**Lab 01: Developing the Class Enrollment Application**

**Scenario**

You are a Visual C# developer working for a software development company that is writing applications for The School of Fine Arts, an elementary school for gifted children.

The school administrators require an application that they can use to enroll students in a class. The application must enable an administrator to add and remove students from classes, as well as to update the details of students.

You have been asked to write the code that implements the business logic for the application.

 **Note:** During the labs for the first two modules in this course, you will write code for this class enrollment application.

When The School of Fine Arts ask you to extend the application functionality, you realize that you will need to test proof of concept and obtain client feedback before writing the final application, so in the lab for Module 3, you will begin developing a prototype application and continue with this until then end of Module 8.

In the lab for Module 9, after gaining signoff for the final application, you will develop the user interface for the production version of the application, which you will work on for the remainder of the course.

**Objectives**

After completing this lab, you will be able to:

1. Write Visual C# code that implements the logic necessary to edit the details of a student.
2. Write Visual C# code that implements the logic necessary to add new students.
3. Write Visual C# code that implements the logic necessary to remove students from a class.
4. Perform simple data transformations for displaying information.

* Estimated Time: 105 minutes

**Exercise 1: Implementing Edit Functionality for the Students List**

**Scenario**

In this exercise, you will write the code that enables an administrator using the application to edit a student’s details.

A list of students is displayed in the user interface of the application. When the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Enter. If they did press Enter, you will write code to display the student’s details in a separate form, which the user can use to modify the details. When the user closes the form, you will copy the updated details back to the list box displaying the list of students. Finally, you will run the application to verify that your code functions as expected, and then use the debugging tools to examine code as it runs.

The main tasks for this exercise are as follows:

1. Detect whether the user has pressed the Enter key.
2. Initialize the StudentForm window and populate it with the details of the currently selected student.
3. Display the StudentForm window and copy the updated student details entered back to the Student object.
4. Run the application and verify that the edit functionality works as expected.
5. Use the Visual Studio Debugger to step through the code.

 **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 and log on as **Student** with the password **Pa$$w0rd**.
3. Start File Explorer, navigate to the **E:\Mod01\Labfiles\Databases** folder, and then run

**SetupSchoolDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio and from the **E:\Mod01\Labfiles\Starter\Exercise 1** folder, open the **School.sln**

solution.

1. In the code for the **MainWindow.xaml.cs** window, find the **studentsList\_KeyDown** method.
2. In this method, add a switch statement to detect whether the user has pressed Enter.

The second argument passed to this method is a KeyEventArgs object named e. This object has a Key property which returns the keyboard key associated with the event. You can use this in conjunction with the Key enumeration to determine which key initiated the KeyDown event.

1. If the user has pressed Enter, store the selected student in a **Student** object variable.

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

1. If the user has pressed the Enter key, create a new instance of the **StudentForm** window named **sf**

and set the **Title** property of the window to **Edit Student Details**.

1. Populate the following text boxes on the form with the corresponding properties of the current student:
   1. **firstName**
   2. **lastName**
   3. **dateOfBirth**

To store data in a text box in a window, set the **Text** property of the text box to the required string.

1. Display the date of birth by using the standard short date format without the time element by using the “d” format specifier as shown in the following code.

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. At the end of the **case Key.Enter** block, display the **StudentForm** window by using the **ShowDialog**

method of the form.

1. If the user clicks OK in the StudentForm window, copy the updated student details from the StudentForm window back to the Student object.

You can detect whether the user clicked the OK button by examining the return value of the ShowDialog method. If the Value property of this is true, the user clicked OK, otherwise the clicked

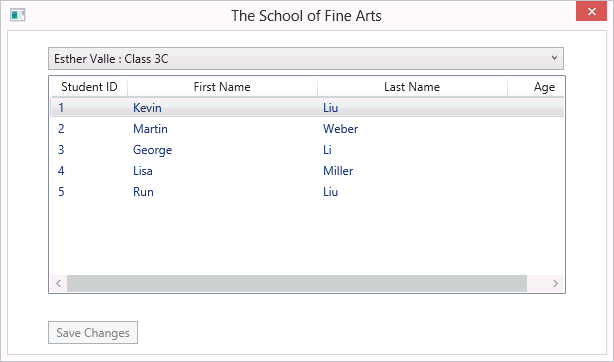
Cancel.

You can use the DateTime.Parse method to convert the date of birth string from the text box to a DateTime type.

1. If the user clicks OK, also enable the Save Changes button in the user interface.

To enable an item in a user interface, set the IsEnabled property of the item to true.

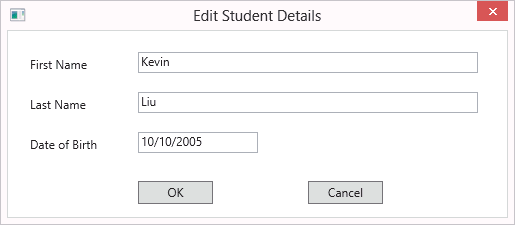
 Task 4: Run the application and verify that the edit functionality works as expected

1. Build the solution and resolve any compilation errors.
2. Run the application and verify that it displays the initial list of students. The initial students list should look like this:

**FIGURE 01.1:THE INITIAL STUDENTS LIST**

1. Edit the row for Kevin Liu 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**

1. Change the last name of **Kevin Liu** to **Cook** and verify that the updated data is copied back to the students list.
2. Verify that the **Save Changes** button is now enabled.
3. Close the application.

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

1. In Visual Studio, in the **studentsList\_KeyDown** method, insert a breakpoint at the statement that sets the **Title** property of the **StudentForm**.
2. Debug the application.
3. Edit the row for George Li.
4. When Visual Studio enters break mode, open the **Watch 1** window that automatically appears in the tab group in the bottom left window and populate the grid with a row for each of the following:
   * **sf.Title**
   * **sf.firstName.Text**
   * **sf.lastName.Text**
   * **sf.dateOfBirth.Text**
5. Step over the next code statement four times.
6. Use the **Immediate Window** that automatically appears in the tab group in the bottom middle window to view the value of **sf.firstName.Text** and to verify that it contains the value **George**.
7. In the **Watch 1** window, change the value **George** to **Dominik**.
8. In the **Immediate Window**, enter **sf.lastName.Text** and verify that the value **"Li"** is displayed.
9. Enter code to change the **sf.lastName.Text** value to **"Dubicki"**, and then verify that value changes in the Watch 1 window.
10. Continue debugging and verify that the following information is displayed in the Edit Student Details form:

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

1. Stop debugging the application.
2. In Visual Studio, on the **Debug** menu, click **Delete All Breakpoints**, and then close the solution.

**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

**Scenario**

In this exercise, you will write code that enables an administrator using the application to add a new student to the students list.

A list of students is displayed in the user interface of the application. When the user presses a key on the keyboard, you will check whether the key they pressed was Insert. If they did press Insert, you will write code to display a form in which the user can enter the details of a new student, including their first name, last name, and date of birth. When the user closes the form, you will add the new student to the list of

students and display the details in the list box. Finally, you will run the application to verify that your code functions as expected.

The main tasks for this exercise are as follows:

1. Add logic to the key down method to detect if the Insert key has been pressed.
2. Initialize the student form.
3. Display the StudentForm window and enable the user to provide the details of the new student.
4. Assign the new student to a class and enable the user to save the details of the new student.
5. Run the application and verify that the insert functionality works as expected.

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

1. In Visual Studio, from the **E:\Mod01\Labfiles\Starter\Exercise 2** folder, open the **School.sln**

solution.

1. In the code for the **MainWindow.xaml.cs** window, locate the **studentsList\_KeyDown** method.
2. In this method, add a statement to detect whether the user has pressed Insert.

 Task 2: Initialize the student form

1. If the user has pressed Insert, create a new instance of the StudentForm window.
2. Set the **Title** property of the window to **New Student for Class** appended to the **Class** property of the **teacher** object. Use code similar to the following to create the string for the **Title** property.

"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. Display the StudentForm window by using the **ShowDialog** method.
2. If the user clicks the **OK** button in the StudentForm window, create a new student object and copy the student details from the **StudentForm** window to the new student object.

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

1. If the user clicks the **OK** button in the **StudentForm** window, use the **Students.Add** method of the current teacher to assign the new student to a class. You can use **This.Teacher** to access the current teacher.
2. Add the new student object to the list of students displayed on the form.
3. Enable the **Save Changes** button in the user interface.

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

1. Build the solution and resolve any compilation errors.
2. Run the application and verify that it displays the initial list of students.
3. Display the new student window and verify that it contains no data.
4. Insert the details for **Darren Parker**, date of birth is **02/03/2006**, and verify that the new student is added to the students list. The ID of a new student will be 0 until they are saved to the database in the next lab.
5. Verify that the **Save Changes** button is now enabled.
6. Close the application.
7. In Visual Studio, close the 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

**Scenario**

In this exercise, you will write code that enables an administrator to remove a student from the students list.

A list of students is displayed in the user interface of the application. If the user selects a student and then presses a key on the keyboard, you will check whether the key they pressed was Delete. If they did press Delete, you will write code to prompt the user to confirm that they want to remove the selected student from the class. If they do, the student will be deleted from the students list for the appropriate class, otherwise nothing changes. Finally, you will run the application to verify that your code functions as expected.

The main tasks for this exercise are as follows:

1. Add logic to the key down method to detect if the Delete key has been pressed.
2. Prompt the user to confirm that they want to remove the selected student from the class.
3. Remove the student and enable the user to save the changes.
4. Run the application and verify that the delete functionality works as expected.

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

1. In Visual Studio, from the **E:\Mod01\Labfiles\Starter\Exercise 3** folder, open the **School.sln**

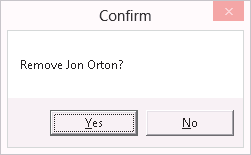
solution.

1. In the code for the **MainWindow.xaml.cs** window, find the **studentsList\_KeyDown** method.
2. In this method, add a statement to detect whether the user has pressed Delete.

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

1. If the user presses Delete, find the details of the student that the user has selected and display a message box showing the selected student’s name. Ask the user to confirm that they want to remove the student.

The confirmation prompt should look like this.



**FIGURE 01.3:PROMPT TO CONFIRM THE DELETION OF A STUDENT.**

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

1. If the user confirms that they want to delete the student, delete the current student object from the

**schoolContext.Students** collection and enable the **Save Changes** button in the user interface.

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

1. Build the solution and resolve any compilation errors.
2. Run the application and verify that it displays the initial list of students.
3. Delete the student **Jon Orton** from class **4B**.
4. Verify that the prompt window appears, the student is removed from the list, and that the **Save Changes** button is enabled.
5. Close the application.
6. In Visual Studio, close the solution.

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

###### Exercise 4: Displaying a Student’s Age

**Scenario**

In this exercise, you will update the application to display a student’s age instead of their date of birth.

You will write code in the **AgeConverter** class that is linked to the grid column displaying student ages. In this class, you will write code to work out the difference between the current date and the date of birth of the student, and then convert this value into years. Then you will run the application to verify that the Age column now displays age in years instead of the date of birth.

The main tasks for this exercise are as follows:

1. Examine the MainWindow XAML.
2. Add logic to the AgeConverter class to calculate a student’s age from their date of birth.
3. Run the application and verify that the student’s age now appears correctly.

 **Task 1: Examine the MainWindow XAML**

1. In Visual Studio, open the **School.sln** solution from the **E:\Mod01\Labfiles\Starter\Exercise 4**

folder.

1. Build the solution.
2. View the **MainWindow.xaml** code.
3. Note how the **Age** column in the **GridView** uses databinding with a value converter (**AgeConverter**).

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

1. In the code for the **MainWindow.xaml.cs** window, find the **Convert** method in the **AgeConverter**

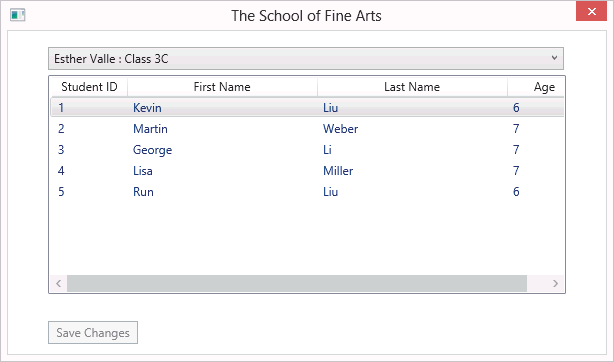
class.

1. In this method, add code that checks that the **value** parameter of the method contains data. If it does not, return an empty string.
2. If the **value** parameter is not null, convert the **value** parameter to a **DateTime** object.
3. Calculate the difference between the current date and the student’s date of birth by using the **DateTime.Now.Subtract** method to subtract the date of birth from the current date and store the result in a **TimeSpan** object.
4. Convert the result into a number of years by using the **TimeSpan.Days** method to retrieve the difference in days and then using the following formula to calculate the age in years.

Age in years = difference in days / 365.25

1. Convert the number of years into a string and return it to the calling method.

 Task 3: Run the application and verify that the student’s age now appears correctly

1. Build the solution and resolve any compilation errors.
2. Run the application and verify it displays the initial list of students, with their ages. The student list should now look similar to the following:

**FIGURE 01.4:THE STUDENT LIST DISPLAYING THEIR AGES.**

1. Add yourself as a student and verify that your age displays correctly in the student list.
2. Close the application.
3. In Visual Studio, close the solution.

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

### Module Review and Takeaways

In this module, you learned about some of the core features provided by the .NET Framework and Microsoft Visual Studio®. You also learned about some of the core Visual C#® constructs that enable you to start developing .NET Framework applications.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **What Visual Studio template would you use to create a .dll?** | |
| Select the correct answer. | |
|  | Console application |
|  | Windows Forms application |
|  | WPF application |
|  | Class library |
|  | WCF Service application |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Given the following for loop statement, what is the value of the count variable once the loop has finished executing?**  var count = 0;  for (int i = 5; i < 12; i++)  {  count++;  } | |
| Select the correct answer. | |
|  | 3 |
|  | 5 |
|  | 7 |
|  | 9 |
|  | 11 |

**Lab 02: Extending the Class Enrollment Application Functionality**

**Scenario**

You have been asked to refactor the code that you wrote in the lab exercises for module 1 into separate methods to avoid the duplication of code in the Class Enrollment Application.

Also, you have been asked to write code that validates the student information that the user enters and to enable the updated student information to be written back to the database, handling any errors that may occur.

**Objectives**

After completing this lab, you will be able to:

1. Refactor code to facilitate reusability.
2. Write Visual C# code that validates data entered by a user.
3. Write Visual C# code that saves changes back to a database.

* Estimated Time: 90 minutes

**Exercise 1: Refactoring the Enrollment Code**

**Scenario**

In this exercise, you will refactor the existing code to avoid writing duplicate code.

The application currently enables a user to edit a student’s details by pressing Enter, but you now want them to also be able to initiate the edit process by double-clicking on a student in the list. You will begin by creating a new method that contains the code for editing a student’s details. This will avoid duplicating and maintaining the code in both event handlers. You will then call the new method from both the **studentsList\_MouseDoubleClick** and **StudentsList\_Keydown** events. While doing this, you also decide to refactor the code for adding and deleting students into separate methods, so that it can be called from other parts of the application if the need arises. You will then run the application and verify that users can press Enter or double-click on a student to edit the student’s details, can press Insert to add a new student, and can press Delete to remove a student.

The main tasks for this exercise are as follows:

1. Copy the code for editing a student into the studentsList\_MouseDoubleClick event handler.
2. Run the application and verify that the user can now double-click a student to edit their details.
3. Use the Analyze Solution for Code Clones wizard to detect the duplicated code.
4. Refactor the logic that adds and deletes a student into the addNewStudent and deleteStudent methods.
5. Verify that students can still be added and removed from the application.
6. Debug the application and step into the new method calls.

 **Task 1: Copy the code for editing a student into the studentsList\_MouseDoubleClick event handler**

1. Start File Explorer, navigate to the **E:\Mod02\Labfiles\Databases** folder, and then run

**SetupSchoolDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio and from the **E:\Mod02\Labfiles\Starter\Exercise 1** folder, open the **School.sln**

solution.

1. In the code for the **MainWindow.xaml.cs** window, in the **studentsList\_KeyDown** event, locate the code for editing student details which is in the **case Key.Enter** block.
2. Copy the code in this block to the clipboard and then paste it into the

**StudentsList\_MouseDoubleClick** method.

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

1. Build the solution and resolve any compilation errors.
2. Change **Kevin Liu’s** last name to **Cook** by pressing Enter in the main application window.
3. Verify that the updated data is copied back to the students list and that the **Save Changes** button is now enabled.
4. Change **George Li**’s name to **Darren Parker** by double-clicking on his row in the main application window.
5. Verify that the updated data is copied back to the student list.
6. 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. Using the results of the analysis in the **Code Clone Analysis Results** window, refactor the duplicated code into a method called **editStudent** that takes a **Student** as a parameter.
4. Call this method from the **studentsList\_MouseDoubleClick** and **studentsList\_KeyDown** methods.

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

1. Refactor the code in the **case Key.Insert** code block in the **studentsList\_KeyDown** method into a method called **addNewStudent** that takes no parameters.
2. Call this method from the **case Key.Insert** code block in the **studentsList\_KeyDown** method.
3. Refactor the code in the **case Key.Delete** code block in the **studentsList\_KeyDown** method into a method called **removeStudent** that takes a **Student** as a parameter.
4. Call this method from the case **Key.Delete** code block in the **studentsList\_KeyDown** method.

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Add a new student by pressing Insert to display the New Student for Class 3C window, and verify that it contains no data.
4. Enter details for **Dominik Dubicki**, whose date of birth is **02/03/2006**, and verify that the new student is added to the students list.
5. Delete the student **Run Liu** and verify that the prompt window appears and the student is removed from the student list.
6. Close the application.

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

1. Add a breakpoint at the start of the **switch** statement in the **studentsList\_KeyDown** method.
2. Debug the application.
3. Edit the row for **Kevin Liu** by pressing Enter.
4. Step over the code, watching the **Call Stack** window and **Locals** window, until you reach the

**editStudent** method call, and then step into that method.

1. Step out of the **editStudent** method.
2. Cancel editing the student’s details, and then continue debugging.
3. Add a new student by pressing Insert.
4. Step over the code until you reach the **addNewStudent** method call, and then step into that method.
5. Step out of the **addNewStudent** method.
6. Cancel adding a new student, and then continue debugging.
7. Delete the row for **George Li** by pressing Delete.
8. Step over the code until you reach the **removeStudent** method call, and then step into that method.
9. Step out of the **removeStudent** method.
10. Cancel deleting the student.
11. Stop debugging the application.
12. In Visual Studio, delete all breakpoints and then close the solution.

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

###### Exercise 2: Validating Student Information

**Scenario**

In this exercise, you will write code that validates the information that a user enters for a student.

Up until this point, almost anything can be entered as student data, and fields can be left blank. This means, for example, that a student could be added to the student list with no last name or with an invalid date of birth.

You will write code to check that when adding or editing a student, the first name and last name fields for the student contain data. You will also write code to check that the date of birth entered is a valid date and that the student is at least five years old. Finally, you will run the application and test your validation code.

The main tasks for this exercise are as follows:

1. Run the application and observe that student details that are not valid can be entered.
2. Add code to validate the first name and last name fields.
3. Add code to validate the date of birth.
4. Run the application and verify that student information is now validated correctly.

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

1. In Visual Studio, from the **E:\Mod02\Labfiles\Starter\Exercise 2** folder, open the **School.sln**

solution.

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Press Insert to display the new student window.
4. Leave the **First Name** and **Last Name** boxes empty, and type **10/06/3012** in the **Date of Birth** box.
5. Click **OK** and verify that a new row has been added to the student list, containing a blank first name, blank last name, and a negative age.
6. Close the application.

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

1. In the **ok\_Click** method in **StudentForm.xaml.cs** code, add a statement to check if the First Name box is empty.
2. If it is empty, display a message box with a caption of **Error** containing the text **The student must have a first name**, and then exit the method.
3. In the **ok\_Click** method in **StudentForm.xaml.cs** code, add a statement to check if the **Last Name**

box is empty.

1. If it is empty, display a message box with a caption of **Error** containing the text **The student must have a last name**, and then exit the click method.

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

1. In the **ok\_Click** method in **StudentForm.xaml.cs** code, add a statement to check if the **Date of Birth**

box is empty.

1. If the entered date is invalid, display a message box with a caption of **Error** containing the text **The date of birth must be a valid date**, and then exit the method.
2. In the **ok\_Click** method in **StudentForm.xaml.cs** code, add a statement to calculate the student’s age in years, and check if the age is less than five years.
3. If the age is less than five years, display a message box with a caption of **Error** containing the text **The student must at least 5 years old**, and then exit the method. Use the following formula to calculate the age in years.

Age in years = age in days / 365.25

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Press Insert to display the new student window.
4. Leave the **First Name**, **Last Name**, and **Date of Birth** boxes empty.
5. Click **OK**, verify that an error message appears containing the text **The student must have a first name**, then close the error message box.
6. Type **Darren** into the **First Name** box, and then click **OK**.
7. Verify that an error message appears containing the text **The student must have a last name**, and then close the error message box.
8. Type **Parker** into the **Last Name** box, and then click **OK**.
9. Verify that an error message appears containing the text **The date of birth must be a valid date**, and then close the error message box.
10. Type **10/06/3012** into the **Date of Birth** box, and then click **OK**.
11. Verify that an error message appears containing the text **The student must be at least 5 years old**, and then close the error message box.
12. Amend the date to **10/06/2006**, click **OK**, and then verify that Darren Parker is added to the student list with an age appropriate to the current date.
13. Close the application.
14. In Visual Studio, close the solution.

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

###### Exercise 3: Saving Changes to the Class List

**Scenario**

In this exercise, you will write code that saves changes in the student list to the database.

Every time the user closes and opens the application, they are presented with the original student list as it existed when they first ran the application, regardless of any changes they may have made. You will write code to save changes back to the database when the user clicks the **Save Changes** button. You will then add exception handling code to catch concurrency, update, and general exceptions, and handle the exceptions gracefully. Finally, you will run your application and verify that changes you make to student data are persisted between application sessions.

The main tasks for this exercise are as follows:

1. Verify that data changes are not persisted to the database.
2. Add code to save changes back to the database.
3. Add exception handling to the code to catch concurrency, update, and general exceptions.
4. Run the application and verify that data changes are persisted to the database.

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

1. In Visual Studio, from the **E:\Mod02\Labfiles\Starter\Exercise 3** folder, open the **School.sln**

solution.

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Change **Kevin Liu’s** last name to **Cook** by pressing Enter in the main application window.
4. Verify that the updated data is copied to the student list and that the **Save Changes** button is enabled.
5. Click **Save Changes**.
6. Delete the student **George Li**, and then click the **Save Changes** button.
7. Close the application.
8. Run the application again and verify that it displays the original list of students, without the changes that you just made.
9. Close the application.

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

1. In the **MainWindow.xaml.cs** code bring the **System.Data** and **System.Data.Objects** namespaces into scope.
2. Add code to perform the following tasks when a user clicks **Save Changes**:
   1. Call the **SaveChanges** method of the **schoolContext** object.
   2. Disable the **Save Changes** button.

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

1. Enclose the lines of code that call the **SaveChanges** method of the **schoolContext** object and disable the **Save Changes** button in a **try** block.
2. Below the **try** block, add a **catch** block to catch any **OptimisticConcurrencyException** exceptions that may occur.
3. In the **catch** block, add the following code:

// 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();

1. Add another **catch** block to catch any **UpdateException** exceptions that may occur, storing the exception in a variable named **uEx**.
2. In the **catch** block, add the following code:

// 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);

1. Add another **catch** block to catch any other type of exception that might occur, storing the exception in a variable named **ex**.
2. In the **catch** block, add the following code:

// 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. Build the solution and resolve any compilation errors.
2. Run the application.
3. Change **Kevin Liu’s** last name to **Cook** by pressing Enter in the main application window.
4. In the main application window, click **Save Changes**.
5. Delete the student **George Li** by pressing Delete.
6. Click **Save Changes** and note that the button is disabled when no changes are pending.
7. Close the application.
8. Run the application and verify that the changes you made to the student data have been saved to the database and are reflected in the student list.
9. Close the application
10. In Visual Studio, close the solution.

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

### Module Review and Takeaways

In this module, you learned how to create and use methods, and how to handle exceptions. You also learned how to use logging and tracing to record the details of any exceptions that occur.

**Review Question(s)**

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| The return type of a method forms part of a methods signature. |  |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **When using output parameters in a method signature, which one of the following statements is true?** | |
| Select the correct answer. | |
|  | You cannot return data by using a return statement in a method that use output parameters. |
|  | You can only use the type object when defining an output parameter. |
|  | You must assign a value to an output parameter before the method returns. |
|  | You define an output parameter by using the output keyword. |

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| A finally block enables you to run code in the event of an error occurring? |  |

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| **Trace** statements are active in both **Debug** and **Release** mode builds. |  |

**Lab 03: Writing the Code for the Grades Prototype Application**

**Scenario**

The School of Fine Arts has decided that they want to extend their basic class enrollment application to enable teachers to record the grades that students in their class have achieved for each subject, and to allow students to view their own grades. This functionality necessitates implementing application log on functionality to authenticate the user and to determine whether the user is a teacher or a student.

You decide to start by developing parts of a prototype application to test proof of concept and to obtain client feedback before embarking on the final application. The prototype application will use basic WPF views rather than separate forms for the user interface. These views have already been designed and you must add the code to navigate among them.

You also decide to begin by storing the user and grade information in basic structs, and to use a dummy data source in the application to test your log on functionality.

**Objectives**

After completing this lab, you will be able to:

1. Navigate among views.
2. Create and use collections of structs.
3. Handle events.

* Estimated Time: 90 minutes

**Exercise 1: Adding Navigation Logic to the Grades Prototype Application**

**Scenario**

In this exercise, you will add navigation logic to the **Grades Prototype** application.

First, you will examine the window and views in the application so that you are familiar with the existing structure of the application. You will define a public event handler named **LogonSuccess** that will be raised when a user successfully logs on to the application. You will add dummy code to the **Logon\_Click** event handler to store the username and role of the logged on user and raise the **LogonSuccess** event. Then you will add markup to the **LogonPage** XAML code to connect the **Logon** button to the **Logon\_Click** event handler. Next, you will add code to the **GotoLogon** method to display the logon view and to hide the other views. You will implement the **Logon\_Success** method to handle a successful log on by displaying the logged on views, and then you will add markup to the **MainWindow** XAML code to connect the **LogonSuccess** event to the **Logon\_Success** method. You will add code to the **MainWindow** to determine whether the user is a teacher or a student, display their name in the application, and display either the **StudentsPage** view for teachers or the **StudentProfile** view for students. You will then add code to the **StudentsPage** view that catches a student name being clicked and raises the **StudentSelected** event for that student and displays their student profile. Finally, you will run the application and verify that the appropriate views are displayed for students and teachers upon a successful log on.

The main tasks for this exercise are as follows:

1. Examine the window and views in the application.
2. Define the LogonSuccess event and add dummy code for the Logon\_Click event.
3. Add code to display the Log On view.
4. Add code to determine the type of user.
5. Handle the Student\_Click event.
6. Build and test the application.

 **Task 1: Examine the window and views in the application**

1. Start Visual Studio and open the **GradesPrototype.sln** solution from the

**E:\Mod03\Labfiles\Starter\Exercise 1** folder.

1. Build the solution.
2. Review **MainWindow.xaml**, which is the main window for the application and will host the following views:

* LogonPage.xaml
* StudentProfile.xaml
* StudentsPage.xaml

1. In the **Views** folder, review the **LogonPage.xaml**. 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.
2. In the **Views** folder, review the **StudentProfile.xaml**. 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.
3. In the **View** folder, review the **StudentsPage.xaml**. 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 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. In the **LogonPage.xaml.cs** class, in the **Event Members** region, define a public event handler named

**LogonSuccess**.

1. In the **Logon Validation** region, add an event handler for the **Logon\_Click** event, which takes an

**object** parameter named **sender** and a **RoutedEventArgs** parameter named **e**.

1. In the **Logon\_Click** event handler, add code to do the following:
2. Save the username and role that the user specified on the form in the relevant properties of the

**SessionContext** object.

1. If the user is a student, set the **CurrentStudent** property of the **SessionContext** object to the string

**Eric Gruber**.

1. Raise the **LogonSuccess** event.
2. In the **LogonPage.xaml** XAML editor, locate the definition of the **Log on** button.
3. Modify the definition to call the **Logon\_Click** method when the button is clicked.

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

1. In the **MainWindow.xaml.cs** code, locate the **GotoLogon** method, and then add code to display the

**logonPage** view and to hide the **studentsPage** and **studentProfile** views.

1. In the **Event Handlers** region, add code to the **Logon\_Success** method to handle a successful log on. This method should take an **object** parameter named **sender** and an **EventArgs** parameter named **e**. The method should update the display and show the data for the logged on user.
2. In the **MainWindow.xaml** XAML editor, locate the definition of the **LogonPage** page.
3. Modify the definition to call the **Logon\_Success** method for the **LogonSuccess** event.

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

1. In the **MainWindow.xaml.cs** file, in the **Refresh** method, add code to determine the type of user, display **Welcome <username>** in the **txtName** box, and then call either the **GotoStudentProfile** method (for students) or **GotoStudentsPage** method (for teachers) to display the appropriate view.
2. In the **GotoStudentProfile** method, add code to hide the **studentsPage** view, and then to display and refresh the **studentProfile** view.
3. In the **GotoStudentsPage** method, add code to hide the **studentProfile** view, and then add code to display and refresh the **studentsPage** view.
4. In the **StudentProfile.xaml.cs** file, in the **Refresh** method, add code to:
   1. Parse the student name into the first name and last name by using a regular expression as shown in the following example.

Match matchNames = Regex.Match(SessionContext.CurrentStudent, @"([^ ]+) ([^

]+)");

* 1. If a name is successfully parsed, display the first name and last name of the student in the appropriate boxes.
  2. If the user is a student, hide the **btnBack** button, or if the user is a teacher, display the **btnBack**

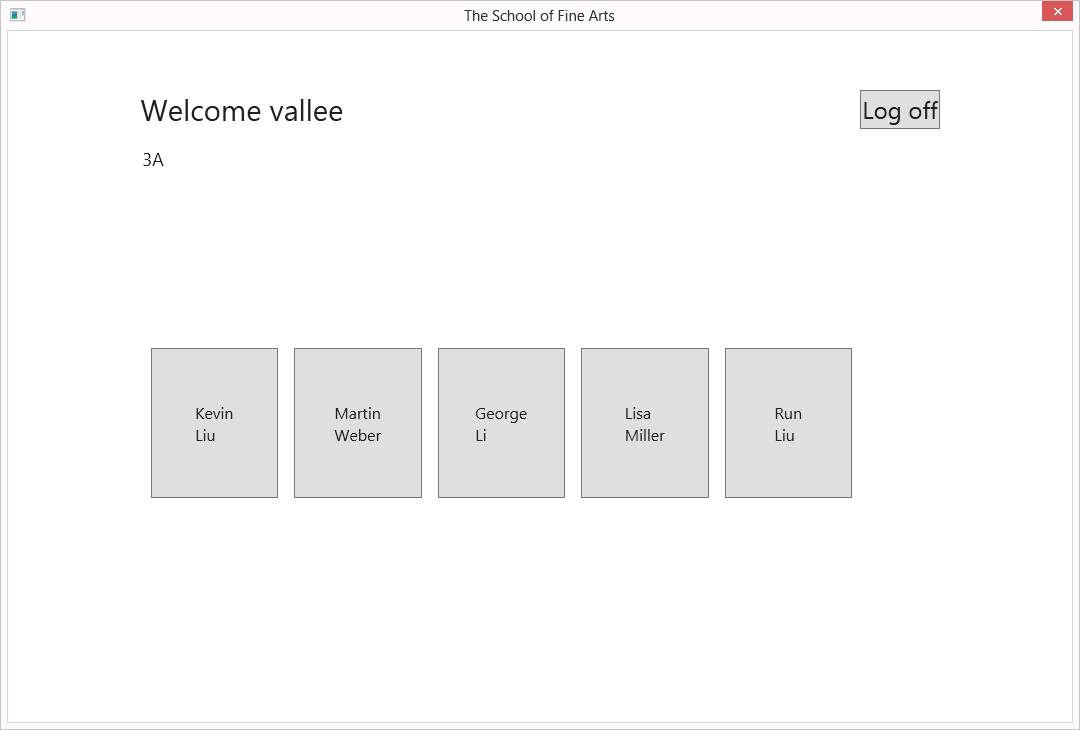
button.

 Task 5: Handle the Student\_Click event

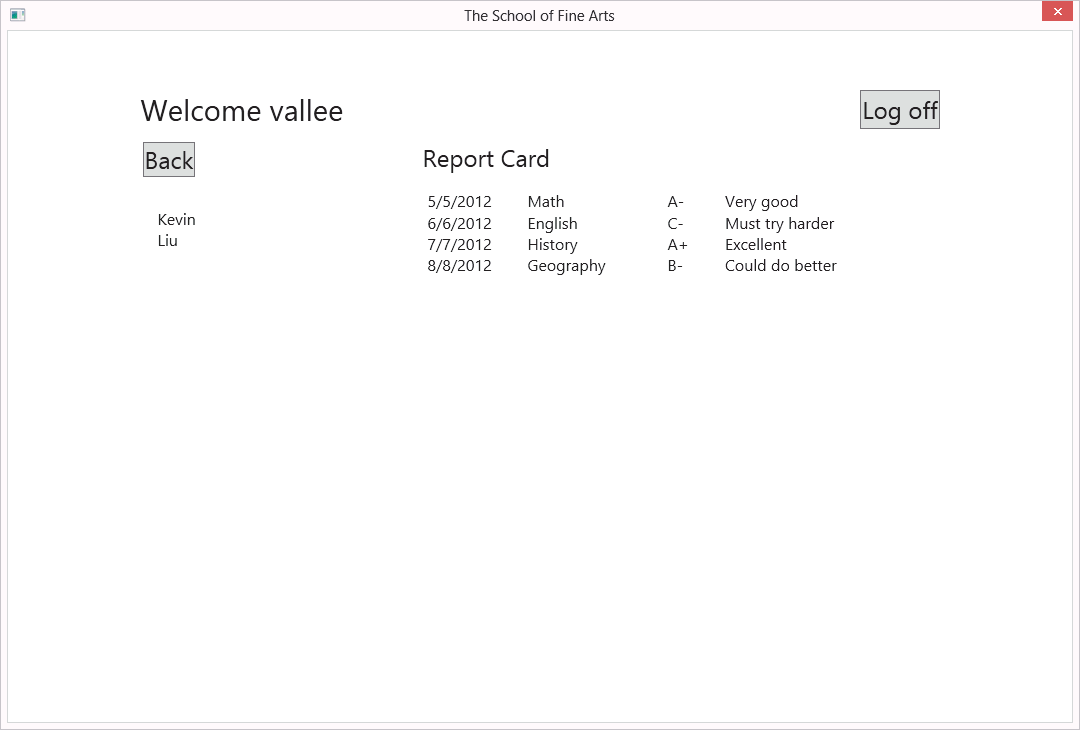
1. In the **StudentsPage.xaml.cs** file, in the **StudentsPage** class, locate the **Student\_Click** method.
2. Add code to this method to identify which student was clicked by using the **Tag** property of the button, and then raise the **StudentSelected** event, passing the student name as the second parameter.
3. In the **MainWindow.xaml.cs** file, in the **studentsPage\_StudentSelected** method, add code to set the **CurrentStudent** property of the **SessionContext** object to the student who was clicked by using the **Child** property of the **e** argument.
4. Call the **GotoStudentProfile** method.
5. In the **MainWindow.xaml** XAML editor, locate the definition of the **StudentsPage** page.
6. Modify the definition to call the **studentsPage\_StudentSelected** method for the **StudentSelected**

event.

 Task 6: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as the teacher, **vallee** with a password of **password**.
4. Verify that the application displays the **StudentPage** view. The Students page should look like this:

**FIGURE 3.1:THE STUDENTS PAGE**

1. Click **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**

1. Log off the application.
2. Log on as the student, **grubere**, with a password of **password**.
3. Verify that the application displays the student profile page for Eric Gruber.
4. Close the application and then close the 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

**Scenario**

In this exercise, you will define basic structs that will hold the teacher, student, and grade information for the application. You will then examine the dummy data source that the application uses to populate the collections in this exercise.

The main tasks for this exercise are as follows:

1. Define basic structs for holding Grade, Student, and Teacher information
2. Examine the dummy data source used to populate the collections

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

1. Open the **GradesPrototype** solution from the **E:\Mod03\Labfiles\Starter\Exercise 2** folder.
2. In the **Data** folder, open **Grade.cs**.
3. In the **GradesPrototype.Data** namespace, create a **struct** named **Grade** that contains the following fields:

* **StudentID** as an integer.
* **AssessmentDate** as a string.
* **SubjectName** as a string.
* **Assessment** as a string.
* **Comments** as a string.

1. In the **GradesPrototype.Data** namespace, create a **struct** named **Student** that contains the following fields:

* **StudentID** as an integer.
* **UserName** as a string.
* **Password** as a string.
* **TeacherID** as an integer.
* **FirstName** as a string.
* **LastName** as a string.

1. In the **GradesPrototype.Data** namespace, create a **struct** named **Teacher** that contains the following fields:

* **TeacherID** as an integer.
* **UserName** as a string.
* **Password** as a string.
* **FirstName** as a string.
* **LastName** as a string.
* **Class** as a string.

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

1. In the **Data** folder, in the **DataSource.cs** file, expand the **Sample Data** region.
2. Note how the **Teachers ArrayList** is populated with **Teacher** data, each containing **TeacherID**, **UserName**, **Password**, **FirstName**, **LastName**, and **Class** fields.
3. Note how the **Students ArrayList** is populated with **Student** data, each containing a **StudentID**, **UserName**, **Password**, **TeacherID**, **FirstName**, and **LastName** fields.
4. 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

**Scenario**

In this exercise, you will first define a public event handler named **LogonFailed** that will be raised when a user fails to log on successfully. You will add code to the **Logon\_Click** event handler to validate the username and password entered by the user against the Users collection in the **MainWindow** window. If the user is a teacher or a student, you will store their details in the global context and then raise the **LogonSuccess** event, but if the user is not validated, you will raise the **LogonFailed** event. You will handle log on failure in the **Logon\_Failed** method to display a message to the user and then you will add markup to the **MainWindow** XAML code to connect the **LogonFailed** event to the **Logon\_Failed** method. You will add code to the **StudentsPage** view to display students for the current teacher, and to display the details for a student when the user clicks their name. You will then use data binding to display the details and grades for the current student in the **StudentProfile** view, and to display only the **Back** button if the user is a teacher. Finally, you will run the application and verify that only valid users can log on and that valid users can see only data appropriate to their role.

The main tasks for this exercise are as follows:

1. Add the LogonFailed event.
2. Add the Logon\_Failed event handler.
3. Display the students for the current teacher.
4. Set the DataContext for the page.
5. Build and test the application.

 **Task 1: Add the LogonFailed event**

1. Open the **GradesPrototype** solution from the **E:\Mod03\Labfiles\Starter\Exercise 3** folder.
2. In the **LogonPage.xaml.cs** file, in the **Event Members** region, define a public event handler named

**LogonFailed**.

1. In the **Logon\_Click** event, add code to do the following:
2. Determine whether the user is a teacher by using a LINQ query to retrieve teachers with the same user name and password as the current user. If the LINQ query returns a result, then the user is a teacher.
3. If the user is a teacher, set the **UserID**, **UserRole**, **UserName**, and **CurrentTeacher** properties of the **SessionContext** object to the appropriate fields from the data source, and then raise the **LogonSuccess** event.
4. If the user is not a teacher, determine whether the user is a student by using a LINQ query to retrieve students with the same user name and password as the current user.
5. If the user is a student, set the **UserID**, **UserRole**, **UserName**, and **CurrentStudent** properties of the **SessionContext** object to the appropriate fields from the data source, and then raise the **LogonSuccess** event.
6. If the credentials do not match any teachers or students, raise the **LogonFailed** event.

 Task 2: Add the Logon\_Failed event handler

1. In the **MainWindow.xaml.cs** class, in the **Event Handlers** region, add an event handler for the **Logon\_Failed** event that takes an **object** parameter named **sender** and an **EventArgs** parameter named **e**.
2. In the **Logon\_Failed** event handler, add code to display an error message to the user.
3. In the **MainWindow.xaml** XAML editor, locate the definition of the **LogonPage** page.
4. Modify the definition to call the **Logon\_Failed** method for the **LogonFailed** event.
5. In the **MainWindow.xaml.cs** code, locate the **Refresh** method.
6. In the **case** statement for a student, add code to display the student name in **txtName** text box at the top of the page.
7. In the **case** statement for a teacher, add code to display the teacher name in the banner at the top of the page.

 Task 3: Display the students for the current teacher

1. In **StudentsPage.xaml** 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.

1. In the **StudentsPage.xaml.cs** code, locate the **Refresh** method, and then add code to do the following:
   1. Find all the students for the current teacher and store them in a new **ArrayList** object.
   2. Bind the collection to the **ItemsSource** property of the **list** control.
   3. Display the class name.
2. Locate the **Student\_Click** event and then add code to do the following:
   1. Identify which student was clicked by using the **Tag** property of the button.
   2. Find the details of that student by examining the **DataContext** of the button.
   3. Raise the **StudentSelected** event, passing the student as the second parameter.
   4. In the **StudentsPage\_StudentSelected** event handler, add code to set the **CurrentStudent** property of the **SessionContext** object to the student passed to the event by using the **Child** property of the **e** argument.

 Task 4: Set the DataContext for the page

1. In the **StudentProfile.xaml.cs** file, in the **Refresh** method, add code to display the details of the current student in the **studentName StackPanel** object and to display the **Back** button only if the user is a teacher.
2. In the **StudentProfile.xaml** XAML editor, locate the definition of the **firstName** text block.
3. Modify the definition to bind the **Text** property to the **FirstName** field.
4. Locate the definition of the **lastName** text block.
5. Modify the definition to bind the **Text** property to the **LastName** field.
6. In the **StudentProfile.xaml.cs** file, at the end of the **Refresh** method, add code to iterate the grades for the current student in **DataSource.Grades** list and then display them in the **studentGrades** control by using data binding.

 Task 5: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **parkerd** with a password of **password** and verify that the **Logon Failed** message box appears.
4. Log on as **vallee** with a password of **password** and verify that the Students page appears.
5. Click **Kevin Liu**, verify that the Student Profile page appears, and then log off.
6. Log on as **grubere** with a password of **password** and verify that the Student Profile page appears.
7. Close the application and then close the 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 Review and Takeaways

In this module, you have learned how to implement structs and enums, organize data into collections, and work with events and delegates.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You want to create a string property named CountryOfOrigin. You want to be able to read the property value from any code, but you should only be able to write to the property from within the containing struct. How should you declare the property?** | |
| Select the correct answer. | |
|  | public string CountryOfOrigin { get; set; } |
|  | public string CountryOfOrigin { get; } |
|  | public string CountryOfOrigin { set; } |
|  | public string CountryOfOrigin { get; private set; } |
|  | private string CountryOfOrigin { get; set; } |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You want to create a collection to store coffee recipes. You must be able to retrieve each coffee recipe by providing the name of the coffee. Both the name of the coffee and the coffee recipe will be stored as strings. You also need to be able to retrieve coffee recipes by providing an integer index. Which collection class should you use?** | |
| Select the correct answer. | |
|  | ArrayList |
|  | Hashtable |
|  | SortedList |
|  | NameValueCollection |
|  | StringDictionary |

Test Your Knowledge

**Question**

**You are creating a method to handle an event named OutOfBeans. The delegate for the event is as follows:**

**public delegate void OutOfBeansHandler(Coffee coffee, EventArgs args); Which of the following methods should you use to subscribe to the event?**

|  |  |
| --- | --- |
| **Question** | |
| Select the correct answer. | |
|  | public void HandleOutOfBeans(delegate OutOfBeansHandler)  {  } |
|  | public void HandleOutOfBeans(Coffee c, EventArgs e)  {  } |
|  | public Coffee HandleOutOfBeans(EventArgs e) |
|  | public Coffee HandleOutOfBeans(Coffee coffee, EventArgs args) |
|  | public void HandleOutOfBeans(Coffee c, EventArgs e) |

**Lab 04: Adding Data Validation and Type-Safety to the Application**

**Scenario**

Now that the user interface navigation features are working, you decide to replace the simple structs with classes to make your application more efficient and straightforward.

You have also been asked to include validation logic in the application to ensure that when a user adds grades to a student, that the data is valid before it is written to the database. You decide to create a unit test project that will perform tests against the required validation for different grade scenarios.

Teachers who have seen the application have expressed concern that the students in their classes are displayed in a random order. You decide to use the IComparable interface to enable them to be displayed in alphabetical order.

Finally, you have been asked to add functionality to the application to enable teachers to add students to and remove students from a class, and to add student grades to the database.

**Objectives**

After completing this lab, you will be able to:

1. Create classes.
2. Write data validation code and unit tests.
3. Implement the **IComparable** interface.
4. Use generic collections.

* Estimated Time: 75 minutes

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

**Scenario**

In this exercise, you will convert the existing Teacher, Student, and Grade structs into classes. This will enable you to implement the additional functionality required for each class, such as adding constructors, properties, and methods. In the Teacher and Student classes, you will make the password property write- only, add the **VerifyPassword** method, and then define their respective constructors. You will also modify the **Logon\_Click** method to use the **VerifyPassword** method to verify passwords when a user logs on.

Finally, you will run the application and verify that it still functions correctly, allowing a student or a teacher to log on.

The main tasks for this exercise are as follows:

1. Convert the Grades struct into a class.
2. Convert the Students and Teachers structs into classes.
3. Use the VerifyPassword method to verify the password when a user logs in.
4. Build and run the application, and verify that a teacher or student can still log on.

 **Task 1: Convert the Grades struct into a class**

1. Start Visual Studio and open the **GradesPrototype.sln** solution from the

**E:\Mod04\Labfiles\Starter\Exercise 1** folder.

1. In the **Data** folder, in **Grade.cs**, convert the **Grade** struct into a class.
2. Define a class constructor that takes the following parameters and uses them to populate the public properties of the class:
   * studentID
   * assessmentDate
   * subject
   * assessment
   * comments
3. Define a default class constructor that takes no parameters and assigns the following default values to the public properties:
   * Student ID: 0
   * AssessmentDate: the current date
   * SubjectName: Math
   * Assessment: A
   * Comments: an empty string

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

1. Convert the **Students** struct into a class.
2. Create a write-only password property that generates a new GUID for the default value.
3. Create a **VerifyPassword** method that uses the **String.Compare** method to check that the password passed to it as a parameter matches the stored password.

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

1. Define a class constructor that takes the following parameters and uses them to populate the public properties of the class:
   * studentID
   * userName
   * password
   * firstName
   * lastName
   * teacherID
2. Define a default class constructor that takes no parameters and assigns the following default values to the public properties:
   * Student ID: 0
   * UserName: an empty string
   * Password: an empty string
   * FirstName: an empty string
   * LastName: an empty string
   * TeacherID: 0
3. Convert the **Teachers** struct into a class.
4. Create a write-only password property that generates a new GUID for a default value and a **VerifyPassword** method that uses the **String.Compare** method to check that the password passed to it as a parameter matches the stored password.
5. Define a class constructor that takes the following parameters and uses them to populate the public properties of the class:
   * teacherID
   * userName
   * password
   * firstName
   * lastName
   * className
6. Define a default class constructor that takes no parameters and assigns the following default values to the public properties:
   * TeacherID: 0
   * UserName: an empty string
   * Password: an empty string
   * FirstName: an empty string
   * LastName: an empty string
   * Class: an empty string

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

1. In the **Views** folder, in the **LogonPage.xaml.cs** code, modify the code in the **Logon\_Click** method to call the **VerifyPassword** method to verify the teacher’s password.
2. Modify the code to check whether **teacher** is null before examining the **UserName** property.
3. In the **Student** class, modify the code in the **Logon\_Click** method to use the **VerifyPassword**

method to verify the student’s password.

1. Modify the code to check whether **student** is null before examining the **UserName** property.

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

1. Build the solution and resolve any compilation errors.
2. Log in as **vallee** with a password of **password**.
3. Verify that you can log on as a teacher.
4. Log off from the application.
5. Log in as **grubere** with a password of **password**.
6. Verify that you can log on as a student.
7. Close the application.
8. In Visual Studio, close the 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

**Scenario**

In this exercise, you will define a public list of strings called **Subjects** to hold the names of each subject that can be assessed and then populate it with valid subject names. You will then add validation logic to the **Grade** class to ensure that the subject name appears in the list you created and that the assessment date and assessment grade contain allowed values. Finally, you will create a unit test project to verify that your validation code functions as expected.

The main tasks for this exercise are as follows:

1. Create a list of valid subject names.
2. Add validation logic to the Grade class to check the data entered by the user.
3. Add a unit test to verify that the validations defined for the Grade class functions as expected.

 **Task 1: Create a list of valid subject names**

1. In Visual Studio, from the **E:\Mod04\Labfiles\Starter\Exercise 2** folder, open the

**GradesPrototype.sln** solution.

1. In the **Data** folder, in the **DataSource** class, define a generic **List** collection to hold the names of valid subjects.
2. In the **CreateData** method in that class, populate the list with the following subject names:
   * 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 **Data** folder, in the **Grade.cs** code, add validation code to the **AssessmentDate** property to ensure that the following requirements have been met:

* Verify that the user has provided a valid date.
* Check that the date is no later than the current date. If it is, throw an

**ArgumentOutOfRangeException** exception.

* If the date is valid, then save it in the appropriate format.
* If the date is not in a valid format, throw an **ArgumentException** exception.

1. Add validation code to the **Subject** property to ensure that the following requirements are met:

* Check that the specified subject is in the list that you have defined.
* If the subject is valid, store the subject name.
* If the subject is not valid, then throw an **ArgumentException** exception.

1. Add validation to the **Assessment** property to ensure that the following requirements are met:

* Verify that the grade is in the range **A+** to **E-** by using the following regular expression.

Match matchGrade = Regex.Match(value, @"[A-E][+-]?$");

* If the grade is not valid, then throw an **ArgumentOutOfRangeException** exception.

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

1. Add a **Unit Test Project** called **GradesTest** to the solution and reference the **GradesPrototype**

project from it.

1. In the **UnitTest1** class, define the following tests and support methods:

* Init: to call the **CreateData** method to initialize the **DataSource**
* TestValidGrade: to check that valid data passes the validation logic successfully
* TestBadDate: to check that dates in the future are not valid
* TestDateNotRecognized: to check that non-dates are not valid
* TestBadAssessment: to check that assessment values outside the permitted range are not valid
* TestBadSubject: to check that subject names not in the list are not valid

1. Build the solution and resolve any compilation errors.
2. Run all tests and verify that they all pass.
3. Close **Test Explorer** and then close the solution.

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

###### Exercise 3: Displaying Students in Name Order

**Scenario**

In this exercise, you will write code to display the students in alphabetical order of last name and then first name.

The application currently displays students in no specific order when logged on as a teacher, but you now want them to be displayed in alphabetical order of last name and first name. To achieve this, you decide that the Student class should implement the **IComparable<>** interface to enable comparison of student data. You can then add code to the **CompareTo** method in the **Student** class, enabling students to be sorted based on their last name and first name. Currently, Students are stored in a non-type-safe **ArrayList** collection. You decide to modify this so they are stored in a type-safe **List** collection. Finally, you will sort the Students data and then run the application to verify that the students are retrieved and displayed in alphabetical order of their last name and first name.

The main tasks for this exercise are as follows:

1. Run the application and verify that the students are not displayed in any specific order when logged on as a teacher.
2. Implement the IComparable<Student> interface to enable comparison of students.
3. Change the Students ArrayList collection into a List<Student> collection.
4. Sort the data in the Students collection.
5. Verify that Students are retrieved and displayed in order of their first name and last name.

 **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, from the **E:\Mod04\Labfiles\Starter\Exercise 3** folder, open the

**GradesPrototype.sln** solution.

1. Build the solution and resolve any compilation errors.
2. Log in as **vallee** with a password of **password**. Verify that the students are not displayed in any specific order.
3. Close the application.

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

1. In the **Grade.cs** code window, locate the **Student** class definition, and modify it to implement the

**IComparable<Student>** interface.

1. In the **CompareTo** method, concatenate the **FirstName** and **LastName** properties of each of the students being compared, and then use the **String.Compare** method to establish the order that they should be displayed in the list.

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

1. In the **Data** folder, in the **DataSource.cs** code, in the **DataSource** class, modify the **Students ArrayList** collection to be a generic **List** collection.
2. In the **CreateData** method, modify the creation of the **Students** collection to create a new generic

**List** collection.

 Task 4: Sort the data in the Students collection

1. In **MainWindow.xaml.cs**, in the **MainWindow** constructor, after calling the **DataSource.CreateData**

method, add a method call to sort the data in the **Students** collection.

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

1. Build the solution and resolve any compilation errors.
2. Run the application without debugging.
3. Log in as **vallee** with a password of **password**.
4. Verify that the students are displayed in order of ascending last name.
5. Close the application.
6. In Visual Studio, close the 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

**Scenario**

In this exercise, you will write code that enables a teacher to add a student and then enroll them in a class. This will be implemented as two separate steps, because a teacher may want to add a student before knowing which class they will be enrolled in. You will also enable a teacher to remove a student from a class. When adding or removing a student, you will display a prompt to confirm that the teacher wants to perform the action.

To enroll a student in a class or remove them from a class, you modify the **TeacherID** property of that student. The application now includes the **AssignStudentDialog** window, which displays a list of students who are not assigned to a class. You need to add code to this window to assign a student to the teacher’s class and to update the list of students as appropriate. You also need to add code to remove a student from a class and to enable teachers to add grades to their students. After a student has been added to the database, that student will be able to log on to view their own grades.

The main tasks for this exercise are as follows:

1. Change the Teachers and Grades collections to be generic List collections.
2. Add the EnrollInClass and RemoveFromClass methods for the Teacher class.
3. Add code to enroll a student in a teacher’s class.
4. Add code to enable a teacher to remove the student from the assigned class.
5. Add code to enable a teacher to add a grade to a student.
6. Run the application and verify that students can be added to and removed from classes, and that grades can be added to students.

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

1. In Visual Studio, from the **E:\Mod04\Labfiles\Starter\Exercise 4** folder, open the

**GradesPrototype.sln** solution.

1. In the **Data** folder, in the **DataSource.cs** code, change the **Teachers** collection to be a generic **List**

collection.

1. Change the **Grades** collection to be a generic **List** collection.
2. In the **CreateData** method, modify the creation of the **Teachers** collection to create a new generic

**List** collection.

1. In the **CreateData** method, modify the creation of the **Grades** collection to create a new generic **List**

collection.

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

1. In the **Data** folder, in the **Grade.cs** code, in the **Teacher** class, implement the **EnrollInClass** method as follows:

* Verify that the student is not already enrolled in another class.
* If the student is not in another class, set the **TeacherID** property of the student to the current

**TeacherID**.

* If the Student is in another class, throw an **ArgumentException** exception to show that the student is already assigned to a class.

1. In the **Teacher** class, add code to the **RemoveFromClass** method as follows:

* Verify that the **Student** is actually assigned to the class for the given teacher.
* If the student is part of the class, reset the **TeacherID** property of the student to zero.
* If the student is not part of the class, throw an **ArgumentException** exception to show that the student is not assigned to this class.

1. In the **Teacher** class, implement the **AddGrade** method as follows:

* Verify that the **Grade** object passed to the method does not belong to another student.
* If it does not belong to another student, add the grade to the student’s record by setting the

**StudentID** property of the **Grade** object.

* If it does belong to another student, throw an **ArgumentException** exception to show that the grade belongs to a different student.

 Task 3: Add code to enroll a student in a teacher’s class

1. In the **Controls** folder, in the **AssignStudentDialog.xaml.cs** code, in the **Student\_Click** method, write code as follows:

* Add a **try** block.
* Inside the **try** block, determine which student the user clicked by using the **Tag** property of the

**studentClicked** button.

* Find this student in the **Students** collection and prompt the user to confirm that they wish to add the student to their class.
* If the user confirms this, add the student to the class by calling the **EnrollInClass** method, and then refresh the display.
* Add a **catch** block to display a message to the user if an exception occurs.

1. In the **Refresh** method, write code as follows:

* Find all unassigned students with a **TeacherID** of zero.
* If there are no unassigned students, show the **txtMessage** box and hide the **list** control.
* If there are unassigned students, hide the **txtMessage** box and show the **list** control bound to the list of unassigned students.

1. In the **StudentsPage.xaml.cs** code, in the **EnrollStudent\_Click** method, add code to use the **AssignStudentDialog** to display the unassigned students. Then refresh the display to show any newly enrolled students.

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

1. In the **StudentProfile.xaml.cs** code, in the **Remove\_Click** method, write code as follows:

* Detect if the user is a teacher. If they are not, exit the method.
* Add a **try** block.
* Inside the **try** block, display a message box to prompt the user to confirm that the current student should be removed from their class.
* If the user confirms, call the **RemoveFromClass** method of the current teacher to remove this student from their class, and then return to the previous page.
* Add a **catch** block to display a message to the user if an exception occurs.

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

1. In the **StudentProfile.xaml.cs** code, in the **AddGrade\_Click** method, write code to add a grade to a student as follows:

* Detect if the user is a teacher. If they are not, exit the method.
* Add a **try** block.
* Inside the **try** block, use the **GradeDialog** to get the details of the assessment grade and use them to create a new **Grade** object.
* Save the grade to the list of grades.
* Add the grade to the current student.
* Refresh the display so that the new grade appears.
* Add a **catch** block to display a message to the user if an exception occurs.

 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. Build the solution and resolve any compilation errors.
2. Log in as **vallee** with a password of **password**.
3. Add a student with the following details:

* First name: **Darren**
* Last name: **Parker**
* Password: **password**

1. Verify that Darren Parker is added to the student list.
2. Remove the student Kevin Liu from the student list.
3. For the student Darren Parker, add a new grade by using the following information:

* Date: current date
* Subject: **English**
* Assessment: **B**
* Comments: **Good**

1. Verify that the grade information is added to the Report Card.
2. Log on as the student **Darren Parker** and verify that the grade information from the previous steps is displayed in the Report Card.

 **Note:** A username is generated by taking a user’s last name and the first initial of their first name. The username for Darren Parker is parkerd.

1. Close the application.
2. In Visual Studio, close the 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 Review and Takeaways

In this module, you have learned how to work with classes, interfaces, and generic collections in Visual C#.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Which of the following types is a reference type?** | |
| Select the correct answer. | |
|  | Boolean |
|  | Byte |
|  | Decimal |
|  | Int32 |
|  | Object |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Which of the following types of member CANNOT be included in an interface?** | |
| Select the correct answer. | |
|  | Events |
|  | Fields |
|  | Indexers |
|  | Methods |
|  | Properties |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You want to create a custom generic class. The class will consist of a linear collection of values, and will enable developers to queue items from either end of the collection. Which of the following should your class declaration resemble?** | |
| Select the correct answer. | |
|  | public class DoubleEndedQueue<T> : IEnumerable<T> |
|  | public class DoubleEndedQueue<T> : ICollection<T> |

|  |  |
| --- | --- |
| **Question** | |
|  | public class DoubleEndedQueue<T> : IList<T> |
|  | public class DoubleEndedQueue<T> : IList<T>, IEnumerable<T> |
|  | public class DoubleEndedQueue<T> : IDictionary<TKey,TValue> |

**Lab 05: Refactoring Common Functionality into the User Class**

**Scenario**

You have noticed that the Student and Teacher classes in the Grades application contain some duplicated functionality. To make the application more maintainable, you decide to refactor this common functionality to remove the duplication.

You are also concerned about security. Teachers and students all require a password, but it is important to maintain confidentiality and at the same time ensure that students (who are children) do not have to remember long and complex passwords. You decide to implement different password policies for teachers and students; teachers' passwords must be stronger and more difficult to guess than student passwords.

Also, you have been asked to update the application to limit the number of students that can be added to a class. You decide to add code that throws a custom exception if a user tries to enroll a student in a class that is already at capacity.

**Objectives**

After completing this lab, you will be able to:

1. Use inheritance to factor common functionality into a base class.
2. Implement polymorphism by using an abstract method.
3. Create a custom exception class.

* Estimated Time: 60 minutes

**Exercise 1: Creating and Inheriting from the User Base Class**

**Scenario**

In this exercise, you will create an abstract base class called **User** that contains the **UserName** and **Password** properties, and the **VerifyPassword** method that is common to the **Student** and **Teacher** classes.

You will modify the definitions of the **Student** and **Teacher** classes to inherit from the **User** class, and remove the **UserName** and **Password** properties and the **VerifyPassword** method from these classes. Finally, you will build and run the application without making any other changes to the application, and then verify that it still works correctly.

The main tasks for this exercise are as follows:

1. Create the User abstract base class.
2. Modify the Student and Teacher classes to inherit from the User class.
3. Run the application and test the log on functionality.

 **Task 1: Create the User abstract base class**

1. Start Visual Studio and open the **GradesPrototype.sln** solution from the

**E:\Mod05\Labfiles\Starter\Exercise 1** folder.

1. In the **Grade.cs** file in the **Data** folder, create a new abstract class called **User**.
2. Add the **UserName** and **Password** properties to the **User** class. You can copy the code for the **UserName** and **Password** properties and the private **\_password** field from either the **Student** class or the **Teacher** class.
3. Add the **VerifyPassword** method to the **User** class. You can copy the code for the **VerifyPassword**

method from either the **Student** class or the **Teacher** class.

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

1. In the **Grade.cs** file, modify the **Student** class to inherit from the **User** class. Remove the **UserName** and **Password** properties, and the private **\_password** field. Also remove the **VerifyPassword** method from the **Student** class.
2. Modify the **Teacher** class to inherit from the **User** class. Remove the **UserName** and **Password** properties, and the private **\_password** field. Also remove the **VerifyPassword** method from the **Teacher** class.

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** (a teacher) with a password of **password**.
4. Verify that a list of students for this teacher appears in **The School of Fine Arts** window.
5. Select student Kevin Liu and verify that the report card listing the grades for this student appears.
6. Log off and then log on as **liuk** (a student) with a password of **password**.
7. Verify that the report card for Kevin Liu is displayed again.
8. Log off and then close the application
9. In Visual Studio, close the 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

**Scenario**

In this exercise, you will add an abstract method called **SetPassword** to the **User** class. In the **Teacher** and **Student** classes you will implement the **SetPassword** method. This method will set the password for the user (either a teacher or a student). The **SetPassword** method for a teacher will check that the password is at least eight characters long and contains at least two numeric characters. The **SetPassword** method for a student will check that the password is at least six characters long. If the password meets these requirements, it is set and the method will return true, otherwise it will return false. You will then modify the set accessor of the **Password** property in the **User** class to call the **SetPassword** method to change the user's password. Next, you will integrate this feature into the user interface of the application to enable a user to change their password. Finally, you will build and run the application to test the password functionality.

The main tasks for this exercise are as follows:

1. Define the SetPassword abstract method.
2. Implement the SetPassword method in the Student and Teacher classes.
3. Set the password for a new student.
4. Change the password for an existing user.
5. Run the application and test the change password functionality.

 **Task 1: Define the SetPassword abstract method**

1. In Visual Studio, open the **GradesPrototype.sln** solution from the

**E:\Mod05\Labfiles\Starter\Exercise 2** folder.

1. In the **Data** folder, in the **Grade.cs** file, in the **User** class, define a public abstract method called **SetPassword**. This method should take a string parameter containing the password and return a Boolean value indicating whether the password has been set successfully.
2. In the **User** class, modify the set accessor of the **Password** property to call the **SetPassword** method rather than directly writing to the **\_password** field. Throw an **ArgumentException** exception if the **SetPassword** method returns false.

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

1. In the **User** class, make the **\_password** field protected rather than private; it needs to be accessible in the **Student** and **Teacher** classes.
2. In the **Student** class, implement the **SetPassword** method. The method should verify that the password specified as the parameter is at least six characters long. If the password is of sufficient length, then populate the **\_password** field and return true; otherwise, return false.
3. In the **Teacher** class, implement the **SetPassword** method. The method should verify that the password specified as the parameter is at least eight characters long and contains at least two numeric characters by using the following regular expression.

Match numericMatch = Regex.Match(pwd, @".\*[0-9]+.\*[0-9]+.\*");

1. If the password is of sufficient complexity, then populate the **\_password** field and return true; otherwise, return false.

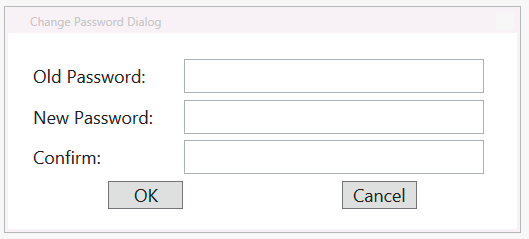
 Task 3: Set the password for a new student

1. In the code for the **StudentsPage** view, locate the **NewStudent\_Click** method. This method runs when a teacher creates a new student.
2. In this method, modify the statement that sets the password for the new student to call the **SetPassword** method. If the password is not sufficiently complex and the method returns **false**, throw an **Exception** with a suitable error message.

 Task 4: Change the password for an existing user

1. Build the solution.
2. In the XAML definition of the **MainWindow** window, find the definition of the **Change Password** button. When the application runs, this button appears at the top of the page. If the user clicks this button, the **ChangePassword\_Click** method runs.
3. In the **MainWindow.xaml.cs** file, review the **ChangePassword\_Click** method. This method displays a dialog called **ChangePasswordDialog** that enables a user to change their password.
4. In the Controls folder, review both the UI and the XAML code for the **ChangePasswordDialog.xaml** window. This window contains three text boxes that prompt the user to provide their old password, enter a new password, and confirm the new password. When the user clicks **OK** the new password is set.

The **ChangePasswordDialog** window looks like this:



**FIGURE 5.1:THE CHANGEPASSWORDDIALOG WINDOW**

1. Examine the code in the **ok\_Click** method in the **ChangePassword.xaml.cs** file. This method runs when the user clicks **OK** in the Change Password dialog box. Currently, this method does nothing.
2. Implement the logic for the **ok\_Click** method:
   1. Get the details for the current user. You can use the **SessionContext.UserRole** property to determine if the current user is a teacher or a student and then use either the **CurrentTeacher** property or **CurrentStudent** property of the **SessionContext** object to access the user details.
   2. Verify that the old password specified in the dialog is correct by using the **VerifyPassword** method of the User class. If the password is incorrect, display a message and return from the method without changing the password.
   3. Verify that the new password and confirm password text boxes in the dialog contain the same value. If they are different, display a message and return from the method without changing the password.
   4. Set the password by using the **SetPassword** method of the current user. If this method returns false, display a message and return without changing the password.

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** (a teacher) with a password of **password99**.

 **Note:** The passwords for all teachers have changed to **password99** to ensure that they meet the complexity requirements. The password for all students is still **password.**

1. Change the password for the current user. First try setting it to a password that is insufficiently complex, and then change it to **password101**.
2. Log out and then log back in again as **vallee**, and verify that the password has been changed to

**password101**.

1. Create a new student and verify that the student password must be at least six characters long. Use the **Enroll Student** feature to verify that the student is successfully created.
2. Log off and then close the application.
3. In Visual Studio, close the 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

**Scenario**

In this exercise, you will create a new custom exception class called **ClassFullException**. You will modify the **EnrollInClass** method of the **Teacher** class to raise this exception if too many students are added to a teacher's class. You will update the application to catch this exception, and then you will build and run the application to test this feature.

The main tasks for this exercise are as follows:

1. Implement the ClassFullException class.
2. Throw and catch the ClassFullException.
3. Build and test the solution.

 **Task 1: Implement the ClassFullException class**

1. In Visual Studio, open the **GradesPrototype.sln** solution from the

**E:\Mod05\Labfiles\Starter\Exercise 3** folder.

1. Review the **ClassFullException** class in the **Services** folder. Notice that the class inherits from the

**Exception** class, but most of the functionality has yet to be defined.

1. Add a private string field called **\_className** and a public virtual read-only string property called **ClassName** to the **ClassFullException** class. This property should return the value in the **\_className** field. The **\_className** field will hold the name of the class that is full when the exception is raised.
2. Add a default public constructor to the **ClassFullException** class. This constructor should simply delegate its responsibilities to the equivalent constructor in the **Exception** class.
3. Add a public constructor to the **ClassFullException** class that takes a string parameter containing the exception message. This constructor should also delegate its functionality to the equivalent constructor in the **Exception** class.
4. Add a public constructor to the **ClassFullException** that takes a string parameter holding the exception message and an **Exception** object containing an inner exception. Like the previous constructors, this constructor should delegate its functionality to the equivalent constructor in the **Exception** class.
5. Add a public custom constructor that takes the exception message and the name of the class that is full as parameters. Invoke the **Exception** constructor with the exception message, but store the name of the class in the **\_className** field.
6. Add a public custom constructor that takes the exception message, the name of the class that is full, and an **Exception** object containing an inner exception as parameters. Invoke the **Exception** constructor with the exception message and the inner exception, but store the name of the class in the **\_className** field.

 Task 2: Throw and catch the ClassFullException

1. In the **Teacher** class, add a private constant integer field called **MAX\_CLASS\_SIZE** and initialize it with the value 8. This field specifies the maximum class size for a teacher.
2. In the **EnrollInClass** method of the **Teacher** class, if the current number of students is already equal to the value in **MAX\_CLASS\_SIZE**, then throw a **ClassFullException** with a suitable message and the name of the class that is full (the name of the class is available in the **Class** property of the Teacher).
3. Students are enrolled in a class by using the **AssignStudentDialog** window. Open the **AssignStudentDialog.xaml.cs** file and review the code in the **Student\_Click** method. This method runs when the user selects a student to add to a class. Notice that the **try** block in this method includes the following statement:

SessionContext.CurrentTeacher.EnrollInClass(student);

1. Add a **catch** handler after the **try** block that catches the **ClassFullException**. In this **catch** handler, display a suitable message that includes the exception message and class name from the exception.

 Task 3: Build and test the solution

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** (a teacher) with a password of **password99**.
4. Create four new students.
5. Try to enroll all four students in the class for Esther Valle; this teacher currently has five students, so attempting to add the final student should fail with a **ClassFullException** exception.
6. Log off and then close the application.
7. In Visual Studio, close the 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 Review and Takeaways

In this module, you have learned how to use inheritance and extension methods to extend the functionality of existing types.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Which of the following types of method *must* you implement in derived classes?** | |
| Select the correct answer. | |
|  | Abstract methods. |
|  | Protected methods. |
|  | Public methods. |
|  | Static methods. |
|  | Virtual methods. |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You want to create an extension method for the String class. You create a static method within a static class. How do you indicate that your method extends the String type?** | |
| Select the correct answer. | |
|  | The return type of the method must be a String. |
|  | The first parameter of the method must be a String. |
|  | The class must inherit from the String class. |
|  | The method declaration must include String as a type argument. |
|  | The method declaration must be preceded by String. |

**Lab 06: Generating the Grades Report**

**Scenario**

You have been asked to upgrade the Grades Prototype application to enable users to save a student’s grades as an XML file on the local disk. The user should be able to click a new button on the StudentProfile view that asks the user where they would like to save the file, displays a preview of the data to the user, and asks the user to confirm that they wish to save the file to disk. If they do, the application should save the grade data in XML format in the location that the user specified.

**Objectives**

After completing this lab, you will be able to:

1. Serialize data to a memory stream.
2. Deserialize data from a memory stream.
3. Save serialized data to a file.

* Estimated Time: 60 minutes

**Exercise 1: Serializing Data for the Grades Report as XML**

**Scenario**

In this exercise, you will write code that runs when the user clicks the **Save Report** button on the **Student Profile** view. You will enable a user to specify where to save the Grade Report, and to serialize the grades data so it is ready to save to a file.

You will use the **SaveFileDialog** object to ask the user for the file name and location where they want to save the file. You will extract the grade data from the application data source and store it in a list of Grade objects.

You will then write the **FormatAsXMLStream** method. This method will use an **XmlWriter** object to create an XML document and populate it with grade information from the list of Grade objects. Finally, you will debug the application and view the data held in the memory stream.

The main tasks for this exercise are as follows:

1. Prompt the user for a filename and retrieve the grade data.
2. Serialize the grade data to a memory stream.
3. Debug the application.

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

1. Start Visual Studio and from the **E:\Mod06\Labfiles\Starter\Exercise 1** folder, open the

**GradesPrototype.sln** solution.

1. In the **Views** folder, open the **StudentProfile.xaml** user interface, and note that it has been updated to include a **Save Report** button that users will click to generate and save the Grades Report.
2. In **StudentProfile.xaml.cs**, in the **SaveReport\_Click** method, add code to store the return value from the dialog box in a nullable Boolean variable.
3. Check if the return value from the **SaveFileDialog** contains data. If it does, do the following:
   1. Get the grades for the currently selected student and store them in a generic list.
   2. Call the **FormatAsXMLStream** method, passing the list of grades to it, and store the returned data in a **MemoryStream** object.

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

1. In **StudentProfile.xaml.cs** code file, locate the **FormatAsXMLStream** method.
2. Add code to save the XML document to a **MemoryStream** object by using an **XmlWriter** object.
3. Add code to create the root node of the XML document in the following format:

<Grades Student="Eric Gruber">

1. Add code to enumerate the grades for the student and add them as child elements of the root node, using the following format:

<Grade Date="01/01/2012" Subject="Math" Assessment="A-" Comments="Good" />

1. Add code to finish the XML document with the appropriate end elements.
2. Add code to flush the **XmlWriter** object and then close it to ensure that all the data is written to the

**MemoryStream** object.

1. Add code to reset the **MemoryStream** object so that it can be read from the start, and then return it to the calling method.
2. Delete the line of code that throws a **NotImplementedException** exception.

 Task 3: Debug the application

1. Build the solution and resolve any compilation errors.
2. In the **SaveReport\_Click** method, add a breakpoint to the closing brace of the **if** block.
3. Debug the application.
4. Log in as **vallee** with a password of **password99**.
5. View **Kevin Liu’s** report, and then click **Save Report** to generate the XML document.
6. 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.

1. When you enter Break Mode, use the Immediate Window to view the contents of the

**MemoryStream** object by using the following code:

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

1. Review the grade data that is returned.
2. Stop debugging, delete the breakpoint, and then close the 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

**Scenario**

In this exercise, you will write code to display a preview of the report to the user before saving it.

First, you will add code to the **SaveReport\_Click** method to display the XML document to the user in a message box. To display the document, you need to build a string representation of the XML document that is stored in the **MemoryStream** object. Finally, you will verify that your code functions as expected by running the application and previewing the contents of a report.

The main tasks for this exercise are as follows:

1. Display the string to the user in a message box.
2. Build a string representation of the XML document.
3. Run the application and preview the data.

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

1. In Visual Studio, from the **E:\Mod06\Labfiles\Starter\Exercise 2** folder, open the

**GradesPrototype.sln** solution.

1. In **StudentProfile.xaml.cs**, in the **SaveReport\_Click** method, add code to the end of the **if** block that calls the **FormatXMLData** method, passing the **MemoryStream** object that you created in the previous task, and storing the return value in a string variable.
2. Add code to preview the string version of the report data in a message box with a caption of **Preview Report**, an **OK** button, and an information image.

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

1. In **StudentProfile.xaml.cs**, in the **FormatXMLData** method, add code to create a new **StringBuilder**

object used to construct the string.

1. Add code to create a new **XmlTextReader** object used to read the XML data from the stream.
2. Add code to read the XML data one node at a time, construct a string representation of the node, and append it to the **StringBuilder**. The possible nodes that can be encountered are **XmlDeclaration**, **Element**, and **EndElement**. Each element may have one or more attributes.
3. Add code to reset the **MemoryStream** object and return the string containing the formatted data to the calling method.
4. Delete the line of code that throws a **NotImplementedException** exception.

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log in as **vallee** with a password of **password99**.
4. View **Kevin Liu’s** report, and then click **Save Report** to generate the XML document.
5. Specify to save the file in the default location.

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

1. Review the XML data displayed in the message box and close the application.
2. In Visual Studio, close the 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

**Scenario**

In this exercise, you will write the grade data to a file on the local disk.

You will begin by modifying the existing preview dialog box to ask the user if they wish to save the file. If they wish to save the file, you will use a **FileStream** object to copy the data from the **MemoryStream** to a physical file. Then you will run the application, generate and save a report, and then verify that the report has been saved in the correct location in the correct format.

The main tasks for this exercise are as follows:

1. Save the XML document to disk.
2. Run the application and verify that the XML document is saved correctly.

 **Task 1: Save the XML document to disk**

1. In Visual Studio, from the **E:\Mod06\Labfiles\Starter\Exercise 3** folder, open the

**GradesPrototype.sln** solution.

1. In **StudentProfile.xaml.cs**, in the **SaveReport\_Click** method, locate the line of code that displays the report data to the user in a message box.
2. Modify this line of code as follows:
   1. Save the return value of the **MessageBox.Show** method in a **MessageBoxResult** variable
   2. Set the caption of the message box to **Save Report?**
   3. Include **Yes** and **No** buttons in the message box.
   4. Display a question mark image.
3. If the user clicks **Yes** to save the report, open the file that the user specified and create a **FileStream**

object to write data to this file.

1. Write the data from the **MemoryStream** object to the file by using the **MemoryStream.CopyTo**

method.

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

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. View **Kevin Liu’s** report card and then click **Save Report** to generate the XML document.
5. Specify to save the file in the Documents folder by using the default name.
6. Review the XML data displayed in the message box, and then confirm that you want to save the file.
7. Close the application.
8. Open the saved report in Internet Explorer and verify that it contains the expected grade data.
9. In Visual Studio, close the solution.

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

### Module Review and Takeaways

In this module, you have learned how to work with the file system by using a number of classes in the System.IO namespace, and how to serialize application data to different formats.

**Review Question(s)**

**Question:** You are a developer working on the Fourth Coffee Windows Presentation Foundation (WPF) client application. You have been asked to store some settings in a plain text file in the user’s temporary folder on the file system. Briefly explain which classes and methods you could use to achieve this.

**Question:** You are a developer working for Fourth Coffee. A bug has been raised and you have been asked to investigate. To help reproduce the error, you have decided to add some logic to persist the state of the application to disk, when the application encounters the error. All the types in the application are serializable, and it would be advantageous if the persisted state was human readable. What approach will you take?

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You are a developer working for Fourth Coffee. You have been asked to write some code to process a 100 GB video file. Your code needs to transfer the file from one location on disk, to another location on disk, without reading the entire file into memory. Which classes would you use to read and write the file?** | |
| Select the correct answer. | |
|  | The MemoryStream, BinaryReader and BinaryWriter classes. |
|  | The FileStream, BinaryReader and BinaryWriter classes. |
|  | The BinaryReader and BinaryWriter classes. |
|  | The FileStream, StreamReader and StreamWriter classes. |
|  | The MemoryStream, StreamReader and StreamWriter classes. |

**Lab 07: Retrieving and Modifying Grade Data**

**Scenario**

You have been asked to upgrade the prototype application to use an existing SQL Server database. You begin by working with a database that is stored on your local machine and decide to use the Entity Data Model Wizard to generate an EDM to access the data. You will need to update the data access code for the Grades section of the application, to display grades that are assigned to a student and to enable users to assign new grades. You also decide to incorporate validation logic into the EDM to ensure that students cannot be assigned to a full class and that the data that users enter when they assign new grades conforms to the required values.

**Objectives**

After completing this lab, you will be able to:

1. Create an EDM from an existing database.
2. Update data by using the .NET Entity Framework.
3. Extend an EDM to validate data.

* Estimated Time: 75 minutes

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

**Scenario**

In this exercise, you will use the Entity Data Model Wizard to generate an EDM from the **SchoolGradesDB**

SQL Server database and then review the model and the code that the wizard generates. The main tasks for this exercise are as follows:

1. Build and generate an EDM by using a table from the SchoolGradesDB database.
2. Review the generated code.

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

1. Start File Explorer, navigate to the **E:\Mod07\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and from the E:\Mod07\Labfiles\Starter\Exercise 1 folder, open the GradesPrototype.sln solution.
3. Add a new class library project named Grades.DataModel to the solution.
4. Add a new ADO.NET Entity Data Model named GradesModel to the Grades.DataModel project.
5. Generate the model from the **SchoolGradesDB** database on the (localdb)\v11.0 server and include the following tables:

* **Grades**
* **Students**
* **Subjects**
* **Teachers**
* **Users**

1. If the **Security Warning** dialog box appears, click **Do not show this message again**, and then click

**OK**.

1. Build the solution.

 Task 2: Review the generated code

1. In the designer window, review the entities that the wizard generated.
2. Review the properties and navigation properties of the **Grade** entity.
3. Review the mapping details for the Grade entity.
4. In the GradesModel.Context.tt folder, in GradesModel.Context.cs file, review the code for the

**SchoolGradesDBEntities DbContext** object.

1. In the GradesModel.tt folder, in the Grade.cs file, review the properties of the **Grade** entity.
2. Save all of the files, and then close the 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

**Scenario**

In this exercise, you will add functionality to the prototype application to display the grades for a user. The grade information in the database stores the subject ID for a grade, so you will add code to the application to convert this to the subject name for display purposes. You will also add code to display the **Add Grade** view to the user and then use the information that the user enters to add a grade for the current student. Finally, you will run the application and verify that the grade display and grade-adding functionality works as expected.

The main tasks for this exercise are as follows:

1. Display grades for the current student.
2. Display the subject name in the UI.
3. Display the GradeDialog view and use the input to add a new grade.
4. Run the application and test the grade-adding functionality.

 **Task 1: Display grades for the current student**

1. In Visual Studio, from the E:\Mod07\Labfiles\Starter\Exercise 2 folder, open the GradesPrototype.sln solution.
2. Set the GradesPrototype project to be the startup project.
3. In the **Views** folder, in **StudentProfile.xaml.cs**, in the **Refresh** method, add code to the end of the method to:
   1. Iterate through the grades in the session context object and, if they belong to the current student, add them to a new list of grades.
   2. Use data binding to display the list of grades in the **studentGrades ItemsControl** control by setting the **studentGrades.ItemsSource** property to the list of grades that you have just created.
4. Build the solution and resolve any compilation errors.
5. Run the application.
6. Log on as **vallee** with a password of **password99**.
7. Click **Kevin Liu**, and then verify that his grades appear in the list.
8. 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 StudentProfile.xaml.cs, in the **SubjectConverter** class, in the **Convert** method, add code to the method to:
   1. Convert the subject ID that is passed into the method into the subject name.
2. Return the subject name or, if there is no subject name matching the subject ID, the string “N/A”.

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

1. In StudentProfile.xaml.cs, in the **AddGrade\_Click** method, add code to:
2. Create and display a new instance of the **GradeDialog** view.
3. If the user clicks **OK** in the **GradeDialog** view, retrieve the data they have entered and use it to create a new **Grade** object.
4. Save the grade and refresh the display so that the new grade appears.

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

1. Build the solution, and then resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Click **Kevin Liu**, and verify that the list of grades now displays the subject name, not the subject ID.
5. Add a new grade for Kevin Liu using the following information:

* Subject: **Geography**
* Assessment: **A+**
* Comments: **Well done!**

1. Verify that the new grade is added to the list, and then close the application.
2. In Visual Studio, close the 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

**Scenario**

In this exercise, you will update the application to validate data that the user enters.

First, you will add code to check whether a class is full before enrolling a student and throw an exception if it is. Then you will add validation code to check that a user enters a valid date and assessment grade when adding a grade to a student. Finally, you will run the application and verify that the data validation works as expected.

The main tasks for this exercise are as follows:

1. Throw the ClassFullException exception.
2. Add validation logic for the Assessment and AssessmentDate properties.
3. Run the application and test the validation logic.

 **Task 1: Throw the ClassFullException exception**

1. In Visual Studio, open the GradesPrototype.sln solution from the E:\Mod07\Labfiles\Starter\Exercise 3 folder.
2. Set the GradesPrototype project to be the startup project.
3. Add a new class named **customTeacher.cs** to the Grades.DataModel project.
4. Modify the class declaration to make it a public partial class named **Teacher**.
5. Add a private integer constant named **MAX\_CLASS\_SIZE** with a value of **8** to the **Teacher** class.
6. Add an **EnrollInClass** method that takes a **Student** object as a parameter and returns **void**.
7. In the **EnrollInClass** method, add code to:
   1. Use a LINQ query to determine how many students are currently in the class. You can execute a Count query of the students in a particular class by selecting only those students with a **TeacherUserId** property equal to the contents of the **UserId** variable.
   2. If the class is full, throw a new **ClassFullException** exception.
   3. If the student who is passed to the method is not already enrolled in a class, set the **TeacherID**

property of the **Student** object to the **UserID** of the current teacher.

* 1. Otherwise, throw a new **ArgumentException** exception.

1. In the **Views** folder, in the **AssignStudentDialog.xaml.cs**, locate the **Student\_Click** method.
2. Towards the end of the method, before the call to the **Refresh** method, add code to:
   1. Call the **EnrollInClass** method to assign the student to this teacher’s class, passing the student as a parameter.
   2. Save the updated student/class information back to the database.

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

1. Add a new class named **customGrade.cs** to the **Grades.DataModel** project.
2. Modify the class declaration to make it a public partial class named **Grade**.
3. Add a **ValidateAssessmentDate** method that takes a **DateTime** object as a parameter and returns a

**boolean**.

1. In the **ValidateAssessmentDate** method, add code to:
   1. If the **DateTime** object passed to the method is later than the current date, throw a new

**ArgumentOutOfRangeException** exception.

* 1. Otherwise, return **true**.

1. Bring the **System.Text.RegularExpressions** namespace into scope.
2. Add a **ValidateAssessmentGrade** method that takes a **string** as a parameter and returns a **boolean**.
3. In the **ValidateAssessmentGrade** method, add code to:
   1. Use the following regular expression to check that the string passed to the method is in the range of A+ to E-.

Match matchGrade = Regex.Match(assessment, @"^[A-E][+-]?$");

* 1. If the string passed is not in the valid range, throw a new **ArgumentOutOfRangeException**

exception.

* 1. Otherwise, return **true**.

1. In the **Controls** folder, in the **GradeDialog.xaml.cs** class, locate the **ok\_Click** method.
2. In this method, add code to:
   1. Create a new **Grade** object.
   2. Call the **ValidateAssessmentDate** method, passing the selected date in the **assessmentDate**

date picker control.

* 1. Call the **ValidateAssessmentGrade** method, passing the text in the **assessmentGrade** text box control.

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

1. Build the solution, and then resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Attempt to enroll a new student into the class, and then verify that an error message is displayed explaining that the class is already full.
5. Click **Kevin Liu**, and then add a new grade for him by using the following information:

* Date: **tomorrow’s date**
* Subject: **Math**
* Assessment: **F+**
* Comments: **Well done!**

1. Verify that an error message is displayed explaining that the assessment date must be on or before the current date.
2. Modify the new grade date by using the following information:

 Date: **8/19/2012**

* Subject: **Math**
* Assessment: **F+**
* Comments: **Well done!**

1. Verify that an error message is displayed explaining that the assessment grade must be in the range A+ to E-.
2. Modify the new grade date by using the following information:

 Date: **8/19/2012**

* Subject: **Math**
* Assessment: **A+**
* Comments: **Well done!**

1. Verify that the new grade is added to the list, and then close the application.
2. In Visual Studio, close the solution.

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

### Module Review and Takeaways

In this module, you learned how to create and use EDMs and how to query many types of data by using LINQ.

**Review Question(s)**

**Question:** What advantages does LINQ provide over traditional ways of querying data?

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Fourth Coffee wants you to add custom functionality to an existing EDM in its Coffee Sales application. You need to write a method for adding a new product to the application. In which of the following locations should you write your code?** | |
| Select the correct answer. | |
|  | In the relevant generated class in the EDM project. |
|  | In a partial class in the EDM project. |

**Lab 07: Retrieving and Modifying Grade Data in the Cloud**

**Scenario**

Currently, the application retrieves data from a local database. However, you have decided to store the data in the cloud and must configure the application so that it can retrieve data across the web.

You must create a WCF Data Service for the **SchoolGrades** database that will be integrated into the application to enable access to the data.

Finally, you have been asked to write code that displays student images by retrieving them from across the web.

**Objectives**

After completing this lab, you will be able to:

1. Create a WCF Data Service.
2. Use a WCF Data Service.
3. Retrieve data over the web.

* Estimated Time: 60 minutes

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

**Scenario**

In this exercise, you will create a WCF Data Service for the **SchoolGrades** database so that the client application can connect to the database over the web.

First, you will add a new ASP.NET Web Application project to the solution and configure it for the client application. You will then expose the entities in the EDM from a data service in the new project. Next, you will specify the data context for the data service and configure access rights to the entities that it exposes. Finally, you will add an operation to the data service that returns all of the students in a specified class.

The main tasks for this exercise are as follows:

1. Create the Grades.Web project.
2. Add a data service to the Grades.Web project.
3. Specify the GradesDBEntities data context for the data service.
4. Add an operation to retrieve all of the students in a specified class.
5. Build and test the data service.

 **Task 1: Create the Grades.Web project**

1. Start File Explorer, navigate to the **E:\Mod08\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and from the **E:\Mod08\Labfiles\Starter\Exercise 1** folder, open the

**GradesPrototype.sln** solution.

1. Add a Visual C# ASP.NET Empty Web Application project called Grades.Web to the solution.
2. Configure the new project as follows:
   1. Start Action: **Don’t open a page.**
   2. Servers: **Use Local IIS Web server**
   3. Project Url: **http://localhost:1650/**
3. Set the following projects to start at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
4. Save all of the files.

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

1. Add a new folder named Services to the Grades.Web project.
2. Add a new WCF Data Service named GradesWebDataService to the Services folder.
3. Add a reference to the Grades.DataModel project in the Grades.Web project.
4. Add a reference to the EntityFramework assembly. This assembly is located in the E:\Mod08\Labfiles\Starter\Exercise 1\packages\EntityFramework.5.0.0\lib\net45 folder.
5. Copy the **<connectionStrings>** element from the App.config file in the GradesPrototype project and paste it into the Web.config file in the Grades.Web project.

 **Note:** The data service in the Grades.Web project needs to connect to the same data source that the data model uses.

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

1. In the code in the Grades.WebDataService.svc file, add a **using** directive to bring the

**Grades.DataModel** namespace into scope.

1. Modify the class declaration of the **GradesWebDataService** to use the **SchoolGradesDBEntities**

class as the data source.

 **Note:** The **GradesDBEntities** class provides the object context for the EDM. The GradesWebDataService data service will retrieve data by using this object context and expose the various entities by using REST URIs.

1. In the **InitializeService** method, set the access rules for each of the following entities to

**EntitySetRights.All**:

* **Grades**
* **Teachers**
* **Students**
* **Subjects**
* **Users**

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

1. In the **GradesWebDataService** class, add an operation named **StudentsInClass** that takes a class name as a string and returns an **IEnumerable<Student>** collection. This operation should be annotated with the **WebGet** attribute.
2. In this operation, use a LINQ query against the **CurrentDataSource** object to retrieve and return all of the students in the class.
3. In the **InitializeService** method, set the access rule for the **StudentsInClass** operation to

**ServiceOperationRights.AllRead**.

 Task 5: Build and test the data service

1. Build the solution, and then resolve any compilation errors.
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)**.

1. Verify that Internet Explorer displays an XML description of the entities that the data service exposes.
2. Close Internet Explorer.
3. In Visual Studio, close the 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

**Scenario**

In this exercise, you will integrate the WCF Data Service into the Grades Prototype application.

First, you will add a service reference in the GradesPrototype project that references the running WCF Data Service. You will then modify the code that accesses the local EDM to use the WCF Data Service instead. Next, you will modify the code that saves changes back to the database to do so through the data service. Finally, you will test the application to verify that it runs the same as if the data was being called locally.

The main tasks for this exercise are as follows:

1. Add a service reference for the WCF Data Service to the GradesPrototype application.
2. Modify the code that accesses the EDM to use the WCF Data Service.
3. Modify the code that saves changes back to the database to use the WCF Data Service.
4. Build and test the application to verify that the application still functions correctly.

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

1. In Visual Studio, open the GradesPrototype.sln solution from the E:\Mod08\Labfiles\Starter\Exercise 2 folder.
2. Set the following projects to start at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
3. Rebuild the solution.
4. In the GradesPrototype project, remove the reference to the Grades.DataModel project.
5. Add a service reference to http://localhost:1650, using the namespace of **Grades.DataModel**.
6. Update the namespace declaration in the Reference.cs file for the service reference to Grades.DataModel. The Reference.cs file is generated in the Service References\Grades.DataModel\Reference.datasvcmap folder in Solution Explorer. You need to enable Solution Explorer to show all files to see this folder.

 **Note:** The Add Service Reference Wizard prepends the namespace that you specify with the namespace of the project, so the result in this case is **GradesPrototype.Grades.DataModel.** The existing code in the GradesPrototype project expects the various entity classes to be located in the **Grades.DataModel** namespace. You can either update every reference throughout the project, or you can change the namespace of the data service; this lab opts for the latter approach.

There is one drawback with this approach; if you regenerate the data service reference (this will be necessary if, for example, you modify the WCF data service and add a new entity class), you will have to edit the Reference.cs file and update the namespace again because any manual changes you make to this file will be lost.

1. Add a new folder named **DataModel** to the **GradesPrototype** project.
2. Copy the following code files from the **Grades.DataModel** project to the

**GradesPrototype\DataModel** folder:

* 1. Classes.cs
  2. customGrade.cs
  3. customTeacher.cs

 **Note:** The Classes.cs, Grade.cs, and Teacher.cs files contain custom functionality for the **Grade** and

**Teacher** classes that you implemented in an earlier lab. WCF Data Services does not propagate any

custom functionality that is defined for a data model, so you must manually copy these files to the Grades.DataModel project. You will also have to make some small changes to this code to access data through the WCF Data Service rather than by referencing the entities themselves. You will do this in the next task.

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

1. In the Grades.Web project, in the Services folder, in SessionContext.cs, modify the **DBContext** declaration to pass a new **Uri** object pointing to http://localhost:1650/Services/GradesWebDataService.svc to the **SchoolGradesDBEntities** constructor.

 **Note:** The **DBContext** object provides the object context for accessing the data source. Previously this object context retrieved data directly from a local EDM. Now the data service provides this object context, and the constructor requires the URL of the data service.

1. Add the following static constructor to the **SessionContext** class.

static SessionContext()

{

DBContext.MergeOption = System.Data.Services.Client.MergeOption.PreserveChanges;

}

This constructor ensures that any changes made by the user are not lost if multiple users try and make simultaneous changes.

1. In the Views folder, in StudentsPage.xaml.cs, locate the **Refresh** method.
2. Modify the code in the **foreach** loop that populates the **list ItemsControl** with the details of the students for the current teacher. The user and grades data for a student are held in separate entities and they are not fetched automatically by WCF Data Services (this is to save network resources by not retrieving data unnecessarily). Your code should retrieve the related data in the **User** and **Grades** properties for each student by using the **LoadProperty** method of the data context (available in **SessionContext.DBContext** object).
3. In LogonPage.xaml.cs, in the **Logon\_Click** method, modify the statement that loads teacher data to also load the user and student data for that teacher. As an alternative to using the **LoadProperty** method of the data context, use the **Expand** method when the data is fetched by using the LINQ query.
4. In LogonPage.xaml.cs, in the **Logon\_Click** method, modify the statement that loads student data to also load the user and grades data.
5. In the **GradesPrototype** solution, in the **customTeacher.cs** file, add a **using** directive to bring the

**GradesPrototype.Services** namespace into scope.

1. In the **EnrollInClass** method, modify the **from** statement to reference the **Students** collection in the

**SessionContext.DBContext** object.

1. In the **AssignStudentDialog** dialog, in the **Refresh** method, modify the code that finds unassigned students to reference the **SessionContext.DBContext.Students** collection rather than **SessionContext.DBContext.Students.Local**. This change is necessary because the data model implemented by the data service does not provide the **Local** property. You should also use the **Expand** method to retrieve the **User** and **Grades** information for the students.
2. In the **StudentProfile** view, in the **AddGrade\_Click** method, find the code that uses the **Add** method of the **Grades** collection to add a grade to a student. Modify this code to use the **AddToGrades** method of the **DBContext** class. This change is necessary because the **Grades** collection implemented by WCF Data Services does not provide the **Add** method.
3. In the **SaveReport\_Click** method, modify the LINQ query that retrieves the grades for the report to also fetch the **Subject** details by using the **Expand** method.
4. In the **StudentsPage** view, in the **NewStudent\_Click** method, find the code that uses the **Add** method of the **Students** collection to add a new student. Modify this code to use the **AddToStudents** method of the **DBContext** class.

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

1. In the code for the **AssignStudentDialog** view, in the **Student\_Click** method, add code to specify that the selected student has been changed by using the **UpdateObject** method before the call to the **SessionContext.Save** method.

**Note:** WCF Data Services requires that you explicitly mark an entity as updated, otherwise any changes will not be saved.

1. In the **StudentProfile** view, in the **Remove\_Click** method, add code to specify that the current student has been changed before the call to the **SessionContext.Save** method.
2. In the Controls folder, in **ChangePasswordDialog.xaml.cs**, in the **ok\_Click** method, add code to specify that the current user has been changed before the call to the **SessionContext.Save** method.

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

1. Build the solution, and then resolve any compilation errors.
2. Log on as **vallee** with a password of **password99**.
3. Perform the following tasks to verify that the application still updates the data correctly:
   1. Remove Eric Gruber from the class.
   2. Enroll **Jon Orton** into the class.
   3. Change the password to **password88**, and then verify that you can log on with the new password.
   4. Log on as **grubere** with a password of **password**.
   5. Verify that his student profile appears, and then log off.
4. Close the application.
5. In Visual Studio, close the 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)

**Scenario**

In this exercise, you will write code that displays student images by retrieving the image from across the web. You will modify the **StudentsPage** window (that displays the list of students in a class), the **StudentProfile** window (that displays the details for an individual student), and the **AssignStudentDialog** window (that displays a list of unassigned students) to include the student photographs. The data for each student contains an **ImageName** property that specifies the filename of the photograph for the student on the web server. These files are located in the **Images\Portraits** folder on the same web server that hosts the data service (in the Web.Grades project.) You will build a value converter class that generates the URL of an image from the **ImageName** property and then use an **Image** control to use the URL to fetch and render the image in each of the specified windows. Finally, you will run the application to verify that the images appear.

The main tasks for this exercise are as follows:

1. Create the ImageNameConverter value converter class.
2. Add an Image control to the StudentsPage view and bind it to the ImageName property.
3. Add an Image control to the StudentProfile view and bind it to the ImageName property.
4. Add an Image control to the AssignStudentDialog control and bind it to the ImageName property.
5. Build and test the application, verifying that student’s photographs appear in the list of students for the teacher.

 **Task 1: Create the ImageNameConverter value converter class**

1. In Visual Studio, open the GradesPrototype.sln solution from the E:\Mod08\Labfiles\Starter\Exercise 3 folder.
2. Set the following projects to start at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
3. Rebuild the solution.
4. In the GradesPrototype project, in StudentsPage.xaml.cs, create a new public class named

**ImageNameConverter** that implements the **IValueConverter** interface.

1. In the **ImageNameConverter** class, define a string constant named **webFolder** that contains the string “http://localhost:1650/Images/Portraits/”.
2. Implement the **IValueConverter** interface.
3. In the **Convert** method, add code to check whether the value that is passed to the method contains a string, and if so, append it to the **webFolder** string and return the result. If the value passed to the method is **null**, return **string.Empty**. There is no need to add any code to the **ConvertBack** method.
4. Build the solution and resolve any compilation errors.

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

1. In the XAML markup for the StudentsPage view, add a reference to the **clr- namespace:GradesPrototype.Views** namespace. Assign this to **xmlns:local**. This is the namespace that contains the **ImageNameConverter** class.
2. Add an instance of the **ImageNameConverter** class as a resource to the view as shown in the following code.

<UserControl.Resources>

<local:ImageNameConverter x:Key="ImageNameConverter"/>

</UserControl.Resources>

1. At the top of the **StackPanel** control, add an **Image** control. The contents of the image should use a data binding that references the **ImageNameConverter** class to convert the value in the **ImageName** property into a URL, and then display the data retrieved from this URL. Set the height of the control to **100**. The markup for the control should look like this:

<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 the XAML markup for the StudentProfile view, add an instance of the **ImageNameConverter** class as a resource to the view. Use the **app** namespace (this namespace has already been defined at the top of the XAML markup).
2. At the top of the **StackPanel** control, add an **Image** control. Use the **ImageNameConverter** to convert the value in the **ImageName** property into a URL and display the image retrieved from this URL. Set the height of the **Image** control to 150.

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

1. In the XAML markup for the AssignStudentDialog control, add a reference to the **clr- namespace:GradesPrototype.Views** namespace. Assign this to **xmlns:local**.
2. Add an instance of the **ImageNameConverter** class as a resource to the view.
3. At the top of the **StackPanel** control. As before, use the **ImageNameConverter** to convert the value in the **ImageName** property into a URL and display the image retrieved from this URL. Set the height of the **Image** control to 100.

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

1. Build the solution, and then resolve any compilation errors.
2. Log on as **vallee** with a password of **password88**.
3. Verify that the students list now includes images.
4. View George Li’s profile and verify that his image appears.
5. Remove George Li from the class.
6. Enroll George Li in the class, and then verify that the **Assign Student** dialog box now includes images, and new student icons in the main application window include images.
7. Close the application.
8. In Visual Studio, close the 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 Review and Takeaways

In this module, you have learned how to use the request and response classes in the **System.Net** namespace to manipulate remote data sources directly and how to use WCF Data Services to expose and consume an entity data model over the web.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Which of the following correctly describes how to access data that is provided in an HTTP response?** | |
| Select the correct answer. | |
|  | Invoke the GetResponseStream static method on the HttpWebResponse class. |
|  | Read the ContentLength instance property on the HttpWebResponse object. |
|  | Invoke the GetRequestStream instance method on the HttpWebResponse object. |
|  | Invoke the GetResponseStream instance method on the HttpWebResponse object. |
|  | Invoke the GetResponseStream instance method on the HttpWebRequest object. |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **When you create a WCF Data Service to provide remote access to an EDM, how do you specify which entity sets the data service should make available to client applications?** | |
| Select the correct answer. | |
|  | Do nothing. All entity sets in the EDM are automatically available to client applications. |
|  | In the InitializeService method of the data service, use the SetEntityAccessRule method of the DataServiceConfiguration object to specify which entity sets should be made available to client applications. |
|  | Create a certificate for each client that can connect to the service. Configure the service to only allow authenticated clients to connect and retrieve data. |
|  | Define a data contract for each entity set. |
|  | Configure the service to enable HTTP GET requests for each entity set. |

**Lab 09: Customizing Student Photographs and Styling the Application**

**Scenario**

Now that you and The School of Fine Arts are happy with the basic functionality of the application, you need to improve the appearance of the interface to give the user a nicer experience through the use of animations and a consistent look and feel.

You decide to create a **StudentPhoto** control that will enable you to display photographs of students in the student list and other views. You also decide to create a fluid method for a teacher to remove a student from their class. Finally, you want to update the look of the various views, keeping their look consistent across the application.

**Objectives**

After completing this lab, you will be able to:

1. Create and use user controls.
2. Use styles and animations.

* Estimated Time: 90 minutes

**Exercise 1: Customizing the Appearance of Student Photographs**

**Scenario**

In this exercise, you will customize the appearance of student photographs in the production application.

You will begin by creating a **StudentPhoto** user control that will host the photographs on the various pages in the UI. Then you will lay out the user controls and write code to raise the **Student\_Click** method when a user clicks a student photograph.

Next, you will add a remove button with a red X to the user control that users can click to remove a student from a class. When a user hovers over the button, the opacity of the button and the photograph will change.

Finally, you will run the application to verify that the student’s image is displayed correctly on the

**StudentsPage** view.

The main tasks for this exercise are as follows:

1. Create the StudentPhoto user control.
2. Display the students’ photographs in the StudentsPage view.
3. Enable the user to display the details for a student.
4. Add a Remove button to the StudentsPage view.
5. Display all students for the current teacher.
6. Build and test the application.

 **Task 1: Create the StudentPhoto user control**

1. Start File Explorer, navigate to the **E:\Mod09\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and then open the Grades.sln solution from the E:\Mod09\Labfiles\Starter\Exercise 1 folder.
3. Set the following projects to start without debugging at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
4. Add a new WPF user control named **StudentPhoto.xaml** to the **Controls** folder in the **Grades.WPF**

project.

1. Modify the XAML markup for the user control as follows:
   1. Add an **Image** control to the **Grid**. This **Image** control will use data binding to display the photograph, and the source of the **Image** should be the **File** property of the data source. The **Image** should fill the user control except for a margin of **8** points all the way around to allow for a frame.
   2. Add a second **Image** control to the **Grid**. This **Image** control will display the frame around the student photograph, and it should completely fill the **Grid**, so specify a margin of **0** points. Use the **Image\_Frame.png** file in the **Images** folder as the source for the Image; this image has a transparent center that enables the student photograph to show through.
   3. Add a **TextBlock** control to display the name of the student underneath the photo frame. This control will also use data binding, and the name will be provided by the **Name** property of the data source. Use the static resource **LabelCenter** to style the text and set the **FontSize** to **16**. Set the **VerticalAlignment** to **Bottom** to ensure that the name appears underneath the photograph, and specify a margin of **8, 0, 14.583, 8** to add a bit of space around the name.
   4. Change the **Class** name of the control to **Grades.WPF.StudentPhoto**. The completed markup should look like the following:

<UserControl x:Class="Grades.WPF.StudentPhoto" xmlns=["http://schemas.microsoft.com/winfx/2006/xaml/presentatio](http://schemas.microsoft.com/winfx/2006/xaml/presentation)n"

xmlns:x[="h](http://schemas.microsoft.com/winfx/2006/xaml)tt[p://schemas.microsoft.com/winfx/2006/xaml"](http://schemas.microsoft.com/winfx/2006/xaml) xmlns:mc="[http://schemas.openxmlformats.org/markup-compatibility/2006"](http://schemas.openxmlformats.org/markup-compatibility/2006) xmlns:d[="h](http://schemas.microsoft.com/expression/blend/2008)tt[p://schemas.microsoft.com/expression/blend/2008"](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>

1. In **StudentPhoto.xaml.cs**, remove the existing **using** directives and add **using** directives to bring the following namespaces into scope:

* **System.Windows.Controls**
* **System.Windows.Media.Animation**

1. Change the namespace of the control to **Grades.WPF**.

 Task 2: Display the students’ photographs in the StudentsPage view

1. In the **Views** folder, in **StudentsPage.xaml**, in the **ScrollViewer** element, locate the **ItemsControl** control named **list**. You will use this control to display the list of photographs for students in a class. In a later step, you will use data binding to associate the list of students with this control.
2. Add an **ItemTemplate** element to the **ItemsControl** below the **ItemsControl.ItemsPanel** control. This element will specify how each photograph is displayed and formatted.
3. In the ItemTemplate element, define a DataTemplate that displays the StudentPhoto control in a grid with a Margin property of 8 points.

Remember that the StudentPhoto control is defined in the Grades.WPF namespace. The XAML markup for the StudentsPage control contains the following namespace definition to bring the types in the Grades.WPF namespace into scope with the alias local: so you should refer to the StudentPhoto control as local:StudentPhoto.

xmlns:local="clr-namespace:Grades.WPF"

1. Use the following information to set the properties of the **StudentPhoto** control.

* Height: **150**

 Width: **127.5**

* Cursor: **Hand**

The completed markup should look like the following:

<ItemsControl.ItemTemplate>

<DataTemplate>

<Grid Margin="8">

<local:StudentPhoto Height="150" Width="127.5" Cursor="Hand" />

</Grid>

</DataTemplate>

</ItemsControl.ItemTemplate>

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

1. In **StudentPage.xaml**, modify the instance of the **StudentPhoto** control in the **DataTemplate**

element to invoke the **Student\_Click** event handler when a user clicks a photo.

The XAML markup for the **StudentPhoto** control should look like this:

<local:StudentPhoto Height="150" Width="127.5" Cursor="Hand" MouseLeftButtonUp="Student\_Click"/>

1. In **StudentsPage.xaml.cs**, in the **Events** region, locate the **Student\_Click** method.
2. Review this 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**, add another **Grid** control to the existing **Grid** control in the **DataTemplate** element. You will add controls to this grid to display a customized "remove" icon in the top right corner of each photograph. If the user clicks this icon, the student will be removed from the class.
2. Use the following information to set properties of the **Grid** control.

* VerticalAlignment: **Top**
* HorizontalAlignment: **Right**
* Background: **#00000000**
* Opacity: **0.3**
* Width: **20**
* Height: **20**
* ToolTipService.Tooltip: **Remove from class**
* Tag: **{Binding}**

 **Note:** The **Tag** property will contain a reference to the student, so that the "remove" functionality knows which student to remove. This property will use the data binding of the **ItemsControl** that contains the **DataTemplate** to reference the student.

1. Add an **Image** control as a child of the **Grid** control and use it to display the **delete.png** picture in the **Images** folder. Set the **Stretch** property to **Uniform**. This image contains a cross symbol that will be displayed by the remove icon.

The XAML markup for the **Grid** control should look like this:

<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>

1. In **StudentsPage.xaml.cs**, locate the **RemoveStudent\_MouseEnter** method. This code 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.
2. In **StudentsPage.xaml.cs**, locate the **RemoveStudent\_MouseLeave** method. This code 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.
3. In **StudentsPage.xaml.cs**, locate the **RemoveStudent\_Click** method. This code removes a student from the current teacher’s class when a user clicks the remove icon.
4. In **StudentsPage.xaml**, in the **Grid** control for the remove icon, specify the event handlers to use when the mouse traverses the control: when the mouse enters the control raise the **RemoveStudent\_MouseEnter** event, and when the mouse leaves the control, raise the **RemoveStudent\_MouseLeave** event.
5. In **StudentsPage.xaml**, in the **Grid** control for the remove icon, specify to raise the

**RemoveStudent\_Click** event handler when the user clicks the remove icon.

The completed XAML markup for **ItemsControl.ItemTemplate** element in **StudentsPage.xaml**, including the student photograph and the remove icon, should look like this:

<ItemsControl.ItemTemplate>

<DataTemplate>

<Grid Margin="8">

<local:StudentPhoto Height="150" Width="127.5" Cursor="Hand" MouseLeftButtonUp="Student\_Click" />

<Grid VerticalAlignment="Top" HorizontalAlignment="Right" Background="#00000000" Opacity="0.3" Width="20" Height="20"

ToolTipService.ToolTip="Remove from class" Tag="{Binding}" MouseEnter="RemoveStudent\_MouseEnter"

MouseLeave="RemoveStudent\_MouseLeave" MouseLeftButtonUp="RemoveStudent\_Click">

<Image Source="../Images/delete.png" Stretch="Uniform" />

</Grid>

</Grid>

</DataTemplate>

</ItemsControl.ItemTemplate>

 Task 5: Display all students for the current teacher

1. In the **StudentsPage.xaml.cs** file, locate the **Refresh** method.

This method runs each time the **StudentsPage** view is displayed. The purpose of this method is to ensure that the view displays a correct and up-to-date list of students for the teacher.

1. In this method, review the code which finds all students for the current teacher and constructs a list of students.
2. Add code after the **foreach** loop to bind the list of students to the **list ItemsControl** control by using the **ItemsSource** property of the **list** object.

 Task 6: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
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.

**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

**Scenario**

In this exercise, you will update the **LogonPage** control to have the same look and feel as the rest of the application.

First, you will define styles for the username and password text boxes on the **LogonPage** of the application. You will use the **Style** property of each control to apply the styles that you have defined. Then you will define some global styles for use across the entire application. You will define a style for labels and a style for text. Finally, you will run the application to verify that the styling of the text elements has changed throughout the application.

The main tasks for this exercise are as follows:

1. Define and apply styles for the LogonPage view.
2. Define global styles for the application.
3. Build and test the application.

 **Task 1: Define and apply styles for the LogonPage view**

1. In Visual Studio, open the Grades.sln solution from the E:\Mod09\Labfiles\Starter\Exercise 2 folder.
2. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. In the **Grades.WPF** project, in the **Views** folder, open **LogonPage.xaml**.
2. In the **LogonPage** user control, create a **Resources** section.
3. In the **Resources** section, define a style named **LoginTextBoxStyle**, based on the **TextBoxStyle**

style, and targeting text boxes.

1. Use the following information to set properties of the style:

* Margin: **5**
* FontSize: **24**
* MaxLength: **16**

The XAML markup for the style should look like this:

<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>

</UserControl.Resources>

1. Locate the definition of the **username** text box, delete the **FontSize** property of the control, and then apply the **LoginTextBoxStyle** to the control.
2. In the **Resources** section, define another style named **PasswordBoxStyle** targeting password boxes.
3. Use the following information to set properties of the style:

* Margin: **5**
* FontSize: **24**
* MaxLength: **16**

1. Locate the definition of the password text box, delete the **FontSize** property of the control, and then apply the **PasswordBoxStyle** to the control.

 Task 2: Define global styles for the application

1. In the **Themes** folder, open the **Generic.xaml** file.
2. Locate the **<!-- TODO: Exercise 2: Task 2a: Define the label styling used throughout the application -->** comment near the end of the file.
3. Below this comment, set the properties of the **LabelStyle** style by using the following information:

* TextWrapping: **NoWrap**
* FontFamily: **Buxton Sketch**
* FontSize: **19**
* Foreground: **#FF303030**

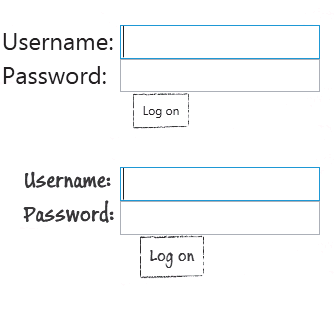
1. Locate the **<!-- TODO: Exercise 2: Task 2b: Define the text styling used throughout the application -->** comment.
2. Below this comment, set the properties of the **TextBoxStyle** style by using the following information:

* TextWrapping: **NoWrap**
* FontFamily: **Buxton Sketch**
* FontSize: **12**
* TextAlignment: **Left**
* Foreground: **#FF303030**

 Task 3: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
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**

1. Close the application, and then in Visual Studio, close the 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)

**Scenario**

In this exercise, you will update the **StudentPhoto** control to animate when a user hovers over it.

First you will define an animation for the **StudentPhoto** control, which will cause a student’s photograph to pulse when a user hovers over it. You will then add event handlers for this animation and apply the animation to the control. Finally, you will run the application to verify that the animation executes correctly.

The main tasks for this exercise are as follows:

1. Define animations for the StudentPhoto control.
2. Add event handlers to trigger the animations.
3. Build and test the application.

 **Task 1: Define animations for the StudentPhoto control**

1. In Visual Studio, open the Grades.sln solution from the E:\Mod09\Labfiles\Starter\Exercise 3 folder.
2. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. In the **Grades.WPF** project, in the **Controls** folder, open the **StudentPhoto.xaml** file.
2. Create a **RenderTransform** element in the **UserControl**.
3. Inside the **RenderTransform** element, add a **ScaleTransform** named **scale**. You will use this transform to change the size of the **StudentPhoto** user control when the mouse moves over it.
4. Create a **Resources** element in the user control.
5. Inside the **Resources** element, add a **Storyboard** that will contain animations that are performed when the mouse enters the control; set the **x:Key** property to **sbMouseEnter**.
6. Inside the **Storyboard** element, add a **DoubleAnimation** element. Use the following information to define the properties of the animation:

 To: **1.1**

* BeginTime: **00:00:00**

 Duration: **00:00:00.05**

* Storyboard.TargetName: **scale**
* Storyboard.TargetProperty: **ScaleX**

1. Inside the **Storyboard** element, add another **DoubleAnimation** element. Use the following information to define the properties of the animation:

 To: **1.1**

* BeginTime: **00:00:00**

 Duration: **00:00:00.15**

* Storyboard.TargetName: **scale**
* Storyboard.TargetProperty: **ScaleY**

1. After the existing Storyboard element, add another **Storyboard** that will contain animations that are performed when the mouse leaves the control; set the **x:Key** property to **sbMouseLeave**.
2. Inside the **Storyboard** element, add a **DoubleAnimation** element. Use the following information to define the properties of the animation:

* To: **1**
* BeginTime: **00:00:00**

 Duration: **00:00:00.05**

* Storyboard.TargetName: **scale**
* Storyboard.TargetProperty: **ScaleX**

1. Inside the **Storyboard** element, add another **DoubleAnimation** element. Use the following information to define the properties of the animation:

* To: **1**
* BeginTime: **00:00:00**

 Duration: **00:00:00.15**

* Storyboard.TargetName: **scale**
* Storyboard.TargetProperty: **ScaleY**

 Task 2: Add event handlers to trigger the animations

1. In the **StudentPhoto.xaml.cs** file, locate the **Storyboard** region.
2. Add code to define the **OnMouseEnter** event handler that triggers the mouse enter animation (**sbMouseEnter**), as follows:

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();

}

1. Add code to define the **OnMouseLeave** event handler that triggers the mouse leave animation (**sbMouseLeave**).
2. In the **Views** folder, open the **StudentsPage.xaml.cs** file.
3. In the **Events** region, add an event handler called **Student\_MouseEnter** to forward the **MouseEnter**

event to the **StudentPhoto** control, as follows:

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();

}

1. Add another event handler called **Student\_MouseLeave** that forwards the **MouseLeave** event to the

**StudentPhoto** control.

1. In the XAML markup for the **StudentsPage**.view, specify that the **MouseEnter** event for the **StudentPhoto** control in the **DataTemplate** should trigger the **Student\_MouseEnter** event handler method, and the **MouseLeave** event should trigger the **Student\_MouseLeave** event handler.

 Task 3: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
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, and then in Visual Studio, close the solution.

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

### Module Review and Takeaways

In this module, you learned how to create engaging UIs for graphical applications. You learned how to use XAML to create windows and user controls and how you bind controls to data items and collections. You also learned how to provide a consistent user experience by creating styles as reusable resources.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You want to use rows and columns to lay out a UI. Which container control should you use?** | |
| Select the correct answer. | |
|  | The Canvas control. |
|  | The DockPanel control. |
|  | The Grid control. |
|  | The StackPanel control. |
|  | The WrapPanel control. |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You are creating an application that enables users to place orders for coffees. The application should allow users to select the drink they want from a list. Each list item should display the name of the coffee, the description, the price, and an image of the coffee. How should you proceed?** | |
| Select the correct answer. | |
|  | Create a ListBox control. Add child controls to the ListBox control to represent each field. |
|  | Create a ListBox control. Use a DataTemplate to specify how each field is displayed within a list item. |
|  | Create a ListBox control. Create a custom control that inherits from ListBoxItem, and use this custom control to specify how each field is displayed. |
|  | Create a ListBox control. Use the DisplayMemberPath property to specify the fields you want to display in each list item. |
|  | Create a ListBox control. Use a Style to specify how each field is displayed within a list item. |

Test Your Knowledge

|  |
| --- |
| **Question** |
| **You want to apply a highlighting effect to selected items in a ListBox. How should you proceed?** |
| Select the correct answer. |

|  |  |
| --- | --- |
| **Question** | |
|  | Create a Style element and set the TargetType attribute to ListBox. Use a Setter element to apply the highlighting effect. |
|  | Create a Style element and set the TargetType attribute to ListBox. Use a Trigger element to apply the highlighting effect when a list box item is selected. |
|  | Create a Style element and set the TargetType attribute to ListBox. Use an EventTrigger element to apply the highlighting effect when a list box item is selected. |
|  | Create a Style element and set the TargetType attribute to ListBox. Use a Storyboard element to apply the highlighting effect when a list box item is selected. |
|  | Create a Style element and set the TargetType attribute to ListBox. Use a DoubleAnimation element to apply the highlighting effect when a list box item is selected. |

**Lab 10: Improving the Responsiveness and Performance of the Application**

**Scenario**

You have been asked to update the Grades application to ensure that the UI remains responsive while the user is waiting for operations to complete. To achieve this improvement in responsiveness, you decide to convert the logic that retrieves the list of students for a teacher to use asynchronous methods. You also decide to provide visual feedback to the user to indicate when an operation is taking place.

**Objectives**

After completing this lab, you will be able to:

1. Use the **async** and **await** keywords to implement asynchronous methods.
2. Use events and user controls to provide visual feedback during long-running operations.

* Estimated Time: 75 minutes

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

**Scenario**

In this exercise, you will modify the functionality that retrieves data for teachers to make use of asynchronous programming techniques. First, you will modify the code that gets the details of the current user (when the user is a teacher) to run asynchronously. You will use an asynchronous task to run the LINQ query and use the **await** operator to return the results of the query. Next, you will modify the code that retrieves the list of students for a teacher. In this case, you will configure the code that retrieves the list of students to run asynchronously. When the operation is complete, your code will invoke a callback method to update the UI with the list of students. Finally, you will build and test the application and verify that the UI remains responsive while the application is retrieving data.

The main tasks for this exercise are as follows:

1. Build and run the application.
2. Modify the code that retrieves teacher data to run asynchronously.
3. Modify the code that retrieves and displays the list of students for a teacher to run asynchronously.
4. Build and test the application.

 **Task 1: Build and run the application**

1. Start File Explorer, navigate to the **E:\Mod10\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and then open the Grades.sln solution from the E:\Mod10\Labfiles\Starter\Exercise 1 folder.
3. Set the following projects to start without debugging at startup:
4. **Grades.Web**
5. **Grades.WPF**
6. Build the solution, and then resolve any compilation errors.
7. Run the application.
8. Log on as **vallee** with a password of **password99**.
9. 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).
10. Close the application.

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

1. In the Grades.WPF project, in the Services folder, open **ServiceUtils.cs**.
2. Modify the **GetTeacher** method declaration so that it:
   1. Runs asynchronously.
   2. Returns a **Task<Teacher>** object.
3. In the **GetTeacher** method, modify the LINQ query to use the **await** operator to run the query asynchronously and to return a **Task<Teacher>** object.
4. In the **MainWindow** class, modify the **Refresh** method declaration so that the method runs asynchronously.
5. In the **Refresh** method, in the **case "Teacher"** block, modify the call to the **GetTeacher** method to run asynchronously.

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

1. In the **StudentsPage** class, modify the **Refresh** method declaration so that the method runs asynchronously.
2. In the **Refresh** method, in the **Callbacks** region, locate the code that iterates through the returned set of students and updates the UI.
3. Factor out this code into a new method named **OnGetStudentsByTeacherComplete**. This method should:
   1. Accept a single argument of type **IEnumerable<Student>**.
   2. Return **void**.
   3. Use a **Dispatcher** object to update the UI.
4. In the **ServiceUtils** class, modify the **GetStudentsByTeacher** method declaration so that it:
   1. Runs asynchronously.
   2. Returns a **Task** object.
   3. Accepts a delegate argument that can represent the **OnGetStudentsByTeacherComplete**

callback method.

1. Modify the **GetStudentsByTeacher** method so that it returns without passing a value if the call to

**IsConnected** returns **false**.

1. In the **GetStudentsByTeacher** method, use the **await** operator to run the LINQ query asynchronously and return a **Task<Student>** object.
2. Modify the **GetStudentsByTeacher** method so that it invokes the callback method delegate asynchronously rather than returning the results.
3. In the **StudentsPage** class, modify the **Refresh** method so that it:
   1. Calls the **GetStudentsByTeacher** method asynchronously.
   2. Passes the **OnGetStudentsByTeacherComplete** method as the second argument.

 Task 4: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Verify that the application is more responsive than before while fetching the list of students for Esther Valle.
5. Close the application, and then close the 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

**Scenario**

In this exercise, you will create a user control that displays a progress indicator while the Grades application is retrieving data. You will add this user control to the main page but will initially hide it from view. Next, you will modify the code that retrieves data so that it raises one event when the data retrieval starts and another event when the data retrieval is complete. You will create handler methods for these events that toggle the visibility of the progress indicator control, so that the application displays the progress indicator when data retrieval starts and hides it when data retrieval is complete. Finally, you will build and test the application and verify that the UI displays the progress indicator while the application is retrieving data.

The main tasks for this exercise are as follows:

1. Create the BusyIndicator user control.
2. Add StartBusy and EndBusy event handler methods.
3. Raise the StartBusy and EndBusy events.
4. Build and test the application.

 **Task 1: Create the BusyIndicator user control**

1. In Visual Studio, and then open the Grades.sln solution from the E:\Mod10\Labfiles\Starter\Exercise 2 folder.
2. Set the following projects to start without debugging at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
3. Build the solution.
4. In the **Grades.WPF** project, create a new user control named **BusyIndicator.xaml**.
5. Move the **BusyIndicator.xaml** file 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.

1. Ensure that the user control does not specify dimensions by deleting the **DesignWidth** and

**DesignHeight** properties.

1. In the BusyIndicator.xaml file, modify the existing **Grid** element to have a background color of

#99000000.

1. In the **Grid** element, add a **Border** element with the following characteristics:
   1. Set the corner radius to **6** units.
   2. Set the horizontal and vertical alignments to **Center**.
   3. Set the background to a horizontal linear gradient from light gray to dark gray.
   4. Add a drop shadow effect with opacity of 0.75.
2. In the **Border** element, add a **Grid** element with the following characteristics:
   1. Set the margin to **10** units.
   2. Define two rows with automatically determined heights.
3. In the first row of the **Grid** element, add a **ProgressBar** element with the following characteristics:
   1. Set the width to **200** units.
   2. Set the height to **25** units.
   3. Set the margin to **20** units.
   4. Set the name of the control to **progress**.
   5. Set the progress bar to provide generic, continuous feedback rather than actual values (hint: use the **IsIndeterminate** attribute).
4. In the second row of the **Grid** element, add a **TextBlock** element with the following characteristics:
   1. Set the font size to **14** points.
   2. Apply the **Verdana** font.
   3. Set the text alignment to **Center**.
   4. Display the message **Please Wait…**
   5. Set the name of the control to **txtMessage**.
5. Your completed XAML markup should look like the following code:

<UserControl x:Class="Grades.WPF.BusyIndicator" xmlns=["http://schemas.microsoft.com/winfx/2006/xaml/presentatio](http://schemas.microsoft.com/winfx/2006/xaml/presentation)n" xmlns:x="[http://schemas.microsoft.com/winfx/2006/xaml"](http://schemas.microsoft.com/winfx/2006/xaml)>

<Grid Background="#99000000">

<Border CornerRadius="6" VerticalAlignment="Center" HorizontalAlignment="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>

<Grid Margin="10">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

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

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

</Grid>

</Border>

</Grid>

</UserControl>

1. Save all files.
2. In the Grades.WPF project, in the MainWindow.xaml file, add a **BusyIndicator** element with the following characteristics:
   1. Set the margin to **0**.
   2. Set the visibility to **Collapsed**.
   3. Set the name of the control to **busyIndicator**.
   4. Build the solution.

 Task 2: Add StartBusy and EndBusy event handler methods

1. In the **MainWindow** class, add a private method named **StartBusy**. The method should:
   1. Return **void**.
   2. Accept two arguments, of type **object** and **EventArgs** respectively.
2. In the **StartBusy** method, make the **busyIndicator** control visible.
3. In the **MainWindow** class, add a private method named **EndBusy**. The method should:
   1. Return **void**.
   2. Accept two arguments, of type **object** and **EventArgs** respectively.
4. In the **EndBusy** method, make the **busyIndicator** control hidden.

 Task 3: Raise the StartBusy and EndBusy events

1. In the **StudentsPage** class, add a public event named **StartBusy**.
2. Add a public event named **EndBusy**.
3. Add a method named **StartBusyEvent** that takes no arguments and returns **void**.
4. In the **StartBusyEvent** method, raise the **StartBusy** event.
5. Add a method named **EndBusyEvent** that takes no arguments and returns **void**.
6. In the **EndBusyEvent** method, raise the **EndBusy** event.
7. In the MainWindow.xaml file, in the **StudentsPage** element, associate the **StartBusy** event handler method with the **StartBusy** event.
8. In the **StudentsPage** element, associate the **EndBusy** event handler method with the **EndBusy** event.
9. In the **StudentsPage** class, at the start of the **Refresh** method, raise the **StartBusy** event by calling the **StartBusyEvent** method.
10. At the end of the **Refresh** method, raise the **EndBusy** event by calling the **EndBusyEvent** method.

 Task 4: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Verify that the application displays the busy indicator while waiting for the list of students to load.
5. Close the application, and then close the 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 Review and Takeaways

In this module, you have learned a variety of asynchronous programming techniques for Visual C#, including how to use the Task Parallel Library, how to use the **async** and **await** keywords, and how to use synchronization primitives.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You create and start three tasks named task1, task2, and task3. You want to block the joining thread until all of these tasks are complete. Which code example should you use to accomplish this?** | |
| Select the correct answer. | |
|  | task1.Wait();  task2.Wait();  task3.Wait(); |
|  | Task.WaitAll(task1, task2, task3); |
|  | Task.WaitAny(task1, task2, task3); |
|  | Task.WhenAll(task1, task2, task3); |
|  | Task.WhenAny(task1, task2, task3); |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You have a synchronous method with the following signature: public IEnumerable<string> GetCoffees(string country, int strength)**  **You want to convert this method to an asynchronous method. What should the signature of the asynchronous method be?** | |
| Select the correct answer. | |
|  | public async IEnumerable<string> GetCoffees(string country, int strength) |
|  | public async Task<string> GetCoffees(string country, int strength) |
|  | public async Task<IEnumerable<string>> GetCoffees(string country, int strength) |
|  | public async Task GetCoffees(string country, int strength, out string result) |
|  | public async Task GetCoffees(string country, int strength, out IEnumerable<string> result) |

Test Your Knowledge

**Question**

**You want to ensure that no more than five threads can access a resource at any one time. Which synchronization primitive should you use?**

|  |  |
| --- | --- |
| **Question** | |
| Select the correct answer. | |
|  | The ManualResetEventSlim class. |
|  | The SemaphoreSlim class. |
|  | The CountdownEvent class. |
|  | The ReaderWriterLockSlim class. |
|  | The Barrier class. |

**Lab 11: Upgrading the Grades Report**

**Scenario**

You have been asked to upgrade the grades report functionality to generate reports in Word format. In Module 6, you wrote code that generates reports as an XML file; now you will update the code to generate the report as a Word document.

**Objectives**

After completing this lab, you will be able to:

1. Use dynamic types.
2. Manage object lifetime and resources.

* Estimated Time: 60 minutes

**Exercise 1: Generating the Grades Report by Using Word**

**Scenario**

In this exercise, you will update the reporting functionality to generate reports in Word format.

First you will review the existing code in the **WordWrapper** class that appends headings, text, breaks, and carriage returns to a document. You will write code to create an instance of Word, create a blank Word document, and save a Word document. You will then review the code in the **GenerateStudentReport** method to create a blank document, add a heading and grade data to the document, and save the document using the methods that you reviewed and wrote in the **WordWrapper** class. You will run this method as a separate task. Finally, you will build and test the application and verify that the reports are now generated in Word format.

The main tasks for this exercise are as follows:

1. Examine the WordWrapper class that provides a functional wrapper around the dynamic (COM) API for Word.
2. Review the code in the GeneratedStudentReport method to generate a Word document.
3. Build and test the application.

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

1. Start File Explorer, navigate to the **E:\Mod11\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio and open the Grades.sln solution from the E:\Mod11\Labfiles\Starter\Exercise 1 folder.
3. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. In the Grades.Utilities project, review the contents of the WordWrapper.cs file.
2. In the **WordWrapper** class, add code to create a dynamic variable named **\_word** for activating Word.
3. In the class constructor, add code to instantiate the **\_word** variable as a new Word **Application**

object, but ensure that the application is not visible to the user.

1. In the **CreateBlankDocument** method, add code to create a new Word document and then activate the new document.
2. In the **SaveAs** method, add code to do the following:

* Save the active Word document using the file name that is passed to the method.
* Close the document.

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

1. In the Grades.WPF project, in the Views folder, open **StudentProfile.xaml.cs**.
2. In the **Utility and Helper methods** region, locate the **GenerateStudentReport** method.
3. Examine the code that is in this method to generate the student report.
4. In the **Events** region, in the **SaveReport\_Click** method, add code to generate the report by using a separate task, passing the student and the file name as parameters to the **GenerateStudentReport** method.

 Task 3: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Generate a grade report for Kevin Liu, save the report in the E:\Mod11\Labfiles\Starter\Exercise 1 folder, close the application, and in Visual Studio, close the solution.
5. Open Kevin Liu’s grade report in Word, review the 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

**Scenario**

In this exercise, you will write code to ensure that Word is correctly terminated after generating a grades report.

You will begin by observing that currently the Word object remains in memory after the application has generated a report. You will verify this by observing the running tasks in Task Manager. You will update the code in the **WordWrapper** class to implement the dispose pattern to ensure correct termination of the Word instance. You will then update the code in the **GenerateStudentReport** method to ensure that the **WordWrapper** object is disposed of when the method finishes. Finally, you will build and test the application and verify that Word now closes after the report is generated.

The main tasks for this exercise are as follows:

1. Run the application to generate a grades report and view the Word task in Task Manager.
2. Update the WordWrapper class to terminate Word correctly.
3. Wrap the object that generates the Word doc in a using statement.
4. Use Task Manager to observe that Word terminates correctly after generating a report.

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

1. In Visual Studio, open the Grades.sln solution from the E:\Mod11\Labfiles\Starter\Exercise 2 folder.
2. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Generate a grade report for Kevin Liu, save the report in the E:\Mod11\Labfiles\Starter\Exercise 2 folder, and then close the application.
5. Use File Explorer to verify that the report has been generated.
6. Open Task Manager, and then verify that Word is still running.
7. In Task Manager, end the Word task, and then close Task Manager.

 Task 2: Update the WordWrapper class to terminate Word correctly

1. In the Grades.Utilities project, open **WordWrapper.cs**.
2. Modify the definition of the **WordWrapper** class so that it implements the **IDisposable** interface.
3. In the **WordWrapper** class, create a protected virtual void method named **Dispose** that takes a Boolean parameter.
4. Add code to this method to do the following:

* Check that the class is not yet disposed of.
* Check whether the Boolean parameter that is passed to the method is **true**.
* Check whether the **\_word** variable is not null.

1. If all of these checks are valid, use the **Quit** method of the **Word** object to release the managed resources.
2. If the **\_word** variable is still not null, use the **ReleaseComObject** method to release the unmanaged resources.
3. Set the **isDisposed** property to **true**.
4. In the **WordWrapper** class, create a public method named **Dispose**.
5. Add code to the method to do the following:

* Call the **Dispose** method, passing **true** as a parameter.
* Call the **SuppressFinalize** method of the GC.

1. In the **WordWrapper** class, create a private Boolean variable named **isDisposed** and assign a default value of **false**.

 Task 3: Wrap the object that generates the Word doc in a using statement

1. In the Grades.WPF project, in the Views folder, open **StudentProfile.xaml.cs**.
2. In the **Utility and Helper methods** region, locate the **GenerateStudentReport** method.
3. Replace the current declaration for the wrapper variable with a **using** statement that declares the

**wrapper** object as a **var** and encloses all of the code in the method.

 Task 4: Use Task Manager to observe that Word terminates correctly after generating a report

1. Build the solution and resolve any compilation errors.
2. Open Task Manager.
3. Run the application.
4. Log on as **vallee** with a password of **password99**.
5. View George Li’s grade report.
6. Save the report in the E:\Mod11\Labfiles\Starter\Exercise 2 folder. As you click **Save**, in the **Task Manager** window, verify that **Microsoft Word (32 bit)** appears and then disappears from the **Background processes** group.
7. Close Task Manager, close the application, and then in Visual Studio, close the solution.

**Results**: After completing this exercise, the application will terminate Word correctly after it has generated a grades report.

### Module Review and Takeaways

In this module, you have learned how to interoperate with COM assemblies and how to ensure that your objects dispose of any resources they consume.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Which of the following statements best describes the dynamic keyword?** | |
| Select the correct answer. | |
|  | It defines an object of type object and instructs the compiler to perform type checking. |
|  | It defines a nullable object and instructs the compiler to defer type checking. |
|  | It defines an object of type object and instructs the compiler to defer type checking. |
|  | It defines a nullable object and instructs the compiler to perform type checking. |

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| You can use a **using** statement to implicitly invoke the **Dispose** method on an object that implements the **IDisposable** pattern. |  |

**Lab 12: Specifying the Data to Include in the Grades Report**

**Scenario**

You decide to update the Grades application to use custom attributes to define the fields and properties that should be included in a grade report and to format them appropriately. This will enable further reusability of the Microsoft Word reporting functionality.

You will host this code in the GAC to ensure that it is available to other applications that require its services.

**Objectives**

After completing this lab, you will be able to:

1. Define custom attributes.
2. Use reflection to examine metadata at run time.
3. Sign an assembly and deploy it to the GAC.

* Estimated Time: 75 minutes

**Exercise 1: Creating and Applying the IncludeInReport attribute**

**Scenario**

In this exercise, you will create the **IncludeInReport** attribute to specify the fields and the format of each field that is included in the grades report.

First, you will write code for the **IncludeInReportAttribute** class and define the members that are contained in it. Next, you will apply the attribute to the appropriate properties in the **LocalGrade** class in the Data.cs file. Finally, you will build the application and then use the MSIL Disassembler utility (IL DASM) to examine the metadata that the attribute generates.

The main tasks for this exercise are as follows:

1. Write the code for the IncludeInReportAttribute class.
2. Apply the IncludeInReportAttribute attribute to the appropriate properties.
3. Build the application and review the metadata for the LocalGrades class.

 **Task 1: Write the code for the IncludeInReportAttribute class**

1. Start File Explorer, navigate to the **E:\Mod12\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and then open the Grades.sln solution from the E:\Mod12\Labfiles\Starter\Exercise 1 folder.
3. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. In the Grades.Utilities project, in the **IncludeInReport.cs** class, add code to specify that the

**IncludeInReportAttribute** class is an attribute class.

1. Add code to specify that the possible targets to which the **IncludeInReport** attribute can be applied are fields and properties and that the attribute can be applied only once to each entity.
2. In the **IncludeInReportAttribute** class, define a private Boolean variable named **\_include** to hold the value of the attribute.
3. In the **IncludeInReportAttribute** class, define two public read/write Boolean properties named

**Underline** and **Bold**.

1. In the **IncludeInReportAttribute** class, define a public read/write string property named **Label**.
2. In the **IncludeInReportAttribute** class, create a default constructor that sets the properties as follows:

* \_include: **true**
* Underline: **false**
* Bold: **false**
* Label: **string.Empty**

1. Create another constructor that takes a Boolean parameter named **includeInReport** and sets the properties as follows:

* \_include: **\_includeInReport**
* Underline: **false**
* Bold: **false**
* Label: **string.Empty**

 Task 2: Apply the IncludeInReportAttribute attribute to the appropriate properties

1. In the Grades.WPF project, in the Data.cs file, in the **LocalGrade** class, add the **IncludeInReport**

attribute to the appropriate properties of the **LocalGrade** class as follows:

* **SubjectName** property:
* Label: **Subject Name**
* Bold: **true**
* Underline: **true**
* **AssessmentDate** property:
* Label: **Date**
* **Assessment** property:
* Label: **Grade**
* **Comments** property:
* Label: **Comments**

 Task 3: Build the application and review the metadata for the LocalGrades class

1. Build the solution, and then resolve any compilation errors.
2. Use the IL DASM utility to examine the metadata of the **LocalGrades** class in the **Grades.WPF.exe** assembly. The IL DASM utility is located in the C:\Program Files (x86)\Microsoft SDKs\Windows\v8.0A\bin\NETFX 4.0 Tools folder, and the **Grades.WPF.exe** assembly is located in the E:\Mod12\Labfiles\Starter\Exercise 1\Grades.WPF\bin\Debug folder.
3. Verify that the **IncludeInReport** attribute has been applied to the specified properties in the class.

**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

**Scenario**

In this exercise, you will update the grades report to include fields and properties only if they are tagged with the **IncludeInReport** attribute.

First, you will implement a method named **GetItemsToInclude** in a static helper class called **IncludeProcessor** that implements the logic that is necessary to discover the fields and properties that are tagged with the **IncludeInReport** attribute. You will then update the code for the **StudentProfile** view to include fields and properties in the report only if they are tagged with the **IncludeInReport** attribute and to format them appropriately.

The main tasks for this exercise are as follows:

1. Implement a static helper class called IncludeProcessor.
2. Update the report functionality for the StudentProfile view.
3. Build and test the application.

 **Task 1: Implement a static helper class called IncludeProcessor**

1. In Visual Studio, open the Grades.sln solution from the E:\Mod12\Labfiles\Starter\Exercise 2 folder.
2. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. In the Grades.Utilities project, open IncludeInReport.cs.
2. Locate the **FormatField** struct declaration, and then add two string members named **Value** and

**Label** and two Boolean members named **IsBold** and **IsUnderlined**.

1. In the **GetItemsToInclude** method of the **IncludeProcessor** class, after the variable declarations, add code to find all of the public fields and properties in the **dataForReport** object and use the **AddRange** method add them to the **fieldsAndProperties** list.
2. Add code to iterate through all of the public fields and properties.
3. In the loop, add code to retrieve all the custom attributes applied to an item.
4. Then use the **Array.Find** method to test whether any of the custom attributes applied to the item are the **IncludeInReport** attribute.
5. At the end of the loop, add code to find the value of the item that is tagged with the

**IncludeInReport** attribute.

1. At the end of the loop, add code to construct a **FormatField** item containing the value of the item and the label, bold, and underline values from the attribute.
2. At the end of the loop, add code to add the **FormatField** item to the collection to be returned.

 Task 2: Update the report functionality for the StudentProfile view

1. In the **Grades.WPF** project, in the **StudentProfile.xaml.cs** class, in the **GenerateStudentReport** method, at the start of the **foreach** loop, add code to use the **IncludeProcessor** class to determine which fields in the **Grade** object are tagged with the **IncludeInReport** attribute.
2. Add code to include each tagged item in the output, using the format that is specified by the properties of the **IncludeInReport** attribute for each item.

 Task 3: Build and test the application

1. Build the solution, and then resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Create and save a grades report for Kevin Liu.
5. Close the application, and then close the solution.
6. Open the grades report that you just created in Word and verify that the document contains a correctly formatted report.
7. 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)

**Scenario**

In this exercise, you will store the **Grades.Utilities** assembly in the GAC.

First, you will use Sn.exe to generate a key pair and then use the key pair to sign the **Grades.Utilities** assembly. Next, you will use Gacutil.exe to add the assembly to the GAC. You will then update the reference for the **Grades.Utilities** assembly in the Grades.WPF project to use the new signed assembly that is hosted in the GAC, and finally you will test the application to ensure that the reports are correctly generated.

The main tasks for this exercise are as follows:

1. Sign the Grades.Utilities assembly and deploy it to the GAC.
2. Reference the Grades.Utilities assembly in the GAC from the application.

 **Task 1: Sign the Grades.Utilities assembly and deploy it to the GAC**

1. In Visual Studio, open the Grades.sln solution from the E:\Mod12\Labfiles\Starter\Exercise 3 folder.
2. Set the following projects to start without debugging at startup:

* **Grades.Web**
* **Grades.WPF**

1. Run the **VS2012 x86 Native Tools Command Prompt** window as Administrator.
2. Run the Sn.exe utility to create a key pair file named **GradesKey.snk** in the E:\Mod12\Labfiles\Starter folder.
3. In Visual Studio, set the properties of the Grades.Utilities project to use the key pair that you have just created to sign the assembly.
4. Build the solution, and then resolve any compilation errors.
5. At the command prompt, use the Gacutil utility to add the **Grades.Utilities** assembly to the GAC.
6. Close the **Command Prompt** window.

 Task 2: Reference the Grades.Utilities assembly in the GAC from the application

1. In Visual Studio, remove the current reference to the Grades.Utilities project from the Grades.WPF project.
2. Add a new reference to the signed **Grades.Utilities.dll** assembly.
3. Build the solution, and then resolve any compilation errors.
4. Run the application.
5. Log on as **vallee** with a password of **password99**.
6. Create and save a grades report for Kevin Liu.
7. Close the application, and then close the solution.
8. Open the grades report that you just created in Word and verify that the document contains a correctly formatted report.
9. Close Word.

**Results**: After completing this exercise, you will have a signed version of the **Grades.Utilities** assembly deployed to the GAC.

### Module Review and Takeaways

In this module, you learned how to consume existing assemblies by using reflection and how to add additional metadata to types and type members by using attributes. You also learned how to generate code at run time by using CodeDOM and how you can ensure that your assemblies are versioned and available to other applications by using the GAC.

**Review Question(s)**

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| You are developing an application that enables users to browse the object model of a compiled type. At no point will the application attempt to execute any code; it will merely serve as a viewer. You notice the code that loads the assembly uses the **Assembly.LoadFrom** static method. This is the most suitable method taking into account the requirements of the application. |  |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You are developing a custom attribute. You want to derive your custom attribute class from the abstract base class that underpins all attributes. Which class should you use?** | |
| Select the correct answer. | |
|  | Attribute |
|  | ContextAttribute |
|  | ExtensionAttribute |
|  | DataAttribute |
|  | AddInAttribute |

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **You are reviewing some code that uses CodeDOM to generate managed Visual C# at run time. What does the following line of code do?**  **var method = new CodeEntryPointMethod();** | |
| Select the correct answer. | |
|  | Defines an instance method with a random name. |
|  | Defines an instance method named EntryPoint. |
|  | Defines a static method named EntryPoint. |

|  |  |
| --- | --- |
| **Question** | |
|  | Defines an instance method named Main. |
|  | Defines a static method named Main. |

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| The **FourthCoffee.Core.dll** assembly has 2.1.0.24 as its version number. The number 24 in the version number refers to the build number. |  |

**Lab 13: Encrypting and Decrypting the Grades Report**

**Scenario**

You have been asked to update the Grades application to ensure that reports are secure when they are stored on a user's computer. You decide to use asymmetric encryption to protect the report as it is generated, before it is written to disk. Administrative staff will need to merge reports for each class into one document, so you decide to develop a separate application that generates a combined report and prints it.

**Objectives**

After completing this lab, you will be able to:

1. Encrypt data by using asymmetric encryption.
2. Decrypt data.

* Estimated Time: 60 minutes

**Exercise 1: Encrypting the Grades Report**

**Scenario**

In this exercise, you will update the reporting functionality to encrypt the report as it is generated, but before it is saved to disk.

First, you will create an asymmetric certificate by using a prewritten batch file. The batch file uses the MakeCert tool that ships with the Windows Software Development Kit (SDK). You will create a self-signed certificate named Grades using the SHA-1 hash algorithm and store it in the LocalMachine certificate store. You will then write code in the Grades application to retrieve the certificate by looping through the certificates in the LocalMachine store and checking the name of the certificate against the name that is stored in the App.Config file. Next, you will use the classes that are provided in the **System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces to write the **EncryptWithX509** method in the **Grades.Utilities.WordWrapper** class. You will get the public key from the certificate that you created and use it to create an instance of the **RSAPKCS1KeyExchangeFormatter** class. You will use this to encrypt the data for the report and then return the encrypted buffered data to the calling method as a byte array. You will then write code in the **EncryptAndSaveToDisk** method to write the returned data to the file that the user specifies. Finally, you will build and test the application and verify that the reports are now encrypted.

The main tasks for this exercise are as follows:

1. Create an asymmetric certificate.
2. Retrieve the Grade certificate.
3. Encrypt the data.
4. Write the encrypted data to disk.
5. Build and test the application.

 **Task 1: Create an asymmetric certificate**

1. Start File Explorer, navigate to the **E:\Mod13\Labfiles\Databases** folder, and then run

**SetupSchoolGradesDB.cmd**.

1. Close File Explorer.
2. Start Visual Studio, and then open the **Grades.sln** solution from the

**E:\Mod13\Labfiles\Starter\Exercise 1** folder.

1. Set the following projects to start without debugging at startup:
   1. **Grades.Web**
   2. **Grades.WPF**
2. In the Grades.Utilities project, review the contents of the CreateCertificate.cmd file.
3. In a command window running as Administrator, navigate to the

**E:\Mod13\Labfiles\Starter\Exercise 1\Grades.Utilities** folder and then run **CreateCertificate.cmd**.

1. Verify that the command returns a success message, and then close the command window.

 Task 2: Retrieve the Grade certificate

1. In the **Grades.Utilities** project, in the **WordWrapper** class, locate the **GetCertificate** method.
2. Add code to this method to loop through the certificates in the **store.Certificates** collection.
3. Inside the loop, if the **SubjectName.Name** property matches the **\_certificateSubjectName** variable, return the certificate to the calling method.

 Task 3: Encrypt the data

1. In the **Grades.Utilities** project, in the **WordWrapper** class, locate the **EncryptWithX509** method.
2. Add code to this method to get the public key from the X509 certificate by using the **PublicKey.Key**

property, cast it to a **RSACryptoServiceProvider** object, and store it in a variable named **provider**.

1. In the **EncryptWithX509** method, add code to create an instance of the **AesManaged** encryption class named **algorithm**. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
2. Add code to create an instance of the **MemoryStream** class to hold the unencrypted data. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
3. Add the following code to create an AES encryptor based on the key and IV.

using (var encryptor = algorithm.CreateEncryptor())

{

var keyFormatter = new RSAPKCS1KeyExchangeFormatter(provider); var encryptedKey = keyFormatter.CreateKeyExchange(algorithm.Key,

algorithm.GetType());

1. Add a closing brace for the **using** statement at the end of the method.
2. Add the following code to create byte arrays to get the length of the encryption key and IV.

var keyLength = BitConverter.GetBytes(encryptedKey.Length); var ivLength = BitConverter.GetBytes(algorithm.IV.Length);

1. Add code to write the following data to the unencrypted memory stream object by using the **Write**

method of the **MemoryStream** instance.

* 1. The length of the secret key.
  2. The length of the IV.
  3. The encrypted secret key.
  4. The IV.
  5. The encrypted data.

1. Add code to create an instance of a **CryptoStream** object, passing the unencrypted memory stream, the AES encryptor, and the **CryptoStreamMode.Write** constant as parameters. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
2. Add code to call the **Write** and **FlushFinalBlock** methods of the **CryptoStream** object to write all the data to the memory stream.
3. Add code to return the encrypted buffered data to the calling method.

 Task 4: Write the encrypted data to disk

1. In the Grades.Utilities project, in the **WordWrapper** class, in the **EncryptAndSaveToDisk** method, add code to write the encrypted bytes to the file path passed to the method.

 Task 5: Build and test the application

1. Build the solution and resolve any compilation errors.
2. Run the application.
3. Log on as **vallee** with a password of **password99**.
4. Generate grade reports for George Li and Kevin Liu, saving each report in the E:\Mod13\Labfiles\Reports folder.
5. Close the application, and then close the solution.
6. Attempt to open one of the reports that you created in the previous step by using Internet Explorer and Notepad.

**Results**: After completing this exercise, you should have updated the Grades application to encrypt generated reports.

###### Exercise 2: Decrypting the Grades Report

**Scenario**

In this exercise, you will create a separate utility to enable users to print reports. Users will be able to select a folder that contains encrypted reports, and the application will then generate one combined report and send it to the default printer.

First, you will use the classes that are provided in the **System.Security.Cryptography** and **System.Security.Cryptography.X509Certificates** namespaces to write the **DecryptWithX509** method in the **SchoolReports.WordWrapper** class. You will get the private key from the certificate and use it to create an instance of the **RSACryptoServiceProvider** class. You will use this to decrypt the data from the individual reports and then return the decrypted data to the calling method as a byte array. Finally, you will build and test the application and verify that a printed version of the composite report has been generated.

The main tasks for this exercise are as follows:

1. Decrypt the data.
2. Build and test the solution.

 **Task 1: Decrypt the data**

1. In Visual Studio, open the School-Reports.sln solution from the E:\Mod13\Labfiles\Starter\Exercise 2 folder.
2. In the **WordWrapper** class, locate the **DecryptWithX509** method.
3. Add code to this method to get the private key from the X509 certificate by using the **PrivateKey**

property, cast it to a **RSACryptoServiceProvider** object, and store it in a variable named provider.

1. In the **DecryptWithX509** method, add code to create an instance of the **AesManaged** encryption class named **algorithm**. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
2. Add code to create an instance of the **MemoryStream** class, passing the byte array that the method received as a parameter. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
3. Add the following code to create byte arrays to get the length of the encryption key and IV.

var keyLength = new byte[4]; var ivLength = new byte[4];

1. Add code to read the key and IV lengths from index 0 in the memory stream and then convert the two lengths to integers.
2. Add code to determine the starting position and length of the encrypted data.
3. Add code to create byte arrays to store the encrypted key, the IV, and the encrypted data.
4. Add code to read the key, IV, and encrypted data from the memory stream and store them in the byte arrays that you have just created.
5. Add code to decrypt the encrypted AES managed key by calling the **Decrypt** method of the **provider**

object.

1. Add code to create a new instance of the **MemoryStream** class to store the decrypted data. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
2. Add code to create an AES decryptor object, passing the decrypted key and the IV as parameters. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
3. Add code to create an instance of a **CryptoStream** object, passing the memory stream for the decrypted data, the AES decryptor, and the **CryptoStreamMode.Write** constant as parameters. Enclose this line of code in a **using** statement and add a closing brace at the end of the method.
4. Add code to call the Write and **FlushFinalBlock** methods of the **CryptoStream** object to write all of the data to the memory stream.
5. Add code to return the decrypted buffered data to the calling method.

 Task 2: Build and test the solution

1. Build the solution, and then resolve any compilation errors.
2. Run the application, and then print a composite report that contains the two reports that you generated earlier. Save the .oxps file in the E:\Mod13\Labfiles\Reports\ClassReport folder.
3. Close the application, close the solution, and then close Visual Studio.
4. Open the composite report in the XPS Viewer and verify that the data has printed correctly.

**Results**: After completing this exercise, you should have a composite unencrypted report that was generated from the encrypted reports.

### Module Review and Takeaways

In this module, you learned how to implement symmetric and asymmetric encryption and how to use hashes to generate mathematical representations of your data.

**Review Question(s)**

Test Your Knowledge

|  |  |
| --- | --- |
| **Question** | |
| **Fourth Coffee wants you to implement an encryption utility that can encrypt and decrypt large image files. Each image will be more than 200 megabytes (MB) in size. Fourth Coffee envisages that only a small internal team will use this tool, so controlling who can encrypt and decrypt the data is not a concern. Which of the following techniques will you choose?** | |
| Select the correct answer. | |
|  | Symmetric encryption |
|  | Asymmetric encryption |
|  | Hashing |

Verify the correctness of the statement by placing a mark in the column to the right.

|  |  |
| --- | --- |
| **Statement** | **Answer** |
| Is the following statement true or false? Asymmetric encryption uses a public key to encrypt data. |  |