attempt to create a grade with an unrecognized subject
    GradesPrototype.Data.Grade grade = new GradesPrototype.Data.Grade(1, "1/1/2012",
"French", "B-", "OK");
}
```

8. On the **Build** menu, click **Build Solution**.
9. On the **Test** menu, point to **Run**, and then click **All Tests**.
10. In the **Test Explorer** window, verify that all the tests are passed.
11. Close **Test Explorer**.
12. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the Grade class will contain validation logic.

Exercise 3: Displaying Students in Name Order

▶ Task 1: Run the application and verify that the students are not displayed in any specific order when logged on as a teacher

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** Menu, click **Start Without Debugging**.
5. In the **Username** box, type **vallee**.
6. In the **Password** box, type **password**, and then click **Log on**.
7. Verify that the students are not displayed in any specific order.
8. Close the application.

▶ Task 2: Implement the IComparable<Student> interface to enable comparison of students

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Specify that the Student class implements the IComparable<Student> interface** task.
2. In the code editor, click at the end of the **public class Student** declaration, and then type the following code:

```
: IComparable<Student>
```

3. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: Compare Student objects based on their LastName and FirstName properties** task.
4. In the code editor, in the blank line below the comment, type the following code:

```
// Compare Student objects based on their LastName and FirstName properties
public int CompareTo(Student other)
{
    // Concatenate the LastName and FirstName of this student
string thisStudentsFullName = LastName + FirstName;
     // Concatenate the LastName and FirstName of the "other" student
string otherStudentsFullName = other.LastName + other.FirstName;
     // Use String.Compare to compare the concatenated names and return the result
return(String.Compare(thisStudentsFullName, otherStudentsFullName));
}
```

▶ Task 3: Change the Students ArrayList collection into a List<Student> collection

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3a: Change the Students collection into a List<Student>** task.
2. In the code editor, below the comment, modify the **public static ArrayList Students;** code to look like the following code:

```
public static List<Student> Students;
```

3. In the **Task List** window, double-click the **TODO: Exercise 3: Task 3b: Populate the List<Student> collection** task.

4. In the code editor, below the comment, modify the **Students = new ArrayList()** code to look like the following code:

```
Students = new List<Student>()
```

▶ Task 4: Sort the data in the Students collection

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 4a: Sort the data in the Students collection** task.
2. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
DataSource.Students.Sort();
```

▶ Task 5: Verify that Students are retrieved and displayed in order of their first name and last name

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password**, and then click **Log on**.
5. Verify that the students are displayed in order of ascending last name.
6. Log off and then close the application.
7. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the application will display the students in alphabetical order of last name and then first name.

Exercise 4: Enabling Teachers to Modify Class and Grade Data

▶ Task 1: Change the Teachers and Grades collections to be generic List collections

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod04\Labfiles\Starter\Exercise 4**, click **GradesPrototype.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1a: Change the Teachers collection into a generic List** task.
4. In the code editor, below the comment, modify the code **public static ArrayList Teachers;** to look like the following code:

```
public static List<Teacher> Teachers;
```

5. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1b: Change the Grades collection into a generic List** task.
6. In the code editor, below the comment, modify the code **public static ArrayList Grades;** to look like the following code:

```
public static List<Grade> Grades;
```

7. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1c: Populate the Teachers collection** task.
8. In the code editor, below the comment, modify the code **Teachers = new ArrayList()** to look like the following code:

```
Teachers = new List<Teacher>()
```

9. In the **Task List** window, double-click the **TODO: Exercise 4: Task 1d: Populate the Grades collection** task.
10. In the code editor, below the comment, modify the code **Grades = new ArrayList()**to look like the following code:

```
Grades = new List<Grade>()
```

▶ Task 2: Add the EnrollInClass and RemoveFromClass methods for the Teacher class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2a: Enroll a student in the class for this teacher** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
public void EnrollInClass(Student student)
{
    // Verify that the student is not already enrolled in another class
    if (student.TeacherID ==0)
    {
        // Set the TeacherID property of the student
        student.TeacherID = TeacherID;
     }
     else
     {
        // If the student is already assigned to a class, throw an ArgumentException
        throw new ArgumentException("Student", "Student is already assigned to a
class");
     }
```

```
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2b: Remove a student from the class for this teacher** task.

4. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Remove a student from the class for this teacher
public void RemoveFromClass(Student student)
{
   // Verify that the student is actually assigned to the class for this teacher
   if (student.TeacherID == TeacherID)
   {
       // Reset the TeacherID property of the student
       student.TeacherID = 0;
   }
   else
   {
       // If the student is not assigned to the class for this teacher, throw an
ArgumentException
       throw new ArgumentException("Student", "Student is not assigned to this
class");
   }
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 4: Task 2c: Add a grade to a student (the grade is already populated)** task.

6. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Add a grade to a student (the grade is already populated)
public void AddGrade(Grade grade)
{
   // Verify that the grade does not belong to another student - the StudentID should
be zero
   if (grade.StudentID == 0)
   {
      // Add the grade to the student's record
      grade.StudentID = StudentID;
    }
    else
    {
        // If the grade belongs to a different student, throw an ArgumentException
        throw new ArgumentException("Grade", "Grade belongs to a different student");
    }
}
```

▶ Task 3: Add code to enroll a student in a teacher's class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3a: Enroll a student in the teacher's class** task.

2. In the code editor, below the comment, click in the blank line in the **Student_Click** method, and then type the following code:

```
try
{
    // Determine which student the user clicked
    // the StudentID is held in the Tag property of the Button that the user clicked
    Button studentClicked = sender as Button;
int studentID = (int)studentClicked.Tag;
    // Find this student in the Students collection
```

```
    Student student = (from s in DataSource.Students
                       where s.StudentID == studentID
                       select s).First();
    // Prompt the user to confirm that they wish to add this student to their class
    string message = String.Format("Add {0} {1} to your class?", student.FirstName,
student.LastName);
MessageBoxResult reply = MessageBox.Show(message, "Confirm", MessageBoxButton.YesNo,
MessageBoxImage.Question);
    // If the user confirms, add the student to their class
    if (reply == MessageBoxResult.Yes)
    {
        // Get the ID of the currently logged-on teacher
        int teacherID = SessionContext.CurrentTeacher.TeacherID;
        // Assign the student to this teacher's class
SessionContext.CurrentTeacher.EnrollInClass(student);
        // Refresh the display - the new assigned student should disappear from the
list of unassigned students
        Refresh();
     }
}
catch (Exception ex)
{
      MessageBox.Show(ex.Message, "Error enrolling student", MessageBoxButton.OK,
MessageBoxImage.Error);
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3b: Refresh the display of unassigned students** task.

4. In the code editor, below the comment, click in the blank line in the **Refresh** method, and then type the following code:

```
// Find all unassigned students - they have a TeacherID of 0
var unassignedStudents = from s in DataSource.Students
                         where s.TeacherID == 0
                         select s;
// If there are no unassigned students, then display the "No unassigned students"
message
// and hide the list of unassigned students
if (unassignedStudents.Count() == 0)
{
    txtMessage.Visibility = Visibility.Visible;
    list.Visibility = Visibility.Collapsed;
}
else
{
     // If there are unassigned students, hide the "No unassigned students" message
     // and display the list of unassigned students
     txtMessage.Visibility = Visibility.Collapsed;
     list.Visibility = Visibility.Visible;
      // Bind the ItemControl on the dialog to the list of unassigned students
      // The names of the students will appear in the ItemsControl on the dialog
      list.ItemsSource = unassignedStudents;
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 4: Task 3c: Enroll a student in the teacher's class** task.

6. In the code editor, below the comment, click in the blank line in the **EnrollStudent_Click** method, and then type the following code:

```
// Use the AssignStudentDialog to display unassigned students and add them to the
teacher's class
// All of the work is performed in the code behind the dialog
AssignStudentDialog asd = new AssignStudentDialog();
```

```
asd.ShowDialog();
// Refresh the display to show any newly enrolled students
Refresh();
```

▶ Task 4: Add code to enable a teacher to remove the student from the assigned class

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 4a: Enable a teacher to remove a student from a class** task.
2. In the code editor, below the comment, click in the blank line in the **Remove_Click** method, and then type the following code:

```
// If the user is not a teacher, do nothing (the button should not appear anyway)
if (SessionContext.UserRole != Role.Teacher)
{
   return;
}
try
{
    // If the user is a teacher, ask the user to confirm that this student should be
removed from their class
    string message = String.Format("Remove {0} {1}",
SessionContext.CurrentStudent.FirstName, SessionContext.CurrentStudent.LastName);
    MessageBoxResult reply = MessageBox.Show(message, "Confirm",
MessageBoxButton.YesNo, MessageBoxImage.Question);
    // If the user confirms, then call the RemoveFromClass method of the current
teacher to remove this student from their class
    if (reply == MessageBoxResult.Yes)
    {
        SessionContext.CurrentTeacher.RemoveFromClass(SessionContext.CurrentStudent);
        // Go back to the previous page - the student is no longer a member of the
class for the current teacher
        if (Back != null)
        {
             Back(sender, e);
        }
    }
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message, "Error removing student from class",
MessageBoxButton.OK, MessageBoxImage.Error);
}
```

▶ Task 5: Add code to enable a teacher to add a grade to a student

1. In the **Task List** window, double-click the **TODO: Exercise 4: Task 5a: Enable a teacher to add a grade to a student** task.
2. In the code editor, below the comment, click in the blank line in the **AddGrade_Click** method, and then type the following code:

```
// If the user is not a teacher, do nothing (the button should not appear anyway)
if (SessionContext.UserRole != Role.Teacher)
{
    return;
}
try
{
    // Use the GradeDialog to get the details of the assessment grade
    GradeDialog gd = new GradeDialog();
    // Display the form and get the details of the new grade
    if (gd.ShowDialog().Value)
    {
```

```
        // When the user closes the form, retrieve the details of the assessment
grade from the form
        // and use them to create a new Grade object
        Grade newGrade = new Grade();
        newGrade.AssessmentDate = gd.assessmentDate.SelectedDate.Value.ToString("d");
        newGrade.SubjectName = gd.subject.SelectedValue.ToString();
        newGrade.Assessment = gd.assessmentGrade.Text;
        newGrade.Comments = gd.comments.Text;
        // Save the grade to the list of grades
        DataSource.Grades.Add(newGrade);
        // Add the grade to the current student
        SessionContext.CurrentStudent.AddGrade(newGrade);
        // Refresh the display so that the new grade appears
        Refresh();
    }
}
catch (Exception ex)
{
       MessageBox.Show(ex.Message, "Error adding assessment grade",
MessageBoxButton.OK, MessageBoxImage.Error);
}
```

▶ Task 6: Run the application and verify that students can be added to and removed from classes, and that grades can be added to students

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password**, and then click **Log on**.
5. Click **New Student**.
6. In the **First Name** box, type **Darren**.
7. In the **Last Name** box, type **Parker**.
8. In the **Password** box, type **password**, and then click **OK**.
9. Click **Enroll Student**.
10. Verify that the **Assign Student** dialog box appears and that **Darren Parker** is in the list.
11. Click **Darren Parker**.
12. Verify that the **Confirm** message box appears, and then click **Yes**.
13. In the **Assign Student** dialog box, verify that Darren Parker disappears and that the text "No unassigned students" is displayed.
14. Click **Close**.
15. Verify that Darren Parker is added to the student list.
16. Click the student **Kevin Liu**.
17. Click **Remove Student**.
18. Verify that the **Confirm** message box appears, and then click **Yes**.
19. Verify that Kevin Liu is removed from the student list.
20. Click the student **Darren Parker**.
21. Click **Add Grade**.

22. Verify that the **New Grade Details** dialog box appears.
23. Verify that the Date box contains the current date.
24. In the **Subject** list, click **English**.
25. In the **Assessment** box, type **B**.
26. In the **Comments** box, type **Good**, and then click **OK**.
27. Verify that the grade information appears on the Report Card.
28. Click **Log off**.
29. In the **Username** box, type **parkerd**.
30. Click **Log on**.
31. Verify that the **Welcome Darren Parker** screen is displayed, showing the Report Card and the previously added grade.
32. Click **Log off**.
33. Close the application.
34. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, the application will enable teachers to add and remove students from their classes, and to add grades to students.

Module 5: Creating a Class Hierarchy by Using Inheritance

Lab: Refactoring Common Functionality into the User Class

Exercise 1: Creating and Inheriting from the User Base Class

▶ Task 1: Create the User abstract base class

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window.
5. Click **Visual Studio 2012**.
6. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
8. In Visual Studio, on the **View** menu, click **Task List**.
9. In the **Task List** window, in the **Categories** list, click **Comments**.
10. Double-click the **TODO: Exercise 1: Task 1a: Create the User abstract class with the common functionality for Teachers and Students** task.
11. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public abstract class User
{
```

12. Click at the end of the last comment in the block (before the Grade class declaration), press Enter, and then type the following code:

```
}
```

13. In the **Task List** window, double click the **TODO: Exercise 1: Task 1b: Add the UserName property to the User class** task.
14. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public string UserName { get; set; }
```

15. In the **Task List** window, double click the **TODO: Exercise 1: Task 1c: Add the Password property to the User class** task.
16. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
private string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
public string Password
{
   set
   {
      _password = value;
```

```
    }
}
```

17. In the **Task List** window, double click the **TODO: Exercise 1: Task 1d: Add the VerifyPassword method to the User class** task.
18. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public bool VerifyPassword(string pass)
{
    return (String.Compare(pass, _password) == 0);
}
```

▶ Task 2: Modify the Student and Teacher classes to inherit from the User class

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2a: Inherit from the User class** task.
2. In the code editor, modify the statement below this comment as shown below in bold:

```
public class Student: User, IComparable<Student>
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Remove the UserName property (now inherited from User)** task.
4. In the code editor, delete the following statement from below the comment:

```
public string UserName { get; set; }
```

5. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2c: Remove the Password property (now inherited from User)** task.
6. In the code editor, delete the following block of code from below the comment:

```
private string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
public string Password
{
    set
    {
        _password = value;
    }
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2d Remove the VerifyPassword method (now inherited from User)** task.
8. In the code editor, delete the following method from below the comment:

```
public bool VerifyPassword(string pass)
{
    return (String.Compare(pass, _password) == 0);
}
```

9. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2e: Inherit from the User class** task.
10. In the code editor, modify the statement below this comment as shown below in bold:

```
public class Teacher: User
```

11. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2f: Remove the UserName property (now inherited from User)** task.
12. In the code editor, delete the following statement from below the comment:

```
public string UserName { get; set; }
```

13. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2g: Remove the Password property (now inherited from User)** task.
14. In the code editor, delete the following block of code from below the comment:

```
private string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
public string Password
{
    set
    {
        _password = value;
    }
}
```

15. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2h Remove the VerifyPassword method (now inherited from User)** task.
16. In the code editor, delete the following method from below the comment:

```
public bool VerifyPassword(string pass)
{
    return (String.Compare(pass, _password) == 0);
}
```

▶ Task 3: Run the application and test the log on functionality

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application starts, in the **Username** box, type **vallee**, in the **Password** box, type **password**, and then click **Log on**.
4. Verify that the list of students for teacher Esther Valle is displayed.
5. Click **Kevin Liu**, and verify that the report card displaying the grades for Kevin Liu is displayed.
6. Click **Log off**.
7. In the **Username** box, type **liuk**, in the **Password** box, type **password**, and then click **Log on**.
8. Verify that the report card showing the grades for Kevin Liu is displayed again.
9. Click **Log off**.
10. Close the application.
11. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have removed the duplicated code from the **Student** and **Teacher** classes, and moved the code to an abstract base class called **User**.

Exercise 2: Implementing Password Complexity by Using an Abstract Method

▶ Task 1: Define the SetPassword abstract method

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. In Visual Studio, on the **View** menu, click **Task List**.
4. In the **Task List** window, in the **Categories** list, click **Comments**.
5. Double-click the **TODO: Exercise 2: Task 1a: Define an abstract method for setting the password** task.
6. In the code editor, review the comment below this line, click at the end of the comment, press Enter, and then type the following code:

```
public abstract bool SetPassword(string pwd);
```

7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Use the SetPassword method to set the password** task.
8. In the code editor, delete the following statement:

```
_password = value;
```

9. Add the following block of code in the place of the statement that you just deleted:

```
if (!SetPassword(value))
{
    throw new ArgumentException("Password not complex enough", "Password");
}
```

▶ Task 2: Implement the SetPassword method in the Student and Teacher classes

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Make _password a protected field rather than private** task.
2. In the code editor, modify the statement below the comment as shown below in bold:

```
protected string _password = Guid.NewGuid().ToString(); // Generate a random password
by default
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Implement SetPassword to set the password for the student** task.
4. In the code editor, review the comment below this line, click at the end of the comment, press Enter, and then type the following code:

```
public override bool SetPassword(string pwd)
{
    // If the password provided as the parameter is at least 6 characters long then
save it and return true
    if (pwd.Length >= 6)
    {
        _password = pwd;
        return true;
    }
    // If the password is not long enough, then do not save it and return false
    return false;
```

```
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Implement SetPassword to set the password for the teacher** task.
6. In the code editor, review the comment below this line, click at the end of the comment, press Enter, and then type the following code:

```
public override bool SetPassword(string pwd)
{
    // Use a regular expression to check that the password contains at least two
numeric characters
    Match numericMatch = Regex.Match(pwd, @".*[0-9]+.*[0-9]+.*");
    // If the password provided as the parameter is at least 8 characters long and
contains at least two numeric characters then save it and return true
    if (pwd.Length >= 8 && numericMatch.Success)
    {
        _password = pwd;
        return true;
    }
    // If the password is not complex enough, then do not save it and return false
    return false;
}
```

▶ Task 3: Set the password for a new student

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Use the SetPassword method to set the password.** task.
2. In the code editor, delete the statement below this comment and replace it with the following block of code:

```
if (!newStudent.SetPassword(sd.password.Text))
{
    throw new Exception("Password must be at least 6 characters long. Student not
created");
}
```

▶ Task 4: Change the password for an existing user

1. On the **Build** menu, click **Build Solution**.
2. In Solution Explorer, expand the **GradesPrototype** project, and then double-click **MainWindow.xaml**.
3. In the XAML pane, scroll down to line 27 and review the following block of XAML code:

```
<Button Grid.Column="2" Margin="5" HorizontalAlignment="Right"
Click="ChangePassword_Click">
    <TextBlock Text="Change Password" FontSize="24"/>
</Button>
```

4. In Solution Explorer, expand **MainWindow.xaml** and then double-click **MainWindow.xaml.cs**.
5. In the code editor, expand the **Event Handlers** region, and locate the **ChangePassword_Click** method.
6. Review the code in this method:

```
private void ChangePassword_Click(object sender, EventArgs e)
{
    // Use the ChangePasswordDialog to change the user's password
    ChangePasswordDialog cpd = new ChangePasswordDialog();
```

```
    // Display the dialog
    if (cpd.ShowDialog().Value)
    {
        // When the user closes the dialog by using the OK button, the password
should have been changed
        // Display a message to confirm
        MessageBox.Show("Password changed", "Password", MessageBoxButton.OK,
MessageBoxImage.Information);
    }
}
```

7. In Solution Explorer, expand **Controls**, and then double-click **ChangePasswordDialog.xaml**.
8. In Solution Explorer, expand **ChangePasswordDialog.xaml** and then double-click **ChangePasswordDialog.xaml.cs**.
9. Review the code in the **ok_Click** method:

```
// If the user clicks OK to change the password, validate the information that the
user has provided
private void ok_Click(object sender, RoutedEventArgs e)
{
    // TODO: Exercise 2: Task 4a: Get the details of the current user
    // TODO: Exercise 2: Task 4b: Check that the old password is correct for the
current user
    // TODO: Exercise 2: Task 4c: Check that the new password and confirm password
fields are the same
    // TODO: Exercise 2: Task 4d: Attempt to change the password
    // If the password is not sufficiently complex, display an error message
    // Indicate that the data is valid
    this.DialogResult = true;
   }
```

10. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4a: Get the details of the current user** task.
11. In the code editor, in the blank line below this comment, type the following code:

```
User currentUser;
if (SessionContext.UserRole == Role.Teacher)
{
    currentUser = SessionContext.CurrentTeacher;
}
else
{
    currentUser = SessionContext.CurrentStudent;
}
```

12. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4b: Check that the old password is correct for the current user** task.
13. In the code editor, in the blank line below this comment, type the following code:

```
string oldPwd = oldPassword.Password;
if (!currentUser.VerifyPassword(oldPwd))
{
    MessageBox.Show("Old password is incorrect", "Error", MessageBoxButton.OK,
MessageBoxImage.Error);
    return;
}
```

14. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4c: Check that the new password and confirm password fields are the same** task.

15. In the code editor, in the blank line below this comment, type the following code:

```
string newPwd = newPassword.Password;
    string confirmPwd = confirm.Password;
if (String.Compare(newPwd, confirmPwd) != 0)
{
    MessageBox.Show("The new password and confirm password fields are different",
"Error", MessageBoxButton.OK, MessageBoxImage.Error);
    return;
}
```

16. In the **Task List** window, double-click the **TODO: Exercise 2: Task 4d: Attempt to change the password** task.

17. In the code editor, review the comment below this line, click at the end of the comment, press Enter, and then type the following code:

```
if (!currentUser.SetPassword(newPwd))
{
    MessageBox.Show("The new password is not sufficiently complex", "Error",
MessageBoxButton.OK, MessageBoxImage.Error);
    return;
}
```

▶ Task 5: Run the application and test the change password functionality

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application starts, in the **Username** box, type **vallee**, in the **Password** box, type **password99**, and then click **Log on**.
4. In **The School of Fine Arts** window, click **Change Password**.
5. In the **Change Password Dialog** window, in the **Old Password** box, type **password99**, in the **New Password** box, type **pwd101**, in the **Confirm** box, type **pwd101**, and then click **OK**.
6. Verify that the message **The new password is not sufficiently complex** is displayed, and then click **OK**.
7. In the **New Password** box, type **password101**, in the **Confirm** box, type **password101**, and then click **OK**.
8. Verify that the message **Password changed** is displayed, and then click **OK**.
9. Click **Log off**.
10. In the **Username** box, type **vallee**, in the **Password** box, type **password101**, and then click **Log on**.
11. Click **New Student**.
12. In the **New Student Details** window, in the **First Name** box, type **Luka**, in the **Last Name** box, type **Abrus**, in the **Password** box, type **1234**, and then click **OK**.
13. Verify that the message **Password must be at least 6 characters long. Student not created** appears, and then click **OK**.
14. Click **New Student**.
15. In the **New Student Details** window, in the **First Name** box, type **Luka**, in the **Last Name** box, type **Abrus**, in the **Password** box, type **abcdef**, and then click **OK**.
16. Click **Enroll Student**.

17. In the **Assign Student** window, verify that the student Luka Abrus appears.
18. Click **Close**.
19. Click **Log off**.
20. Close the application.
21. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have implemented a polymorphic method named **SetPassword** that exhibits different behavior for students and teachers. You will also have modified the application to enable users to change their passwords.

Exercise 3: Creating the ClassFullException Custom Exception

▶ Task 1: Implement the ClassFullException class

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod05\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. In Visual Studio, on the **View** menu, click **Task List**.
4. In the **Task List** window, in the **Categories** list, click **Comments**.
5. Double-click the **TODO: Exercise 3: Task 1a: Add custom data: the name of the class that is full** task.
6. In the code editor, review the comment below this task, click at the end of the comment, press Enter, and then type the following code:

```
private string _className;
public virtual string ClassName
{
    get
    {
        return _className;
    }
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1b: Delegate functionality for the common constructors directly to the Exception class** task.
8. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public ClassFullException()
{
}
public ClassFullException(string message)
    : base(message)
{
}
public ClassFullException(string message, Exception inner)
    : base(message, inner)
{
}
```

9. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1c: Add custom constructors that populate the _className field.** task.
10. In the code editor, review the comment below this task, click at the end of the comment, press Enter, and then type the following code:

```
public ClassFullException(string message, string cls)
    : base(message)
{
    _className = cls;
}
public ClassFullException(string message, string cls, Exception inner)
    : base(message, inner)
{
    _className = cls;
}
```

▶ Task 2: Throw and catch the ClassFullException

1. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Set the maximum class size for any teacher** task.
2. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
private const int MAX_CLASS_SIZE = 8;
```

3. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: If the class is already full, then another student cannot be enrolled** task.
4. In the code editor, review the comment below this task, click at the end of the comment, press Enter, and then type the following code:

```
if (numStudents == MAX_CLASS_SIZE)
{
    // Throw a ClassFullException and specify the class that is full
    throw new ClassFullException("Class full: Unable to enroll student", Class);
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2c: Catch and handle the ClassFullException** task.
6. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
catch (ClassFullException cfe)
{
    MessageBox.Show(String.Format("{0}. Class: {1}", cfe.Message, cfe.ClassName),
"Error enrolling student", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

▶ Task 3: Build and test the solution

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application starts, in the **Username** box, type **vallee**, in the **Password** box, type **password99**, and then click **Log on**.
4. In **The School of Fine Arts** window, click **New Student**.
5. In the **New Student Details** window, enter the following details, and then click **OK**.

Field	Value
First Name	Walter
Last Name	Harp
Password	abcdef

 **Note:** New students will not be listed in the main application window because this displays students in the users' class, and the new students have yet to be assigned to a class.

6. In **The School of Fine Arts** window, click **New Student**.

7. In the **New Student Details** window, enter the following details, and then click **OK**.

Field	Value
First Name	Andrew
Last Name	Harris
Password	abcdef

8. In **The School of Fine Arts** window, click **New Student**.
9. In the **New Student Details** window, enter the following details, and then click **OK**.

Field	Value
First Name	Toni
Last Name	Poe
Password	abcdef

10. In **The School of Fine Arts** window, click **New Student**.
11. In the **New Student Details** window, enter the following details, and then click **OK**.

Field	Value
First Name	Ben
Last Name	Andrews
Password	abcdef

12. In **The School of Fine Arts** window, click **Enroll Student**.
13. In the **Assign Student** window, click **Walter Harp**.
14. In the **Confirm** message box, click **Yes**.
15. In the **Assign Student** window, click **Andrew Harris**.
16. In the **Confirm** message box, click **Yes**.
17. In the **Assign Student** window, click **Toni Poe**.
18. In the **Confirm** message box, click **Yes**.
19. In the **Assign Student** window, click **Ben Andrews**.
20. In the **Confirm** message box, click **Yes**.
21. Verify that the message **Class full: Unable to enroll student: Class: 3C** is displayed, and then click **OK**.
22. In the **Assign Student** window, click **Close**.
23. Click **Log off**.

24. Close the application.
25. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have created a new custom exception class and used it to report when too many students are enrolled in a class.

Module 6: Reading and Writing Local Data

Lab: Generating the Grades Report

Exercise 1: Serializing Data for the Grades Report as XML

▶ Task 1: Prompt the user for a filename and retrieve the grade data

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Window 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window.
5. Click **Visual Studio 2012**.
6. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
7. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
8. In Solution Explorer, expand **GradesPrototype**, expand **Views**, and then double-click **StudentProfile.xaml**.
9. Note that this view displays and enables users to add grades for a student. The solution has been updated to include a **Save Report** button that users will click to generate and save the Grades Report.
10. On the **View** menu, click **Task List**.
11. In the **Task List** window, in the **Categories** list, click **Comments**.
12. Double-click the **TODO: Exercise 1: Task 1a: Store the return value from the SaveFileDialog in a nullable Boolean variable.** task.
13. In the code editor, click in the blank line below the comment, and then type the following code:

```
Nullable<bool> result = dialog.ShowDialog();
if (result.HasValue && result.Value)
{
```

14. Click at the end of the last comment in this method, press Enter, and then type the following code:

```
}
```

15. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1b: Get the grades for the currently selected student.** task.
16. In the code editor, click in the blank line below the comment, and then type the following code:

```
List<Grade> grades = (from g in DataSource.Grades
where g.StudentID == SessionContext.CurrentStudent.StudentID
select g).ToList();
```

17. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1c: Serialize the grades to a MemoryStream.** task.
18. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
MemoryStream ms = FormatAsXMLStream(grades);
```

▶ Task 2: Serialize the grade data to a memory stream

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2a: Save the XML document to a MemoryStream by using an XmlWriter** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
MemoryStream ms = new MemoryStream();
```

3. XmlWriter writer = XmlWriter.Create(ms);
4. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Create the root node of the XML document.** task.
5. In the code editor, click in the blank line below this and the next comment, and then type the following code:

```
writer.WriteStartDocument();
writer.WriteStartElement("Grades");
```

6. writer.WriteAttributeString("Student", String.Format("{0} {1}", SessionContext.CurrentStudent.FirstName, SessionContext.CurrentStudent.LastName));
7. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2c: Format the grades for the student and add them as child elements of the root node** task.
8. In the code editor, click in the blank line below this and the next comment, and then type the following code:

```
foreach (Grade grade in grades)
{
    writer.WriteStartElement("Grade");
    writer.WriteAttributeString("Date", grade.AssessmentDate);
    writer.WriteAttributeString("Subject", grade.SubjectName);
    writer.WriteAttributeString("Assessment", grade.Assessment);
    writer.WriteAttributeString("Comments", grade.Comments);
    writer.WriteEndElement();
}
```

9. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2d: Finish the XML document with the appropriate end elements** task.
10. In the code editor, click in the blank line below the comment, and then type the following code:

```
writer.WriteEndElement();
```

11. writer.WriteEndDocument();
12. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2e: Flush the XmlWriter and close it to ensure that all the data is written to the MemoryStream** task.
13. In the code editor, click in the blank line below the comment, and then type the following code:

```
writer.Flush();
writer.Close();
```

14. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2f: Reset the MemoryStream so it can be read from the start and then return it** task.
15. In the code editor, click in the blank line below the comment, and then type the following code:

```
ms.Seek(0, SeekOrigin.Begin);
return ms;
```

16. Delete the following line of code from the end of the method:

```
throw new NotImplementedException();
```

▶ Task 3: Debug the application

1. On the **Build** menu, click **Build Solution**.
2. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1c: Serialize the grades to a MemoryStream** task.
3. In the code editor, select the closing brace immediately below the following line of code:

```
MemoryStream ms = FormatAsXMLStream(grades);
```

4. On the **Debug** menu, click **Toggle Breakpoint**.
5. On the **Debug** menu, click **Start Debugging**.
6. In the **Username** box, type **vallee**.
7. In the **Password** box, type **password99**, and then click **Log on**.
8. In the main application window, click **Kevin Liu**.
9. In the **Report Card** view, click **Save Report**.
10. In the **Save As** dialog box, click **Save**.

Note: You will write the code to actually save the report to disk in Exercise 3 of this lab.

11. When you enter Break Mode, in the Immediate Window, type the following code, and then press Enter.

```
?(new StreamReader(ms)).ReadToEnd()
```

12. Review the grade data formatted as XML that is returned to the Immediate Window.
13. On the **Debug** menu, click **Stop Debugging**.
14. On the **Debug** menu, click **Delete All Breakpoints**
15. In the confirmation message box, click **Yes**.
16. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be able to specify the location for the Grades Report file.

Exercise 2: Previewing the Grades Report

▶ Task 1: Display the string to the user in a message box

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. On the **View** menu, click **Task List**.
4. In the **Task List** window, in the **Categories** list, click **Comments**.
5. Double-click the **TODO: Exercise 2: Task 1a: Generate a string representation of the report data** task.
6. In the code editor, click in the blank line below the comment, and then type the following code:

```
string formattedReportData = FormatXMLData(ms);
```

7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Preview the string version of the report data in a MessageBox** task.
8. In the code editor, click in the blank line below the comment, and then type the following code:

```
MessageBox.Show(formattedReportData, "Preview Report", MessageBoxButton.OK,
MessageBoxImage.Information);
```

▶ Task 2: Build a string representation of the XML document

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Use a StringBuilder to construct the string** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
StringBuilder builder = new StringBuilder();
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Use an XmlTextReader to read the XML data from the stream** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
XmlTextReader reader = new XmlTextReader(stream);
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Read and process the XML data a node at a time** task.
6. In the code editor, click in the blank line below the comment, and then type the following code:

```
while (reader.Read())
            {
                switch (reader.NodeType)
                {
                    case XmlNodeType.XmlDeclaration:
                        // The node is an XML declaration such as <?xml
version='1.0'>
                        builder.Append(String.Format("<?{0} {1}>\n", reader.Name,
reader.Value));
                        break;
                    case XmlNodeType.Element:
                        // The node is an element (enclosed between '<' and '/>')
                        builder.Append(String.Format("<{0}", reader.Name));
                        if (reader.HasAttributes)
                        {
```

```
                            // Output each of the attributes of the element in the
form "name='value'"
                            while (reader.MoveToNextAttribute())
                            {
                                builder.Append(String.Format(" {0}='{1}'",
reader.Name, reader.Value));
                            }
                        }
                        builder.Append(">\n");
                        break;
                    case XmlNodeType.EndElement:
                        // The node is the closing tag at the end of an element
                        builder.Append(String.Format("</{0}>", reader.Name));
                        break;
                }
            }
```

7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2d: Reset the stream and return the string containing the formatted data** task.
8. In the code editor, click in the blank line below the comment, and then type the following code:

```
stream.Seek(0, SeekOrigin.Begin);
return builder.ToString();
```

9. Delete the following line of code from the end of the method:

```
throw new NotImplementedException();
```

▶ Task 3: Run the application and preview the data.

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password99**, and then click **Log on**.
5. In the main application window, click **Kevin Liu**.
6. In the **Report Card** view, click **Save Report**.
7. In the **Save As** dialog box, click **Save**.

 **Note:** You will write the code to actually save the report to disk in the next exercise of this lab.

8. Review the XML data displayed in the message box, and then click **OK**.
9. Close the application.
10. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be able to preview a report before saving it.

Exercise 3: Persisting the Serialized Grade Data to a File

▶ Task 1: Save the XML document to disk

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod06\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. On the **View** menu, click **Task List**.
4. In the **Task List** window, in the **Categories** list, click **Comments**.
5. Double-click the **TODO: Exercise 3: Task 1a: Modify the message box and ask the user whether they wish to save the report** task.
6. In the code editor, delete the line of code below the comment, and then type the following code:

```
MessageBoxResult reply = MessageBox.Show(formattedReportData, "Save Report?",
MessageBoxButton.YesNo, MessageBoxImage.Question);
```

7. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1b: If the user says yes, then save the data to the file that the user specified earlier** task.
8. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
if (reply == MessageBoxResult.Yes)
{
    // If the user says yes, then save the data to the file that the user specified
earlier
    // If the file already exists it will be overwritten (the SaveFileDialog box will
already have asked the user whether this is OK)
    FileStream file = new FileStream(dialog.FileName, FileMode.Create,
FileAccess.Write);
    ms.CopyTo(file);
    file.Close();
}
```

▶ Task 2: Run the application and verify that the XML document is saved correctly

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password99**, and then click **Log on**.
5. In the main application window, click **Kevin Liu**.
6. In the **Report Card** view, click **Save Report**.
7. In the **Save As** dialog box, browse to the **Documents** folder, and then click **Save**.
8. Review the XML data displayed in the message box, and then click **Yes**.
9. Close the application.
10. Open Internet Explorer.
11. Press the Alt key, and then on the **File** menu, click **Open**.
12. In the **Open** dialog box, click **Browse**.
13. In the **Windows Internet Explorer** dialog box, browse to the **Documents** folder, click **Grades.xml**, and then click **Open**.

14. In the **Open** dialog box, click **OK**.
15. Verify that the file contains the expected grade data, and then close Internet Explorer.
16. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be able to save student reports to the local hard disk in XML format.

Module 7: Accessing a Database

Lab: Retrieving and Modifying Grade Data

Exercise 1: Creating an Entity Data Model from The School of Fine Arts Database

▶ Task 1: Build and generate an EDM by using a table from the SchoolGradesDB database

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod07\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the Open Project dialog box, browse to E:\Mod07\Labfiles\Starter\Exercise 1, click GradesPrototype.sln, and then click Open.
12. On the **File** menu, point to **Add**, and then click **New Project**.
13. In the **Add New Project** dialog box, in the **Installed Templates** list, click **Visual C#**, and then click **Class Library**.
14. In the **Name** box, type **Grades.DataModel**, and then click **OK**.
15. On the **Project** menu, click **Add New Item**.
16. In the **Add New Item – Grades.DataModel** dialog box, in the templates list, click **ADO.NET Entity Data Model**, in the **Name** box, type **GradesModel**, and then click **Add**.
17. In the Entity Data Model Wizard, on the **Choose Model Contents** page, click **Generate from database**, and then click **Next**.
18. On the **Choose Your Data Connection** page, click **New Connection**.
19. If the **Choose Data Source** dialog box appears, in the **Data source** list, click **Microsoft SQL Server**, and then click **Continue**.
20. In the **Connection Properties** dialog box, in the **Server name** box, type **(localdb)\v11.0**, in the **Select or enter a database name** list, click **SchoolGradesDB**, and then click **OK**.
21. In the Entity Data Model Wizard, on the **Choose Your Data Connection** page, click **Next**.
22. On the **Choose Your Database Objects and Settings** page, expand **Tables**, expand **dbo**, select the following tables, and then click **Finish**:
    - **Grades**

- **Students**
- **Subjects**
- **Teachers**
- **Users**

23. If the **Security Warning** dialog box appears, click **Do not show this message again**, and then click **OK**.
24. On the **Build** menu, click **Build Solution**.

▶ Task 2: Review the generated code

1. In the designer window, review the entities that have been generated.
2. Review the properties and navigation properties of the **Grade** entity.
3. Right-click the heading of the **Grade** entity, and then click **Table Mapping**.
4. In the Mapping Details – Grade pane, review the mappings between the columns in the database table and the properties of the entity.
5. In Solution Explorer, expand **GradesModel.edmx**, expand **GradesModel.Context.tt**, and then double-click **GradesModel.Context.cs**.
6. In the code window, note that the wizard has created a **DbContext** object named **SchoolGradesDBEntities**.
7. In Solution Explorer, expand **GradesModel.tt**, and then double-click **Grade.cs**.
8. Note that the wizard has created one property for each column in the **Grades** database table.
9. On the **File** menu, click **Save All**.
10. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the prototype application should include an EDM that you can use to access The School of Fine Arts database.

Exercise 2: Updating Student and Grade Data by Using the Entity Framework

▶ Task 1: Display grades for the current student

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod07\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. In Solution Explorer, right-click GradesPrototype, and then click Set as StartUp Project.
4. On the **View** menu, click **Task List**.
5. In the **Task List** window, in the **Categories** List, click **Comments**.
6. Double-click the **TODO: Exercise 2: Task 1a: Find all the grades for the student** task.
7. In the code editor, click in the blank line below the comment, and then type the following code:

```
List<Grades.DataModel.Grade> grades = new List<Grades.DataModel.Grade>();
foreach (Grades.DataModel.Grade grade in SessionContext.DBContext.Grades)
{
   if (grade.StudentUserId == SessionContext.CurrentStudent.UserId)
   {
      grades.Add(grade);
   }
}
```

8. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Display the grades in the studentGrades ItemsControl by using databinding** task.
9. In the code editor, click in the blank line below the comment, and then type the following code:

```
studentGrades.ItemsSource = grades;
```

10. On the **Build** menu, click **Build Solution**.
11. On the **Debug** menu, click **Start Without Debugging**.
12. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
13. In the **Class 3C** view, click **Kevin Liu**.
14. Verify that Kevin Liu's grades are listed.
15. Note that the subject column uses the subject ID rather than the subject name, and then close the application.

▶ Task 2: Display the subject name in the UI

1. In Visual Studio, in the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Convert the subject ID provided in the value parameter** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
int subjectId = (int)value;
var subject = SessionContext.DBContext.Subjects.FirstOrDefault(s => s.Id ==
subjectId);
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Return the subject name or the string "N/A"** task.
4. In the code editor, delete the following line of code:

```
return value;
```

5. In the code editor, click in the blank line below the comment, and then type the following code:

```
return subject.Name != string.Empty ? subject.Name : "N/A";
```

▶ Task 3: Display the GradeDialog view and use the input to add a new grade

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Use the GradeDialog to get the details of the new grade** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
GradeDialog gd = new GradeDialog();
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3b: Display the form and get the details of the new grade** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
if (gd.ShowDialog().Value)
{
```

5. Click in the blank line below the final TODO comment in this **try** block, and then type the following code:

```
}
```

6. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3c: When the user closes the form, retrieve the details of the assessment grade from the form and use them to create a new Grade object** task.
7. In the code editor, click in the blank line below the comment, and then type the following code:

```
Grades.DataModel.Grade newGrade = new Grades.DataModel.Grade();
newGrade.AssessmentDate = gd.assessmentDate.SelectedDate.Value;
newGrade.SubjectId = gd.subject.SelectedIndex;
newGrade.Assessment = gd.assessmentGrade.Text;
newGrade.Comments = gd.comments.Text;
newGrade.StudentUserId = SessionContext.CurrentStudent.UserId;
```

8. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3d: Save the grade** task.
9. In the code editor, click in the blank line below the comment, and then type the following code:

```
SessionContext.DBContext.Grades.Add(newGrade);
SessionContext.Save();
```

10. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3e: Refresh the display so that the new grade appears** task.
11. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
Refresh();
```

▶ Task 4: Run the application and test the grade-adding functionality

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.

3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. In the **Class 3C** view, click **Kevin Liu**.
5. Verify that the list of grades now displays the subject name, not the subject ID.
6. In the **Report Card** view, click **Add Grade**.
7. In the **New Grade Details** dialog box, in the **Subject** list, click **Geography**, in the **Assessment** box, type **A+**, in the **Comments** box, type **Well done!**, and then click **OK**.
8. Verify that the new grade is added to the list, and then close the application.
9. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, users will be see the grades for the current student and add new grades.

Exercise 3: Extending the Entity Data Model to Validate Data

▶ Task 1: Throw the ClassFullException exception

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod07\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. In Solution Explorer, right-click GradesPrototype, and then click Set as StartUp Project.
4. In Solution Explorer, right-click **Grades.DataModel**, point to **Add**, and then click **Class**.
5. In the **Add New Item – Grades.DataModel** dialog box, in the **Name** box, type **customTeacher.cs**, and then click **Add**.
6. In the code editor, modify the class declaration as shown in the following code:

```
public partial class Teacher
```

7. In the code editor, in the **Teacher** class, type the following code:

```
private const int MAX_CLASS_SIZE = 8;
```

8. In the code editor, in the **Teacher** class, type the following code:

```
public void EnrollInClass(Student student)
 {
    // Verify that this teacher's class is not already full.
    // Determine how many students are currently in the class.
    int numStudents = (from s in Students
        where s.TeacherUserId == UserId
        select s).Count();
    // If the class is already full, another student cannot be enrolled.
    if (numStudents >= MAX_CLASS_SIZE)
    {
        // So throw a ClassFullException and specify the class that is full.
        throw new ClassFullException("Class full: Unable to enroll student", Class);
    }
    // Verify that the student is not already enrolled in another class.
    if (student.TeacherUserId == null)
    {
        // Set the TeacherID property of the student.
        student.TeacherUserId = UserId;
    }
    else
    {
        // If the student is already assigned to a class, throw an ArgumentException.
        throw new ArgumentException("Student", "Student is already assigned to a
class");
    }
}
```

9. In the **Task List** window, double-click **the TODO: Exercise 3: Task 1a: Call the EnrollInClass method to assign the student to this teacher's class** task.
10. In the code editor, click in the blank line below the comment, and then type the following code:

```
SessionContext.CurrentTeacher.EnrollInClass(student);
```

11. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1b: Save the updated student/class information back to the database** task.
12. In the code editor, click in the blank line below the comment, and then type the following code:

```
SessionContext.Save();
```

▶ Task 2: Add validation logic for the Assessment and AssessmentDate properties

1. In Solution Explorer, right-click **Grades.DataModel**, point to **Add**, and then click **Class**.
2. In the **Add New Item – Grades.DataModel** dialog box, in the **Name** box, type **customGrade.cs**, and then click **Add**.
3. In the code editor, modify the class declaration as shown in the following code:

```
public partial class Grade
```

4. In the code editor, in the Grade class, type the following code:

```
public bool ValidateAssessmentDate(DateTime assessmentDate)
{
    // Verify that the user has provided a valid date.
    // Check that the date is no later than the current date.
    if (assessmentDate > DateTime.Now)
    {
        // Throw an ArgumentOutOfRangeException if the date is after the current
date.
        throw new ArgumentOutOfRangeException("Assessment Date", "Assessment date
must be on or before the current date");
    }
    else
    {
        return true;
    }
}
```

5. In the code editor, below the existing **using** directives, type the following code:

```
using System.Text.RegularExpressions;
```

6. In the code editor, in the **Grade** class, type the following code:

```
public bool ValidateAssessmentGrade(string assessment)
{
    // Verify that the grade is in the range A+ to E-.
    // Use a regular expression: A single character in the range A-E at the start of
the string followed by an optional + or - at the end of the string.
    Match matchGrade = Regex.Match(assessment, @"^[A-E][+-]?$");
    if (!matchGrade.Success)
    {
        // If the grade is not valid, throw an ArgumentOutOfRangeException.
        throw new ArgumentOutOfRangeException("Assessment", "Assessment grade must be
in the range A+ to E-");
    }
    else
    {
        return true;
    }
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Create a Grade object** task.
8. In the code editor, click in the blank line below the comment, and then type the following code:

```
Grades.DataModel.Grade testGrade = new Grades.DataModel.Grade();
```

9. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: Call the ValidateAssessmentDate method** task.

10. In the code editor, click in the blank line below the comment, and then type the following code:

```
testGrade.ValidateAssessmentDate(assessmentDate.SelectedDate.Value);
```

11. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2c: Call the ValidateAssessmentGrade** task.
12. In the code editor, click in the blank line below the comment, and then type the following code:

```
testGrade.ValidateAssessmentGrade(assessmentGrade.Text);
```

▶ Task 3: Run the application and test the validation logic

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. When the application loads, click **Enroll Student**.
5. In the **Assign Student** dialog box, click **Eric Gruber**, in the **Confirm** message box, click **Yes**, and then in the **Error enrolling student** message box, click **OK**.
6. In the **Assign Student** dialog box, click **Close**.
7. In the **Class 3C** view, click **Kevin Liu**, and then click **Add Grade**.
8. In the **New Grade Details** dialog box, in the **Date** box, type tomorrow's date, and then click **OK**.
9. In the **Error creating assessment** message box, click **OK**.
10. In the **New Grade Details** dialog box, in the **Date** box, type **8/19/2012**, in the **Assessment** box, type **F+**, and then click **OK**.
11. In the **Error creating assessment** message box, click **OK**.
12. In the **New Grade Details** dialog box, in the **Assessment** box, type **A+**, in the **Comments** box, type **Well done!**, and then click **OK**.
13. Verify that the new grade is added to the list, and then close the application.
14. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, the application will raise and handle exceptions when invalid data is entered.

Module 8: Accessing Remote Data

Lab: Retrieving and Modifying Grade Data in the Cloud

Exercise 1: Creating a WCF Data Service for the SchoolGrades Database

▶ Task 1: Create the Grades.Web project

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 Start window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. In File Explorer, navigate to the **E:\Mod08\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Microsoft® Visual Studio®, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 1**, click **GradesPrototype.sln**, and then click **Open**.
12. In Solution Explorer, right-click the **GradesPrototype** solution, point to **Add**, and then click **New Project**.
13. In the **Add New Project** dialog box, in the left pane expand **Visual C#**, and then click **Web**.
14. In the templates list, click **ASP.NET Empty Web Application**.
15. In the **Name** box, type **Grades.Web**, and then click **OK**.
16. In Solution Explorer, right-click the **Grades.Web** project, and then click **Properties**.
17. On the **Web** tab, in the **Start Action** section, click **Don't open a page. Wait for a request from an external application**.
18. In the **Servers** section, ensure that **Use Local IIS Web server** is selected.
19. In the **Project Url** box, type **http://localhost:1650/**, and then click **Create Virtual Directory**.
20. In the **Microsoft Visual Studio** dialog box, click **OK**.
21. In Solution Explorer, right-click the **GradesPrototype** solution, and then click **Set StartUp Projects**.
22. In the **Solution 'GradesPrototype' Property Pages** dialog box, click **Multiple startup projects**.
23. In the **Action** column for **Grades.Web** and **GradesPrototype**, click **Start**, and then click **OK**.
24. On the **File** menu, click **Save All**.

▶ Task 2: Add a data service to the Grades.Web project

1. In Solution Explorer, right-click the **Grades.Web** project, point to **Add**, and then click **New Folder**.

2. Delete the existing folder name, type **Services**, and then press Enter.
3. Right-click **Services**, point to **Add**, and then click **New Item**.
4. In the templates list, click **WCF Data Service**.
5. In the **Name** box, type **GradesWebDataService**, and then click **Add**.
6. Right-click **Grades.Web**, and then click **Add Reference**.
7. In the **Reference Manager – Grades.Web** dialog box, expand **Solution**, and then select **Grades.DataModel**.
8. Click **Browse**.
9. In the **Select the files to reference** dialog box, browse to the **E:\Mod08\Labfiles\Starter\Exercise 1\packages\EntityFramework.5.0.0\lib\net45** folder, click **EntityFramework.dll**, and then click **Add**.
10. In the **Reference Manager – Grades.Web** dialog box, click **OK**.
11. In Solution Explorer, expand the **GradesPrototype** project, and then double-click **App.config**.
12. In the code editor, copy the following XML to the clipboard:

```
<connectionStrings>
    <add name="GradesDBEntities"
connectionString="metadata=res://*/GradesModel.csdl|res://*/GradesModel.ssdl|res://*/
GradesModel.msl;provider=System.Data.SqlClient;provider connection string=&quot;data
source=(localdb)\v11.0;initial catalog=SchoolGradesDB;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework&quot;"
providerName="System.Data.EntityClient" />
</connectionStrings>
```

13. In Solution Explorer, in the **Grades.Web** project, double-click **Web.config**.
14. Click at the end of the opening **<configuration>** element, press Enter, and then paste the contents of the clipboard.

▶ Task 3: Specify the GradesDBEntities data context for the data service

1. In Solution Explorer, in Grades.Web, expand **Services**, and then double-click **GradesWebDataService.svc**.
2. In the code editor, click at the end of the **using System.Web;** code, press Enter, and then type the following code:

```
using Grades.DataModel;
```

3. On the **View** menu, click **Task List**.
4. In the **Task List** window, in the **Categories** list, click **Comments**.
5. Double-click the **TODO: put your data source class name here** task.
6. In the code editor, delete the code **/* TODO: put your data source class name here */**, and then type the following code:

```
SchoolGradesDBEntities
```

7. In the **Task List** window, double-click the **TODO: set rules to indicate which entity sets and service operations are visible, updatable, etc.** task.
8. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Configure all entity sets to permit read and write access.
config.SetEntitySetAccessRule("Grades", EntitySetRights.All);
config.SetEntitySetAccessRule("Teachers", EntitySetRights.All);
config.SetEntitySetAccessRule("Students", EntitySetRights.All);
config.SetEntitySetAccessRule("Subjects", EntitySetRights.All);
config.SetEntitySetAccessRule("Users", EntitySetRights.All);
```

▶ Task 4: Add an operation to retrieve all of the students in a specified class

1. Click after the closing brace for the **InitializeService** method, press Enter twice, and then type the following code:

```
// Find all students in a specified class.
[WebGet]
public IEnumerable<Student> StudentsInClass(string className)
{
    var students = from Student s in this.CurrentDataSource.Students
                   where String.Equals(s.Teacher.Class, className)
                   select s;
    return students;
}
```

2. In the **Task List** window, double-click the **TODO: set rules to indicate which entity sets and service operations are visible, updatable, etc.** task.
3. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
// Configure the StudentsInClass operation as read-only.
config.SetServiceOperationAccessRule("StudentsInClass",
ServiceOperationRights.AllRead);
```

▶ Task 5: Build and test the data service

1. On the **Build** menu, click **Build Solution**.
2. In Solution Explorer, in the **Grades.Web** project, in the **Services** folder, right-click **GradesWebDataService.svc**, and then click **View in Browser (Internet Explorer)**.
3. In Internet Explorer, if the message **Intranet settings are turned off by default**, click **Don't show this message again**.
4. Verify that Internet Explorer® displays an XML description of the entities that the data service exposes.
5. Close Internet Explorer.
6. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have added a WCF Data Service to the application to provide remote access to the **SchoolGrades** database.

Exercise 2: Integrating the Data Service into the Application

▶ Task 1: Add a service reference for the WCF Data Service to the GradesPrototype application

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 2**, click **GradesPrototype.sln**, and then click **Open**.
3. In Solution Explorer, right-click the **GradesPrototype** solution, and then click **Set StartUp Projects**.
4. In the **Solution 'GradesPrototype' Property Pages** dialog box, click **Multiple startup projects**.
5. In the **Action** column for **Grades.Web** and **GradesPrototype**, click **Start**, and then click **OK**.
6. On the **Build** menu, click **Rebuild Solution**.
7. In Solution Explorer, expand **GradesPrototype**, expand **References**, right-click **Grades.DataModel,** and then click **Remove**.
8. Right-click **References**, and then click **Add Service Reference**.
9. In the **Add Service Reference** dialog box, in the **Address** box, type **http://localhost:1650**, and then click **Discover**.
10. In the **Namespace** box, type **Grades.DataModel**, and then click **OK**.
11. In the Solution Explorer toolbar, click **Show All Files**.
12. In Solution Explorer, in the **GradesPrototype** project, in the **Service References** folder, expand **Grades.DataModel**, expand **Reference.datasvcmap**, and then double-click **Reference.cs**.
13. In the code editor, modify the **namespace GradesPrototype.Grades.DataModel** code to look like the following code:

```
namespace Grades.DataModel
```

14. In Solution Explorer, right-click the **GradesPrototype** project, point to **Add**, and then click **New Folder**.
15. Delete the existing name, type **DataModel**, and then press Enter.
16. In Solution Explorer, expand the **Grades.DataModel** project, right-click **Classes.cs**, and then click **Copy**.
17. In GradesPrototype, right-click **DataModel**, and then click **Paste**.
18. In Grades.DataModel, right-click **customGrade.cs**, and then click **Copy**.
19. In GradesPrototype, right-click **DataModel**, and then click **Paste**.
20. In Grades.DataModel, right-click **customTeacher.cs**, and then click **Copy**.
21. In GradesPrototype, right-click **DataModel**, and then click **Paste**.

▶ Task 2: Modify the code that accesses the EDM to use the WCF Data Service

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Specify the URL of the GradesWebDataService** task.
2. In the code editor, below the comment, click inside the parentheses, and then type the following code:

```
new Uri("http://localhost:1650/Services/GradesWebDataService.svc")
```

3. Add the following code to the **SessionContext** class, after the **Save** method:

```
static SessionContext()
{
    DBContext.MergeOption = System.Data.Services.Client.MergeOption.PreserveChanges;
}
```

4. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Load User and Grades data with Students** task.
5. In the code editor, at the end of the comment, press Enter, and then type the following code:

```
SessionContext.DBContext.LoadProperty(student, "User");
SessionContext.DBContext.LoadProperty(student, "Grades");
```

6. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Load User and Students data with Teachers** task.
7. In the code editor, below the comment, click at the end of the **SessionContext.DBContext.Teachers** code, and then type the following code:

```
.Expand("User, Students")
```

8. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2d: Load User and Grades data with Students** task.
9. In the code editor, below the comment, click at the end of the **SessionContext.DBContext.Students** code, and then type the following code:

```
.Expand("User, Grades")
```

10. In Solution Explorer, in the **GradesPrototype** project, expand **DataModel**, and then double-click **customTeacher.cs**.
11. Click at the end of the **using System.Threading.Tasks;** code, press Enter, and then type the following code:

```
using GradesPrototype.Services;
```

12. In the code editor, locate the **TODO: Exercise 2: Task 2e: Refer to the Students collection in the SessionContext.DBContext object** comment in the customTeacher.cs file. There are two comments with this text. This is because the comment is located in the customTeacher.cs file that you copied from the Grades.DataModel project. Make sure that you modify the customTeacher.cs file in the GradesPrototype project.
13. In the line below the comment, delete the word **Students**, and then type the following code:
14. SessionContext.DBContext.Students
15. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2f: Reference the SessionContext.DBContext.Students collection**.
16. In the code editor, below the comment, change **SessionContext.DBContext.Students.Local** to the following code:

```
SessionContext.DBContext.Students.Expand("User, Grades")
```

17. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2g: Use the AddToGrades method to add a new grade**.

18. In the code editor, below the comment, change **SessionContext.DBContext.Grades.Add(newGrade);** to the following code:

```
SessionContext.DBContext.AddToGrades(newGrade);
```

19. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2h: Load Subject data with Grades**.
20. In the code editor, below the comment, change **SessionContext.DBContext.Grades** to the following code:

```
SessionContext.DBContext.Grades.Expand("Subject")
```

21. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2i: Use the AddToStudents method to add a new student**.
22. In the code editor, below the comment, change **SessionContext.DBContext.Students.Add(newStudent);** to the following code:

```
SessionContext.DBContext.AddToStudents(newStudent);
```

▶ Task 3: Modify the code that saves changes back to the database to use the WCF Data Service

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Specify that the selected student has been changed** task.
2. In the code editor, click in the blank space below the comment, and then type the following code:
3. SessionContext.DBContext.UpdateObject(student);
4. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3b: Specify that the current student has been changed** task.
5. In the code editor, click in the blank space below the comment, and then type the following code:

```
SessionContext.DBContext.UpdateObject(SessionContext.CurrentStudent);
```

6. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3c: Specify that the current user has been changed** task.
7. Click in the blank space below the comment, and then type the following code:

```
SessionContext.DBContext.UpdateObject(currentUser);
```

▶ Task 4: Build and test the application to verify that the application still functions correctly

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password99**, and then click **Log on**.
5. In the students list, click **Eric Gruber**, and then click **Remove Student**.
6. In the **Confirm** dialog box, click **Yes**.
7. Verify that **Eric Gruber** is removed from the student list.

8. Click **Enroll Student**.
9. In the **Assign Student** dialog box, click **Jon Orton**.
10. In the **Confirm** dialog box, click **Yes**.
11. In the **Assign Student** dialog box, click **Close**, and then verify that **Jon Orton** is added to the student list.
12. Click **Change Password**.
13. In the **Change Password Dialog** dialog box, in the **Old Password** box, type **password99**.
14. In the **New Password** box, type **password88**.
15. In the **Confirm** box, type **password88**, and then click **OK**.
16. In the **Password** dialog box, click **OK**, and then click **Log off**.
17. Click **Log on**, verify that the **Logon Failed** dialog box appears, and then click **OK**.
18. In the **Password** box, type **password88**, and then click **Log on**.
19. Verify that the student list is displayed.
20. Click **Log off**, and then in the **Username** box, type **grubere**.
21. In the **Password** box, type **password**, and then click **Log on**.
22. Verify that the student profile for Eric Gruber appears, and then click **Log off**.
23. Close the application.
24. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have updated the Grades Prototype application to use the WCF Data Service.

Exercise 3: Retrieving Student Photographs Over the Web (If Time Permits)

▶ Task 1: Create the ImageNameConverter value converter class

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod08\Labfiles\Starter\Exercise 3**, click **GradesPrototype.sln**, and then click **Open**.
3. In Solution Explorer, right-click the **GradesPrototype** solution, and then click **Set StartUp Projects**.
4. In the **Solution 'GradesPrototype' Property Pages** dialog box, click **Multiple startup projects**.
5. In the **Action** column for **Grades.Web** and **GradesPrototype**, click **Start**, and then click **OK**.
6. On the **Build** menu, click **Rebuild Solution**.
7. In the **Task List** window, double-click the **TODO: Exercise 3: Task 1: Create the ImageNameConverter value converter to convert the image name of a student photograph into the URL of the image on the Web server** task.
8. Click at the end of the **// Converter class for transforming an image name for a photograph into a URL** comment, press Enter, and then type the following code:

```
public class ImageNameConverter : IValueConverter
{
}
```

9. In the **ImageNameConverter** class, type the following code:

```
const string webFolder = "http://localhost:1650/Images/Portraits/";
```

10. Right-click the **IValueConverter** keyword, point to **Implement Interface**, and then click **Implement Interface**.
11. In the **Convert** method, delete the existing statement that throws a **NotImplementedException**, and then type the following code:

```
string fileName = value as string;
if (fileName != null)
{
    return string.Format("{0}{1}", webFolder, fileName);
}
else
{
    return string.Empty;
}
```

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

▶ Task 2: Add an Image control to the StudentsPage view and bind it to the ImageName property

1. In Solution Explorer, in the **GradesPrototype** project, expand **Views**, and then double-click **StudentsPage.xaml**.
2. In the XAML editor, in the **UserControl** element at the top of the markup, click after the **xmlns:d="http://schemas.microsoft.com/expression/blend/2008"** line, press Enter, and then type the following markup:

```
xmlns:local="clr-namespace:GradesPrototype.Views"
```

3. Locate the **TODO: Exercise 3: Task 2a. Add an instance of the ImageNameConverter class as a resource to the view** comment, click at the end of the comment, press Enter, and then type the following markup:

```
<UserControl.Resources>
    <local:ImageNameConverter x:Key="ImageNameConverter"/>
</UserControl.Resources>
```

4. Locate the **TODO: Exercise 3: Task 2b. Add an Image control to display the photo of the student** comment, click at the end of the comment, press Enter, and then type the following markup:

```
<Image Height="100" />
```

5. Locate the **Exercise 3: Task 2c. Bind the Image control to the ImageName property and use the ImageNameConverter to convert the image name into a URL** comment.

6. In the line above the comment, modify the **<Image Height="100" />** markup to look like the following markup:

```
<Image Height="100" Source="{Binding ImageName, Converter={StaticResource
ImageNameConverter}}" />
```

▶ Task 3: Add an Image control to the StudentProfile view and bind it to the ImageName property

1. In Solution Explorer, double-click **StudentProfile.xaml**.

2. Locate the **TODO: Exercise 3: Task 3a. Add an instance of the ImageNameConverter class as a resource to the view** comment.

3. Click at the end of the comment, press Enter, and then type the following markup:

```
<app:ImageNameConverter x:Key="ImageNameConverter"/>
```

4. Locate the **TODO: Exercise 3: Task 3b. Add an Image control to display the photo of the student and bind the Image control to the ImageName property and use the ImageNameConverter to convert the image name into a URL** comment.

5. Click at the end of the comment, press Enter, and then type the following markup:

```
<Image Height="150" Source="{Binding ImageName, Converter={StaticResource
ImageNameConverter}}" />
```

▶ Task 4: Add an Image control to the AssignStudentDialog control and bind it to the ImageName property

1. In Solution Explorer, expand **Controls**, and then double-click **AssignStudentDialog.xaml**.

2. In the XAML editor, at the top of the markup, click at the end of the **xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"** line, press Enter, and then type the following markup:

```
xmlns:local="clr-namespace:GradesPrototype.Views"
```

3. Locate the **TODO: Exercise 3: Task 4a. Add an instance of the ImageNameConverter class as a resource to the view** comment.

4. Click at the end of the comment, press Enter, and then type the following markup:

```
<Window.Resources>
```

```
    <local:ImageNameConverter x:Key="ImageNameConverter"/>
    </Window.Resources>
```

5. Locate the **TODO: Exercise 3: Task 4b. Add an Image control to display the photo of the student and bind the control to the ImageName property and use the ImageNameConverter to convert the image name into a URL** comment.
6. Click at the end of the comment, press Enter, and then type the following markup:

```
<Image Height="100" Source="{Binding ImageName, Converter={StaticResource
ImageNameConverter}}" />
```

▶ Task 5: Build and test the application, verifying that student's photographs appear in the list of students for the teacher

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. In the **Username** box, type **vallee**.
4. In the **Password** box, type **password88**, and then click **Log on**.
5. Verify that the students list now includes images.
6. Click **George Li**, and then verify that the student profile appears with an image.
7. Click **Remove Student**.
8. In the **Confirm** dialog box, click **Yes**.
9. Click **Enroll Student**.
10. In the **Assign Student** dialog box, the unassigned students each have an image.
11. Click **George Li**.
12. In the **Confirm** dialog box, click **Yes**.
13. In the **Assign Student** dialog box, click **Close**.
14. Verify that George Li is added to the students list with an image.
15. Close the application.
16. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the students list, student profile, and unassigned student dialog box will display the images of students that were retrieved across the web.

Module 9: Designing the User Interface for a Graphical Application

Lab: Customizing Student Photographs and Styling the Application

Exercise 1: Customizing the Appearance of Student Photographs

▶ Task 1: Create the StudentPhoto user control

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 Start window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod09\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod09\Labfiles\Starter\Exercise 1**, click **Grades.sln**, and then click **Open**.
12. In Solution Explorer, right-click **Solution 'Grades'**, and then click **Properties**.
13. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
14. In Solution Explorer, expand **Grades.WPF**, and then expand **Controls**.
15. Right-click **Controls**, point to **Add**, and then click **New Item**.
16. In the **Add New Item – Grades.WPF** dialog box, in the template list, click **User Control (WPF)**.
17. In the **Name** box, type **StudentPhoto**, and then click **Add**.
18. In the XAML editor, modify the markup to look like the following markup (the changes are highlighted as bold text):

```
<UserControl x:Class="Grades.WPF.StudentPhoto"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
     <Image Stretch="UniformToFill" Source="{Binding File}" Margin="8" />
     <Image Margin="0" Source="../Images/Image_Frame.png" Stretch="Fill" />
     <TextBlock Text="{Binding Name}" Style="{StaticResource LabelCenter}"
FontSize="16" VerticalAlignment="Bottom" Margin="8,0,14.583,8" />
```

```
    </Grid>
</UserControl>
```

19. In Solution Explorer, expand **StudentPhoto.xaml**, and then double-click **StudentPhoto.xaml.cs**.
20. In the code editor, delete all of the **using** directives, and then type the following code:

```
using System.Windows.Controls;
using System.Windows.Media.Animation;
```

21. Modify the **namespace Grades.WPF.Controls** code to look like the following code:

```
namespace Grades.WPF
```

▶ Task 2: Display the students' photographs in the StudentsPage view

1. In Solution Explorer, expand **Views**, and then double-click **StudentsPage.xaml**.
2. Locate the **<!-- TODO: Exercise 1: Task 2a: Define the DataTemplate for the "list" ItemsControl including the StudentPhoto user control -->** comment, click at the end of the comment, press Enter, and then type the following markup:

```
<ItemsControl.ItemTemplate>
    <DataTemplate>
        <Grid Margin="8">
             <local:StudentPhoto Height="150" Width="127.5" Cursor="Hand"
```

Note: When you type an opening tag for an element, such as **<ItemsControl.ItemTemplate>**, the XAML editor automatically creates a corresponding closing tag, such as **</ItemsControl.ItemTemplate>**. For the purposes of these instructions, and to ensure that code appears in the correctly commented place, delete any closing tags that are generated automatically; you will add them in at the appropriate point in the XAML markup in later steps.

▶ Task 3: Enable the user to display the details for a student

1. In **StudentsPage.xaml**, locate the **<!-- TODO: Exercise 1: Task 3a: Set the handler for the click event for the StudentPhoto control -->** comment, and above the comment, click at the end of the **Cursor="Hand"** markup.
2. Press Spacebar, and then type the following markup:

```
MouseLeftButtonUp="Student_Click" />
```

3. In the **Task List** window, in the **Categories** list, click **Comments**.
4. Double-click the **TODO: Exercise 1: Task 3b: Review the following event handler.** task.
5. Review the **Student_Click** method which raises the **StudentSelected** event to display the details of the student when a user clicks their photo.

▶ Task 4: Add a Remove button to the StudentsPage view

1. In StudentsPage.xaml, Locate the **<!-- TODO: Exercise 1: Task 4a: Add the "Remove" button to the DataTemplate -->** comment, click at the end of the comment, press Enter, and then type the following markup:

```
<Grid VerticalAlignment="Top" HorizontalAlignment="Right" Background="#00000000"
Opacity="0.3" Width="20" Height="20" ToolTipService.ToolTip="Remove from class"
Tag="{Binding}" >
    <Image Source="../Images/delete.png" Stretch="Uniform" />
</Grid>
```

2. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4b: Review the following event handler** task.
3. The code in this method increases the opacity of the grid containing the remove button and reduces the opacity of the grid containing the photo when the user moves the mouse over the delete image
4. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4c: Review the following event handler** task.
5. The code in this method reduces the opacity of the grid containing the remove button and increases the opacity of the grid containing the photo when the user moves the mouse away from the delete image.
6. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4d: Review the following event handler** task.
7. The code in this method removes a student from the current teacher's class when a user clicks the remove icon.
8. In StudentsPage.xaml, locate the **<!-- TODO: Exercise 1: Task 4d: Add event handlers to highlight the "Remove" button as the mouse enters and exits this control -->** comment, and above the comment, click at the end of the **Tag="{Binding}"** code (before the closing **>** tag), press Enter, and then type the following markup:

```
MouseEnter="RemoveStudent_MouseEnter" MouseLeave="RemoveStudent_MouseLeave"
MouseLeftButtonUp="RemoveStudent_Click"
```

9. Click at the end of the **<Image Source="../Images/delete.png" Stretch="Uniform" /></Grid>** markup, press Enter, and then type the following code:

```
        </Grid>
    </DataTemplate>
</ItemsControl.ItemTemplate>
```

▶ Task 5: Display all students for the current teacher

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 5a: Bind the list of students to the "list" ItemsControl** task.
2. In this method, review the code which finds all students for the current teacher and constructs a list of students.
3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 5a: Bind the list of students to the "list" ItemsControl** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
list.ItemsSource = resultData;
```

▶ Task 6: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.

3. When the application starts, in the **Username** box, type **vallee**, in the **Password** box, type **password99**, and then click **Log on**.
4. Verify that the students list appears with photographs.
5. In the student list, hover over the **red x** for the student Martin Weber.
6. Verify that the student photograph for Martin Weber becomes transparent and that the red x becomes opaque.
7. Move the cursor away from the red x and verify that the student photograph becomes opaque and that the red x becomes transparent.
8. Click the **red x** for Martin Weber, verify that the **Student** message box appears, and then click **Yes**.
9. Verify that Martin Weber is removed from the student list.
10. Close the application, and then close Visual Studio.

Results: After completing this exercise, the application will display the photographs of each student on the Student List page.

Exercise 2: Styling the Logon View

▶ Task 1: Define and apply styles for the LogonPage view

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod09\Labfiles\Starter\Exercise 2**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. In Solution Explorer, expand **Grades.WPF**, expand **Views**, and then double-click **LogonPage.xaml**.
6. In the XAML editor, locate the **<!-- TODO: Exercise 2: Task 1a: Define the LoginTextBoxStyle -->** comment.
7. Click at the end of the comment, press Enter, and then type the following markup:

```
<UserControl.Resources>
    <Style x:Key="LoginTextBoxStyle" BasedOn="{StaticResource TextBoxStyle}"
TargetType="{x:Type TextBox}">
        <Setter Property="Margin" Value="5" />
        <Setter Property="FontSize" Value="24"/>
        <Setter Property="MaxLength" Value="16" />
    </Style>
```

8. Locate the **<!-- TODO: Exercise 2: Task 1b: Apply the LoginTextBoxStyle to the "username" TextBox -->** comment.
9. In the line of markup below the comment, delete the **FontSize** property, and then modify the markup as shown in bold below:

```
<TextBox x:Name="username" Grid.Row="1" Grid.Column="1" Style="{StaticResource
LoginTextBoxStyle}" />
```

10. Locate the **<!-- TODO: Exercise 2: Task 1c: Define the PasswordBoxStyle -->** comment.
11. Click at the end of the comment, press Enter, and then type the following markup:

```
    <Style x:Key="PasswordBoxStyle" TargetType="{x:Type PasswordBox}">
        <Setter Property="Margin" Value="5" />
        <Setter Property="FontSize" Value="24"/>
        <Setter Property="MaxLength" Value="16" />
    </Style>
</UserControl.Resources>
```

12. Locate the **TODO: Exercise 2: Task 1d: Apply the PasswordBoxStyle to the "password" TextBox** comment.
13. In the line of markup below the comment, delete the **FontSize** property, and then modify the markup as shown in bold below:

```
<PasswordBox x:Name="password" Grid.Row="2" Grid.Column="1" Style="{StaticResource
PasswordBoxStyle}" />
```

▶ Task 2: Define global styles for the application

1. In Solution Explorer, expand **Themes**, and then double-click **Generic.xaml**.
2. In the XAML editor, locate the **<!-- TODO: Exercise 2: Task 2a: Define the label styling used throughout the application -->** comment near the end of the file.

3. Click in the blank line below the comment, and type the following markup:

```
<Setter Property="TextWrapping" Value="NoWrap"/>
<Setter Property="FontFamily" Value="Buxton Sketch"/>
<Setter Property="FontSize" Value="19"/>
<Setter Property="Foreground" Value="#FF303030" />
```

4. Locate the **<!-- TODO: Exercise 2: Task 2b: Define the text styling used throughout the application -->** comment.

5. Click in the blank line below the comment, then type the following markup:

```
<Setter Property="TextWrapping" Value="NoWrap"/>
<Setter Property="FontFamily" Value="Buxton Sketch"/>
<Setter Property="FontSize" Value="12"/>
<Setter Property="TextAlignment" Value="Left" />
<Setter Property="Foreground" Value="#FF303030" />
```

▶ Task 3: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging.**
3. When the application starts, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. In **The School of Fine Arts** window, verify that the styling of the text elements of the application has changed.

Comparison of the Logon views

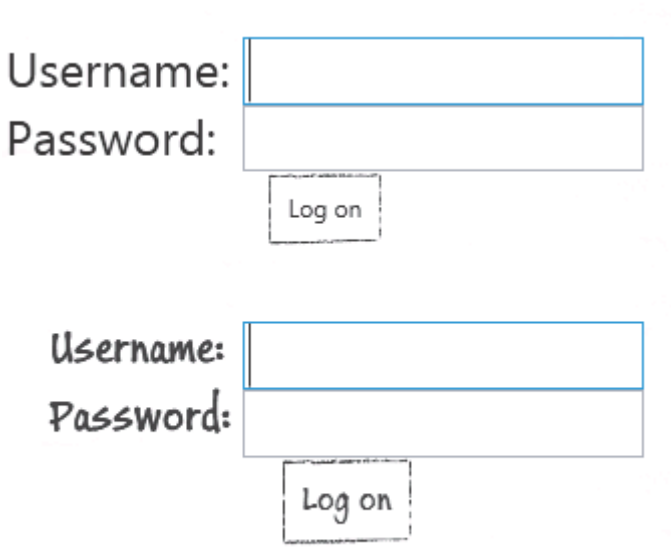

FIGURE 9.1:UPPER: OLD STYLE LOGON VIEW. LOWER: NEW STYLE LOGON VIEW

5. Close the application.
6. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the Logon view will be styled with a consistent look and feel.

Exercise 3: Animating the StudentPhoto Control (If Time Permits)

▶ Task 1: Define animations for the StudentPhoto control

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod09\Labfiles\Starter\Exercise 3**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. In Solution Explorer, in the **Grades.WPF** project, expand **Controls**, and then double-click **StudentPhoto.xaml**.
6. In the XAML editor, locate the **<!-- TODO: Exercise 3: Task 1a: Define a ScaleTransform called "scale" -->** comment.
7. Click in the blank line below the comment, then type the following markup:

```
<UserControl.RenderTransform>
    <ScaleTransform x:Name="scale" />
</UserControl.RenderTransform>
```

8. In the XAML editor, locate the **<!-- TODO: Exercise 3: Task 1b: Define animations for the "scale" transform-->** comment.
9. Click in the blank line below the comment, then type the following markup.

```
<UserControl.Resources>
    <Storyboard x:Key="sbMouseEnter">
        <DoubleAnimation To="1.1" BeginTime="00:00:00" Duration="00:00:00.05"
Storyboard.TargetName="scale" Storyboard.TargetProperty="ScaleX" />
        <DoubleAnimation To="1.1" BeginTime="00:00:00" Duration="00:00:00.15"
Storyboard.TargetName="scale" Storyboard.TargetProperty="ScaleY" />
    </Storyboard>
    <Storyboard x:Key="sbMouseLeave">
        <DoubleAnimation To="1" BeginTime="00:00:00" Duration="00:00:00.05"
Storyboard.TargetName="scale" Storyboard.TargetProperty="ScaleX" />
        <DoubleAnimation To="1" BeginTime="00:00:00" Duration="00:00:00.15"
Storyboard.TargetName="scale" Storyboard.TargetProperty="ScaleY" />
    </Storyboard>
</UserControl.Resources>
```

▶ Task 2: Add event handlers to trigger the animations

1. In Solution Explorer, expand **StudentPhoto.xaml**, and then double-click **StudentPhoto.xaml.cs**.
2. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2a: Handle mouse events to trigger the storyboards that animate the photograph** task.
3. In the code editor, click in the blank line below the comment, and then type the following code:

```
public void OnMouseEnter()
{
    // Trigger the mouse enter animation to grow the size of the photograph currently
under the mouse pointer
    (this.Resources["sbMouseEnter"] as Storyboard).Begin();
}
public void OnMouseLeave()
{
    // Trigger the mouse leave animation to shrink the size of the photograph
currently under the mouse pointer to return it to its original size
```

```
    (this.Resources["sbMouseLeave"] as Storyboard).Begin();
}
```

4. In Solution Explorer, in **Views**, expand **StudentsPage.xaml**, and then double-click **StudentsPage.xaml.cs**.
5. In the **Task List** window, double-click the **TODO: Exercise 3: Task 2b: Forward the MouseEnter and MouseLeave events to the photograph control** task.
6. In the code editor, click in the blank space below the comment, and then type the following code:

```
private void Student_MouseEnter(object sender, MouseEventArgs e)
{
    // Call the OnMouseEnter event handler on the specific photograph currently under
the mouse pointer
    ((StudentPhoto)sender).OnMouseEnter();
}
private void Student_MouseLeave(object sender, MouseEventArgs e)
{
    // Call the OnMouseLeave event handler on the specific photograph currently under
the mouse pointer
    ((StudentPhoto)sender).OnMouseLeave();
}
```

7. In Solution Explorer, double-click **StudentsPage.xaml**.
8. In the XAML editor, locate the **<!-- TODO: Exercise 3: Task 2c: Specify the handlers for the MouseEnter and MouseLeave events -->** comment.
9. Below the comment, click at the end of the **MouseLeftButtonUp="Student_Click"** markup, press Spacebar, and then type the following markup:

```
MouseEnter="Student_MouseEnter" MouseLeave="Student_MouseLeave"
```

▶ Task 3: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging.**
3. When the application starts, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. Hover over one of the students in the student list and verify that the photograph animates—it should expand and contract as the mouse passes over it.
5. Close the application.
6. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, the Photograph control will be animated.

Module 10: Improving Application Performance and Responsiveness

Lab: Improving the Responsiveness and Performance of the Application

Exercise 1: Ensuring That the UI Remains Responsive When Retrieving Teacher Data

▶ Task 1: Build and run the application

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows® 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod10\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Microsoft® Visual Studio®, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod10\Labfiles\Starter\Exercise 1**, click **Grades.sln**, and then click **Open**.
12. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
13. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
14. On the **Build** menu, click **Build Solution**.
15. On the **Debug** menu, click **Start Without Debugging**.
16. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
17. Notice that the UI briefly freezes while fetching the list of students for Esther Valle (try moving the application window after logging on but before the list of students appears).
18. Close the application window.

▶ Task 2: Modify the code that retrieves teacher data to run asynchronously

1. On the **View** menu, click **Task List**.
2. In the **Task List** window, in the **Categories** list, select **Comments**.
3. Double-click the **TODO: Exercise 1: Task 2a: Convert GetTeacher into an async method that returns a Task<Teacher>** task.
4. In the code editor, delete the following line of code:

```
public Teacher GetTeacher(string userName)
```

5. In the blank line below the comment, type the following code:

```
public async Task<Teacher> GetTeacher(string userName)
```

6. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Perform the LINQ query to fetch Teacher information asynchronously** task.

7. In the code editor, modify the statement below the comment as shown in bold below:

```
var teacher = await Task.Run(() =>
(from t in DBContext.Teachers
where t.User.UserName == userName
select t).FirstOrDefault());
```

8. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2c: Mark MainWindow.Refresh as an asynchronous method** task.

9. In the code editor, modify the statement below the comment as shown in bold below:

```
public async void Refresh()
```

10. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2d: Call GetTeacher asynchronously** task.

11. In the code editor, modify the statement below the comment as shown in bold below:

```
var teacher = await utils.GetTeacher(SessionContext.UserName);
```

▶ Task 3: Modify the code that retrieves and displays the list of students for a teacher to run asynchronously

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Mark StudentsPage.Refresh as an asynchronous method** task.

2. In the code editor, modify the statement below the comment as shown in bold below:

```
public async void Refresh()
```

3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Implement the OnGetStudentsByTeacherComplete callback to display the students for a teacher here** task.

4. In the blank line below the comment, type the following code:

```
private void OnGetStudentsByTeacherComplete(IEnumerable<Student> students)
{
}
```

5. In the **Task List** window, double-click the **Exercise 1: Task 3b: Relocate the remaining code in this method to create the OnGetStudentsByTeacherComplete callback (in the Callbacks region)** task.

6. In the code editor, move all of the code between the comment and the end of the **Refresh** method to the Clipboard.

7. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Implement the OnGetStudentsByTeacherComplete callback to display the students for a teacher here** task.

8. Click in the blank line between the curly braces and paste the code from the Clipboard.

9. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3c: Use a Dispatcher object to update the UI** task.
10. In the code editor, click at the end of the comment line, press Enter, and then type the following code:

```
this.Dispatcher.Invoke(() => {
```

11. Immediately after the last line of code in the method, type the following code:

```
});
```

12. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3d: Convert GetStudentsByTeacher into an async method that invokes a callback** task.
13. In the code editor, delete the following line of code:

```
public List<Student> GetStudentsByTeacher(string teacherName)
```

14. In the blank line below the comment, type the following code:

```
public async Task GetStudentsByTeacher(string teacherName,
Action<IEnumerable<Student>> callback)
```

15. In the code editor, modify the return statement below the **if(!IsConnected())** line to return without passing a value to the caller:

```
   return;
```

16. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3e: Perform the LINQ query to fetch Student data asynchronously** task.
17. In the code editor, modify the statement below the comment as shown in bold below:

```
var students = await Task.Run(() =>
(from s in DBContext.Students
where s.Teacher.User.UserName == teacherName
select s).OrderBy(s => s.LastName).ToList());
```

18. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3f: Run the callback by using a new task rather than returning a list of students** task.
19. In the code editor, delete the following code:

```
return students;
```

20. In the blank line below the comment, type the following code:

```
await Task.Run(() => callback(students));
```

21. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3g: Invoke GetStudentsByTeacher asynchronously and pass the OnGetStudentsByTeacherComplete callback as the second argument** task.
22. In the code editor, modify the statement below the comment as shown in bold below:

```
await utils.GetStudentsByTeacher(SessionContext.UserName,
OnGetStudentsByTeacherComplete);
```

▶ Task 4: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. Verify that the application is more responsive than before while fetching the list of students for Esther Valle, and then close the application window.
5. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have updated the Grades application to retrieve data asynchronously.

Exercise 2: Providing Visual Feedback During Long-Running Operations

▶ Task 1: Create the BusyIndicator user control

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod10\Labfiles\Starter\Exercise 2**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. On the **Build** menu, click **Build Solution**.
6. In Solution Explorer, right-click **Grades.WPF**, point to **Add**, and then click **User Control**.
7. In the **Name** box, type **BusyIndicator.xaml**, and then click **Add**.
8. In Solution Explorer, expand **Grades.WPF**, and then drag **BusyIndicator.xaml** into the Controls folder.

 **Note:** It is better to create the user control at the project level and then move it into the Controls folder when it is created. This ensures that the user control is created in the same namespace as other project resources.

9. In the BusyIndicator.xaml file, in the **UserControl** element, delete the following attributes:

```
d:DesignWidth="300" d:DesignHeight="300"
```

10. Modify the **Grid** element to include a **Background** attribute, as the following markup shows:

```
<Grid Background="#99000000">
</Grid>
```

11. Type the following markup between the opening and closing **Grid** tags:

```
<Border CornerRadius="6"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
   <Border.Background>
      <LinearGradientBrush>
         <GradientStop
             Color="LightGray"
             Offset="0" />
         <GradientStop
             Color="DarkGray"
             Offset="1" />
      </LinearGradientBrush>
   </Border.Background>
   <Border.Effect>
      <DropShadowEffect
          Opacity="0.75" />
   </Border.Effect>
</Border>
```

12. On the blank line before the closing **Border** tag, type the following code:

```
<Grid Margin="10">
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
```

```
      <RowDefinition Height="Auto" />
   </Grid.RowDefinitions>
</Grid>
```

13. On the blank line before the closing **Grid** tag, type the following code:

```
<ProgressBar x:Name="progress"
    IsIndeterminate="True"
    Width="200"
    Height="25" Margin="20" />
```

14. Click after the end of the **ProgressBar** element, and then press Enter.
15. In the new line, type the following code:

```
<TextBlock x:Name="txtMessage"
    Grid.Row="1" FontSize="14"
    FontFamily="Verdana"
    Text="Please Wait..."
    TextAlignment="Center" />
```

16. On the **File** menu, click **Save All**.
17. In Solution Explorer, expand **Grades.WPF**, and then double-click **MainWindow.xaml**.
18. Towards the bottom of the MainWindow.xaml file, locate the **TODO: Exercise 2: Task 1b: Add the BusyIndicator control to MainWindow** comment.
19. Click at the end of the comment, press Enter, and then type the following code:

```
<y:BusyIndicator
   x:Name="busyIndicator"
   Margin="0"
   Visibility="Collapsed" />
```

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

▶ Task 2: Add StartBusy and EndBusy event handler methods

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Implement the StartBusy event handler** task.
2. In the blank line below the comment, type the following code:

```
private void StartBusy(object sender, EventArgs e)
{
   busyIndicator.Visibility = Visibility.Visible;
}
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Implement the EndBusy event handler** task.
4. In the blank line below the comment, type the following code:

```
private void EndBusy(object sender, EventArgs e)
{
   busyIndicator.Visibility = Visibility.Hidden;
}
```

▶ Task 3: Raise the StartBusy and EndBusy events

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3a: Add the StartBusy public event** task.

2. In the blank line below the comment, type the following code:

```
public event EventHandler StartBusy;
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3b: Add the EndBusy public event** task.
4. In the blank line below the comment, type the following code:

```
public event EventHandler EndBusy;
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3c: Implement the StartBusyEvent method to raise the StartBusy event** task.
6. In the blank line below the comment, type the following code:

```
private void StartBusyEvent()
{
   if (StartBusy != null)
      StartBusy(this, new EventArgs());
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3d: Implement the EndBusyEvent method to raise the EndBusy event** task.
8. In the blank line below the comment, type the following code:

```
private void EndBusyEvent()
{
   if (EndBusy != null)
      EndBusy(this, new EventArgs());
}
```

9. In Solution Explorer, double-click **MainWindow.xaml**.
10. In the MainWindow.xaml file, locate the **TODO: Exercise 2: Task 3e: Wire up the StartBusy and EndBusy event handlers for the StudentsPage view** comment.
11. Immediately below the comment, modify the **StudentsPage** element to include **StartBusy** and **EndBusy** attributes, as the following code shows:

```
<y:StudentsPage x:Name="studentsPage" StartBusy="StartBusy" EndBusy="EndBusy"
StudentSelected="studentsPage_StudentSelected" Visibility="Collapsed" />
```

12. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3f: Raise the StartBusy event** task.
13. In the blank line below the comment, type the following code:

```
StartBusyEvent();
```

14. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3g: Raise the EndBusy event** task.
15. In the blank line below the comment, type the following code:

```
EndBusyEvent();
```

▶ Task 4: Build and test the application

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

2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. Verify that the application displays the busy indicator while waiting for the list of students to load, and then close the application window.
5. On the **File** menu, click **Close Solution**.

Results: After completing this exercise, you should have updated the Grades application to display a progress indicator while the application is retrieving data.

Module 11: Integrating with Unmanaged Code

Lab: Upgrading the Grades Report

Exercise 1: Generating the Grades Report by Using Word

▶ Task 1: Examine the WordWrapper class that provides a functional wrapper around the dynamic (COM) API for Word

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows® 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. In File Explorer, navigate to the **E:\Mod11\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 1**, click **Grades.sln**, and then click **Open**.
12. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
13. In the **Solutions 'Grades' Properties Pages** dialog box, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
14. In Solution Explorer, expand **Grades.Utilities**, and then double-click **WordWrapper.cs**.
15. Examine the code that is currently contained within this class.
16. On the **View** menu, click **Task List.**
17. In the **Task List** window, in the **Categories** list, click **Comments**.
18. Double-click the **TODO: Exercise 1: Task 1a: Create a dynamic variable called _word for activating Word** task.
19. In the code editor, click in the blank line below the comment, and then type the following code:

```
dynamic _word = null;
```

20. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1b: Instantiate _word as a new Word Application object** task.
21. In the code editor, click in the blank line below the comment, and then type the following code:

```
this._word = new Application { Visible = false };
```

22. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1c: Create a new Word document** task.

23. In the code editor, click in the blank line below the comment, and then type the following code:

```
var doc = this._word.Documents.Add();
doc.Activate();
```

24. In the **Task List** window, double-click **TODO: Exercise 1: Task 1d: Save the document using the specified filename.** task.
25. In the code editor, click in the blank line below the comment, and then type the following code:

```
var currentDocument = this._word.ActiveDocument;
currentDocument.SaveAs(filePath);
```

26. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1e: Close the document** task.
27. In the code editor, click in the blank line below the comment, and then type the following code:

```
currentDocument.Close();
```

▶ Task 2: Review the code in the GeneratedStudentReport method to generate a Word document

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2a: Generate a student grade report as a Word document.** task.
2. Examine the code that is in this method to generate the student report.
3. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2b: Generate the report by using a separate task**.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
Task.Run(() => GenerateStudentReport(SessionContext.CurrentStudent,
dialog.FileName));
```

▶ Task 3: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. Click **Kevin Liu**, and then click **save report**.
5. In the **Save As** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 1**.
6. In the **File name** box, delete the existing contents, type **Kevin Liu Grades Report**, and then click **Save**.
7. Close the application, and then in Microsoft® Visual Studio®, on the **File** menu, click **Close Solution**.
8. Open File Explorer, browse to the **E:\Mod11\Labfiles\Starter\Exercise 1** folder, and then verify that the report has been generated.
9. Double-click **Kevin Liu Grades Report.docx**.
10. Review the grade report, and then close Word.

Results: After completing this exercise, the application will generate grade reports in Word format.

Exercise 2: Controlling the Lifetime of Word Objects by Implementing the Dispose Pattern

▶ Task 1: Run the application to generate a grades report and view the Word task in Task Manager

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 2**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. In the **Solutions 'Grades' Properties Pages** dialog box, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. On the **Build** menu, click **Build Solution**.
6. On the **Debug** menu, click **Start Without Debugging**.
7. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
8. Click **Kevin Liu**, and then click **save report**.
9. In the **Save As** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 2**.
10. In the **File name** box, delete the existing contents, type **Kevin Liu Grades Report**, and then click **Save**.
11. Close the application.
12. Open File Explorer, browse to the **E:\Mod11\Labfiles\Starter\Exercise 2** folder, and then verify that the report has been generated.
13. Right-click the **taskbar**, and then click **Task Manager**.
14. In the **Task Manager** window, click **More details**.
15. In the **Name** column, in the **Background processes** group, verify that **Microsoft Word (32 bit)** is still running.
16. Click **Microsoft Word (32 bit)**, and then click **End task**.
17. Close Task Manager.

▶ Task 2: Update the WordWrapper class to terminate Word correctly

1. In Visual Studio, in the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Specify that the WordWrapper class implements the IDisposable interface** task.
2. In the code editor, on the line below the comment, click at the end of the **public class WordWrapper** code, and then type the following code:

```
:IDisposable
```

3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Create the protected Dispose(bool) method** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
protected virtual void Dispose(bool isDisposing)
{
      if (!this.isDisposed)
      {
            if (isDisposing)
            {
                  // Release managed resources here
                  if (this._word != null)
                  {
                      this._word.Quit();
                  }
             }
             // Release unmanaged resources here
             if (this._word != null)
             {

System.Runtime.InteropServices.Marshal.ReleaseComObject(this._word);
             }
             this.isDisposed = true;
      }
}
```

5. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2c: Create the public Dispose method** task.
6. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public void Dispose()
{
       this.Dispose(true);
       GC.SuppressFinalize(this);
}
```

7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2d: Create a finalizer that calls the Dispose method** task.
8. In the code editor, click in the blank line below the comment, and then type the following code:

```
private bool isDisposed = false;
```

▶ Task 3: Wrap the object that generates the Word doc in a using statement

1. In the **Task List** window, double-click the **TODO: Exercise 2: Task 3: Ensure that the WordWrapper is disposed when the method finishes** task.
2. Below the comment, modify the **WordWrapper wrapper = new WordWrapper();** code to look like the following:

```
using (var wrapper = new WordWrapper())
{
```

3. At the end of the method, after the **wrapper.SaveAs(reportPath);** line of code, add a closing brace to end the **using** block.
4. Your code should look like the following:

```
public void GenerateStudentReport(LocalStudent studentData, string reportPath)
{
    // TODO: Exercise 2: Task 3: Ensure that the WordWrapper is disposed when the
method finishes
    using (var wrapper = new WordWrapper())
    {
        // Create a new Word document in memory
```

```
        wrapper.CreateBlankDocument();
        // Add a heading to the document
wrapper.AppendHeading(String.Format("Grade Report: {0} {1}", studentData.FirstName,
studentData.LastName));
        wrapper.InsertCarriageReturn();
        wrapper.InsertCarriageReturn();
        // Output the details of each grade for the student
        foreach (var grade in SessionContext.CurrentGrades)
        {
wrapper.AppendText(grade.SubjectName, true, true);
            wrapper.InsertCarriageReturn();
            wrapper.AppendText("Assessment: " + grade.Assessment, false, false);
            wrapper.InsertCarriageReturn();
            wrapper.AppendText("Date: " + grade.AssessmentDateString, false, false);
            wrapper.InsertCarriageReturn();
            wrapper.AppendText("Comment: " + grade.Comments, false, false);
            wrapper.InsertCarriageReturn();
            wrapper.InsertCarriageReturn();
        }
        // Save the Word document
        wrapper.SaveAs(reportPath);
  }
}
```

▶ Task 4: Use Task Manager to observe that Word terminates correctly after generating a report

1. On the **Build** menu, click **Build Solution**.
2. Right-click the **taskbar**, and then click **Task Manager**.
3. In Visual Studio, on the **Debug** menu, click **Start Without Debugging**.
4. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
5. Click **George Li**, and then click **save report**.
6. In the **Save As** dialog box, browse to **E:\Mod11\Labfiles\Starter\Exercise 2**.
7. In the **File name** box, delete the existing contents, and then type **George Li Grades Report**.
8. As you click **Save**, in the **Task Manager** window, watch the **Background processes** and verify that **Microsoft Word (32 bit)** appears and then disappears from the list.
9. Close Task Manager, and then close the application.
10. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, the application will terminate Word correctly after it has generated a grades report.

Module 12: Creating Reusable Types and Assemblies

Lab: Specifying the Data to Include in the Grades Report

Exercise 1: Creating and Applying the IncludeInReport attribute

▶ Task 1: Write the code for the IncludeInReportAttribute class

1. Start the 20483B-SEA-DEV11 virtual machine.
2. Log on to Windows® 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
3. Switch to the Windows 8 **Start** window and then type Explorer.
4. In the **Apps** list, click **File Explorer**.
5. Navigate to the **E:\Mod12\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
6. Close File Explorer.
7. Switch to the Windows 8 **Start** window.
8. Click **Visual Studio 2012**.
9. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
10. In the **Open Project** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 1**, click **Grades.sln**, and then click **Open**.
11. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
12. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
13. In Solution Explorer, expand **Grades.Utilities**, and then double-click **IncludeInReport.cs**.
14. On the **View** menu, click **Task List**.
15. In the **Task List** window, in the **Categories** list, click **Comments**.
16. Double-click the **TODO: Exercise 1: Task 1a: Specify that IncludeInReportAttribute is an attribute class** task.
17. In the code editor, below the comment, click at the end of the public **public class IncludeInReportAttribute** code, and then type the following code:

```
: Attribute
```

18. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1b: Specify the possible targets to which the IncludeInReport attribute can be applied** task.
19. In the code editor, click in the blank line below the comment, and then type the following code:

```
[AttributeUsage(AttributeTargets.Field | AttributeTargets.Property, AllowMultiple =
false)]
```

20. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1c: Define a private field to hold the value of the attribute** task.

21. In the code editor, click in the blank line below the comment, and then type the following code:

```
private bool _include;
```

22. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1d: Add public properties that specify how an included item should be formatted** task.

23. In the code editor, click in the blank line below the comment, and then type the following code:

```
public bool Underline { get; set; }
public bool Bold { get; set; }
```

24. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1e: Add a public property that specifies a label (if any) for the item** task.

25. In the code editor, click in the blank line below the comment, and then type the following code:

```
public string Label { get; set; }
```

26. In the **Task List** window, double-click the **TODO: Exercise 1: Task 1f: Define constructors** task.

27. In the code editor, click at the end of the comment, press Enter, and then type the following code:

```
public IncludeInReportAttribute()
{
    this._include = true;
    this.Underline = false;
    this.Bold = false;
    this.Label = string.Empty;
}
public IncludeInReportAttribute(bool includeInReport)
{
    this._include = includeInReport;
    this.Underline = false;
    this.Bold = false;
    this.Label = string.Empty;
}
```

▶ Task 2: Apply the IncludeInReportAttribute attribute to the appropriate properties

1. In Solution Explorer, expand **Grades.WPF**, and then double-click **Data.cs**.

2. In the **Task List** window, double-click the **TODO: Exercise 1: Task 2: Add the IncludeInReport attribute to the appropriate properties in the LocalGrade class** task.

3. In the **LocalGrade** class, expand the **Properties** region, and then expand the **Readonly Properties** region.

4. Above the **public string SubjectName** code, click in the blank line, and then type the following code:

```
[IncludeInReport(Label="Subject Name", Bold=true, Underline=true)]
```

5. Above the **public string AssessmentDateString** code, click in the blank line, press Enter, and then type the following code:

```
[IncludeInReport (Label="Date")]
```

6. Expand the **Form Properties** region.

7. Above the **public string Assessment** code, click in the blank line, press Enter, and then type the following code:

```
[IncludeInReport(Label = "Grade")]
```

8. Above the **public string Comments** code, click in the blank space, press Enter, and then type the following code:

```
[IncludeInReport(Label = "Comments")]
```

▶ Task 3: Build the application and review the metadata for the LocalGrades class

1. On the **Build** menu, click **Build Solution**.
2. Open File Explorer and browse to the **C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools** folder.
3. Right-click **ildasm.exe,** and then click **Open**.
4. In the **IL DASM** window, on the **File** menu, click **Open**.
5. In the **Open** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 1\Grades.WPF\bin\Debug**, click **Grades.WPF.exe**, and then click **Open**.
6. In the **IL DASM** application window, expand **Grades.WPF,** expand **Grades.WPF.LocalGrade**, and then double-click **Assessment : instance string();**.
7. In the **Grades.WPF.LocalGrade::Assessment : instance string()** window, in the **Assessment** method, verify that the **.custom instance void [Grades.Utilities]Grades.Utilities.IncludeInReportAttribute::.ctor()** code is present, and then close the window.
8. In the **IL DASM** application window, double-click **AssessmentDateString : instance string();**.
9. In the **Grades.WPF.LocalGrade::AssessmentDateString : instance string()** window, in the **AssessmentDateString** method, verify that the **.custom instance void [Grades.Utilities]Grades.Utilities.IncludeInReportAttribute::.ctor()** code is present, and then close the window.
10. In the **IL DASM** application window, double-click **Comments : instance string();**.
11. In the **Grades.WPF.LocalGrade::Comments : instance string()** window, in the **Comments** method, verify that the **.custom instance void [Grades.Utilities]Grades.Utilities.IncludeInReportAttribute::.ctor()** code is present, and then close the window.
12. In the **IL DASM** application window, double-click **SubjectName : instance string();**.
13. In the **Grades.WPF.LocalGrade::SubjectName : instance string()** window, in the **SubjectName** method, verify that the **.custom instance void [Grades.Utilities]Grades.Utilities.IncludeInReportAttribute::.ctor()** code is present, and then close the window.
14. Close the IL DASM application.
15. Close File Explorer.
16. In Visual Studio, on the **File** menu, click **Close Solution**.

Results: After completing this exercise, the **Grades.Utilities** assembly will contain an **IncludeInReport** custom attribute and the **Grades** class will contain fields and properties that are tagged with that attribute.

Exercise 2: Updating the Report

▶ Task 1: Implement a static helper class called IncludeProcessor

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 2**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. In Solution Explorer, expand **Grades.Utilities**, and then double-click **IncludeInReport.cs**.
6. Below the **Output** window, click **Task List**.
7. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: Define a struct that specifies the formatting to apply to an item** task.
8. In the code editor, click in the blank line in the **FormatField** struct, and then type the following code:

```
public string Value;
public string Label;
public bool IsBold;
public bool IsUnderlined;
```

9. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Find all the public fields and properties in the dataForReport object** task.
10. In the code editor, click in the blank line below the comment, and then type the following code:

```
Type dataForReportType = dataForReport.GetType();
fieldsAndProperties.AddRange(dataForReportType.GetFields());
fieldsAndProperties.AddRange(dataForReportType.GetProperties());
```

11. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1c: Iterate through all public fields and properties, and process each item that is tagged with the IncludeInReport attribute** task.
12. In the code editor, click in the blank line below the comment, and then type the following code:

```
foreach (MemberInfo member in fieldsAndProperties)
{
```

13. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1d: Determine whether the current member is tagged with the IncludeInReport attribute** task.
14. In the code editor, click in the blank line below the comment, and then type the following code:

```
object[] attributes = member.GetCustomAttributes(false);
IncludeInReportAttribute attributeFound = Array.Find(attributes, a => a.GetType() ==
typeof(IncludeInReportAttribute)) as IncludeInReportAttribute;
```

15. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1e: If the member is tagged with the IncludeInReport attribute, construct a FormatField item** task.
16. In the code editor, click in the blank line below the comment, and then type the following code:

```
if (attributeFound != null)
{
    // Find the value of the item tagged with the IncludeInReport attribute
```

```
    string itemValue;
    if (member is FieldInfo)
    {
        itemValue = (member as FieldInfo).GetValue(dataForReport).ToString();
    }
    else
    {
        itemValue = (member as PropertyInfo).GetValue(dataForReport).ToString();
    }
```

17. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1f: Construct a FormatField item with this data** task.
18. In the code editor, click in the blank line below the comment, and then type the following code:

```
FormatField item = new FormatField()
{
    Value = itemValue,
    Label = attributeFound.Label,
    IsBold = attributeFound.Bold,
    IsUnderlined = attributeFound.Underline
};
```

19. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1g: Add the FormatField item to the collection to be returned** task.
20. In the code editor, click in the blank line below the comment, and then type the following code:

```
        items.Add(item);
    }
}
```

▶ Task 2: Update the report functionality for the StudentProfile view

1. In Solution Explorer, expand **Grades.WPF**, expand **Views,** expand **StudentProfile.xaml**, and then double-click **StudentProfile.xaml.cs**.
2. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2a: Use the IncludeProcessor to determine which fields in the Grade object are tagged** task.
3. In the code editor, click in the blank line below the comment, and then type the following code:

```
List<FormatField> itemsToReport = IncludeProcessor.GetItemsToInclude(grade);
```

4. In the **Task List** window, double-click the **TODO: Exercise 2: Task 2b: Output each tagged item, using the format specified by the properties of the IncludeInReport attribute for each item** task.
5. In the code editor, click in the blank line below the comment, and then type the following code:

```
foreach (FormatField item in itemsToReport)
{
wrapper.AppendText(item.Label == string.Empty ? item.Value : item.Label + ": " +
item.Value, item.IsBold, item.IsUnderlined);
wrapper.InsertCarriageReturn();
}
```

▶ Task 3: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.

3. In the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
4. In the **Class 3C** view, click **Kevin Liu**.
5. Verify that the student report for Kevin Liu appears, and then click **save report**.
6. In the **Save As** dialog box, browse to the **E:\Mod12\Labfiles\Starter\Exercise 2** folder.
7. In the **File name** box, type **KevinLiuGradesReport**, and then click **Save**.
8. Close the application.
9. In Visual Studio, on the **File** menu, click **Close Solution**.
10. Open File Explorer, browse to **E:\Mod12\Labfiles\Starter\Exercise 2**, and then verify that KevinLiuGradesReport.docx has been generated.
11. Right-click **KevinLiuGradesReport.docx**, and then click **Open**.
12. Verify that the document contains the grade report for Kevin Liu and that it is correctly formatted, and then close Word.

Results: After completing this exercise, the application will be updated to use reflection to include only the tagged fields and properties in the grades report.

Exercise 3: Storing the Grades.Utilities Assembly Centrally (If Time Permits)

▶ Task 1: Sign the Grades.Utilities assembly and deploy it to the GAC

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 3**, click **Grades.sln**, and then click **Open**.
3. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
4. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
5. Switch to the Windows 8 **Start** window.
6. In the **Start** window, right-click the background to display the taskbar.
7. On the taskbar, click **All apps**.
8. In the **Start** window, right-click the **VS2012 x86 Native Tools Command** icon.
9. On the taskbar, click **Run as administrator**.
10. In the **User Account Control** dialog box, in the **Password** box, type **Pa$$w0rd**, and then click **Yes**.
11. At the command prompt, type the following code, and then press Enter:

```
E:
```

12. At the command prompt, type the following code, and then press Enter:

```
cd E:\Mod12\Labfiles\Starter
```

13. At the command prompt, type the following code, and then press Enter:

```
sn -k GradesKey.snk
```

14. Verify that the text **Key pair written to GradesKey.snk** is displayed.
15. In Visual Studio, in Solution Explorer, right-click **Grades.Utilities**, and then click **Properties**.
16. On the **Signing** tab, select **Sign the assembly**.
17. In the **Choose a strong name key file** list, click **Browse**.
18. In the **Select File** dialog box, browse to **E:\Mod12\Labfiles\Starter**, click **GradesKey.snk**, and then click **Open**.
19. On the **Build** menu, click **Build Solution**.
20. Switch to the command prompt, type the following code, and then press Enter:

```
cd E:\Mod12\Labfiles\Starter\Exercise 3\Grades.Utilities\bin\Debug
```

21. At the command prompt, type the following code, and then press Enter:

```
gacutil -i Grades.Utilities.dll
```

22. Verify that the text **Assembly successfully added to the cache** is displayed, and then close the Command Prompt window.

▶ Task 2: Reference the Grades.Utilities assembly in the GAC from the application

1. In Visual Studio, in Solution Explorer, expand **Grades.WPF**, expand **References**, right-click **Grades.Utilities**, and then click **Remove**.
2. Right-click **References**, and then click **Add Reference**.
3. In the **Reference Manager – Grades.WPF** dialog box, click the **Browse** button.
4. In the **Select the files to reference** dialog box, browse to **E:\Mod12\Labfiles\Starter\Exercise 3\Grades.Utilities\bin\Debug**, click **Grades.Utilities.dll**, and then click **Add**.
5. In the **Reference Manager – Grades.WPF** dialog box, click **OK**.
6. On the **Build** menu, click **Build Solution**.
7. On the **Debug** menu, click **Start Without Debugging**.
8. In the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.
9. In the **Class 3C** view, click **Kevin Liu**.
10. Verify that the student report for Kevin Liu appears, and then click **save report**.
11. In the **Save As** dialog box, browse to the **E:\Mod12\Labfiles\Starter\Exercise 3** folder.
12. In the **File name** box, type **KevinLiuGradesReport**, and then click **Save**.
13. Close the application.
14. In Visual Studio, on the **File** menu, click **Close Solution**.
15. Open File Explorer, browse to **E:\Mod12\Labfiles\Starter\Exercise 3**, and then verify that KevinLiuGradesReport.docx has been generated.
16. Right-click **KevinLiuGradesReport.docx**, and then click **Open**.
17. Verify that the document contains the grade report for Kevin Liu and that it is correctly formatted, and then close Word.

Results: After completing this exercise, you will have a signed version of the **Grades.Utilities** assembly deployed to the GAC.

Module 13: Encrypting and Decrypting Data

Lab: Encrypting and Decrypting the Grades Report

Exercise 1: Encrypting the Grades Report

▶ Task 1: Create an asymmetric certificate

1. Start the MSL-TMG1 virtual machine if it is not already running.
2. Start the 20483B-SEA-DEV11 virtual machine.
3. Log on to Windows® 8 as **Student** with the password **Pa$$w0rd**. If necessary, click **Switch User** to display the list of users.
4. Switch to the Windows 8 **Start** window and then type Explorer.
5. In the **Apps** list, click **File Explorer**.
6. Navigate to the **E:\Mod13\Labfiles\Databases** folder, and then double-click **SetupSchoolGradesDB.cmd**.
7. Close File Explorer.
8. Switch to the Windows 8 **Start** window.
9. Click **Visual Studio 2012**.
10. In Microsoft® Visual Studio®, on the **File** menu, point to **Open**, and then click **Project/Solution**.
11. In the **Open Project** dialog box, browse to **E:\Mod13\Labfiles\Starter\Exercise 1**, click **Grades.sln**, and then click **Open**.
12. In Solution Explorer, right-click **Solutions 'Grades'**, and then click **Properties**.
13. On the **Startup Project** page, click **Multiple startup projects**. Set **Grades.Web** and **Grades.WPF** to **Start without debugging**, and then click **OK**.
14. In Solution Explorer, expand the **Grades.Utilities** node, and then double-click the **CreateCertificate.cmd** file.
15. Review the contents of this file.
16. Switch to the Windows 8 **Start** window.
17. In the **Start** window, right-click the background to display the task bar.
18. On the task bar, click **All apps**.
19. In the **Start** window, right-click the **VS2012 x86 Native Tools Command** icon.
20. On the task bar, click **Run as administrator**.
21. In the **User Account Control** dialog box, in the **Password** box, type **Pa$$w0rd**, and then click **Yes**.
22. At the command prompt, type the following, and then press Enter.

```
E:
```

23. At the command prompt, type the following, and then press Enter.

```
cd E:\Mod13\Labfiles\Starter\Exercise 1\Grades.Utilities
```

24. At the command prompt, type the following, and then press Enter.

```
CreateCertificate.cmd
```

25. Verify that the command returns a success message, and then close the command window.

▶ Task 2: Retrieve the Grade certificate

1. In Visual Studio, on the **View** menu, click **Task List**.
2. In the **Task List** window, in the **Categories** list, click **Comments**.
3. Double-click the **TODO: Exercise 1: Task 2a: Loop through the certificates in the X509 store to return the one matching _certificateSubjectName** task.
4. In the code editor, click in the blank line below the comment, and then type the following code:

```
foreach (var cert in store.Certificates)
  if (cert.SubjectName.Name.Equals(this._certificateSubjectName,
StringComparison.InvariantCultureIgnoreCase))
    return cert;
```

▶ Task 3: Encrypt the data

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3a: Get the public key from the X509 certificate** task.
2. In the code editor, delete the following line of code:

```
throw new NotImplementedException();
```

3. In the blank line below the comment, type the following code:
4. var provider = (RSACryptoServiceProvider)this._certificate.PublicKey.Key;
5. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3b: Create an instance of the AesManaged algorithm** task.
6. In the code editor, click in the blank line below the comment, and then type the following code:

```
using (var algorithm = new AesManaged())
{
```

7. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3c: Create an underlying stream for the unencrypted data** task.
8. In the code editor, click in the blank line below the comment, and then type the following code:

```
using (var outStream = new MemoryStream())
{
```

9. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3d: Create an AES encryptor based on the key and IV** task.
10. In the code editor, click in the blank line below the comment, and then type the following code:

```
using (var encryptor = algorithm.CreateEncryptor())
{
  var keyFormatter = new RSAPKCS1KeyExchangeFormatter(provider);
  var encryptedKey = keyFormatter.CreateKeyExchange(algorithm.Key,
algorithm.GetType());
```

11. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3e: Create byte arrays to get the length of the encryption key and IV** task.
12. In the code editor, click in the blank line below the comment, and then type the following code:

```
var keyLength = BitConverter.GetBytes(encryptedKey.Length);
var ivLength = BitConverter.GetBytes(algorithm.IV.Length);
```

13. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3f: Write the following to the out stream** task.
14. In the code editor, click in the blank line below the comment block, and then type the following code:

```
outStream.Write(keyLength, 0, keyLength.Length);
outStream.Write(ivLength, 0, ivLength.Length);
outStream.Write(encryptedKey, 0, encryptedKey.Length);
outStream.Write(algorithm.IV, 0, algorithm.IV.Length);
```

15. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3g: Create a CryptoStream that will write the encrypted data to the underlying buffer** task.
16. In the code editor, click in the blank line below the comment, and then type the following code:

```
using (var encrypt = new CryptoStream(outStream, encryptor, CryptoStreamMode.Write))
{
```

17. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3h: Write all the data to the stream** task.
18. In the code editor, click in the blank line below the comment, and then type the following code:

```
encrypt.Write(bytesToEncrypt, 0, bytesToEncrypt.Length);
encrypt.FlushFinalBlock();
```

19. In the **Task List** window, double-click the **TODO: Exercise 1: Task 3i: Return the encrypted buffered data as a byte[]** task.
20. In the code editor, click in the blank line below the comment, and then type the following code:

```
      return outStream.ToArray();
      }
    }
  }
```

21. }

▶ Task 4: Write the encrypted data to disk

1. In the **Task List** window, double-click the **TODO: Exercise 1: Task 4a: Write the encrypted bytes to disk** task.
2. In the code editor, click in the blank line below the comment, and then type the following code:

```
File.WriteAllBytes(filePath, encryptedBytes);
```

▶ Task 5: Build and test the application

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, in the **Username** box, type **vallee**, and in the **Password** box, type **password99**, and then click **Log on**.

4. In the **Class 3C** view, click **George Li**.
5. In the **Report Card** view, click **save report**.
6. In the **Save As** dialog box, browse to the **E:\Mod13\Labfiles\Reports** folder, in the **File name** box, type **GeorgeLi**, and then click **Save**.
7. In the **Report Card** view, click **Back**.
8. In the **Class 3C** view, click **Kevin Liu**.
9. In the **Report Card** view, click **save report**.
10. In the **Save As** dialog box, browse to the **E:\Mod13\Labfiles\Reports** folder, in the **File name** box, type **KevinLiu**, and then click **Save**.
11. In the **Report Card** view, click **Log off**, and then close the application.
12. On the **File** menu, click **Close Solution**.
13. Open Windows Internet Explorer®, and in the address bar, type **E:\Mod13\Labfiles\Reports\KevinLiu.xml**, and then press Enter.
14. Note the page is blank because the file is encrypted, and then close Internet Explorer.
15. Open File Explorer, and then browse to the **E:\Mod13\Labfiles\Reports** folder.
16. Right-click **KevinLiu.xml**, and then click **Edit**.
17. Review the encrypted data, close Notepad, and then close File Explorer.

Results: After completing this exercise, you should have updated the Grades application to encrypt generated reports.

Exercise 2: Decrypting the Grades Report

▶ Task 1: Decrypt the data

1. In Visual Studio, on the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, browse to **E:\Mod13\Labfiles\Starter\Exercise 2**, click **School-Reports.sln**, and then click **Open**.
3. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1a: Get the private key from the X509 certificate** task.
4. In the code editor, delete the following line of code:

```
throw new NotImplementedException();
```

5. In the blank line below the comment, type the following code:

```
var provider = (RSACryptoServiceProvider)this._certificate.PrivateKey;
```

6. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1b: Create an instance of the AESManaged algorithm which the data is encrypted with** task.
7. In the blank line below the comment, type the following code:

```
using (var algorithm = new AesManaged())
{
```

8. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1c: Create a stream to process the bytes** task.
9. In the blank line below the comment, type the following code:

```
using (var inStream = new MemoryStream(bytesToDecrypt))
{
```

10. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1d: Create byte arrays to get the length of the encryption key and IV** task.
11. In the blank line below the comment, type the following code:

```
var keyLength = new byte[4];
var ivLength = new byte[4];
```

12. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1e: Read the key and IV lengths starting from index 0 in the in stream** task.
13. In the blank line below the comment, type the following code:

```
inStream.Seek(0, SeekOrigin.Begin);
inStream.Read(keyLength, 0, keyLength.Length);
inStream.Read(ivLength, 0, ivLength.Length);
```

14. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1f: Convert the lengths to ints for later use** task.
15. In the blank line below the comment, type the following code:

```
var convertedKeyLength = BitConverter.ToInt32(keyLength, 0);
var convertedIvLength = BitConverter.ToInt32(ivLength, 0);
```

16. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1g: Determine the starting position and length of data** task.
17. In the blank line below the comment, type the following code:

```
var dataStartPos = convertedKeyLength + convertedIvLength + keyLength.Length +
ivLength.Length;
var dataLength = (int)inStream.Length - dataStartPos;
```

18. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1h: Create the byte arrays for the encrypted key, the IV, and the encrypted data** task.
19. In the blank line below the comment, type the following code:

```
var encryptionKey = new byte[convertedKeyLength];
var iv = new byte[convertedIvLength];
var encryptedData = new byte[dataLength];
```

20. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1i: Read the key, IV, and encrypted data from the in stream** task.
21. In the blank line below the comment, type the following code:

```
inStream.Read(encryptionKey, 0, convertedKeyLength);
inStream.Read(iv, 0, convertedIvLength);
inStream.Read(encryptedData, 0, dataLength);
```

22. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1j: Decrypt the encrypted AesManaged encryption key** task.
23. In the blank line below the comment, type the following code:

```
var decryptedKey = provider.Decrypt(encryptionKey, false);
```

24. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1k: Create an underlying stream for the decrypted data** task.
25. In the blank line below the comment, type the following code:

```
using (var outStream = new MemoryStream())
{
```

26. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1l: Create an AES decryptor based on the key and IV** task.
27. In the blank line below the comment, type the following code:

```
using (var decryptor = algorithm.CreateDecryptor(decryptedKey, iv))
{
```

28. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1m: Create a CryptoStream that will write the decrypted data to the underlying buffer** task.
29. In the blank line below the comment, type the following code:

```
using (var decrypt = new CryptoStream(outStream, decryptor, CryptoStreamMode.Write))
{
```

30. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1n: Write all the data to the stream** task.
31. In the blank line below the comment, type the following code:

```
decrypt.Write(encryptedData, 0, dataLength);
decrypt.FlushFinalBlock();
```

32. In the **Task List** window, double-click the **TODO: Exercise 2: Task 1o: Return the decrypted buffered data as a byte[]** task.
33. In the blank line below the comment, type the following code:

```
            return outStream.ToArray();
        }
      }
    }
  }
}
```

▶ Task 2: Build and test the solution

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.
3. When the application loads, click **Browse**.
4. In the **Browse For Folder** dialog box, browse to the **E:\Mod13\Labfiles\Reports** folder, and then click **OK**.
5. Click **Print**.
6. In the **Save Print Output As** dialog box, browse to the **E:\Mod13\Labfiles\Reports\ClassReport** folder, in the **File name** box, type **3CReport**, and then click **Save**.
7. In the **The School of Fine Arts** dialog box, click **OK**, and then close the application.
8. Open File Explorer, and browse to the **E:\Mod13\Labfiles\Reports\ClassReport** folder.
9. Right-click **3CReport.oxps**, and then click **Open**.
10. Review the unencrypted report, and then close the XPS Viewer.

Results: After completing this exercise, you should have a composite unencrypted report that was generated from the encrypted reports.