Data processing
420-TDD-ID v1

Version:1.0

Original text: Yves Desharnais;
Translation and adjustments: Daniel Desmeules

Module 1

Graphical user interface programming with WPF

WPF or *Windows Presentation Foundation* is a Microsoft technology with which we can create GUI (*graphical user interface*) applications. In this next lesson, we will learn more about GUI development and how it makes C# applications more attractive to users.

1A –Understanding the technical aspects of WPF

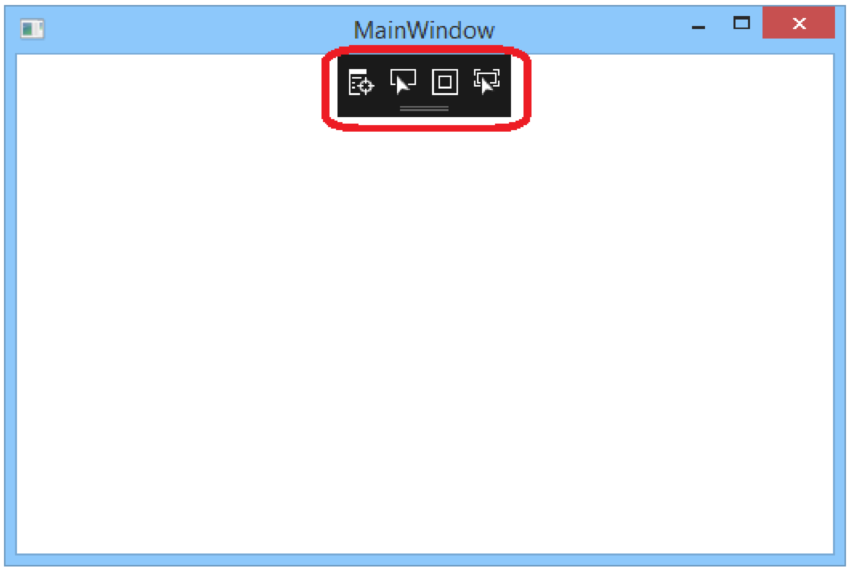
In this first lesson, we will learn about the basics of GUI. We will see how to design such an application and how it works.

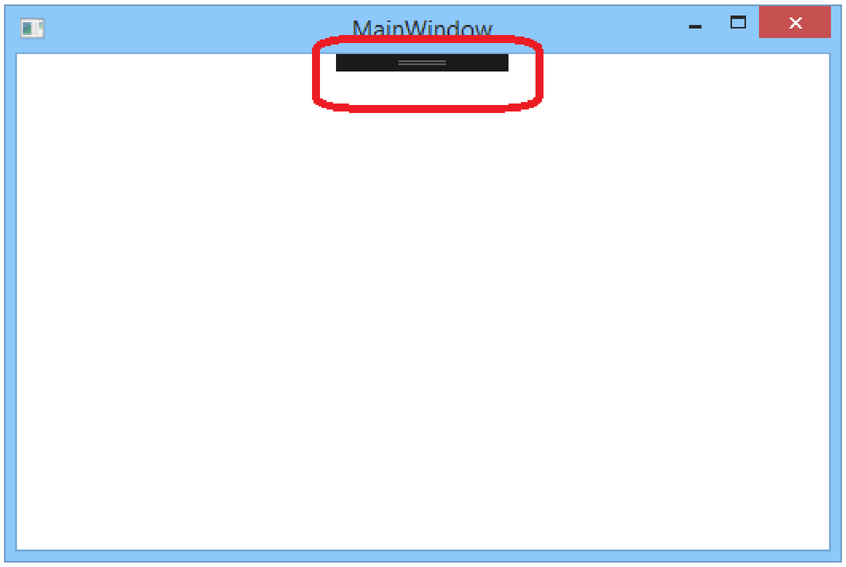
It's important to take a look at some of the technical aspects of Visual Studio to truly understand GUI programming.

OBJECTIVES

- ➢ Design the layout of a GUI application.
- ➢ Configure the Visual Studio design tools

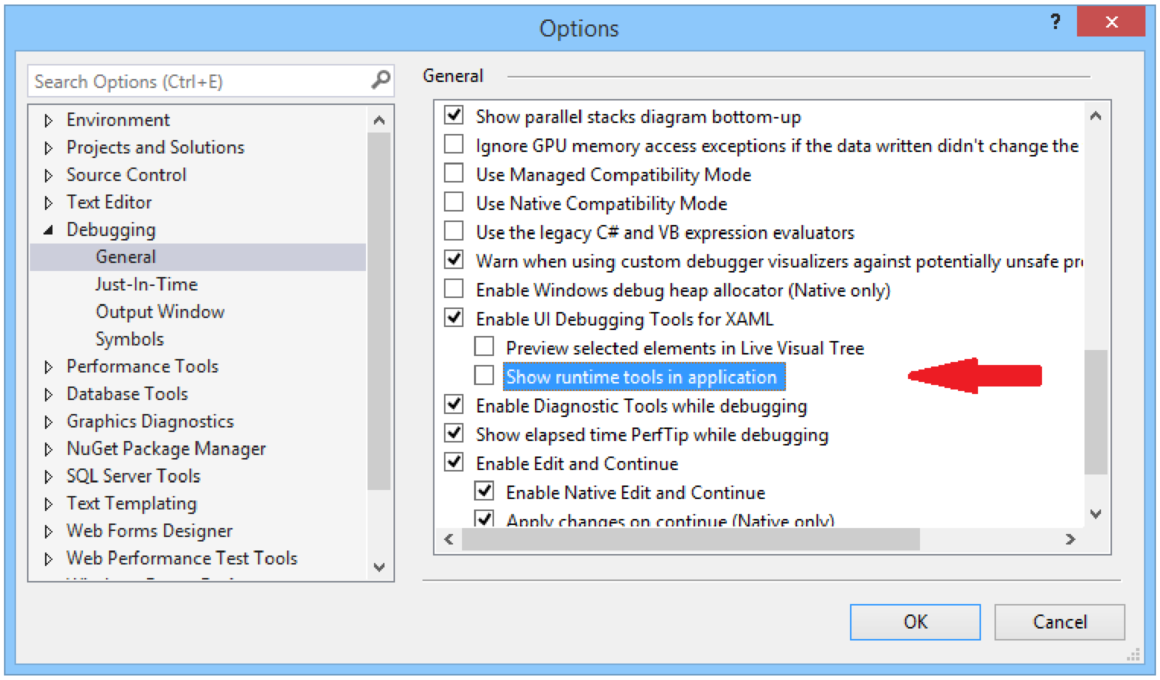
Before we start to go in depth with GUI applications, we will make some small adjustments. First, Using Visual Studio, create a new WPF application and give it any name you want. Then, run the application. You should get one of the two following windows:





These circled items are debugging tool boxes. If you see one in your window, you should disable it as it can become cumbersome while doing this course. To do so, go in the

Debug menu and select ***Options...*** In the ***General*** section, uncheck the ***Show runtime tools in the application*** option:



Options

Search Options (Ctrl+E)

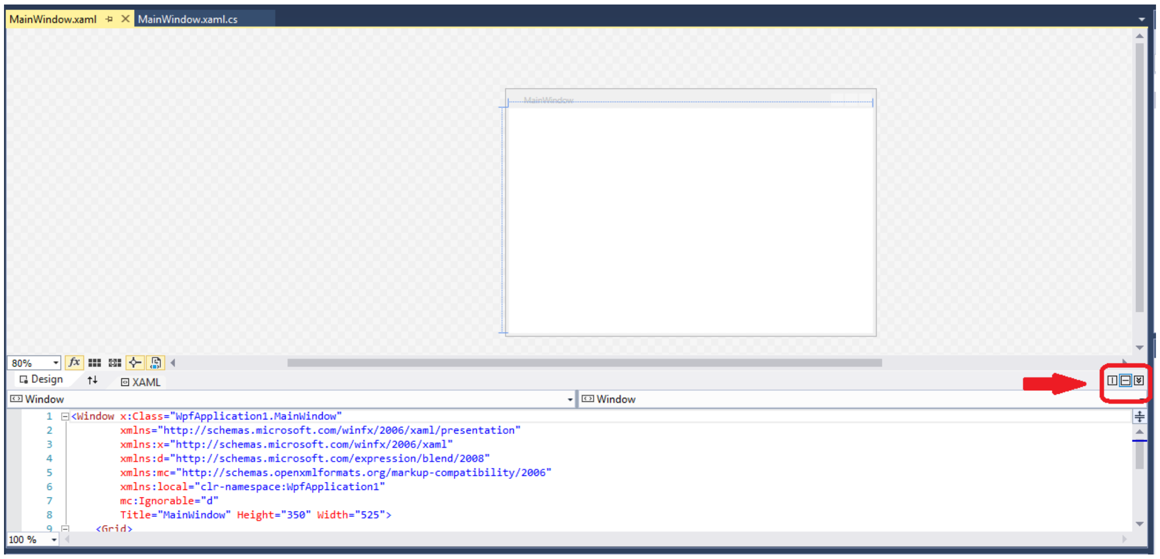
- Environment
- Projects and Solutions
- Source Control
- Text Editor
- Debugging
  - General
  - Just-In-Time
  - Output Window
  - Symbols
- Performance Tools
- Database Tools
- Graphics Diagnostics
- NuGet Package Manager
- SQL Server Tools
- Text Templating
- Web Forms Designer
- Web Performance Test Tools

General

- [✓] Show parallel stacks diagram bottom-up
- [ ] Ignore GPU memory access exceptions if the data written didn't change the
- [ ] Use Managed Compatibility Mode
- [ ] Use Native Compatibility Mode
- [ ] Use the legacy C# and VB expression evaluators
- [✓] Warn when using custom debugger visualizers against potentially unsafe pr
- [ ] Enable Windows debug heap allocator (Native only)
- [✓] Enable UI Debugging Tools for XAML
  - [ ] Preview selected elements in Live Visual Tree
  - [ ] Show runtime tools in application
- [✓] Enable Diagnostic Tools while debugging
- [✓] Show elapsed time PerfTip while debugging
- [✓] Enable Edit and Continue
  - [✓] Enable Native Edit and Continue
  - [✓] Apply changes on continue (Native only)

If you run your application again, you should see that the box has disappeared.

Now let's take a look at the work area itself. When a WPF project is created, Visual Studio should show you the design window which represents your user interface. This screen should be split in two: the top area shows you the design look of the interface and the bottom part shows you the XAML code that makes up this interface (You will learn more about XAML later in the course). Notice that between the two, to the right, there are three buttons:

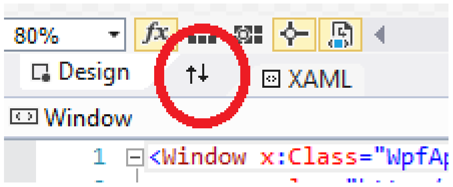


```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfApplication1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
```

These three buttons control how your work area is displayed:

◫	Splits the screen sideways
⊟	Splits the screen top/bottom (by default)
⊻	Shows or hides the XAML code

Another interesting tidbit: on the left side of the screen, between the two sections, you will find these two arrows:



```
<Window x:Class="WpfAp
```

If you click on them, the two sections will switch positions.

Now that you know how to configure your work area, we'll start the creation of an actual application.

1B –The Window control

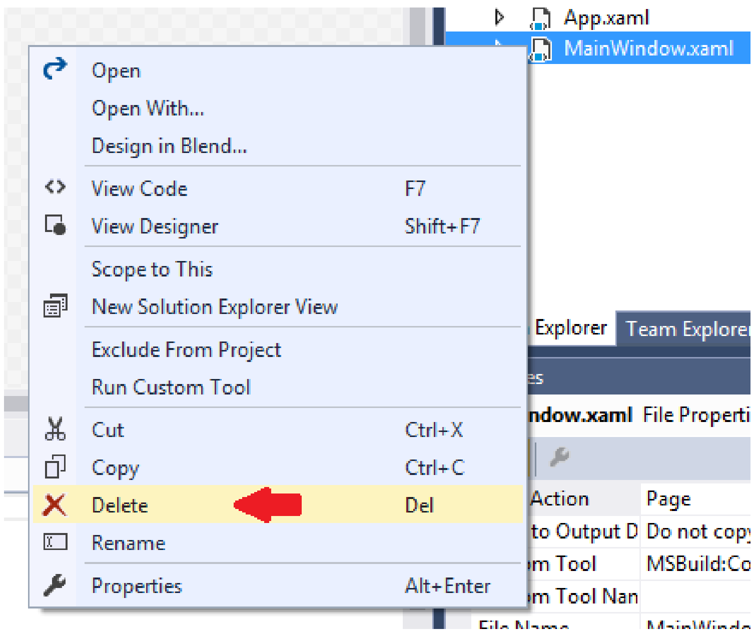
We will now see how to create an actual WPF application. First, we are going to take a look at the main control that holds everything together in a WPF form: the Window control.

OBJECTIVES

- Understand how to create a WPF form.
- Work with properties of a WPF form control.

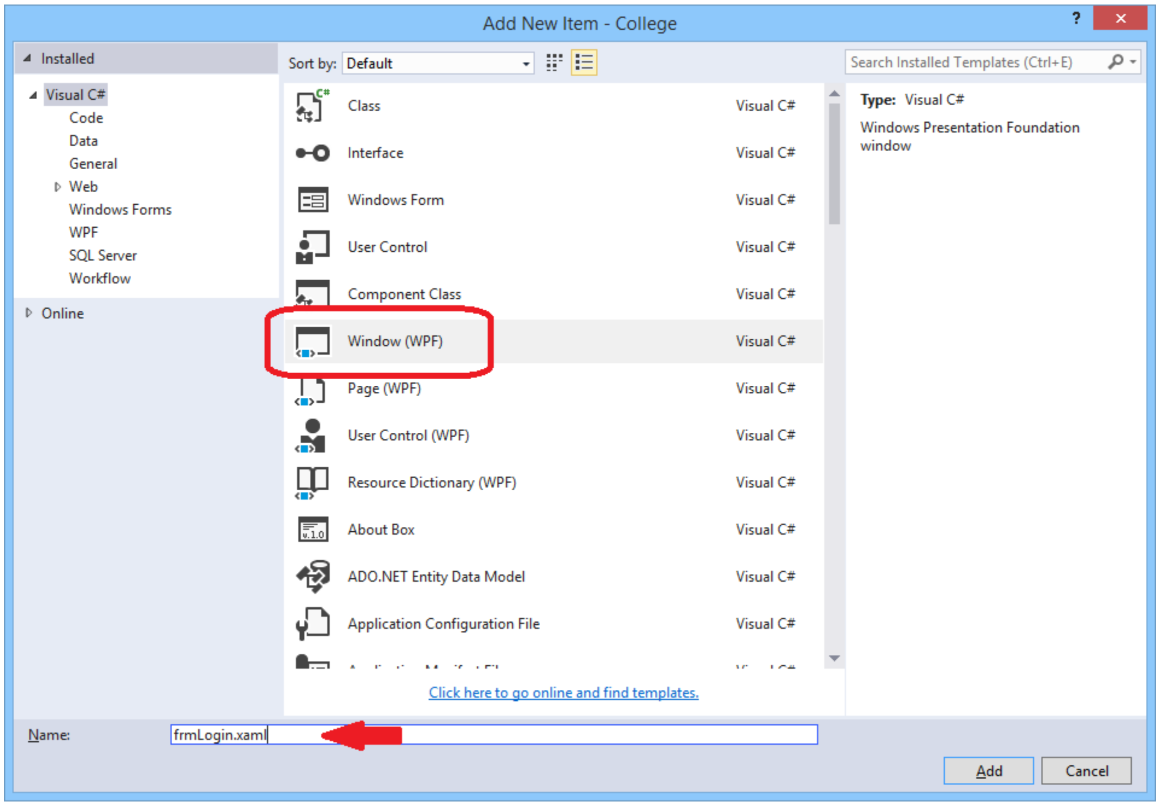
We are now going to create an application that manages students in a school. We will start by designing the graphical interface of this application. In Visual Studio, create a new C# Application of **WPF** type. In the ***Name*** field, enter **College** and in the ***Solution Name*** field, type **CDICollege**. Click the **OK** button. When the solution is created, you will see that a form called ***MainWindow*** has been created automatically. We will delete this form in order to create our own. Let's start with the removal of the ***MainWindow*** form.

In the ***Solution Explorer*** window on the right, right-click on the **MainWindows.xaml** file. Then, click on the **Delete** option of the contextual menu.



We will now add a new form to our solution. Right-click on the **College** element in the ***Solution Explorer*** and select **Add** - ***New Item...***

In the ***Add New Item*** window, select **Window(WPF)** and name it **frmLogin.xaml** :



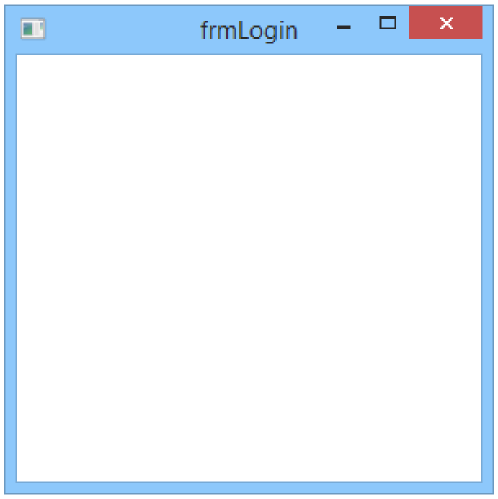
Name: frmLogin.xaml

Your new form will now appear on the screen. We must now set it as the starting form for the application. To do so, we will have to modify the XAML code of the app.

The first thing we need to do is change the **StartupUri** value. ***StartupUri*** is the thing that tells the app what his startup form is. The startup form is the first form that pops up when the application starts. To change this value, you need to go in the **App.xaml** file. In the ***Solution Explorer***, double-click on the ***App.xaml*** file to open it, then update it like so:

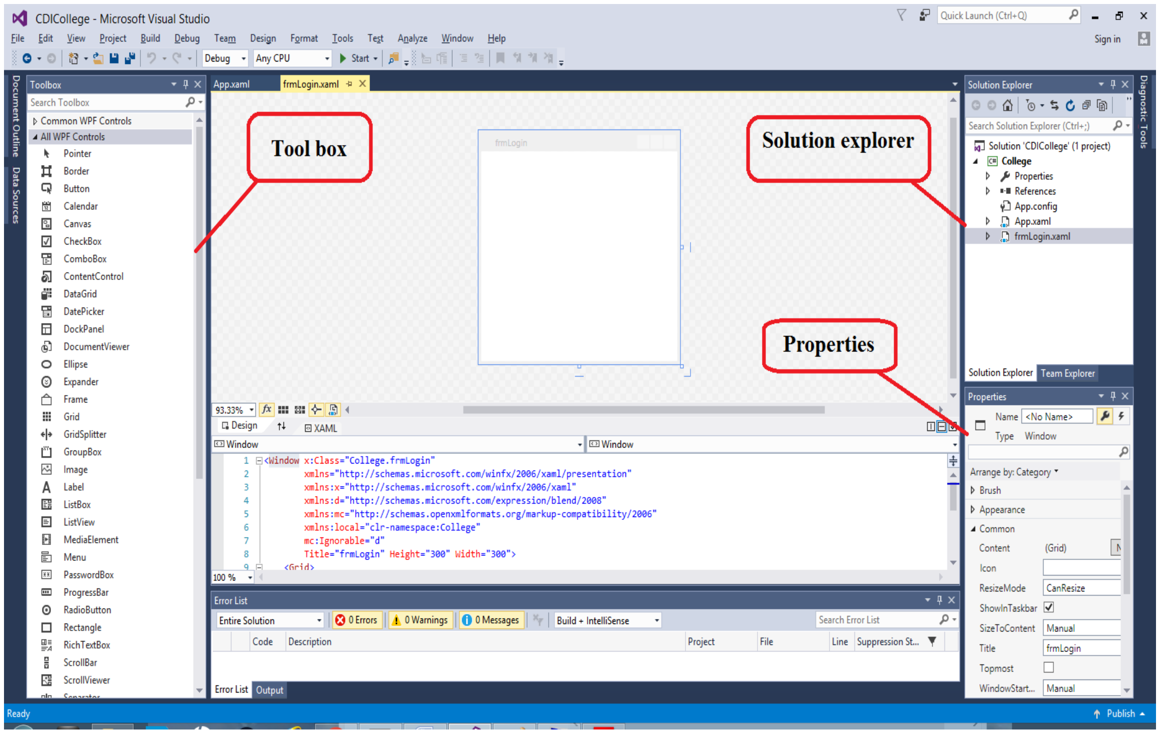
```
<Application x:Class="College.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:College"
             StartupUri="frmLogin.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Now try to execute the app. You should see the following window appear:



Great! The correct form pops up so it means our modification worked.

Now let's take a closer look at the Visual Studio environment when it comes to WPF app development. Here are the general sections of this environment:

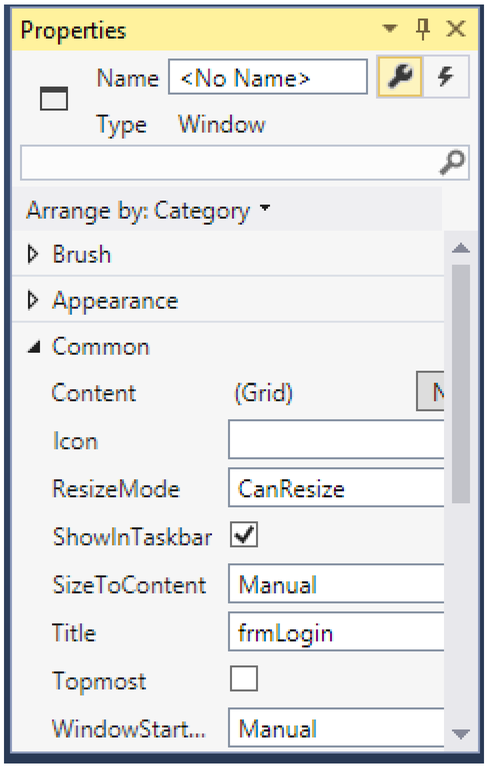


```
<Window x:Class="College.frmLogin"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="frmLogin" Height="300" Width="300">
    <Grid>
```

Name	<No Name>
Type	Window
Content	(Grid)
ResizeMode	CanResize
ShowInTaskbar	✓
SizeToContent	Manual
Title	frmLogin
Topmost	☐
WindowStart...	Manual

The central section of this screen is where we will design the interface of the form. Aside from that, there is also the **tool box**, **Solution explorer** and the **property window**. The tool box area includes two sections: ***Common WPF controls*** and ***All WPF controls.*** This is where you will find the controls that compose the interface. A ***control*** is a small graphical unit that can interact with a user. For example, buttons and text boxes are controls.

Let's take a look at one of the more important controls of a WPF application, the ***Window*** control. Click on the title bar of the ***frmLogin*** window at the center of your screen. This should select the **Window** control element of your interface. Now if you look in the property window, it should display the properties of the **Window** control:



Name <No Name>
Type Window

Content (Grid)
Icon
ResizeMode CanResize
ShowInTaskbar ✓
SizeToContent Manual
Title frmLogin
Topmost
WindowStart... Manual

Any change made in this property window will now affect the Window control. It will also update the XAML code to reflect these changes. In fact, the property window is just an easier way to update the XAML file, anything you do here could also be done directly in the XAML code. So if we want to give a name to this **Window** control, we could do it in the property window or directly in XAML. Let's do it in the property window. Go in the **Name** field and type **frmLogin1**. After that, take a look at the XAML code. You will see that it has been changed automatically:

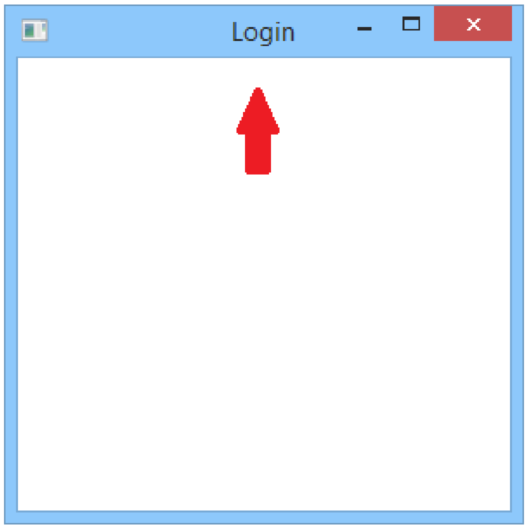
```
<Window x:Name="frmLogin1" x:Class="College.frmLogin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/
2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="frmLogin" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

Note that the reason we named the Window control ***frmLogin1*** instead of just plain ***frmLogin*** is to avoid having an object named the same way as its class. Doing so would generate an error.

Now if we go back to the property window, we should be able to find a **Title** property. Go to this field and change it to ***Login***. The ***Title*** property sets the text that his shown in the title bar of the form. This change will also appear in the XAML code:

```
<Window x:Name="frmLogin1" x:Class="College.frmLogin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/
2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="Login" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

Now run the application. You should see the title ***Login*** in the title bar of the form:



Note: Depending on your Windows/Visual Studio setup, it's possible that the ***Login*** *title appears on the left instead of being centered.*

You will now try to change the following properties yourself by locating them in the property window:

Height: **171**

Width: **507**

WindowsStartupLocation: **CenterScreen**

ResizeMode: **NoResize**

If you are having difficulties finding the properties in the property window, try to reorder them by name. There is an ***arrange by*** dropdown list just for that near the top of the window.

Again, when you are done, you should see the changes made in the XAML code:

```
<Window x:Name="frmLogin1" x:Class="College.frmLogin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/
2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="Login" Height="171" Width="507"
WindowStartupLocation="CenterScreen" ResizeMode="NoResize">
    <Grid>

    </Grid>
```

```
</Window>
```

Run the program again. You should notice two things. First, the application starts at the center of the screen. Second, you should not be able to resize it.

SUMMARY

The main control of a WPF application is the ***Window*** control.

In this lesson, you have seen some of the properties of this control:

- **Height:** Sets or gets the height of the form
- **Width:** Sets or gets the width of the form
- **WindowStartupLocation:** Sets the starting position of the form
- **ResizeMode:** Sets the resizing options of the window
- **Title:** Sets the text that will be shown in the title bar of the window
- **Name:** Sets the name of the Window control. This name must be unique in the solution

1C –The container controls

We will now take a look at another kind of control: the container controls. These controls are made to contain other controls so we can group them and better position them on the screen.

OBJECTIVES

- Understand what a container control is.
- Use a container control in a WPF application.

In a WPF application, some controls are used to contain other controls. They are called container controls. Here is a list of these controls:

- Grid
- StackPanel
- DockPanel
- WrapPanel
- ScrollViewer
- Canvas

When a new WPF application is created, it is the **Grid** control that is used by default. If we take a look at the XAML of your ***Login*** form, we can easily find the **Grid** control tags:

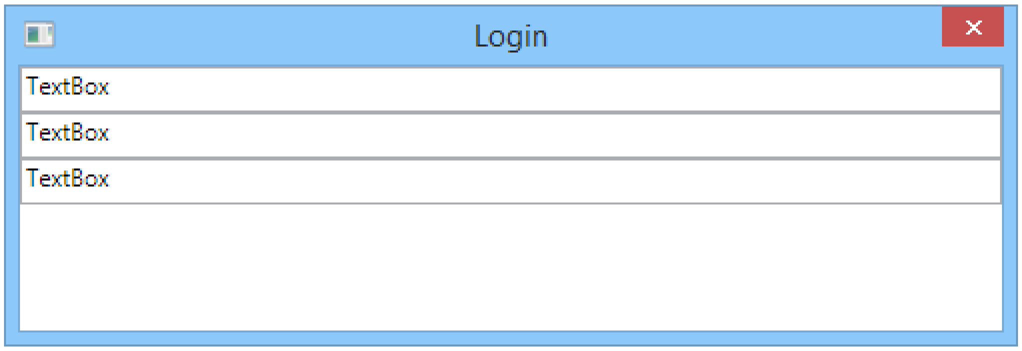
```
<Window x:Name="frmLogin1" x:Class="College.frmLogin"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentatio
n"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/
2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="Login" Height="171" Width="507"
WindowStartupLocation="CenterScreen" ResizeMode="NoResize">
    <Grid>

    </Grid>
</Window>
```

Much like an HTML document, XAML documents use ***tags*** to declare controls. In the case of a container control, everything that is located between the two tags will be contained in the control. The **Grid** control positions its inner controls in a grid-like fashion. We will see more about that when we continue work on our **College** project.

The **StackPanel** control positions its elements by stacking them, either horizontally or vertically, depending on the value of its ***Orientation*** property. The default is vertical. The following XAML code creates a StackPanel containing three textboxes:

```
<StackPanel Orientation="Vertical">
            <TextBox x:Name="textBox" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
            <TextBox x:Name="textBox1" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
            <TextBox x:Name="textBox2" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
</StackPanel>
```

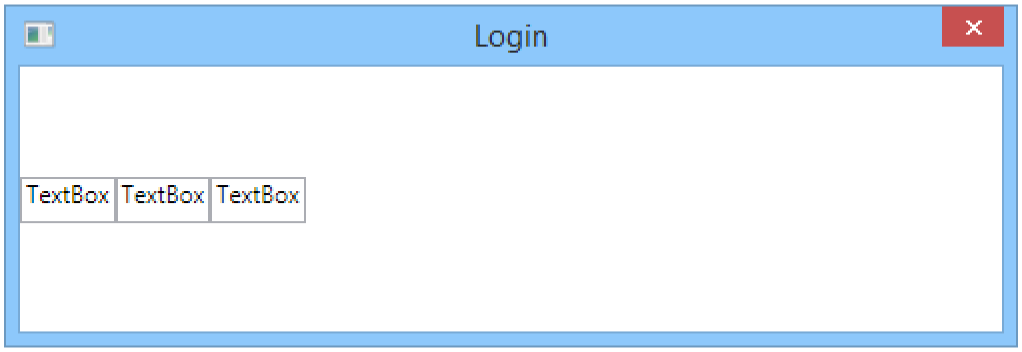
Try to remove the <Grid> </Grid> code in your file and replace it with this code. Run the program and you should see the textboxes stacked vertically:



Now change the **Orientation** to ***Horizontal*** in the code:

```
<StackPanel Orientation="Horizontal">
            <TextBox x:Name="textBox" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
            <TextBox x:Name="textBox1" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
            <TextBox x:Name="textBox2" Height="23"
TextWrapping="Wrap"
            Text="TextBox"/>
</StackPanel>
```

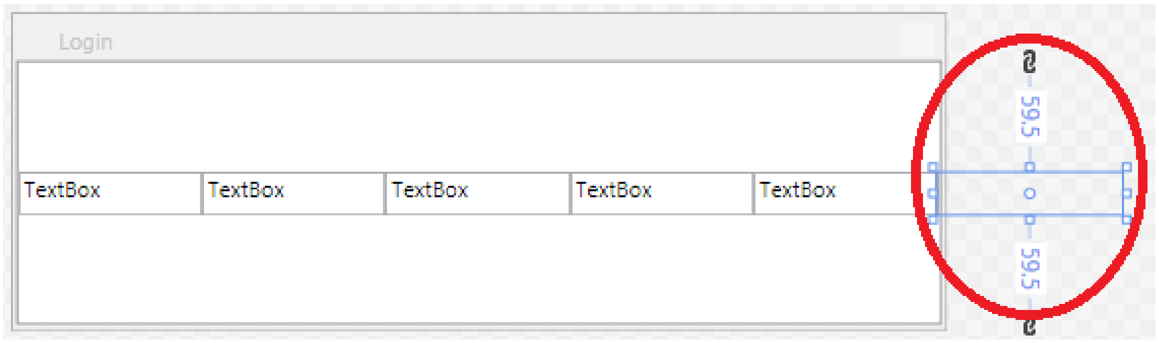
And you will get the following result when you run the app:



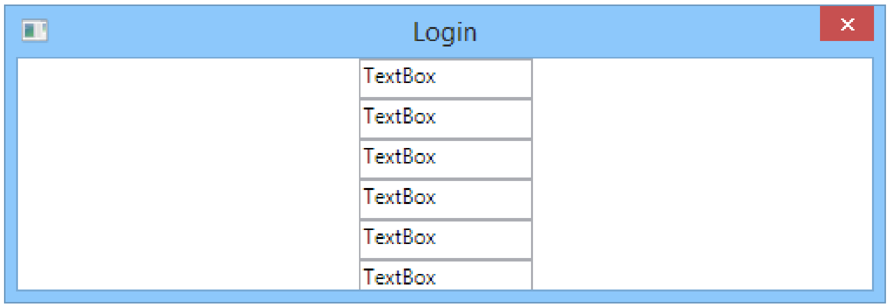
As you can see, all the textboxes are stacked either horizontally or vertically. Let's play a little bit with the TextBoxes' properties to see what happens. Change the **StackPanel** like so:

```
<StackPanel Orientation="Horizontal">
        <TextBox x:Name="textBox" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox1" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox2" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox3" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox4" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox5" Height="23" Width="100"
Text="TextBox"/>
</StackPanel>
```

If you look in the design window on top of the XAML code, you will see that the sixth textbox is outside of the **StackPanel**:



This is troublesome because the textbox will not be shown when you run the app. Now let's try to change the orientation again to ***Vertical*** and see what we have:



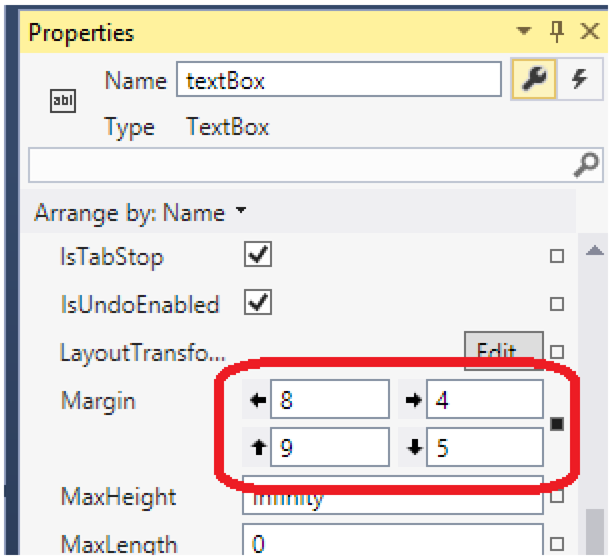
All the textboxes are there, but they are centered in the StackPanel. This is because now the textboxes have an actual width so they don't take the entire space available in the StackPanel. And by default, the StackPanel aligns its content in the middle. Also, have you noticed that there is no space between the textboxes? That's because we haven't set any margins to them. To set a margin to a control, we must give a value to its **Margin** property. Here are three examples of the **Margin** syntax:

Margin = "10"	This will put a 10 pixels spacing all around the control
Margin = "10,5"	This will put a 10 pixels spacing to the left and right of the control and a 5 pixels space at the top and bottom
Margin= "10,10,5,5"	This syntax works like this: Margin = "Left, Top, Right, Bottom". It's clockwork starting at the left

Here we will change the margin property of the first TextBox:

```
<StackPanel Orientation="Vertical">
        <TextBox x:Name="textBox" Height="23" Width="100"
         Text="TextBox" Margin="8,9,4,5"/>
        <TextBox x:Name="textBox1" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox2" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox3" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox4" Height="23" Width="100"
Text="TextBox"/>
        <TextBox x:Name="textBox5" Height="23" Width="100"
Text="TextBox"/>
</StackPanel>
```

Once you have typed this code, click on that TextBox in the designer and take a look at the ***margin*** property in the property window (order the properties by name if you can't find it). It should look like this:

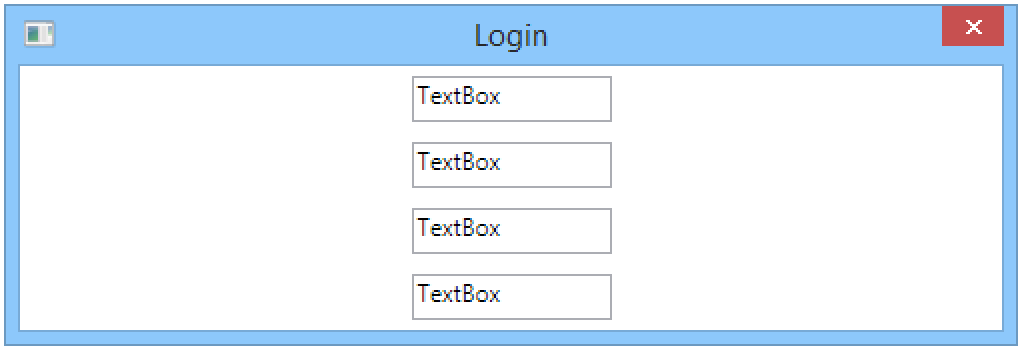


This clearly shows the order in which the margin values are applied to the textbox.

Now that you understand how to set a margin, change the XAML code so that every TextBox object has a margin of 5 pixels all around:

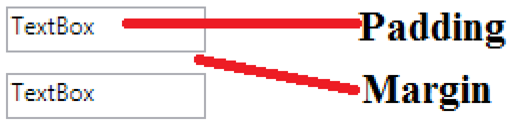
```
<StackPanel Orientation="Vertical">
        <TextBox x:Name="textBox" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox1" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox2" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox3" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox4" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox5" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
</StackPanel>
```

Run the app and you should get this screen:



Once again, we have lost some TextBoxes! Do not worry; this is just an example to show you the behavior of the controls. To fix this, you only have to make the **Window** control a bit taller by changing his ***Height*** property to 250.

Now let's continue the demonstration with the introduction of another property: the **padding**. Like the *margin* property, the ***padding*** property adds spaces around the control. The difference is that while the *margin* property puts spacing between a control and the neighboring controls, the ***padding*** property adds a space between the inner borders of the control and its content:



Here is the syntax of the **Padding** property:

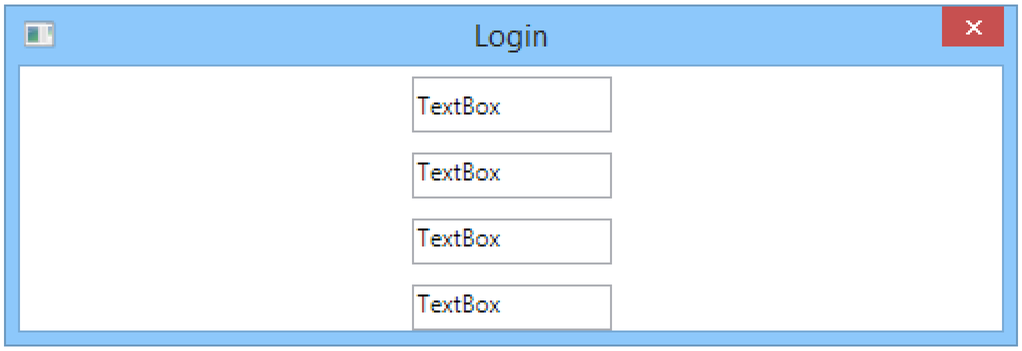
Padding = "10"	Sets a distance of 10 pixels between every border and the content
Padding = "10,5"	Sets 10 pixels of space between the left and right borders and the content. 5 pixels are used for the top and bottom.
Padding = "10,10,5,5"	This syntax works like this: Padding = "Left, Top, Right, Bottom". It's clockwork starting at the left

Let's go back to our XAML code and add padding to the first TextBox. Change the code like so:

```
<StackPanel Orientation="Vertical">
        <TextBox x:Name="textBox" Height="auto" Width="100"
         Text="TextBox" Margin="5" Padding="0,5,0,5"/>
        <TextBox x:Name="textBox1" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox2" Height="23" Width="100"
```

```
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox3" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox4" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
        <TextBox x:Name="textBox5" Height="23" Width="100"
         Text="TextBox" Margin="5"/>
</StackPanel>
```

Note that the Height property has been change to ***auto***. This means that the height of the TextBox will automatically be set to take into account the content and the padding of the textbox. Run the app and you should get this result:



The first TextBox has a padding of 5 pixels on the top and bottom.

The ***auto*** value is the default value so you could completely remove the **Height="auto"** part and you would get the same result.

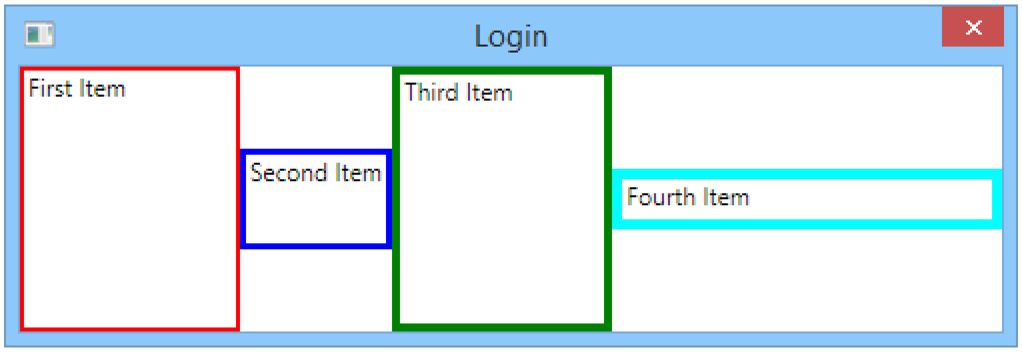
The DockPanel control

The **DockPanel** works differently than the **StackPanel**. It *attaches* or docks its controls on its borders by using its ***Dock*** property. The **Dock** property has a default value of ***Left***. Let's test this by replacing the entire ***StackPanel*** block in your code by this:

```
<DockPanel>
        <TextBox x:Name="textBox" Width="110" Text="First Item"
         BorderBrush="Red" BorderThickness="2"/>
        <TextBox x:Name="textBox1" Height="50" Text="Second
Item"
         BorderBrush="Blue" BorderThickness="3"/>
        <TextBox x:Name="textBox2" Width="110" Text="Third
Item"
         BorderBrush="Green" BorderThickness="4"/>
        <TextBox x:Name="textBox3" Height="30" Text="Fourth
Item"
         BorderBrush="Aqua" BorderThickness="5"/>
```

```
</DockPanel>
```

Run the app and you will get the following result:

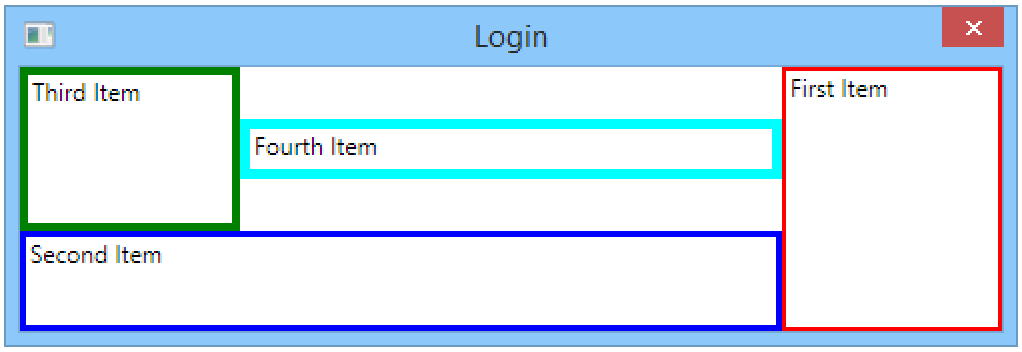


A little side note: In this example, we use two new ***TextBox*** properties: **BorderBrush** assigns a color to the border of the ***TextBox*** and **BorderThickness** sets its thickness. Note that by default, grey is the color used because it is the ***Microsoft Windows*** standard. The default thickness is 1 and a thickness of 0 means that there is no border.

Now back to the **DockPanel**. Examine the code we have now and look at the result. When you are done, make this modification to the code:

```
<DockPanel>
        <TextBox x:Name="textBox" Width="110" Text="First Item"
         DockPanel.Dock="Right" BorderBrush="Red" BorderThickness="2"/>
        <TextBox x:Name="textBox1" Height="50" Text="Second Item"
          DockPanel.Dock="Bottom" BorderBrush="Blue"
BorderThickness="3"/>
        <TextBox x:Name="textBox2" Width="110" Text="Third Item"
          DockPanel.Dock="Left" BorderBrush="Green"
BorderThickness="4"/>
        <TextBox x:Name="textBox3" Height="30" Text="Fourth Item"
         BorderBrush="Aqua" BorderThickness="5"/>
</DockPanel>
```

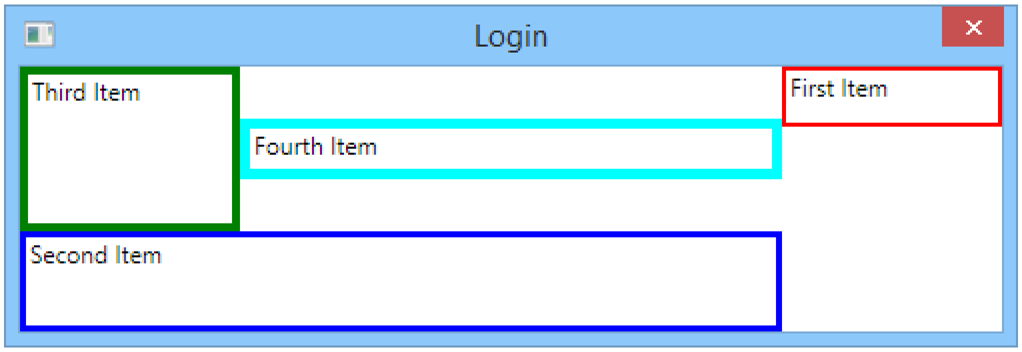
This should give you the following result:



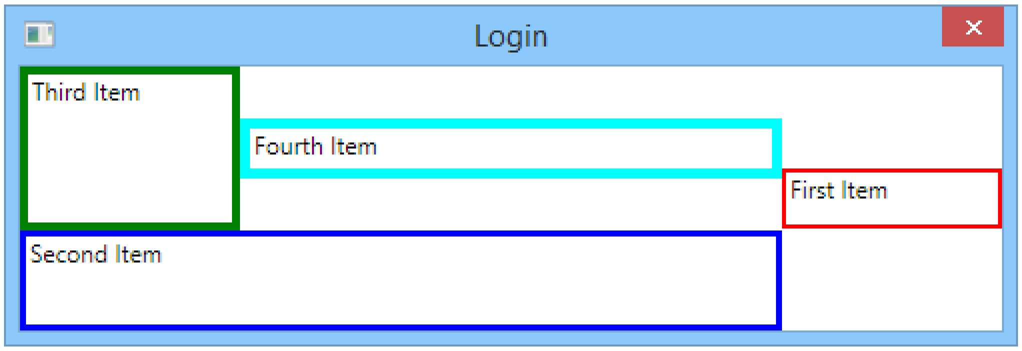
The first TextBox is docked on the right border and has a width of 110 pixels. Since its height has not been defined, it uses the entire space available in the DockPanel. Let's try to give this TextBox an actual height and align it towards the top:

```
<DockPanel>
        <TextBox x:Name="textBox" Height ="30" Width="110" Text="First
Item"
         DockPanel.Dock="Right" BorderBrush="Red" BorderThickness="2"
         VerticalAlignment="Top"/>
        <TextBox x:Name="textBox1" Height="50" Text="Second Item"
          DockPanel.Dock="Bottom" BorderBrush="Blue"
BorderThickness="3"/>
        <TextBox x:Name="textBox2" Width="110" Text="Third Item"
          DockPanel.Dock="Left" BorderBrush="Green"
BorderThickness="4"/>
        <TextBox x:Name="textBox3" Height="30" Text="Fourth Item"
         BorderBrush="Aqua" BorderThickness="5"/>
</DockPanel>
```

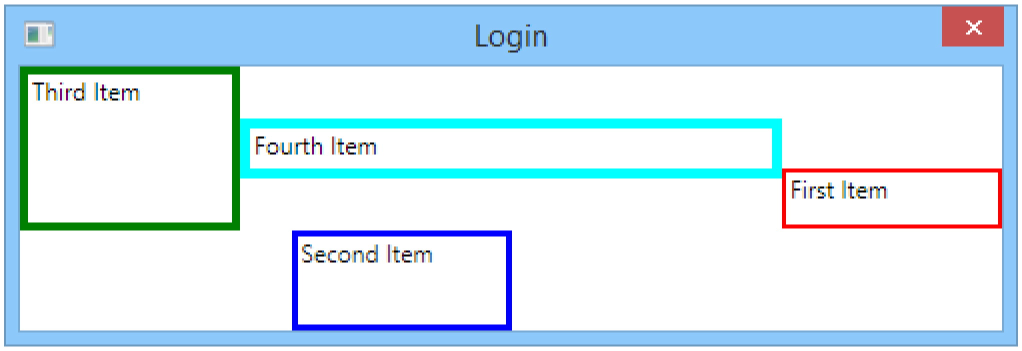
This gives us the following result:



Now if we remove the VerticalAlignment="Top" part and we run again, we get this:



This example shows that a docked control will reserve the width of its entire column, so that other controls cannot *stretch* to occupy that space. Let's now add a width of 110 pixels to the second TextBox. If you run the code, you will get this:



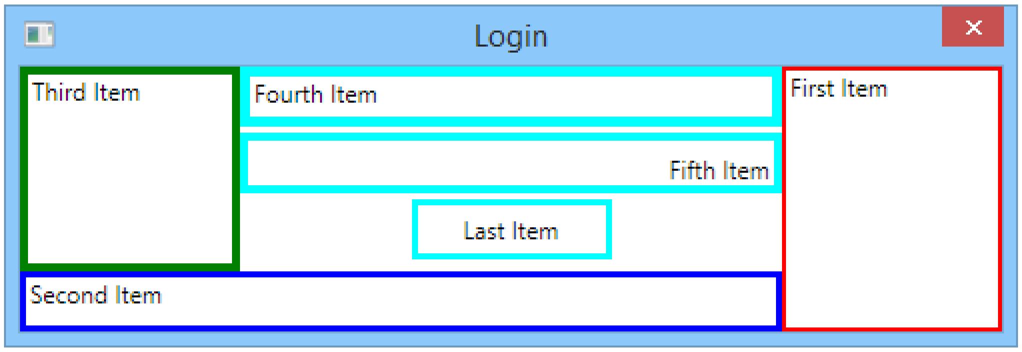
This shows that by default, a control will center itself in the space that is available, either vertically or horizontally depending on the border it is docked to. Now remove the last modifications to return to the original code:

```
<DockPanel>
        <TextBox x:Name="textBox" Width="110" Text="First Item"
         DockPanel.Dock="Right" BorderBrush="Red" BorderThickness="2"/>
        <TextBox x:Name="textBox1" Height="50" Text="Second Item"
          DockPanel.Dock="Bottom" BorderBrush="Blue"
BorderThickness="3"/>
        <TextBox x:Name="textBox2" Width="110" Text="Third Item"
          DockPanel.Dock="Left" BorderBrush="Green"
BorderThickness="4"/>
        <TextBox x:Name="textBox3" Height="30" Text="Fourth Item"
         BorderBrush="Aqua" BorderThickness="5"/>
</DockPanel>
```

A container control can also contain other container controls. For instance, we could insert a **StackPanel** in this **DockPanel**. Take a look at the following code:

```
<DockPanel>
        <TextBox x:Name="textBox" Width="110" Text="First Item"
         DockPanel.Dock="Right" BorderBrush="Red" BorderThickness="2"/>
        <TextBox x:Name="textBox1" Height="30" Text="Second Item"
         DockPanel.Dock="Bottom" BorderBrush="Blue"
BorderThickness="3"/>
        <TextBox x:Name="textBox2" Width="110" Text="Third Item"
         DockPanel.Dock="Left" BorderBrush="Green" BorderThickness="4"/
>
        <StackPanel>
            <TextBox x:Name="textBox3" Height="30" Text="Fourth Item"
            BorderBrush="Aqua" BorderThickness="5"
TextAlignment="Left"/>
            <TextBox x:Name="textBox4" Height="30" Margin="0,3"
Text="Fifth
             Item" BorderBrush="Aqua" BorderThickness="4"
             TextAlignment="Right" VerticalContentAlignment="Bottom"/>
            <TextBox x:Name="textBox5" Height="30" Width="100"
Text="Last
             Item" BorderBrush="Aqua" BorderThickness="3"
             TextAlignment="Center" VerticalContentAlignment="Center"/>
        </StackPanel>
</DockPanel>
```

In this last example, we have replaced the last TextBox with a **StackPanel** control that itself contains three TextBoxes. So the last three TextBoxes will be stacked on top of each other like shown here:



This example has introduced two new TextBox properties: **TextAlignment** and **VerticalContentAlignment**. ***TextAlignment*** aligns the text horizontally in the TextBox by using one of the following values: **Left**, **Right**, **Center** or **Justify**. These are the same options as those to align text in a text editor. ***VerticalContentAlignment*** takes care of the vertical alignment by using one of the following values: **Top** (default), **Bottom**, **Center** and **Stretch**.

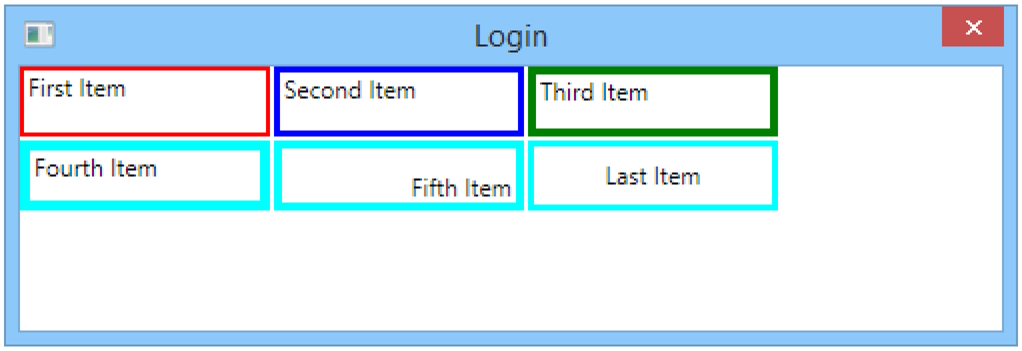
The WrapPanel control

Let's now look at another container control, the **WrapPanel**. The ***WrapPanel*** aligns its content from left to right on the line until there is no more space. At that point, the container makes a new line and continues to align its controls on that new line. Let's look at an example. Replace the **DockPanel** block from the previous example with this:

```
<WrapPanel>
        <TextBox x:Name="textBox" Height="35" Width="125" Text="First
Item"
         BorderBrush="Red" BorderThickness="2"/>
        <TextBox x:Name="textBox1" Height="35" Width="125" Margin="2,0"
         Text="Second Item" BorderBrush="Blue" BorderThickness="3"/>
        <TextBox x:Name="textBox2" Height="35" Width="125" Text="Third
Item"
         BorderBrush="Green" BorderThickness="4"/>
        <TextBox x:Name="textBox3" Height="35" Width="125" Margin="0,2"
         Text="Fourth Item" BorderBrush="Aqua" BorderThickness="5"/>
        <TextBox x:Name="textBox4" Height="35" Width="125" Margin="2,2"
         Text="Fifth Item" BorderBrush="Aqua" BorderThickness="4"
         TextAlignment="Right" VerticalContentAlignment="Bottom"/>
        <TextBox x:Name="textBox5" Height="35" Width="125" Margin="0,2"
```

```
          Text="Last Item" BorderBrush="Aqua" BorderThickness="3"
          TextAlignment="Center" VerticalContentAlignment="Center"/>
</WrapPanel>
```

Run the program and you should see the following:



This type of container is great if we want to display different kinds of control, like images for instance. Let's go back to the XAML at the beginning of the file and make the following change:

```
<Window x:Name="frmLogin1" x:Class="College.frmLogin"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:College"
        mc:Ignorable="d"
        Title="Images" Height="250" Width="500"
        FontSize="11" WindowStartupLocation="CenterScreen"
        ResizeMode="NoResize">
...
```

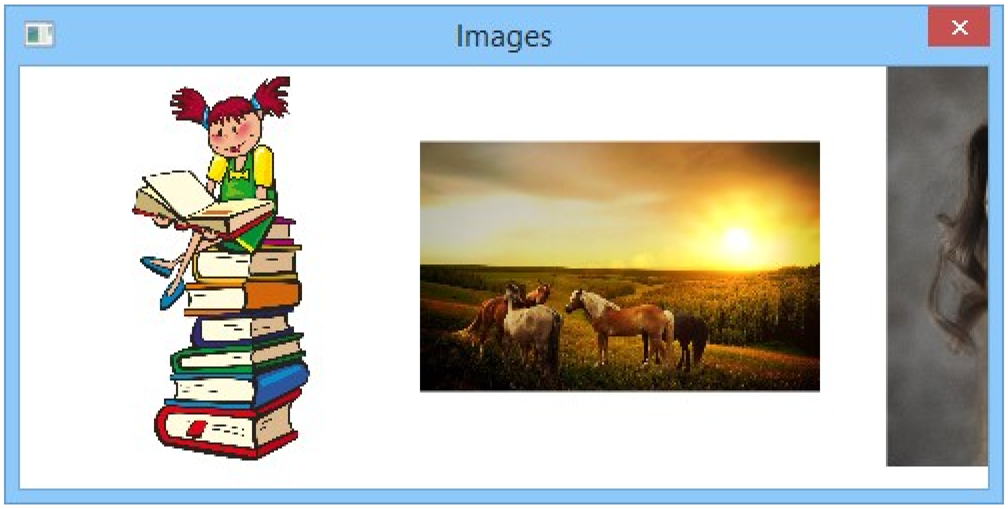
And replace the existing **WrapPanel** block with this new block:

```
<WrapPanel Orientation="Vertical">

</WrapPanel>
```

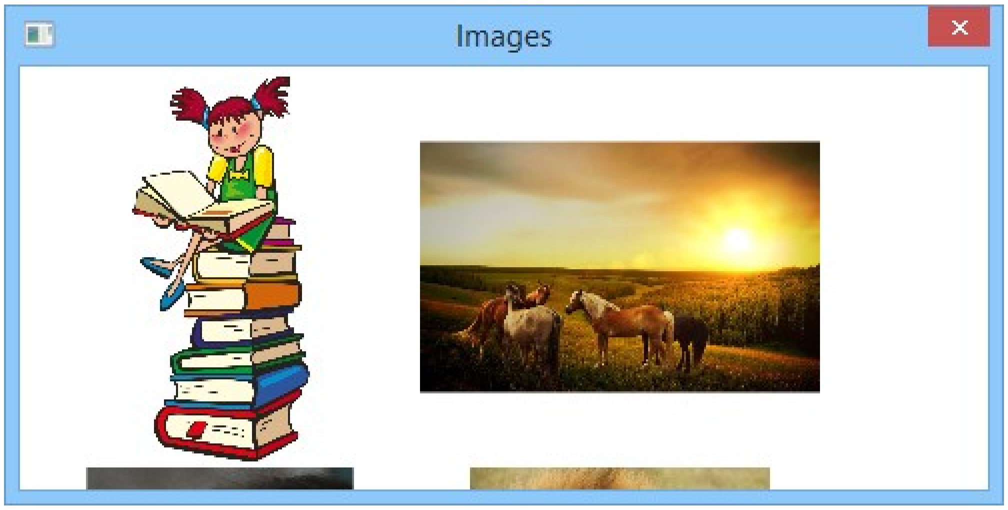
Just like the StackPanel, the **WrapPanel** can also use the **Orientation** property. We will now add six Image controls in the **WrapPanel**. Add the following code:

```
<WrapPanel Orientation="Vertical">
        <Image Source="C:\Images\Books.png" Height="200" Width="200"/>
        <Image Source="C:\Images\Horses.jpg" Height="200" Width="200"/>
        <Image Source="C:\Images\Woman.jpg" Height="200" Width="200"/>
        <Image Source="C:\Images\Lion.jpg" Height="200" Width="200"/>
        <Image Source="C:\Images\Stones.jpg" Height="200" Width="200"/>
        <Image Source="C:\Images\Workers.jpg" Height="200" Width="200"/
>
</WrapPanel>
```

An **Image** control needs a ***Source*** that will tell it where to get the image it must display. We can also add a *height* and a *width* to the control to limit the size of the image. You will find the **Images** folder in the course's **work files**. In the example above, the image files are located on the **C:** drive so make sure you adapt the path to your situation. You will get the following result if you run the app:



Now change the **Orientation** to *Horizontal* and try it again. You should get this:

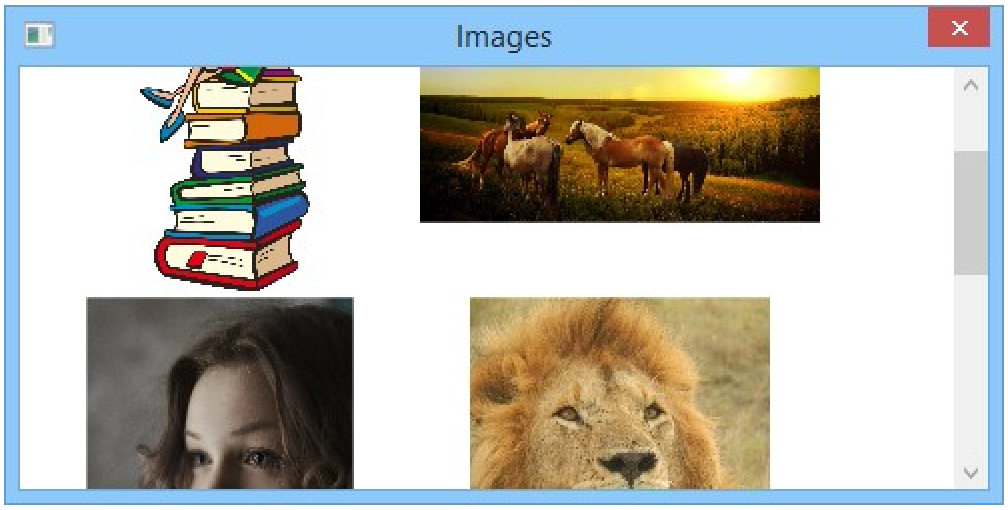


In both cases, the **WrapPanel** is not large enough to accommodate its content. To remedy this, we will introduce a new control, the **ScrollViewer** control.

The **ScrollViewer** control can only contain one item. So obviously, we can't insert all six images in it. However, we can put the entire **WrapPanel** object in the **ScrollViewer**. Look at the following code:

```
<ScrollViewer>
        <WrapPanel Orientation="Horizontal">
            <Image Source="C:\Images\Books.png" Height="200"
Width="200"/>
            <Image Source="C:\Images\Horses.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Woman.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Lion.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Stones.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Workers.jpg" Height="200"
Width="200"/>
        </WrapPanel>
</ScrollViewer>
```

Execute this code. You should see a scrolling bar appear at the right of the window:

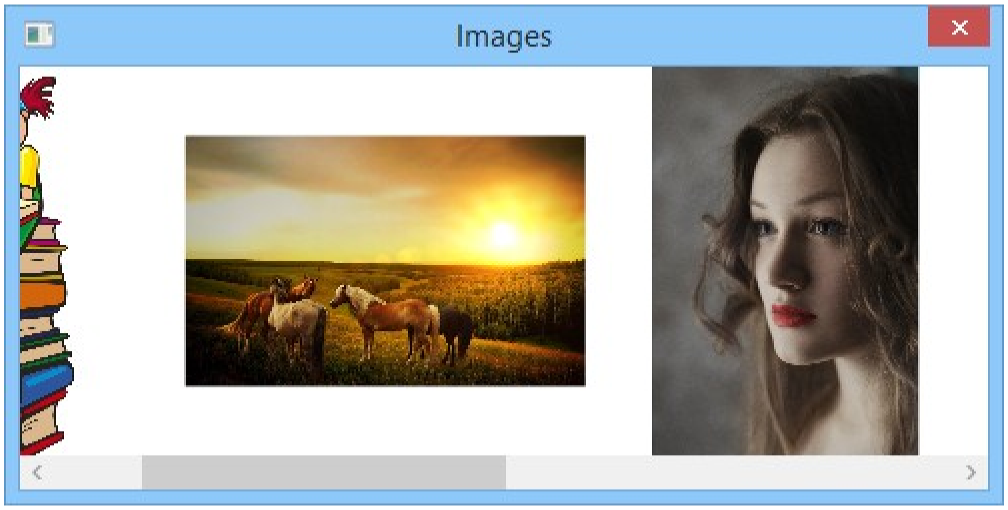


The **ScrollViewer** control has two properties that we can use. The first, **VerticalScrollBarVisibility**, will determine at what point the scroll bar becomes visible (default value: **Visible**). The other property, **HorizontalScrollBarVisibility**, does the same thing but for the horizontal scroll bar (default value: **Disabled**). Modify your code like so:

```
<ScrollViewer HorizontalScrollBarVisibility="Visible"
         VerticalScrollBarVisibility="Disabled">
        <WrapPanel Orientation="Horizontal">
```

```
            <Image Source="C:\Images\Books.png" Height="200"
Width="200"/>
            <Image Source="C:\Images\Horses.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Woman.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Lion.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Stones.jpg" Height="200"
Width="200"/>
            <Image Source="C:\Images\Workers.jpg" Height="200"
Width="200"/>
        </WrapPanel>
</ScrollViewer>
```

Now run the app. You should now see a different behavior from the window:



The horizontal bar is now visible and you can scroll the images from left to right.

We will now examine one last container control: The **Grid** control (the **Canvas** control will not be seen in this course).

The Grid control

The **Grid** control uses rows and columns to position its content. That means we must first begin by defining our rows and columns. We can start by defining either the rows or the columns so we'll start with the columns. Replace the ***ScrollViewer/WrapPanel*** block from the previous example with this:

```
<Grid>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
</Grid>
```

Between the two **<Grid.ColumnDefinitions>** tags, we put as many **<ColumDefinition>** tags as we want columns. So in our case, there will be five columns. The ***Width*** property sets the width of the column. In our example, the first column has a width of 10 pixels and the last one has a width of 50 pixels. The three other columns will adapt themselves depending on their content. Like with arrays, the **Grid** indexes start at 0. So our five columns will range from indexes 0 to 4.

We will now set the number or rows in the Grid. Rows are set the same way as columns except we use the **<Grid.RowDefinitions>** and **<RowDefinition>** tags to define them. We can also use the ***Height*** property to set the height of the rows. Add the following code:

```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
             <RowDefinition Height="*"/>    <!-- Line 0 -->
             <RowDefinition Height="Auto"/> <!-- Line 1 -->
             <RowDefinition Height="Auto"/> <!-- Line 2 -->
             <RowDefinition Height="Auto"/> <!-- Line 3 -->
             <RowDefinition Height="Auto"/> <!-- Line 4 -->
             <RowDefinition Height="*"/>    <!-- Line 5 -->
        </Grid.RowDefinitions>
</Grid>
```

Like columns, rows also have indexes that start at 0. Our six lines will range from indexes 0 to 5. Also note that we have inserted ***comments*** in the previous example. In XAML, comments must be put between <!-- and -->.

We will now add an **Image** control in our **Grid**. As you now know, an image needs to have its ***Source*** property set to the path of the image file. But instead of having the **Source** property point to an absolute path like we did before, let's add the image file to the project as a **resource**. First, let's start by creating a folder that will hold our resources.

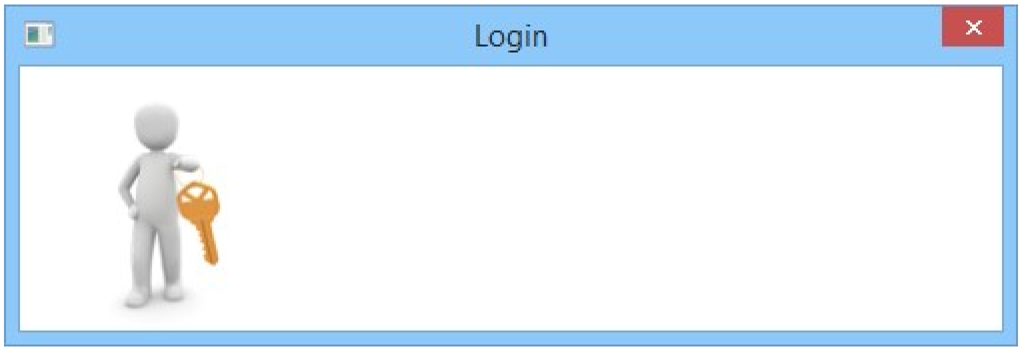
In the ***Solution Explorer***, right-click on **College** and choose **Add** - **New Folder** and name your folder **Resources**. Once the ***Resources*** folder exists, right-click it and choose **Add** - **Existing Item...** In the *Add Existing Item* windows, navigate to the work files and select the **Key.png** (you might need to change the drop down list to ***All Files (*.*)*** at the bottom). That's it! The **Key.png** file is now part of your project as a resource. We can now add the following line to our code:

```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
             <RowDefinition Height="*"/>    <!-- Line 0 -->
             <RowDefinition Height="Auto"/> <!-- Line 1 -->
             <RowDefinition Height="Auto"/> <!-- Line 2 -->
             <RowDefinition Height="Auto"/> <!-- Line 3 -->
             <RowDefinition Height="Auto"/> <!-- Line 4 -->
             <RowDefinition Height="*"/>    <!-- Line 5 -->
        </Grid.RowDefinitions>
        <Image Source="Resources/Key.png"/>
</Grid>
```

We must now position the image in the grid. We will put in row 0, column 1 and make it span 6 columns. Modify the **Image** line like this:

```
<Image Source="Resources/Key.png" Grid.Column="1"
Grid.Row="0" Grid.RowSpan="6"/>
```

Retitle the window ***Login*** like it was before and give it a height of **171** and a width of **507**. Run the app and you should get this result:



We will now start adding labels and textboxes to our app so that users can type in their login information. Let's add a label and a textbox in columns 3 and 4 of the grid:

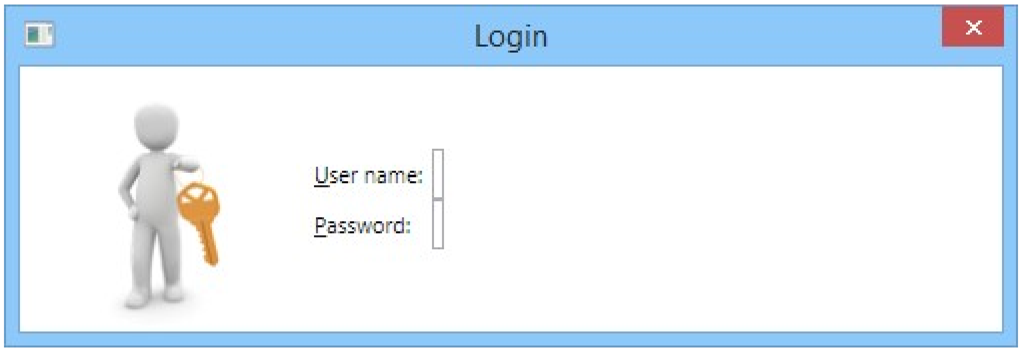
```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
       <Grid.RowDefinitions>
            <RowDefinition Height="*"/>    <!-- Line 0 -->
            <RowDefinition Height="Auto"/> <!-- Line 1 -->
            <RowDefinition Height="Auto"/> <!-- Line 2 -->
            <RowDefinition Height="Auto"/> <!-- Line 3 -->
            <RowDefinition Height="Auto"/> <!-- Line 4 -->
            <RowDefinition Height="*"/>    <!-- Line 5 -->
       </Grid.RowDefinitions>
       <Image Source="Resources/Key.png" Grid.Column="1"
Grid.Row="0"
         Grid.RowSpan="6"/>
       <Label Content="_User name:" Grid.Column="2"
Grid.Row="1"/>
       <TextBox x:Name="txtUser" Grid.Column="3" Grid.Row="1"/>
</Grid>
```

The **x:Name** attribute of the TextBox gives it a unique name. If you click on the textbox in the design window and you look at the property window, you should see that his name is now set. Also, don't worry about the underscore (_) in front the ***user name*** text. This only means that the letter after the underscore (U) will be a shortcut key (alt-U).

We must now add another label and textbox to handle the password. Modify your code like so:

```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
       <Grid.RowDefinitions>
            <RowDefinition Height="*"/>    <!-- Line 0 -->
            <RowDefinition Height="Auto"/> <!-- Line 1 -->
            <RowDefinition Height="Auto"/> <!-- Line 2 -->
            <RowDefinition Height="Auto"/> <!-- Line 3 -->
            <RowDefinition Height="Auto"/> <!-- Line 4 -->
            <RowDefinition Height="*"/>    <!-- Line 5 -->
       </Grid.RowDefinitions>
       <Image Source="Resources/Key.png" Grid.Column="1"
Grid.Row="0"
         Grid.RowSpan="6"/>
       <Label Content="_User name:" Grid.Column="2"
Grid.Row="1"/>
       <TextBox x:Name="txtUser" Grid.Column="3" Grid.Row="1"/>
       <Label Content="_Password:" Grid.Column="2"
Grid.Row="2"/>
      <PasswordBox x:Name="txtPassword" Grid.Column="3"
Grid.Row="2"/>
</Grid>
```

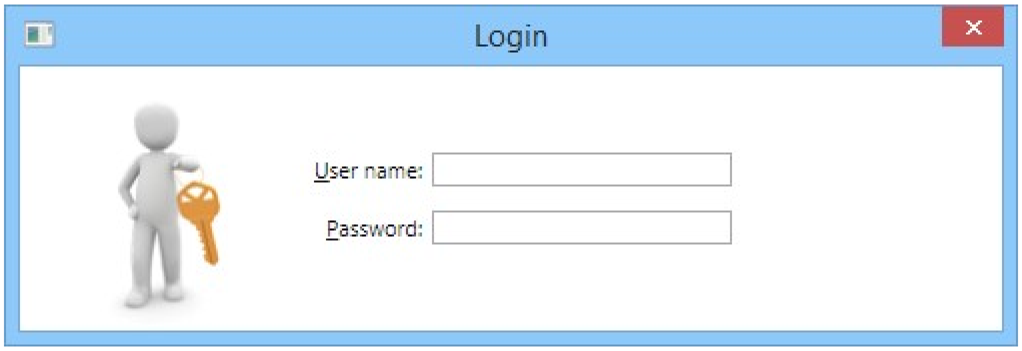
Note the new control we just used: the **PasswordBox**. It's just like a TextBox, except it hides its content which is just what we want with passwords. Run the app and look at the result:



As you can see, there is not enough space for the textbox and passwordbox. Let's make a couple of modifications to the labels and textboxes to solve the issue using properties that we have seen before:

```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="10"/>    <!-- Column 0 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 1 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 2 -->
            <ColumnDefinition Width="Auto"/>  <!-- Column 3 -->
            <ColumnDefinition Width="50"/>    <!-- Column 4 -->
        </Grid.ColumnDefinitions>
       <Grid.RowDefinitions>
            <RowDefinition Height="*"/>    <!-- Line 0 -->
            <RowDefinition Height="Auto"/> <!-- Line 1 -->
            <RowDefinition Height="Auto"/> <!-- Line 2 -->
            <RowDefinition Height="Auto"/> <!-- Line 3 -->
            <RowDefinition Height="Auto"/> <!-- Line 4 -->
            <RowDefinition Height="*"/>    <!-- Line 5 -->
       </Grid.RowDefinitions>
       <Image Source="Resources/Key.png" Grid.Column="1"
Grid.Row="0"
         Grid.RowSpan="6"/>
        <Label Content="_User name:" Grid.Column="2"
Grid.Row="1"
         HorizontalAlignment="Right" Margin="0,2"/>
        <TextBox x:Name="txtUser" Grid.Column="3" Grid.Row="1"
         Width="150" Margin="0,2" VerticalAlignment="Center"/>
        <Label Content="_Password:" Grid.Column="2"
Grid.Row="2"
         HorizontalAlignment="Right" Margin="0,2"/>
        <PasswordBox x:Name="txtPassword" Grid.Column="3"
Grid.Row="2"
         Width="150" Margin="0,2" VerticalAlignment="Center"/>
</Grid>
```

These modifications will give the following result:

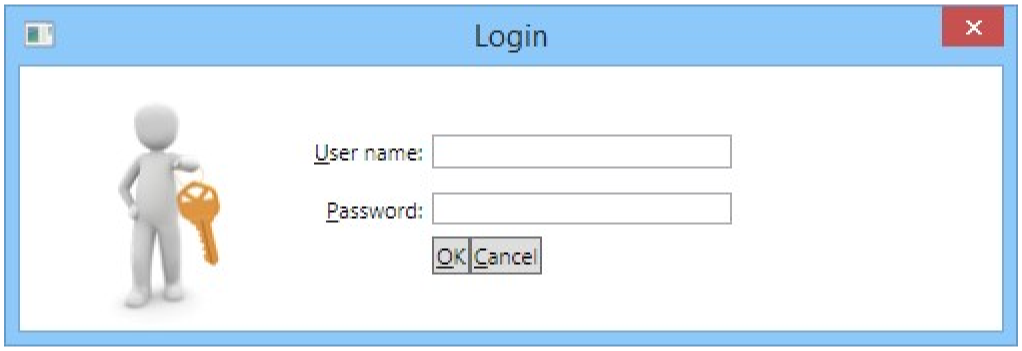


It is now time to add a couple of buttons to our login form. We have seen before that it is possible to put a container within a container. We will do just that in the following code by putting two buttons in a **WrapPanel** and then inserting this ***WrapPanel*** in the **Grid**. Add the following code:

```

...

<Label Content="_User name:" Grid.Column="2" Grid.Row="1"
         HorizontalAlignment="Right" Margin="0,2"/>
        <TextBox x:Name="txtUser" Grid.Column="3" Grid.Row="1"
         Width="150" Margin="0,2" VerticalAlignment="Center"/>
        <Label Content="_Password:" Grid.Column="2"
```

```
Grid.Row="2"
         HorizontalAlignment="Right" Margin="0,2"/>
        <PasswordBox x:Name="txtPassword" Grid.Column="3"
Grid.Row="2"
         Width="150" Margin="0,2" VerticalAlignment="Center"/>
        <WrapPanel Grid.Column="3" Grid.Row="4">
            <Button x:Name="btnOK"  Content="_OK"/>
            <Button x:Name="btnCancel" Content="_Cancel"/>
        </WrapPanel>
</Grid>
```

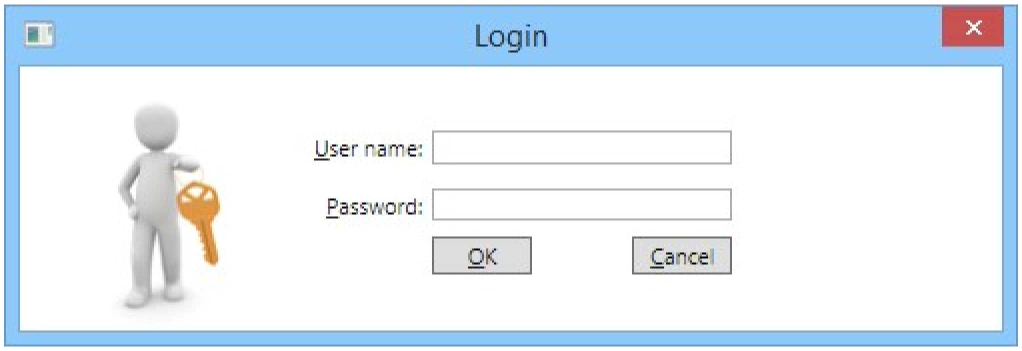
So we added a control at column 3, row 4. But this control contains two other controls (buttons). Run the app and you should get this result:



The buttons are not spaced out, but that's normal **StackPanel** behavior. In the next modification to our code, we will make the buttons 50 pixels wide. We will also add margins to the buttons so that they spread themselves nicely in the form:

```
...
<WrapPanel Grid.Column="3" Grid.Row="4">
            <Button x:Name="btnOK"  Content="_OK" Width="50"
             Margin="0,2"/>
            <Button x:Name="btnCancel" Content="_Cancel"
Width="50"
             Margin="50,2,0,2"/>
</WrapPanel>
...
```

Run the app and you will get this result:



Ok! The form is almost ready for use. Remember we discussed the underscore that was put in front of certain words earlier? These were to activate the shortcut keys using the ALT key. Run your app again, and while you see the screen, press the ALT key. You should see the letters U, P, O and C from the labels and textboxes get underlined. The goal is to make the cursor position itself in the *User name* textbox when the user presses Alt-U. To get that working, we must make it so that the labels target the appropriate textboxes. Modify the following code:

```
...
 <Image Source="Resources/Key.png" Grid.Column="1" Grid.Row="0"
         Grid.RowSpan="6"/>
         <Label Content="_User name:" Grid.Column="2"
Grid.Row="1"
          HorizontalAlignment="Right" Margin="0,2"
          Target="{Binding ElementName=txtUser}"/>
        <TextBox x:Name="txtUser" Grid.Column="3" Grid.Row="1"
         Width="150" Margin="0,2" VerticalAlignment="Center"/>
        <Label Content="_Password:" Grid.Column="2"
Grid.Row="2"
          HorizontalAlignment="Right" Margin="0,2"
          Target="{Binding ElementName=txtPassword}"/>
             <PasswordBox  x:Name="txtPassword"  Grid.Column="3"
Grid.Row="2"

         Width="150" Margin="0,2" VerticalAlignment="Center"/>
...
```

Now try to run the app and then press on ALT-U. You should see the cursor appear in the *User name* textbox. For the buttons, when the shortcut keys are used, it is the same as if we clicked on them.

So we are now ready to start adding C# code to our app and make it functional. To do so, we must first make our buttons recognize the ***click*** event. Modify the XAML code like so:

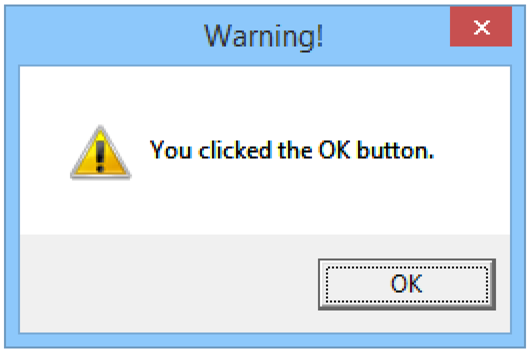
```
...
<WrapPanel Grid.Column="3" Grid.Row="4">
            <Button x:Name="btnOK"  Content="_OK" Width="50"
             Margin="0,2" Click="btnOk_Click"/>
            <Button x:Name="btnCancel" Content="_Cancel"
Width="50"
             Margin="50,2,0,2" Click="btnCancel_Click"/>
</WrapPanel>
...
```

Now the buttons know what to do if they get clicked. For instance, the **Ok** button knows that it must call the ***btnOk_Click*** method if it gets clicked on. It is now time to add these new methods in our C# code. In the designer window, double-click on the **Ok** button. This should open the **frmLogin.xaml.cs** file which is where we are going to type the C# code. You should see that the empty **btnOk_Click** method has been generated for you. Go back to the designer and double-click the **Cancel** button to get the same result.

We will now add the necessary C# code in the methods:

```
public partial class frmLogin : Window
    {
        public frmLogin()
        {
            InitializeComponent();
        }

        private void btnOk_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("You clicked the OK button.", "Warning!",
             MessageBoxButton.OK, MessageBoxImage.Exclamation);
        }

        private void btnCancel_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("You clicked the Cancel button.",
"Warning!",
             MessageBoxButton.OK, MessageBoxImage.Exclamation);
        }
    }
```

Run the application and test it. You should get the appropriate message depending on which button you clicked. Also test the shortcut keys Alt-O and Alt-C to make sure they work.

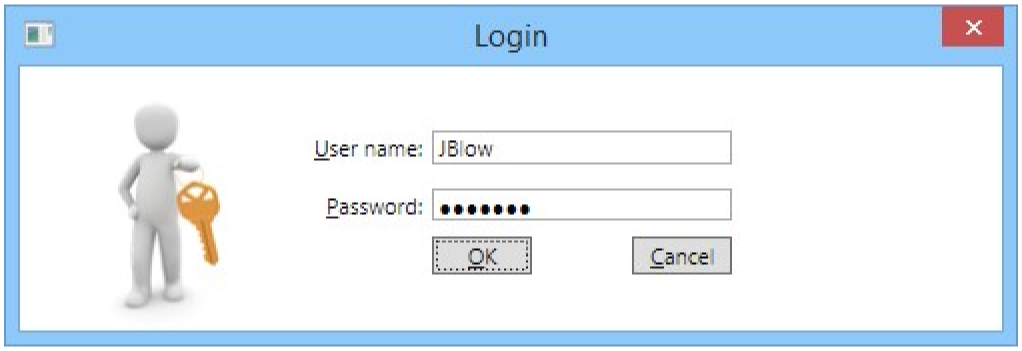


You clicked the OK button.

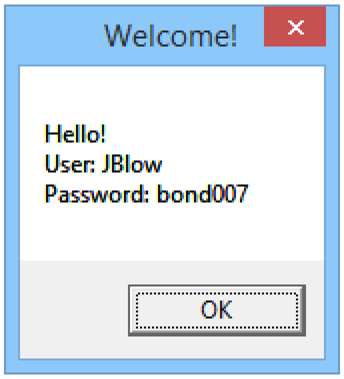
Note that we have used the **MessageBox.Show** method to display message boxes to the user. This method can be used with different parameters. We will now modify the code further to enable us to deal with the content of the textboxes. Make the following adjustments in the **btnOk_Click** method:

```
 private void btnOk_Click(object sender, RoutedEventArgs e)
        {
            //get the user name
            string user = txtUser.Text;
            string passWord = txtPassword.Password;
            //display welcome message
            MessageBox.Show($"Hello! {Environment.NewLine}User:
{user
             + Environment.NewLine}Password: {passWord +
               Environment.NewLine}",
                "Welcome!", MessageBoxButton.OK);
        }
```

Run the app. Enter a user name and password and click **Ok**. You should get the following result:



User name: JBlow
Password: •••••••



Hello!
User: JBlow
Password: bond007

SUMMARY

In this lesson, you have seen the following container controls:

StackPanel	Stacks its content either horizontally or vertically using the ***Orientation*** property.
DockPanel	Docks its content on one of its borders using the ***Dock*** property that can take the values *Left*, *Right*, *Top* or *Bottom*.
WrapPanel	Aligns its content until the end of the container is met, and then starts a new line. Can also use the ***Orientation*** property.
ScrollViewer	Can only contain one item. It can add vertical and horizontal scroll bars depending on its ***visibility*** properties.
Grid	The default container. It will arrange its content based on rows and columns.

- The **Margin** property sets the spacing between controls
- The **Padding** property sets the space between the inner border of a control and its content
- The **Height** and **Width** properties set the height and width of a control
- The **BorderBrush** property sets the border color of a control
- The **BorderThickness** property sets the thickness of the border
- The **Image** control needs its **Source** property to know where to get the image file
- The name of a control is set by using the **x:Name** property in XAML

Progress check

Check your answer in Appendix A

1. What kind of file contains the definition of the graphical interface in WPF?

a) CS
b) XAML
c) XML
d) SLN

2. What is the name of the event that points to the method to execute when a button is clicked?:

a) Action
b) Changes
c) Click
d) btnOK

3. To add spacing around a control, we can use...

a) Height
b) Width
c) Margin
d) Space

4. Which container control attaches its content to one of its borders?

a) StackPanel
b) WrapPanel
c) PadPanel
d) DockPanel

5. True or false: the ScrollViewer control can add both vertical and horizontal scroll bars to its content.

a) True
b) False

Module 2

ADO.NET, basic concepts

Before we start implementing database connectivity in our application, we must first understand what ADO.Net is.

ADO.Net is part of the .NET Framework and it is basically a way to connect and interact with data, wherever it comes from. Up until now, you have mostly connected your programs to files, but you can also get data from databases like SQL Server, MySQL, Oracle, etc.

You will see that databases are much easier to interact with than files. They are also more efficient.

OBJECTIVES

- Describe what ADO.Net is.
- Use ADO.Net in *connected* mode.

Connecting to a database

Before we can start interacting with data from a database, we must first establish a connection between the database server and our application. This connection is created with the ***Connection*** object. Let's see how to do that with an SQL Server database. (**NOTE**: ADO.Net supports connections to many different types of databases. What we'll show here can also apply to them)

For the examples in this module, we will use the **College** database that already has a **tblStudents** table in it. Go in the work files for module 2 and run the **CreateCollege.sql** file in your database server to create the **College** database. Ask your instructor if you need help with this part.

To connect our application to a database, we must use the following syntax:

server*=<Server name>;* ***initial catalog****=<Database name>;* ***user id****=<User name>;* ***password****=<Password>*

Let's examine the parameters:

<Server name> represents the name of the server you want to connect to. In our case, we will use a local server, meaning a server that is installed on our own computer. In this case, you can use either **(local)** or **.** (a dot) as the name of the server. But if we wished to use a remote server, we could do it by using its IP address followed by a port number.

<initial catalog> represents the name of the database to use within the server. In our case, we will use **College**.

The two last parameters (*User name* and *Password*) depend greatly on the authentication method we are using. If we wish to use a registered user in the **SQL Server** user groups, then a user name and password are required. On the other hand, if we wish to use the **Windows** *user* we are currently connected with, we can ignore user names and passwords and use the line ***integrated security = true*** instead. This is what we are going to do in our examples.

This would be a valid connection string:

```
"server=MyServer;    initial    catalog=College;    user=Joe;
password=1234"
```

But this is the one we are going to use later:

```
"server=.; initial catalog=College; integrated security=true"
```

So now we know how to create a ***connection string***. We will now use it in an actual application. Get the **StudentManagement** Visual Studio project from the work files. Copy the project folder somewhere you can work from and open it. Open the **frmLogin.xaml.cs** file in the editor. We will begin by adding the necessary ***using*** statement at the top of the file so we can work with an ***SQL Server*** database:

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Data.SqlClient;
```

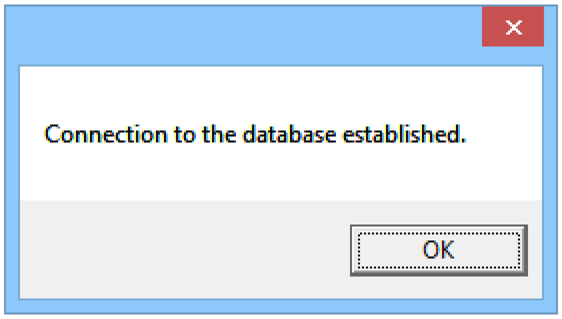
Now we can create an actual **SqlConnection** object and establish a connection using a connection string. Add the following two lines in the code:

```
...
public partial class frmLogin : Window
    {
        SqlConnection connection;
        bool open = false;
        public frmLogin()
        {
            InitializeComponent();
            connection = new SqlConnection("server=.;
initial
             catalog=College; integrated security=true");
            open = true;
        }
...
```

These new lines of code first declare the **SqlConnection** object and then create the instance of the connection using a connection string. We will now try to establish a connection to the database. This will be done in the **btnOk_Click** method so that it is a click on the **OK** button that initiates the connection. We need to close the connection when we are done as it is not a good practice to leave the connection open for too long. We will also put our code in a ***try..catch*** structure to be able to handle the exception if any problem occurs while we open the database. Add the following code in the **btkOk_Click** method:

```
  private void btnOk_Click(object sender, EventArgs e)
        {
            try
            {
                connection.Open();
                MessageBox.Show("Connection to the database
                 established.");
                connection.Close();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);

            }
        }
```

As you can see, it is the **Open**() method that establishes the connection and the **Close**() method that ends it. Run the app, type some text in the ***User name*** TextBox and click **OK**. You should get the following result:


Connection to the database established.

You can also make another test by making a mistake on purpose. Change the connection string for this:
"server=xoxoxo; initial catalog=College; integrated security=true"
And try to click the **OK** button again. The system should now jump in the ***catch*** and display an error message. Try it again with ***Colege*** instead of ***College*** and you should see yet another error message. This happens because the system displays the error message of the generated exception: **MessageBox.Show(ex.Message);**

Now that we know that our connection string works, we will put it somewhere else so that we can easily reuse it throughout our program. Double-click on the **App.config** file in the ***Solution Explorer*** to open it in the editor window. ***App.config*** is a configuration file. It can contain all sorts of parameters that can be used anywhere in the application. It makes it possible to centralize everything. It's the perfect place to put our connection string so we don't have to retype it every time we need it. Make the following modification to the code:

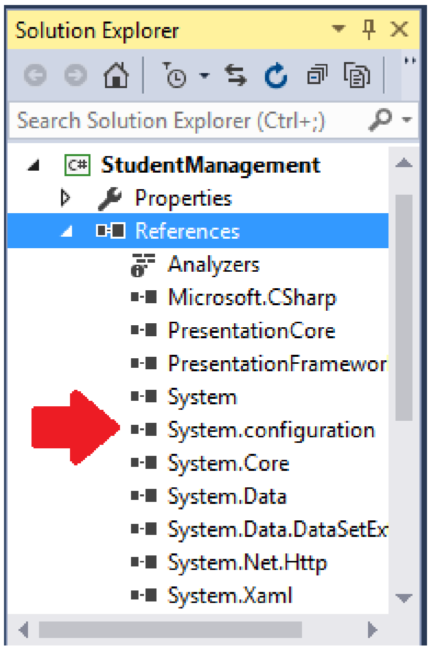
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="constr"
         connectionString="server=.; initial catalog=College;
          integrated security=true"/>
  </connectionStrings>
    <startup>
        <supportedRuntime version="v4.0"
         sku=".NETFramework,Version=v4.5.2" />
    </startup>
</configuration>
```

As you can see, we added our connection string and named it **constr**. So each time we need it, we only need to type ***constr*** instead of the entire connection string. It also makes it easy to add multiple connection strings and organize them here if the need arises.

Now let's go back to the **Login.xaml.cs** file. We need to add a using statement at the top so that we can use the connection string parameter we just created. Add the following line in the using statements:

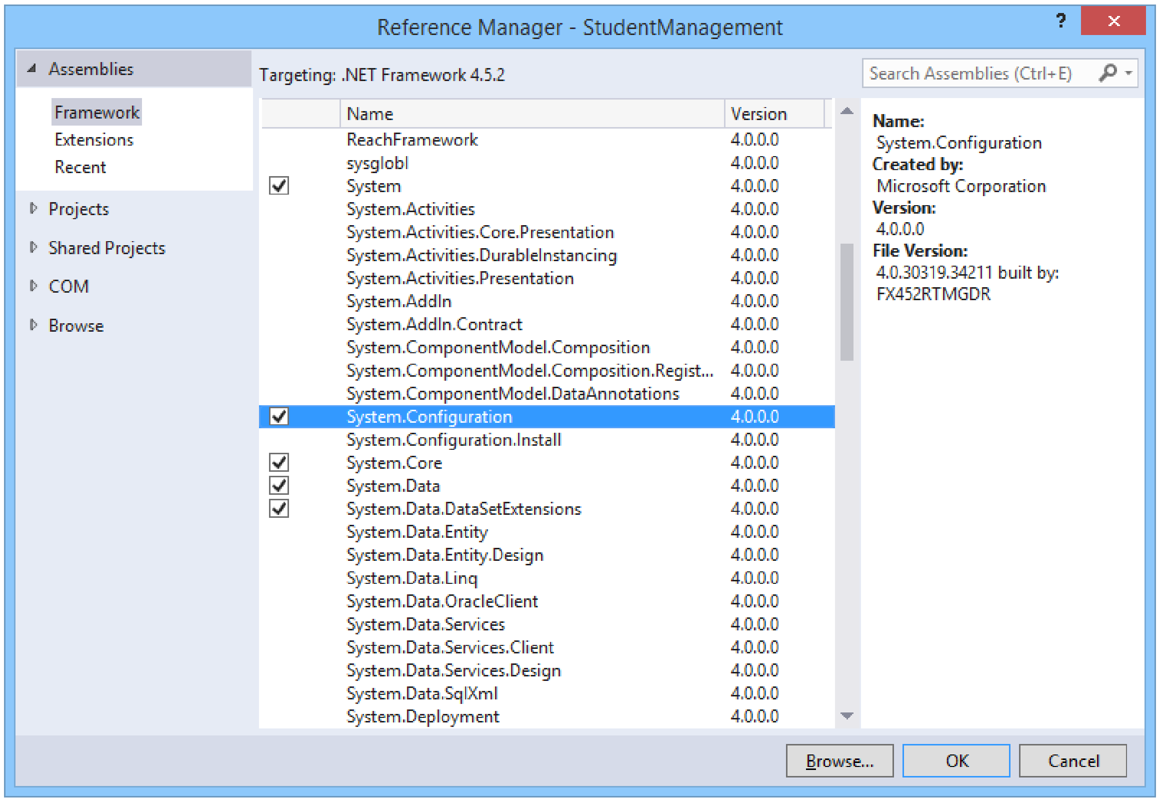
```
using System;
using System.ComponentModel;
using System.Windows;
using System.Data.SqlClient;
using System.Configuration;
```

For this using statement to work, the **System.Configuration** library must be referenced in your project. Check in the Solution Explorer to make sure you see it:



```
◢ StudentManagement
  ▷ Properties
  ◢ References
      Analyzers
      Microsoft.CSharp
      PresentationCore
      PresentationFramewor
      System
      System.configuration
      System.Core
      System.Data
      System.Data.DataSetEx
      System.Net.Http
      System.Xaml
```

If you don't see it in the list, you can add it by right-clicking on **References** at the top of the list and choosing ***Add Reference...*** Then, just select the **System.Configuration** library in the **Framework** list:



Targeting: .NET Framework 4.5.2

	Name	Version
	ReachFramework	4.0.0.0
	sysglobl	4.0.0.0
✓	System	4.0.0.0
	System.Activities	4.0.0.0
	System.Activities.Core.Presentation	4.0.0.0
	System.Activities.DurableInstancing	4.0.0.0
	System.Activities.Presentation	4.0.0.0
	System.AddIn	4.0.0.0
	System.AddIn.Contract	4.0.0.0
	System.ComponentModel.Composition	4.0.0.0
	System.ComponentModel.Composition.Regist...	4.0.0.0
	System.ComponentModel.DataAnnotations	4.0.0.0
✓	System.Configuration	4.0.0.0
	System.Configuration.Install	4.0.0.0
✓	System.Core	4.0.0.0
✓	System.Data	4.0.0.0
✓	System.Data.DataSetExtensions	4.0.0.0
	System.Data.Entity	4.0.0.0
	System.Data.Entity.Design	4.0.0.0
	System.Data.Linq	4.0.0.0
	System.Data.OracleClient	4.0.0.0
	System.Data.Services	4.0.0.0
	System.Data.Services.Client	4.0.0.0
	System.Data.Services.Design	4.0.0.0
	System.Data.SqlXml	4.0.0.0
	System.Deployment	4.0.0.0

Name:
System.Configuration
Created by:
Microsoft Corporation
Version:
4.0.0.0
File Version:
4.0.30319.34211 built by: FX452RTMGDR

Now go back to the code in **Login.xaml.cs**. We will modify the line that uses the connection string so it can take advantage of our new ***constr*** connection string we created earlier:

```
...
public frmLogin()
        {
            InitializeComponent();
            connection = new SqlConnection
(ConfigurationManager.ConnectionStrings["constr"].ConnectionStr
ing);
            open = true;
        }
...
```

Test the program. You should get the same result as before.

Reading a record in the database

If we take a look at the **College** database in **SQL Server**, we see that it contains three tables, one of them being **tblUsers**. This table has five fields: ***UserId***, ***FirstName***, ***LastName***, ***UserName*** and ***PassWord***. The ***UserName*** and ***PassWord*** fields will be used for authentication, but we still need to keep the three other fields to know who the current user is after the login is done. To do so, we will create a small class that will hold this information. Right-click on the **StudentManagement** project name in the ***Solution Explorer***, and then choose **Add - Class...** Name the file **CurrentUser.cs** and add the following code in it:

```
namespace StudentManagement
{
    public class CurrentUser
    {
        public string UserId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

These lines of code will enable us to access the three variables. The **get** and **set** expressions are **C#** keywords that give access to the variable in read(**get**) and write(**set**) mode.

Let's now go back to the **Login.xaml.cs** file and add the code to read a record from a database table. To do so, we will need a ***SqlCommand*** object. Add the following code at the beginning of the class and in the constructor:

```
public partial class frmLogin : Window
    {
        SqlConnection connection;
        SqlCommand command;
        bool open = false;
        public frmLogin()
        {
            InitializeComponent();
            connection = new SqlConnection
(ConfigurationManager.ConnectionStrings["constr"].ConnectionStr
ing);
            txtUser.Focus();
            open = true;
        }
...
```

Note the ***Focus*** instruction that will put the cursor in the **txtUser** TextBox by default.

The **SqlCommand** object needs two things: a connection, and a valid **SQL** instruction. In our case, our instruction will be a **SELECT** command that uses a **WHERE** clause to select only the record that has the correct user name and password. To make this happen, we will create a string that will contain the result of the following concatenation:

```
string authentication =
                    "SELECT * FROM tblUsers WHERE UserName = '"
+
                    txtUser.Text + "' AND PassWord = '" +
                    txtPassword.Password +"'";
```

Here we are concatenating (or gluing together) bits of strings to create a full SQL instruction. So you don't forget the *single quotes*, they have been highlighted in the previous example. So if the **txtUser** TextBox contains ***JBlow*** and the **txtPassword** PasswordBox contains ***1234***, the resulting string will look like this:

SELECT * FROM tblUsers WHERE UserName='JBlow' AND PassWord='1234'

which is a valid SELECT statement!

Add the following code to the **btnOK_Click** method:

```
private void btnOk_Click(object sender, EventArgs e)
        {
            try
            {
                string authentication =
                    "SELECT * FROM tblUsers WHERE UserName = '" +
                    txtUser.Text + "' AND PassWord = '" +
                    txtPassword.Password + "'";
                command = new SqlCommand(authentication, connection);
                connection.Open();
                MessageBox.Show("Connection to the database 
established.");
                connection.Close();
            }
...
```

Now that the **SqlCommand** object is ready, we will need another object to run it. For this, we will use a **SqlDataReader**. This object uses the SELECT command that we just declared and runs it in the database by using the command's **ExecuteReader** method. It then grabs the resulting records from the database table. Add the following code:

```
...
    command = new SqlCommand(authentication, connection);
    connection.Open();
    SqlDataReader reader = command.ExecuteReader();
...
```

After the **ExecuteReader** statement is done, we can check if the ***reader*** received a record from the database. If it did, it means the user exists. If the reader received nothing from the SELECT statement, it means the user doesn't exist in the database. Add the following code to implement this:

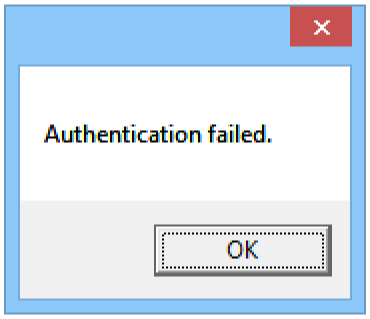
```
...
           SqlDataReader reader = command.ExecuteReader();
           //Check to see if a record was found
           if (reader.Read())
           {
               //Creates the CurrentUser object
               CurrentUser user = new CurrentUser();
               user.UserId = reader["UserId"].ToString();
               user.FirstName = reader["FirstName"].ToString();
               user.LastName = reader["LastName"].ToString();
               MessageBox.Show("Welcome " + user.FirstName + " 
" +
                 user.LastName);
```

```
            }
            else
            {
                //Login failed
                MessageBox.Show("Authentication failed.");
                txtUser.Text = String.Empty;
                txtPassword.Password = String.Empty;
                txtUser.Focus();
            }

            connection.Close();
        }
...
```

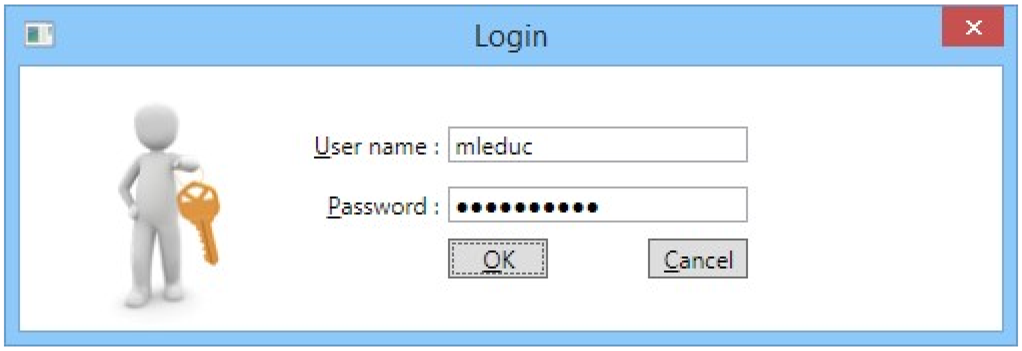
One last thing we want to change before we test is the moment when we close the connection. We want the connection to be closed in all cases so we will move the **connection.Close()** line in a ***finally*** block that we will add to the **try..catch** structure:

```
...
                else
                {
                    //Login failed
                    MessageBox.Show("Authentication failed.");
                    txtUser.Text = String.Empty;
                    txtPassword.Password = String.Empty;
                    txtUser.Focus();
                }


            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);

            }
            finally
            {
                connection.Close();
            }
        }
```

We can now test the application. Try to type anything in the textboxes and click OK. You should get an error message like that:

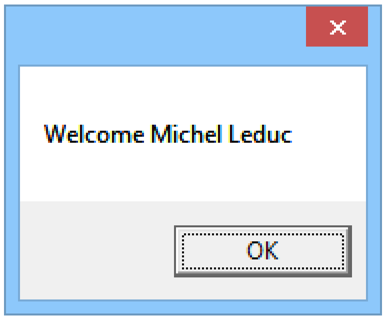

Authentication failed.

Now let's try with a real user. If you go in **SQL Server** and take a look at the content of the **tblUsers** table in the **College** database, you will see that there are two records (you can add more if you want). One of these users is *Michel Leduc*, username: **mleduc** and password: **projet9898**. Let's try with him:



User name : mleduc

Password : ●●●●●●●●●●


Welcome Michel Leduc

Ok, great, it works! We will now modify the code so that a valid user can access the main form of the application. Make the following modifications:

```
...
                              MessageBox.Show("Welcome " + user.FirstName
+ " " +
                               user.LastName);
                              //Create an instance of the form
                              StudentManagementUI studentManagement = new
                               StudentManagementUI(user);
                              //Show the form
                              studentManagement.Show();
```

```
                    open = false;
                    //Close the Login form
                    this.Close();
                }
...
```

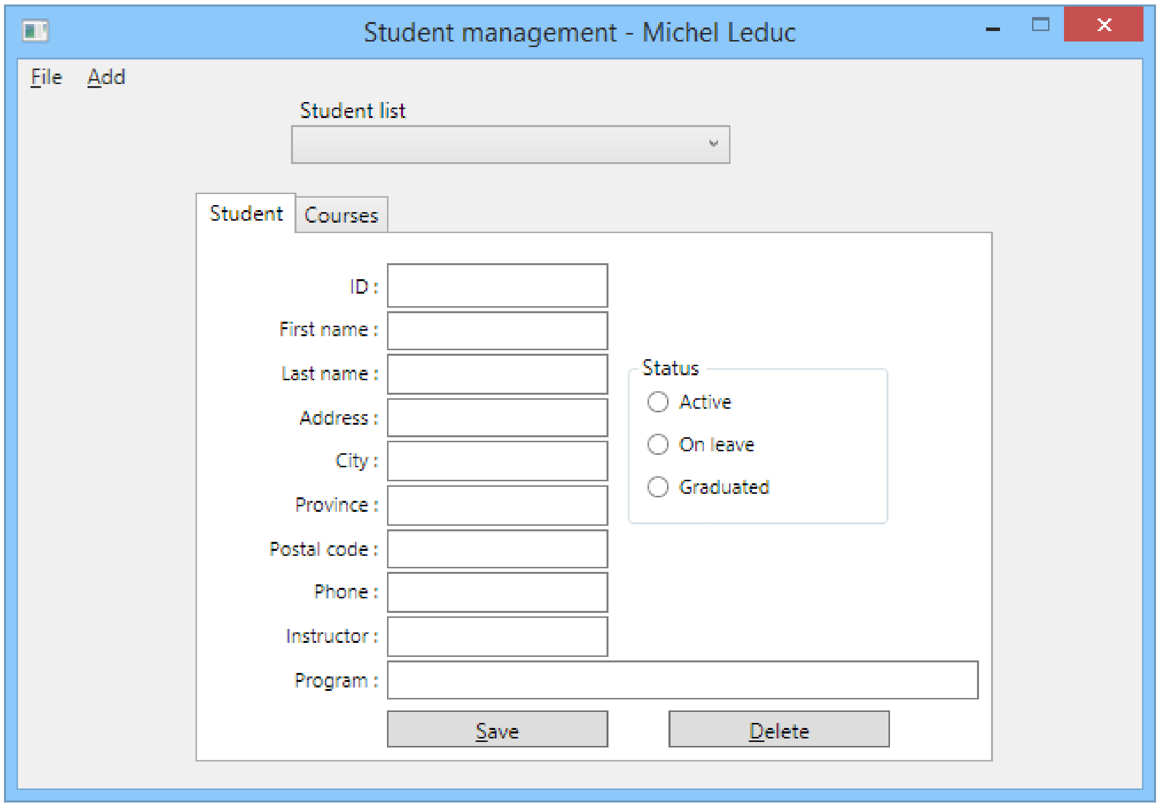
This code will give you an error, but don't worry it will be fixed in the next modification. In the ***Solution Explorer***, double-click on the **StudentManagementUI.xaml.cs** file to open it in the editor. First add the following ***using*** statement at the top:

```
using System.Configuration;
```

Then, make the following modifications in the class itself:

```

public partial class StudentManagementUI : Window
    {
        bool open = false;
        SqlConnection connection;
        CurrentUser user;

        public StudentManagementUI(CurrentUser current)
        {
            InitializeComponent();
            user = current;
            //Display the current user in the window's title
            Title += user.FirstName + " " + user.LastName;
            connection = new
SqlConnection(ConfigurationManager.
            ConnectionStrings["constr"].ConnectionString);
            open = true;
        }
...
```

This should fix the error that appeared in the previous modification as we have now added a ***CurrentUser*** type parameter to the constructor of the class. Run the app and try to login with user ***mleduc***. The following form should appear on screen:



The next step is to get all the students that are managed by the current user (*mleduc* in our example). For that, we will put our code in a new method so that it can be easily called when we need to get a student list. But before that, we need to define what a student is. If we look at the **tblStudents** table in the database, we can see what fields constitute a student:

```
SELECT [StudentId]
      ,[FirstName]
      ,[LastName]
      ,[Address]
      ,[City]
      ,[Province]
      ,[PostalCode]
      ,[Phone]
      ,[ProgramCode]
      ,[InstructorId]
      ,[Status]
  FROM [College].[dbo].[tblStudents]
```

So from this table definition we will create a **Student** class to more easily manage students in our program. We will not include all the fields in our class as only the *first name*, *last name* and *instructor ID* are of interest for now. Add a new class to the project and name the file **Student.cs**. Then, add the following code in it:

```
namespace StudentManagement
{
    class Student
    {
        public string StudentId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string FullName
        {
            get
            {
                return LastName.ToUpper() + ", " +
FirstName;
            }
        }
    }
}
```

Now go back to the **StudentManagementUI.xaml.cs** file and, right after the end of the constructor block, add the following code to create a new method called **FillStudentList**:

```
        public StudentManagementUI(CurrentUser current)
        {
            InitializeComponent();
            user = current;
            //Display the current user in the window's title
            Title += user.FirstName + " " + user.LastName;
            connection = new
SqlConnection(ConfigurationManager.
            ConnectionStrings["constr"].ConnectionString);
            open = true;
        }

        public void FillStudentList()
        {

        }
```

In this new method, add the following code:

```
public void FillStudentList()
        {
            //Create the SELECT query
            string selectStudents = "SELECT
             StudentId,FirstName,LastName FROM tblStudents
WHERE
             InstructorId='" + user.UserId + "' ORDER BY
LastName";
            try
            {
                connection.Open();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            finally
            {
                connection.Close();
            }
        }
```

We will now add a **SqlCommand** object and a **SqlDataReader** object in the code:

```
public partial class StudentManagementUI : Window
    {
        bool open = false;
        SqlConnection connection;
        CurrentUser user;
        SqlCommand command;
...
...
        public void FillStudentList()
        {
            //Create the SELECT query
            string selectStudents = "SELECT
            StudentId,FirstName,LastName FROM tblStudents WHERE
            InstructorId='" + user.UserId + "' ORDER BY
LastName";
            try
            {
                command = new SqlCommand(selectStudents,
connection);
                connection.Open();
                SqlDataReader reader = command.ExecuteReader();
            }
...
```

The last step is to grab all the students returned by the SELECT command and put them in a collection of students. To achieve this, we will use the **List<>** generic class that, like an array, can hold all our students. Make the following modifications:

```
public partial class StudentManagementUI : Window
    {
        bool open = false;
        SqlConnection connection;
        CurrentUser user;
        SqlCommand command;
        Student student;
        List<Student> students = new List<Student>();
...
...
            try
            {
                command = new SqlCommand(selectStudents,
connection);
                connection.Open();
                SqlDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    //Create a new student
                    student = new Student();
                    student.StudentId = reader["StudentId"]
                    .ToString();
                    student.FirstName = reader["FirstName"]
                    .ToString();
                    student.LastName =
reader["LastName"].ToString();
                    //Add to list
                    students.Add(student);

                }
            }
...
```

We can use a **while** loop to read through all the records returned by the **ExecuteReader** method. For each record, we create a student instance, fill it with data and add it to the list of students called ***students***. The final steps after that are to bind this student list to the ***ComboBox*** at the top of the form. This binding should make the ***ComboBox*** display the students that are part of the **students** list. The first thing to do is to assign the **students** list to the ***datacontext*** of the **ComboBox**. When can do that at the end of the **FillStudentList** method:

```
...
 while (reader.Read())
                {
                    //Create a new student
                    student = new Student();
                    student.StudentId = reader["StudentId"].ToString();
                    student.FirstName = reader["FirstName"].ToString();
                    student.LastName = reader["LastName"].ToString();
                    //Add to list
                    students.Add(student);

                }
                StudentList.DataContext = students.ToList();
...
```

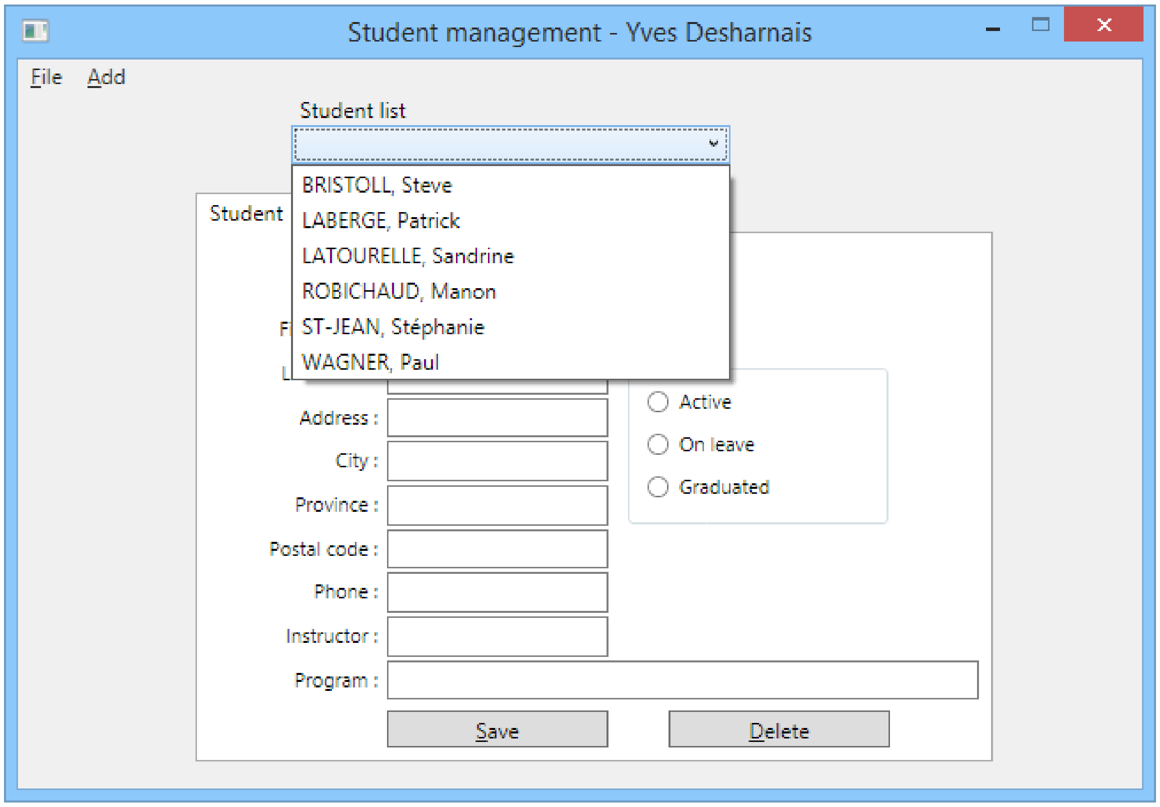
The second thing to do is to go in the XAML code of the form and tell the **ComboBox** what item of its **DataContext** it should display. Make the following modification in the **StudentManagementUI.xaml** file, on the line that creates the **ComboBox**:

```
...
<ComboBox x:Name="StudentList" HorizontalAlignment="Left"
Margin="154,37,0,0" VerticalAlignment="Top" Width="248"
Height="22" ItemsSource="{Binding}"
DisplayMemberPath="FullName"></ComboBox>
...
```

ItemSource={"Binding"} indicates that the source of the **ComboBox** will be established by code with a collection or a ***List***. The ***DisplayMemberPath*** property tells the **ComboBox** which field to use in the ***List***. Now the last step is to call the **FillStudentList** method in the constructor. Go back to the **StudentManagementUI.xaml.cs** file and add the following line:

```
...
public StudentManagementUI(CurrentUser current)
        {
            InitializeComponent();
            user = current;
            //Display the current user in the window's title
            Title += user.FirstName + " " + user.LastName;
            connection = new
SqlConnection(ConfigurationManager.
             ConnectionStrings["constr"].ConnectionString);
            FillStudentList();
            open = true;
        }
...
```

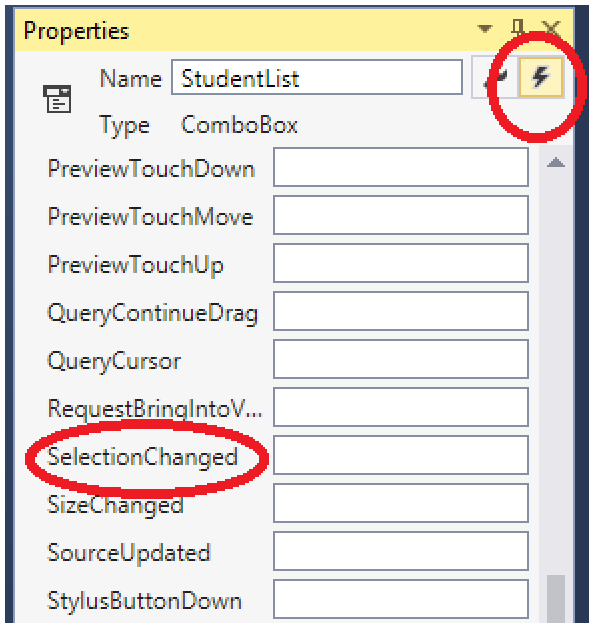
You can now try to run the application. First, login as ***mleduc***. When you get to the main form, click on the students **ComboBox**. You should see all of ***Michel Leduc's*** students. Exit the program and try it again with another user (try ***ydesharnais*** with password ***Chanelle2010***). You should see that user's students in the list.



BRISTOLL, Steve
LABERGE, Patrick
LATOURELLE, Sandrine
ROBICHAUD, Manon
ST-JEAN, Stéphanie
WAGNER, Paul

We are now ready for the next phase which is to enable student selection. When the user selects a student in the list, we want the app to retrieve that student's info in the database and display it in the appropriate TextBoxes. To do that, we will use the **SelectionChanged** event of our ComboBox.

Go in the **StudentManagementUI.xaml** file in design mode and click on the *Student list* ComboBox to select it. Now look at the property windows and click on the small lightning symbol. This will show you the events associated with the ComboBox:



Name StudentList
Type ComboBox

Locate the **SelectionChanged** event and double-click on the textbox to its right. This should bring you in the C# code that will handle the ***SelectionChanged*** event for that ComboBox. A ***SelectionChanged*** event is triggered each time the user selects a student in the ComboBox. In fact, anything that changes which student is currently selected will trigger that event.

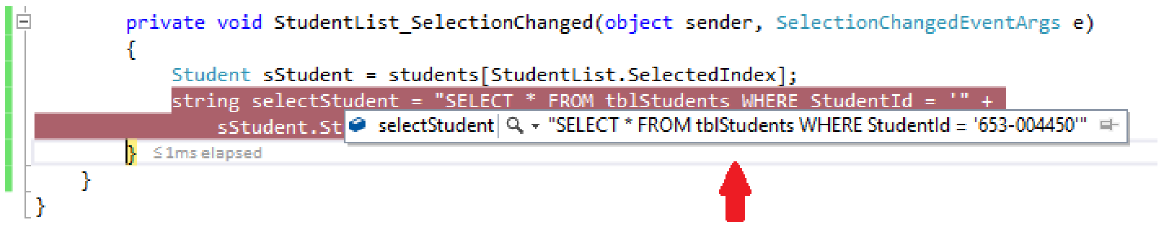
The objective here is to grab whatever student has been selected in the list and store it in a ***Student*** variable. Keep in mind that although it is the student's name that is displayed in the list, the ComboBox can also return the index position of that student. So in the **StudentList_SelectionChanged** event handler method that has just been generated, write the following code:

```
private void StudentList_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
              Student sStudent =
students[StudentList.SelectedIndex];
        }
```

Now that we have grabbed our student and stored it in the **sStudent** variable, we can create an SQL statement that will retrieve its full info from the database:

```
private void StudentList_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
            Student sStudent =
students[StudentList.SelectedIndex];
            string selectStudent = "SELECT * FROM tblStudents
WHERE
                        StudentId = '" + sStudent.StudentId +
"'";
        }
```

At that point, it would be nice to know if our SELECT statement makes sense. To find out, let's put a breakpoint on that last line of code and run the program. Login as any user and select any student in the list. Once the breakpoint is reached, press F10 to make a step and then put your mouse pointer on the **selectStudent** variable. You should see this:



```
private void StudentList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Student sStudent = students[StudentList.SelectedIndex];
    string selectStudent = "SELECT * FROM tblStudents WHERE StudentId = '" +
        sStudent.St
}  ≤1ms elapsed
    }
}
```

selectStudent "SELECT * FROM tblStudents WHERE StudentId = '653-004450'"

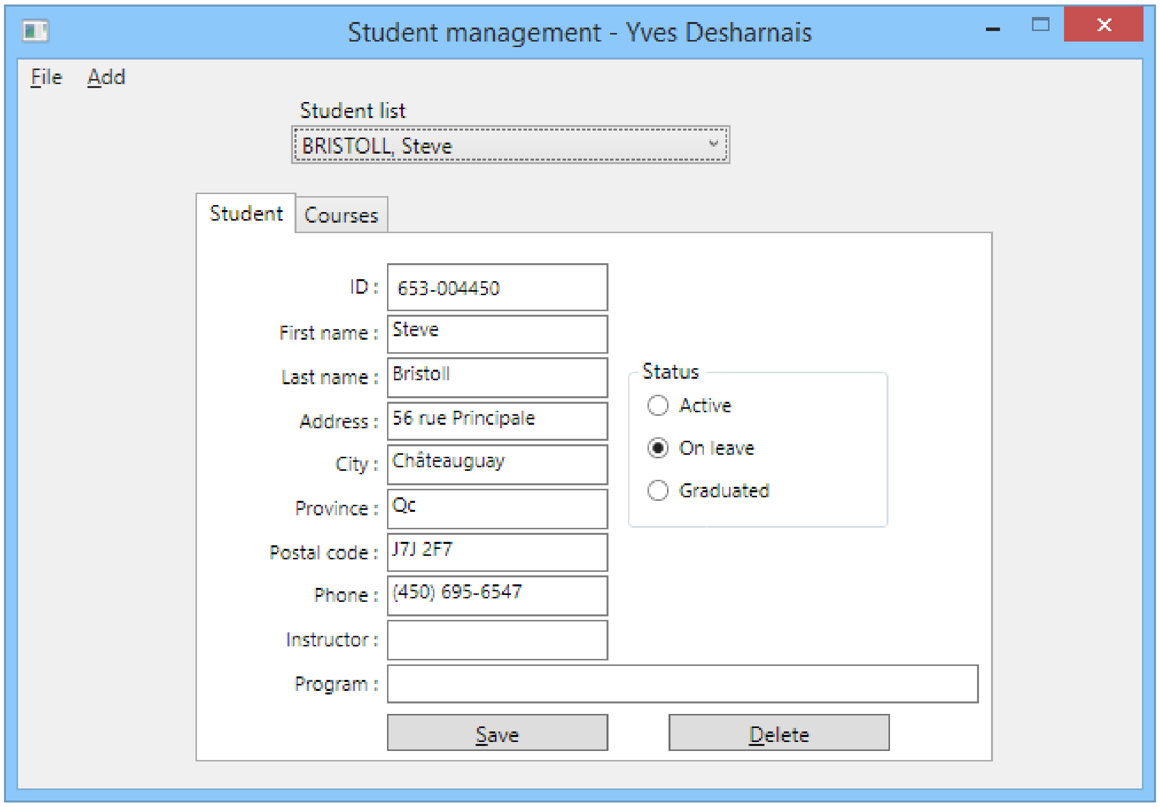
The SQL SELECT statement looks ok.We can now proceed with the rest of the code to actually retrieve the student from the database and fill the form:

```
private void StudentList_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
            Student sStudent =
students[StudentList.SelectedIndex];
            string selectStudent = "SELECT * FROM tblStudents
WHERE
                StudentId = '" + sStudent.StudentId + "'";
            command = new SqlCommand(selectStudent,
connection);
            connection.Open();
            SqlDataReader reader = command.ExecuteReader();
            if (reader.Read())
            {
                lblID.Content = reader["StudentId"];
                txtFirst.Text = reader["FirstName"].ToString();
                txtLast.Text = reader["LastName"].ToString();
                txtAddress.Text = reader["Address"].ToString();
                txtCity.Text = reader["City"].ToString();
                txtProvince.Text =
reader["Province"].ToString();
                txtPostalCode.Text =
reader["PostalCode"].ToString();
                txtPhone.Text = reader["Phone"].ToString();
                //Selects the correct radiobutton
                switch (reader["Status"].ToString())
                {
                    case "0":
                        rbActive.IsChecked = true;
                        break;
                    case "1":
                        rbOnLeave.IsChecked = true;
                        break;
                    case "2":
                        rbGraduated.IsChecked = true;
                        break;
                }

            }
            connection.Close();
        }
```

Before we test this code, let's add a little protection in the case that the **SelectionChanged** event is triggered and there is no student selected in the list (Yes that can happen!). When nothing is selected in a ComboBox, its **SelectedIndex** value is at -1. Let's test for that. Add the following code at the start of the method:

```

private void StudentList_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
            //If no student is selected
            if (StudentList.SelectedIndex==-1)
            {
                return;
            }
...

```

Now we are ready to test the app. Run the program and login as any of the two users. Then select a student in the list. You should see the form get filled with the information related to the selected student:



BRISTOLL, Steve

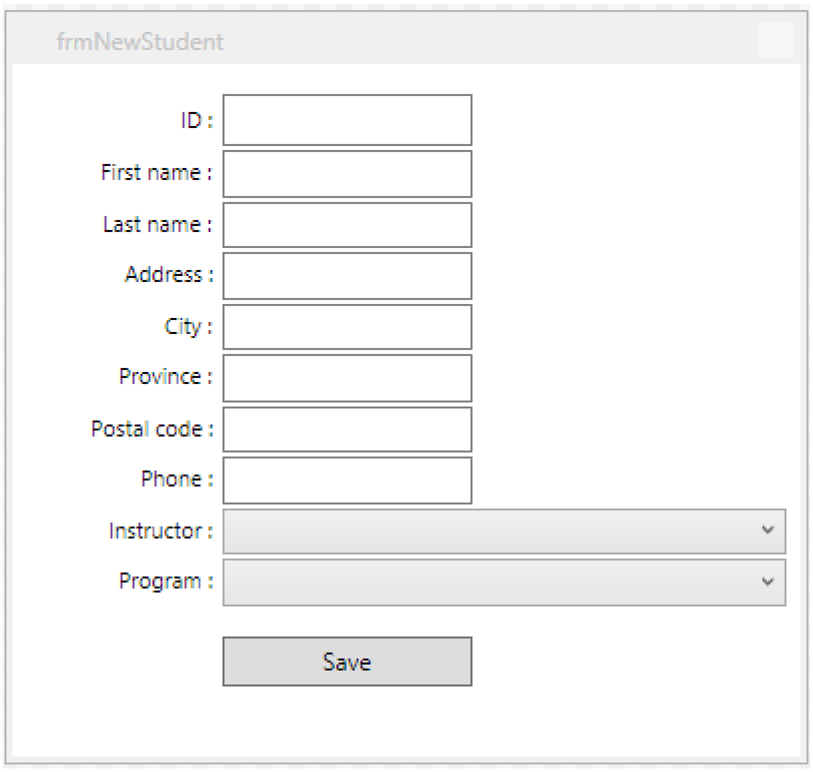
ID :	653-004450
First name :	Steve
Last name :	Bristoll
Address :	56 rue Principale
City :	Châteauguay
Province :	Qc
Postal code :	J7J 2F7
Phone :	(450) 695-6547
Instructor :	
Program :	

( ) Active [✓ On leave] ( ) Graduated

As you can see, all the fields except *Instructor* and *Program* have been filled. We will not worry about those two for the moment.

Make sure that the information displayed in the form is accurate. Validate by going in the database table **tblStudents** and check if the information on the screen matches with that content.

We will now modify the application so that it can add a new student. An ***Add student*** form has already been created for you. It's called **frmNewStudent.xaml**. Display it on the screen to inspect it:



Now go in the **frmNewStudent.xaml.cs** file to see the code. We will add a couple of lines to establish the connection to the database and to manually fill the *Instructor* and *Program* ComboBoxes (we could fill these ComboBoxes directly from the database, but we'll keep it simple for now).

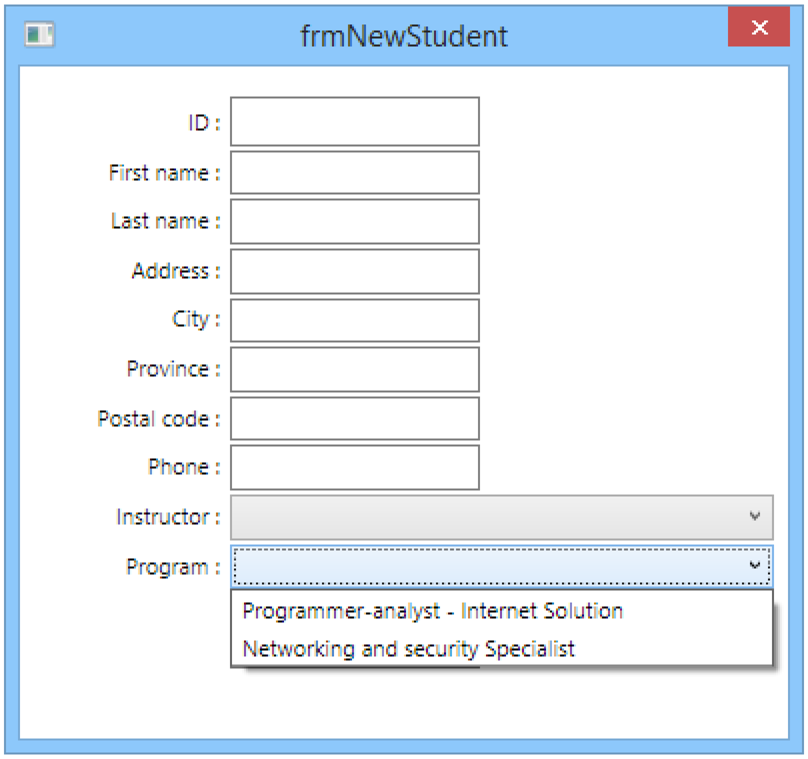
```
...
using System.Configuration;
...
public frmNewStudent()
        {
            InitializeComponent();
            connection = new
SqlConnection(ConfigurationManager.

ConnectionStrings["constr"].ConnectionString);
            cmbInstructor.Items.Add("Yves Desharnais");
            cmbInstructor.Items.Add("Michel Ledeuc");
            cmbProgram.Items.Add("Programmer-analyst - Internet
                                       Solution");
            cmbProgram.Items.Add("Networking and security
                                       Specialist");
        }
```

```
...
```

Now let's add code to make the **Add - Student** menu option work. Go in the **frmStudentManagementUI.xaml.cs** and locate the **mnuAddStudent_Click** method. Then, add the following code:

```
  private void mnuAddStudent_Click(object sender,
RoutedEventArgs e)
        {
            //Create new form instance
            frmNewStudent addStudent = new frmNewStudent();
            //Display the form
            addStudent.ShowDialog();
        }
```

Notice that we used **ShowDialog** instead of **Show** to display the form. ***ShowDialog*** displays the form in modal mode, meaning that the form under it (frmStudentManagementUI) cannot be accessed as long as the new student form is open. Now run the program. Login as any user and click the **Add - Student** menu option. You should see the ***Add new student*** form. Inspect the two ComboBoxes at the bottom to make sure they are filled:



As you did earlier for the ***Student*** ComboBox, generate the **SelectionChanged** event handler methods for both ComboBoxes (*Instructor* and *Program*). Then, add the following code:

```
        private void cmbProgram_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
            switch (cmbProgram.SelectedIndex)
            {
                case 0:
                    programId = "LEA.9C";
                    break;
                case 1:
                    programId = "LEA.AE";
                    break;
            }
        }

        private void cmbInstructor_SelectionChanged(object
sender,

SelectionChangedEventArgs e)
        {
            switch (cmbInstructor.SelectedIndex)
            {
                case 0:
                    InstructorId = "yd001";
                    break;
                case 1:
                    InstructorId = "ml001";
                    break;
            }
        }
```

We are now going to add a method that will make sure that the user input is valid when a new student is added. Go at the bottom of the code, right after the end of the last method but before the end of the class, and add the method **ValidateInput**:

```
public bool ValidateInput()
        {
            bool OK = true;
            if (txtID.Text.Trim()==string.Empty ||
                txtFirst.Text.Trim()==string.Empty ||
                txtLast.Text.Trim() == string.Empty ||
                txtAddress.Text.Trim() == string.Empty ||
                txtCity.Text.Trim() == string.Empty ||
                txtProvince.Text.Trim() == string.Empty ||
                txtPostalCode.Text.Trim() == string.Empty ||
                txtPhone.Text.Trim() == string.Empty ||
                cmbInstructor.SelectedIndex==-1 ||
                cmbProgram.SelectedIndex==-1)
            {
                OK = false;
            }
            return OK;
        }
```

This method will make sure that all the textboxes are not empty and that a selection has been made in the two comboboxes. If any of the controls are left empty, the method will return ***false***.

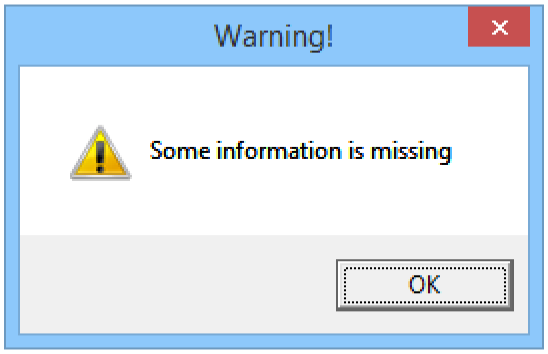
Now let's turn our attention to the **Save** button. We will generate a **Click** event handler method the same way that we did for the comboboxes. In the **frmNewStudent.xaml** file, click on the **Save** button and list its events in the property window (by clicking on the *lightning* symbol). Double-click right to the **Click** event. This should bring you in the **btnSave_Click** method in the C# code.

The first thing we'll do in this method is to call the **ValidateInput** method we just created and display an error message if the method returns **false**. Add the following code:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            bool OK = ValidateInput();
            if (OK)
            {

            }
            else
            {
                MessageBox.Show("Some information is missing",
                 "Warning!", MessageBoxButton.OK,
                 MessageBoxImage.Exclamation);
            }
        }
```

We can test what we just did right away. Run the app and login. Make the ***Add new student*** form appear. Type something in all the textboxes and select something in both comboboxes. Then, if you click on **Save**, nothing should happen (which is normal!).

Now empty one of the textboxes and click **Save** again. An error message should appear which proves that the validation is taking place:



Some information is missing

We are now ready to add the code that will add a new student in the **tblStudents** table. To do so, we will need a **SqlCommand** object to run a SQL INSERT instruction. Add the following code to the **btnSave_Click** method:

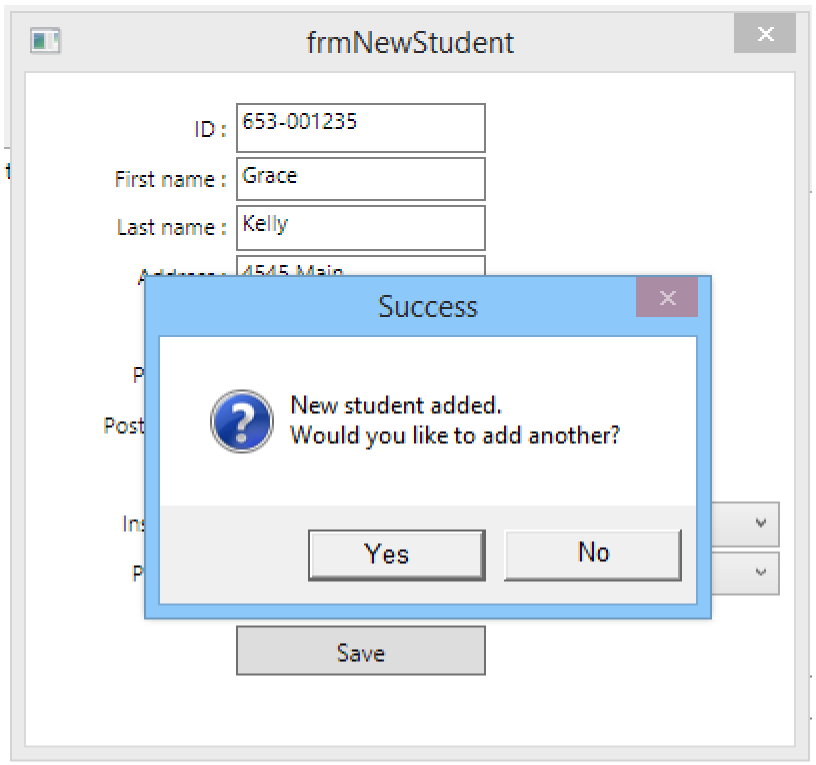
```
private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            bool OK = ValidateInput();
            if (OK)
            {
                string insertStudent = $"INSERT INTO
tblStudents(StudentId,

FirstName,LastName,Address,City,Province,PostalCode,Phone,
                  ProgramCode,InstructorId)

Values('{txtID.Text}','{txtFirst.Text}','{txtLast.Text}',

'{txtAddress.Text}','{txtCity.Text}','{txtProvince.Text}',

'{txtPostalCode.Text}','{txtPhone.Text}','{programId}',
                  '{InstructorId}')";
                command = new SqlCommand(insertStudent, connection);
                try
                {
                    connection.Open();
                    int line = command.ExecuteNonQuery();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
                finally
                {
                    connection.Close();
                }
```

```
           }
           else
...
```

The instruction that actually runs the INSERT command is ***command.ExecuteNonQuery()*** which is a method that executes a SQL command and doesn't expect any records to be returned, unlike a SELECT command. The INSERT command itself is a pretty big chunk of text because we are inserting in a table that has a lot of fields. The ***Add student*** functionality now works, but we will add a bit more code to inform the user that the student was added if everything went well:

```
...
                try
                {
                    connection.Open();
                    int line = command.ExecuteNonQuery();
                    if (line != 0)
                    {
                        if (MessageBox.Show("New student 
added."+
                            Environment.NewLine +
                            "Would you like to add another?",
                            "Success",MessageBoxButton.YesNo,
                            MessageBoxImage.Question)==
                            MessageBoxResult.No)
                        {
                            this.Close();
                        }
                    }
                }
...
```

Try the app. You should now be able to add a new student and get a confirmation message. Check the database afterwards to make sure that the new student is in the **tblStudents** table.



ID : 653-001235
First name : Grace
Last name : Kelly

New student added.
Would you like to add another?

You surely noticed that the app asks you if you want to add another student. This is possible because the **MessageBox** object returns a value that we can evaluate with an ***if*** statement. In this case, if the **MessageBox** returns ***No***, we just close the form, which will return us to the main form.

Let's add some code to clear the textboxes if the user chooses to add another student (just add an ***else*** to the ***if*** we just discussed):

```
...
                        {
                            this.Close();
                        }
                        else
                        {
                           txtID.Text = txtFirst.Text = string.Empty;
                           txtLast.Text = txtAddress.Text =
string.Empty;
                           txtCity.Text = txtProvince.Text =
string.Empty;
                           txtPostalCode.Text = txtPhone.Text =
string.Empty;
                           cmbInstructor.SelectedIndex = -1;
                           cmbProgram.SelectedIndex = -1;
                           txtID.Focus();
                        }
```

```
...
```

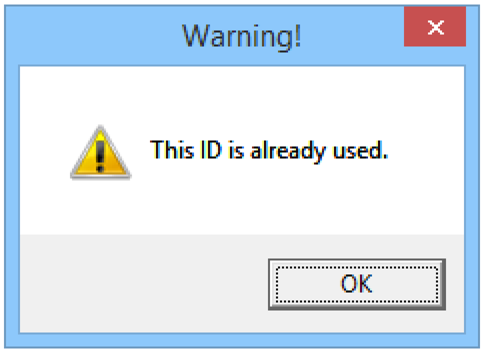
Try to add another student and then answer ***yes*** to add another. The form should be cleared and ready for another entry.

We will now make another test: let's try to add a student using a student ID that already exists. For instance, student ID ***653-002525*** is student *Gary Fisher*'s ID. Run the app and create a student with the same ID. You should get a ***PRIMARY KEY VIOLATION*** error. The fact that the app didn't crash is a good sign, it proves that our ***try..catch*** structure works. Indeed, if we would have done the INSERT outside of a ***try***, the application would have crashed.

There are two problems that can occur with our app at this point: the one you just witnessed (*primary key violation*) and if you try to enter a province with more than two characters (the field's limit in the table). We will make our app a little more robust by fixing those issues. Go near the beginning of the **btnSave_Click** method and add the following code:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            bool OK = ValidateInput();
            if (OK)
            {
                //Query to check if ID exists
                string check = $"SELECT StudentId FROM
tblStudents
                              WHERE StudentId='{txtID.Text}'";
                command = new SqlCommand(check, connection);
                try
                {
                    connection.Open();
                    SqlDataReader reader =
command.ExecuteReader();
                    if (reader.Read())
                    {
                        MessageBox.Show("This ID is already
used.",
                        "Warning!", MessageBoxButton.OK,
                        MessageBoxImage.Exclamation);
                        return;
                    }
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
                finally
                {
                    connection.Close();
                }
...
```

This extra code simply checks the database to see if a record with the student id already exists by using a SELECT command. If a record is found, an error message is displayed and then a **return** statement is executed to leave the method right away. The interesting thing is that the **finally** block will still be executed before leaving the method, making sure that the connection will be closed. Run the app and try to create a new student with an existing ID. You should get the following result:


Warning!

This ID is already used.

Now let's fix the other issue, the 2 character limit for the ***province*** field. Make the following modification in the last **catch** statement of the method:

```

...
  txtID.Focus();
                          }
                      }
                  }
                  catch (Exception ex)
                  {
                      MessageBox.Show("The province can only
contain 2
                       characters.", "Warning!",
MessageBoxButton.OK,
                       MessageBoxImage.Exclamation);
                  }
                  finally
```

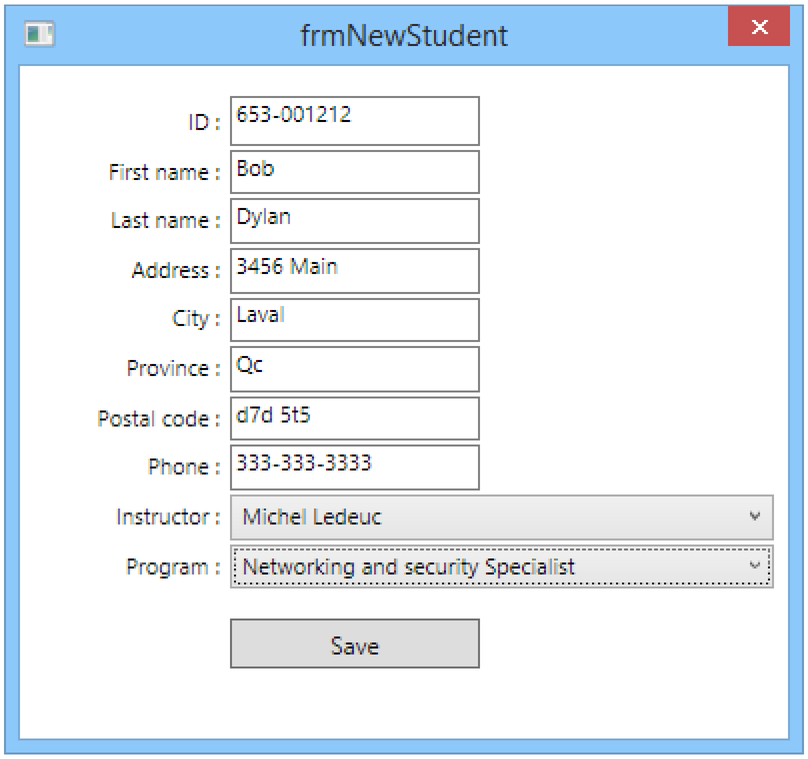
```
                 {
                     connection.Close();
...
```

At that point, the *province* is the only field that can make the INSERT statement fail, so we'll give the user an appropriate error message for that case. Test your program again and you should see that everything now works fine.

But the ***Add student*** functionality gives rise to a new problem! The new student doesn't appear in the instructor's list unless we close the application and start it again. For example, login as ***mleduc***, add a new student that has ***Michel Leduc*** for his instructor and return to the main form. Your new student will not be in ***mleduc***'s list. This is because the ComboBox that displays the students is bound to the **List** object that contains the students and it needs to be refreshed. To do so, we will go back to the **mnuAddStudent_Click** method in the **StudentManagementUI.xaml.cs** file and we'll make the following modification:

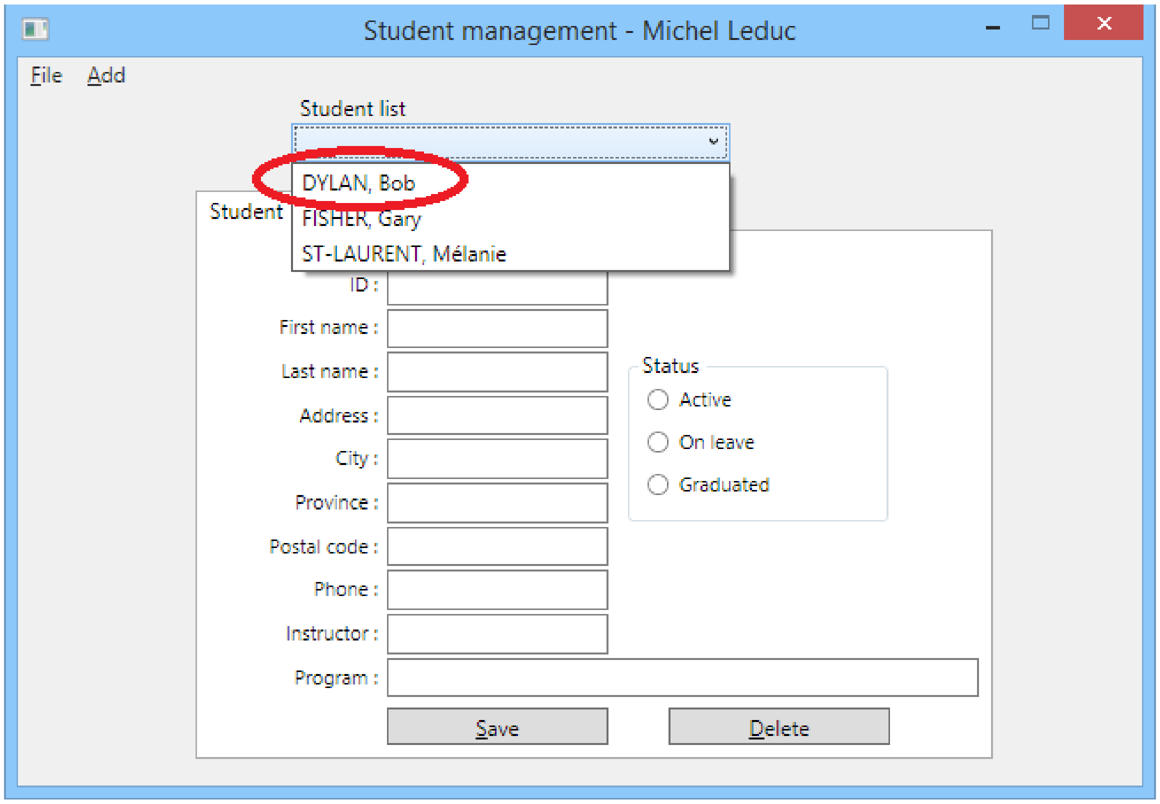
```
private void mnuAddStudent_Click(object sender, RoutedEventArgs
e)
        {
            //Create new form instance
            frmNewStudent addStudent = new frmNewStudent();
            //Display the form
            addStudent.ShowDialog();
            //Refresh the student list
            students.Clear();
            FillStudentList();
        }
```

Now login as ***mleduc*** and create a new student that has Michel Leduc as his instructor. When you return to the main form, you will see this student in the ComboBox:



frmNewStudent

ID :	653-001212
First name :	Bob
Last name :	Dylan
Address :	3456 Main
City :	Laval
Province :	Qc
Postal code :	d7d 5t5
Phone :	333-333-3333
Instructor :	Michel Ledeuc
Program :	Networking and security Specialist



Student management - Michel Leduc

Student list

DYLAN, Bob
FISHER, Gary
ST-LAURENT, Mélanie

We will now work on the **Save** button of the main form. The goal of this button is to save the modifications made to a student's information in the main form. First, generate the **btnSave_Click** method using the technique you have learned before. You should then be sent in the C# code of that method. Add the following code:

```
  private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            int status = -1;
            //Get the status from the radioButtons
            status = (rbActive.IsChecked == true ? 0 :
                rbOnLeave.IsChecked == true ? 1 :
                rbGraduated.IsChecked == true ? 2 : -1);
        }
```

Instead of verifying the three RadioButtons with many ***if*** statements, here we are using ***ternary operators*** to make the code less bulky. The goal is to assign a number to the **status** variable depending on which RadioButton has been selected (0,1 or 2). A value of **-1** indicates that no RadioButton has been selected.

The following code checks the **status** variable to make sure it is not set to -1. If it's not the case, an error message is displayed:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            int status = -1;
            //Get the status from the radioButtons
            status = (rbActive.IsChecked == true ? 0 :
                rbOnLeave.IsChecked == true ? 1 :
                rbGraduated.IsChecked == true ? 2 : -1);
            if (status!=-1)
            {

            }
            else
            {
                MessageBox.Show("You must select a status!",
                "Warning!", MessageBoxButton.OK,
                 MessageBoxImage.Exclamation);
            }
        }
```

Inside the ***if*** statement that checks the status, we will put a nested ***if*** statement that checks if the other controls have been properly filled. For that, we will use a **ValidateInput** method, much like we have done before:

```
...
if (status!=-1)
            {
                if (ValidateInput())
                {

                }
                else
                {
                    MessageBox.Show("Some information is
missing",
                    "Warning!", MessageBoxButton.OK,
                    MessageBoxImage.Exclamation);
                }

            }
            else
...
```

And then let's add the **ValidateInput** method at the end of the class:

```
public bool ValidateInput()
        {
            bool OK = true;
            if(txtFirst.Text.Trim() == string.Empty ||
               txtLast.Text.Trim() == string.Empty ||
               txtAddress.Text.Trim() == string.Empty ||
               txtCity.Text.Trim() == string.Empty ||
               txtProvince.Text.Trim() == string.Empty ||
               txtPostalCode.Text.Trim() == string.Empty ||
               txtPhone.Text.Trim() == string.Empty)
            {
                OK = false;
            }
            return OK;
        }
```

Now let's go back to the **btnSave_Click** method and add the code for the actual database update. You will see that it is pretty much the same as the INSERT command we previously did, but we will use a UPDATE command instead:

```
...
                if (ValidateInput())
                {
                    try
                    {
                        connection.Open();
                        try
                        {
                            //Create the UPDATE command
                            string saveStudent = "UPDATE 
tblStudents
                             SET FirstName='" +
                             txtFirst.Text + "',LastName='" +
                             txtLast.Text + "',Address='" +
                             txtAddress.Text + "',City='" +
                             txtCity.Text + "',Province='" +
                             txtProvince.Text + 
"',PostalCode='" +
                             txtPostalCode.Text + "',Phone='" +
                             txtPhone.Text + "',Status='" + 
status +
                             "' WHERE StudentId='" +
                             lblID.Content + "'";
                            command = new 
SqlCommand(saveStudent,
                              connection);
                            command.ExecuteNonQuery();
                            MessageBox.Show("The student has 
been
                              updated.", "Success!",
                            MessageBoxButton.OK,
                            MessageBoxImage.Information);
                        }
                        catch
                        {
                            MessageBox.Show("The province can 
only
                            contain 2 characters.", "Warning!",
                            MessageBoxButton.OK,
                            MessageBoxImage.Exclamation);
                        }

                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show(ex.Message);
                    }
                    finally
                    {
                        connection.Close();
                    }
                }
                else
...
```

There are two levels of error handling happening in this last bit of code. The first ***try..catch*** structure handles the database connection errors and the second ***try..catch*** structure handles the UPDATE error which, in our case, involves the province field size issue.

Now you must test the app using different scenarios: a valid update, a problem due to province length, a problem due to no status selection, etc. Make sure you get the correct response from the app in all situations.

And now we will conclude this exercise with the implementation of the **Delete** button on the main form. As usual, generate the click event of the button using the property window. We will then start by making sure that there is an actual student selected in the list. After that, we will use a MessageBox to confirm that the user really wants to delete the selected student. Add the following code to the **btnDelete_Click** method:

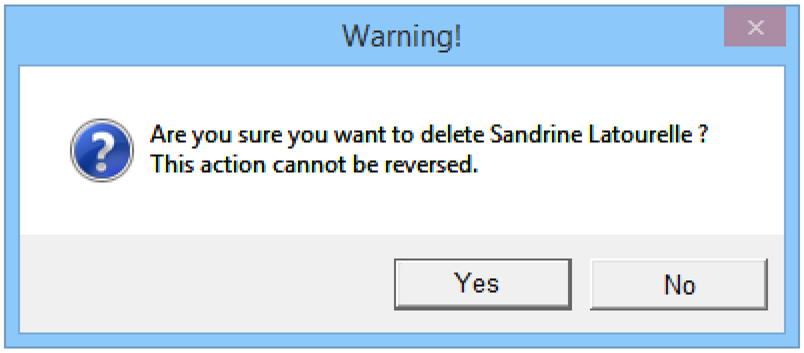
```
private void btnDelete_Click(object sender, RoutedEventArgs e)
        {
            if (StudentList.SelectedIndex!=-1)
            {
                if (MessageBox.Show($"Are you sure you want to
delete
                   {txtFirst.Text} {txtLast.Text} ?"+
                    Environment.NewLine+"This action cannot be
                    reversed.",
"Warning!",MessageBoxButton.YesNo,

MessageBoxImage.Question)==MessageBoxResult.Yes)
                {

                }
            }
        }
```

This way, we can make sure that the user is not deleting anyone by accident. If the user confirms, then we can execute a DELETE command in the database. You will be happy to know that the DELETE instruction is much shorter than an INSERT or UPDATE:

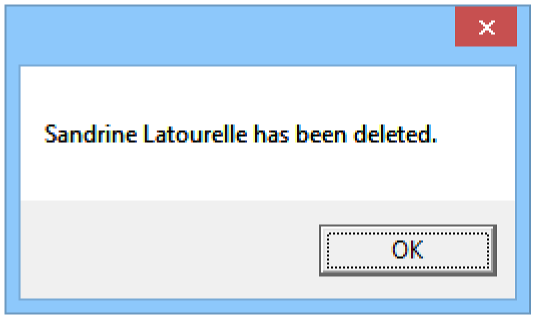
```
...
```

```
if (MessageBox.Show($"Are you sure you want to delete
                   {txtFirst.Text} {txtLast.Text} ?"+
                    Environment.NewLine+"This action cannot be
                    reversed.",
"Warning!",MessageBoxButton.YesNo,

MessageBoxImage.Question)==MessageBoxResult.Yes)
                {
                    try
                    {
                        connection.Open();
                        string delete = $"DELETE FROM
tblStudents
                        WHERE
StudentId='{lblID.Content.ToString()}'";
                        command = new SqlCommand(delete,
connection);
                        int line = command.ExecuteNonQuery();
                        if (line!=0)
                        {
                            MessageBox.Show($"{txtFirst.Text}
                             {txtLast.Text} has been
deleted.");
                            //Refresh the list
                            connection.Close();
                            students.Clear();
                            FillStudentList();
                            //Empty the form
                            lblID.Content = "";
                            txtFirst.Text = txtLast.Text = "";
                            txtAddress.Text = txtCity.Text =
"";
                            txtProvince.Text =
txtPostalCode.Text= "";
                            txtPhone.Text = "";
                        }
                    }
                    catch (Exception ex)
                    {
                        MessageBox.Show(ex.Message);
                    }
                    finally
                    {
                        connection.Close();
                    }
                }
...
```

You can now test the app by trying to delete a student. Again, make sure you test all the possibilities and check the database to see if the DELETE command did its job.


Warning!

Are you sure you want to delete Sandrine Latourelle ?
This action cannot be reversed.


Sandrine Latourelle has been deleted.

We will end this example here. If you want a little challenge, you can try to add the missing code to display the instructor and program of a student in the main form.

SUMMARY

- To connect to an SQL Server database, we must use a **SqlConnection** object.
- The **SqlConnection** object needs the following information to establish a connection: Server name, database name, a user name and a password (or the Windows credentials).
- The connection is opened using the **Open**() method from the **SqlConnection** object.
- It is important to close a connection when we are done with it to release the resource from memory. If we try to open a connection that is already open, an error will occur.
- All the SQL instructions that you have seen in previous courses can be used in your C# applications
- To execute an SQL command, we can use a **SqlCommand** object.
- We can read the result of a SELECT command with a **SqlDataReader** object and his **ExecuteReader** method.
- The **Read** method of a **SqlDataReader** object can read a result set one record at a time.
- All SQL commands that do not return a result set like INSERT, UPDATE and DELETE can be executed with the help of the **ExecuteNonQuery** method. This method returns the number of rows affected.

Progress check

Check your answer in Appendix A

Read carefully the statements and circle the letter that corresponds to the right answer.

1. True or false: the ExecuteNonQuery() method returns an integer.

- **a**. True
- **b**. False

2. True or false: a SqlCommand object can execute a query.

- **a**. True
- **b**. False

3. From which object does a SqlDataReader get his records?

- **a**. Connection
- **b**. SqlConnection
- **c**. SqlCommand
- **d**. None of the above

4. Which method loads the data in a SqlDataReader object?

- **a**. Read()
- **b**. ExecuteReader()
- **c**. Open()
- **d**. Select()

5. Which part of the connection string contains the database name?

- **a**. database
- **b**. userDB
- **c**. initial catalog
- **d**. startup

It is now time for your first exam. Go see your instructor and ask for the midterm exam.

Module 3

Adding parameters to commands and using stored procedures

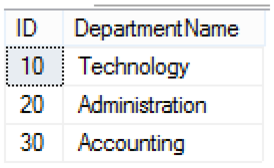
With what you have seen up to this point, it`s not hard to imagine how someone with bad intentions who knows the structure of a database can be viewed as a security risk. In this lesson, we will see how to use parameters to prevent problems such as attacks by SQL injection.

OBJECTIVES

- Add parameters to commands.
- Understand and use stored procedures.

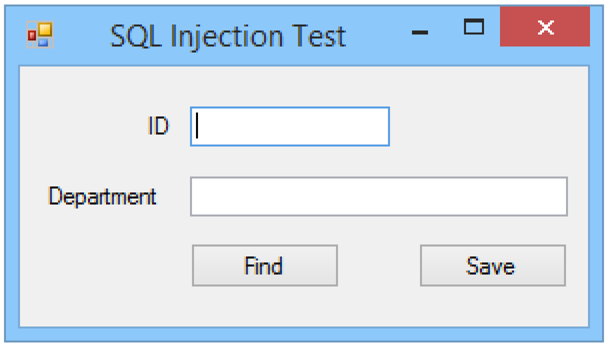
SQL injection

More and more, we read about people trying to cause problems by attacking servers and databases. One of the more popular and easier ways to do that is by ***SQL injection attack***. An **SQL injection attack** is someone trying to change the meaning of an SQL command executed by an application by injecting SQL keywords and expressions in it. To see an example of that, we will use the **InjectionSQL** project that already exists in the ***work files*** folder. With the project, you will find a file called **SQL_Injection.sql**. Execute it in **SQL Server** to create our test database. This database only contains one table called **tblDepartments**. This is a very simple table that only has two fields: ***ID*** and ***DepartmentName***. Look at the content of the table. You should see three records:



ID	DepartmentName
10	Technology
20	Administration
30	Accounting

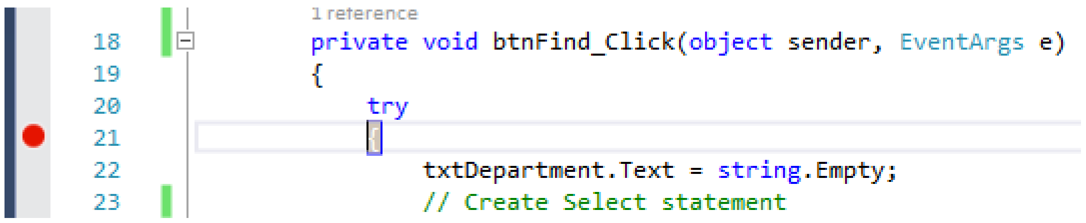
The Application itself is also very simple. Open it with Visual Studio. You will see a simple form that contains a few controls. If you enter a department ID in the ID TextBox and press **Find**, the department name will be displayed in the other TextBox. If the department doesn't exist, you will get an error message. **(NB: This application is of *Windows Forms* type and not *WPF*. Don't worry, the C# code remains the same.)**



Run the application and test it. Inspect the code in the **btnFind_Click** method. Everything seems to work fine. Yet, if a malicious user who knows the structure of your database wanted to cause trouble, he could! How would he do it? Simply by injecting SQL code in his textbox entry. Imagine that the user writes this in the ID Textbox:

```
20; DELETE FROM tblDepartments; --
```

This simple line contains an instruction that will delete everything from the ***tblDepartments*** table and nobody will know about it until the next time the database in searched. Let's try it ourselves. First, insert a *breakpoint* on the line just below the ***try*** instruction at the beginning of the **btnFind_Click** method:

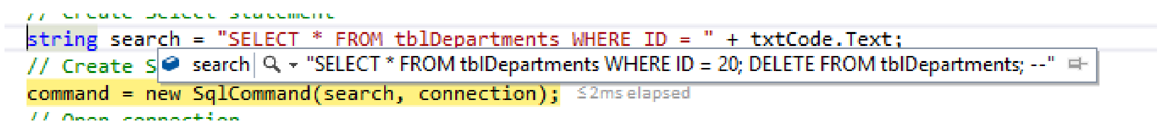


```
1 reference
private void btnFind_Click(object sender, EventArgs e)
{
    try
    {
        txtDepartment.Text = string.Empty;
        // Create Select statement
```

After that, run the program. Type this in the ID Textbox:

```
20; DELETE FROM tblDepartments; --
```

And click ***Find***. The app should stop on your breakpoint. Make a few steps (**F10**) until you go past where your ***SELECT*** statement is and put your mouse pointer on the **search** variable:



```
string search = "SELECT * FROM tblDepartments WHERE ID = " + txtCode.Text;
// Create S   search  "SELECT * FROM tblDepartments WHERE ID = 20; DELETE FROM tblDepartments; --"
command = new SqlCommand(search, connection);  ≤2ms elapsed
```

The SQL command now holds two instructions: a SELECT and a DELETE. If you continue with the steps, you will see that the application ends normally with no errors, but all the rows are now gone from the table. If you end the program, remove the breakpoint and run it again, you will see that there are no departments in the database. Go in the database in **SQL Server** and check the ***tblDepartments*** table. You will see that it is empty.

We will now see some solutions to that problem. Start by putting the deleted records back in the table by executing the three INSERT commands from the **SQL_injection.sql** file. Then make the following modifications in the **cmdFind_Click** method:

```
private void btnFind_Click(object sender, EventArgs e)
```

```
        {
            try
            {
                txtDepartment.Text = string.Empty;
                // Create Select statement
                string search = "SELECT * FROM
tblDepartments
                 WHERE ID = '" + txtCode.Text+ "'";
...
```

Run the app and try to use it normally. You will see that everything works fine. Now try to do the SQL injection entry that you tested earlier. You should get an error message saying that the conversion to ***smallint*** has failed. This happens because the system is able to automatically convert a ***string*** to a ***smallint*** when the string is a valid number ('20', for example). But it fails when it tries to convert the SQL injection string in a ***smallint***. The system needs to make that conversion since we have now enclosed the value in single quotes, making that value a ***char*** type item. This could be a valid solution to the SQL injection problem, but it doesn't always work well and better methods exist. Let's take a look at them.

Adding parameters to commands

In the previous lesson, you have created SQL commands by using concatenation. This means that you have *glued* bits of strings together to form a valid SQL statement. It is a fast but cumbersome way to work (as you may have noticed) because it is easy to get mixed up in all the single quotes and double quotes required to make the statement work. It is also not very safe because it leaves your application vulnerable to SQL injection attacks.

A solution to that is to use ***parameters*** in our SQL commands. To demonstrate this, we will use the previous example, but we will change the code to take advantage of parameters. The initial way to build the SQL command in that example was to use concatenation like this:

```
string search = "SELECT * FROM tblDepartments WHERE ID
= " +
                  txtCode.Text;
```

We will change this to use a parameter instead. First, add the following line to the ***using*** statements at the top of the code:

```
using System.Data;
```

Then change the code in **btnFind_Click** like this:

```
private void btnFind_Click(object sender, EventArgs e)
        {
            try
            {
                txtDepartment.Text = string.Empty;
                // Create Select statement
                string search = "SELECT * FROM tblDepartments 
WHERE ID 
                                 = @deptid";
                // Create SqlCommand
                command = new SqlCommand(search, connection);
                //Add the parameter
                command.Parameters.Add("@deptid", 
SqlDbType.TinyInt);
                //asign value to parameter
                command.Parameters["@deptid"].Value = 
txtCode.Text;
                // Open connection
                connection.Open();
...
```

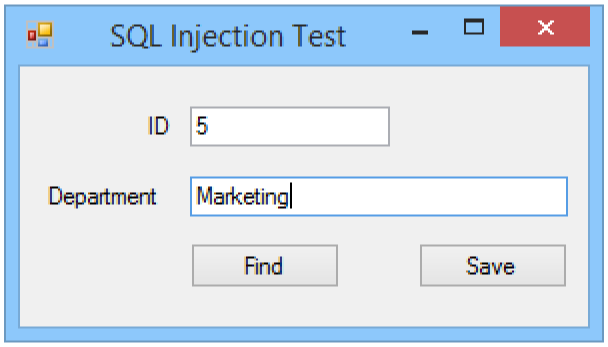
We have now modified the code to use a parameter instead of concatenation. A parameter must begin with the **@** symbol. We first put the parameter where we want it in the SQL statement. Then, using the ***Parameters.add*** method of the **command** object, we declare the parameter. Finally, we put the value in it. In our case, that value comes from a textbox. The rest of the code remains the same.

Execute the application. It should work the same as before. Test the SQL injection string again. The system will display an error message and the table will remain intact.

We will now complete the code of the application by making the ***Save*** button work, thus making it possible to add new departments. In design mode, double click on the ***Save*** button and it should bring you in the **btnSave_Click** method. Add the following code to it:

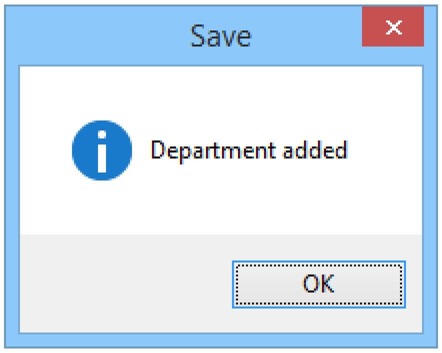
```
private void btnSave_Click(object sender, EventArgs e)
        {
            try
            {
                string line = "INSERT INTO
                tblDepartments(ID,DepartmentName)
                  VALUES(@deptID,@deptName)";
                command = new SqlCommand(line, connection);
                command.Parameters.Add("@deptID",
                  SqlDbType.TinyInt);
                command.Parameters.Add("@deptName",
                  SqlDbType.VarChar);
                command.Parameters["@deptID"].Value =
                  txtCode.Text;
                command.Parameters["@deptName"].Value =
                  txtDepartment.Text;
                connection.Open();
                command.ExecuteNonQuery();
                MessageBox.Show("Department added", "Save",
                    MessageBoxButtons.OK,
                     MessageBoxIcon.Information);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            finally
            {
                connection.Close();
            }
        }
```

Run the application. Try to add a new department with an ID of **5** and the name ***Marketing***:



ID: 5
Department: Marketing

You will get this result:


Save

Department added

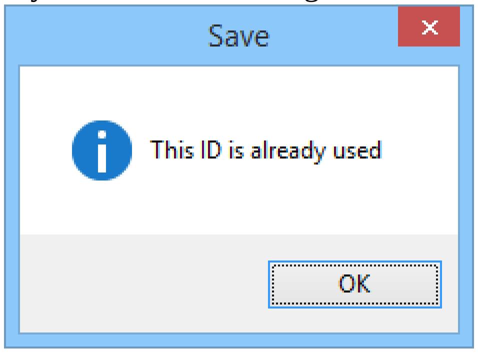
However, if you try to add a new department using an already existing code, you will get a ***Primary key violation*** error message, which is normal. To make this message a little nicer to the common user, we will first search for the ID before we try to use it. Make the following modifications:

```
    private void btnSave_Click(object sender, EventArgs e)
         {
             string search = "SELECT ID FROM tblDepartments 
WHERE
                                  ID=@deptID";
             command = new SqlCommand(search, connection);
             command.Parameters.Add("@deptID",
                   SqlDbType.TinyInt);
             command.Parameters["@deptID"].Value = 
txtCode.Text;
             try
             {
                 connection.Open();
                 var result =command.ExecuteScalar();
                 if (result!=null)
                 {
                     //if the ID is used
                     MessageBox.Show("This ID is already 
used",
                      "Save", MessageBoxButtons.OK,
                      MessageBoxIcon.Information);
                     return;
                 }
                 string line = "INSERT INTO 
                   tblDepartments(ID,DepartmentName) 
                   VALUES(@deptID,@deptName)";
                 command = new SqlCommand(line, connection);
                 command.Parameters.Add("@deptID",
                       SqlDbType.TinyInt);
                 command.Parameters.Add("@deptName",
                       SqlDbType.VarChar);
                 command.Parameters["@deptID"].Value = 
                       txtCode.Text;
```

```
                    command.Parameters["@deptName"].Value =
                           txtDepartment.Text;
                    command.ExecuteNonQuery();
...
```

Notice that we use **ExecuteScalar** instead of **ExecuteReader** in the code above. ***ExecuteScalar*** only returns the first column of the first row of the query so it is perfect to test if something is found or not. If nothing is returned by the SELECT statement, **ExecuteScalar** will return **null**.

Make a test by running the program and trying to add a new department with an ID of 20, which is already in use. You should get the following result:

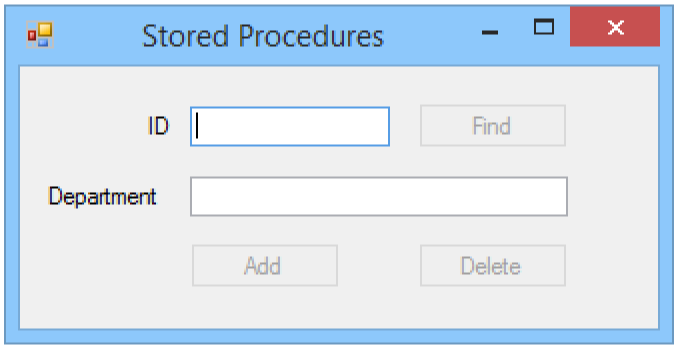


This ID is already used

Make sure the program still works well if you use a valid ID.

Using stored procedures

We will now take a look at stored procedure and how they make database interactions much more efficient. You will need to get the **Stored Procedures** folder from the ***work files***. In it, you will find four **.sql** files that can be executed in **SQL Server** in order to create the stored procedures. Execute them all except for **ModifyAddUpdateDepartment.sql,** we will use this one later. You will also find a Visual Studio project that you should open once your stored procedures are created. Run the application and you should see the following window:



Again, this is a ***Windows Form*** application but the C# code is exactly the same as a ***WPF*** application. You will see that all the buttons are disabled by default. So before we start to use our stored procedures, we'll create four small methods that will help us manage the form. In the Solution Explorer, right click on **Form1.cs** and choose ***View code***. Then, at the bottom of the class, add the following method:

```
private void Reset()
        {
            txtCode.Text = txtDepartment.Text =
string.Empty;
            btnAdd.Text = "Add";
            btnAdd.Enabled = false;
            btnFind.Enabled = false;
            btnDelete.Enabled = false;
            txtCode.ReadOnly = false;
        }
```

Then add this method right under:

```
private string ValidateDeptName(string deptName)
        {
            deptName = deptName.Trim();
            if (deptName!=string.Empty)
            {
                deptName = txtDepartment.Text;
                deptName = deptName[0].ToString().ToUpper()
+
                    deptName.Substring(1).ToLower();
            }
            return deptName;
        }
```

This method is to make sure that the department name starts with a capital letter and isn't surrounded with unwanted spaces. Go back in design mode and double-click on the **txtDepartment** textbox. Then add the following code in the **txtDepartment_TextChanged**:

```
private void txtDepartment_TextChanged(object sender, EventArgs e)
        {
            if (txtCode.Text.Trim()!=string.Empty)
            {
                btnAdd.Enabled = true;
            }
        }
```

And do the same for the **txtCode** textbox:

```
private void txtCode_TextChanged(object sender, EventArgs e)
        {
            btnFind.Enabled = true;
            if (txtDepartment.Text.Trim()!=string.Empty)
            {
                btnAdd.Enabled = true;
            }
        }
```

These two methods will make sure that both textboxes have been filled before enabling the **Add** button.

We will now fix the connection string for the database. Go in the **App.config** file by double-clicking it in the Solution Explorer and change the connection string to this:

```
<add name="constr"
         connectionString="server=.; initial
catalog=TestSQL;
                         integrated security=True"/>
```

Selecting data using stored procedures

Open the file **FindDepartment.sql** in a text editor. You executed this code earlier and it created a stored procedure called ***spFindDepartment***. This stored procedure expects one parameter, the department ID, and then uses a SELECT statement to find the corresponding record in the **tblDepartments** table. The procedure then returns the **DepartmentName** field associate with the ID:

```
     USE TestSQL
     GO
     CREATE PROCEDURE spFindDepartement
          -- Declare parameters
          @IdDept tinyint,
          @Dept nvarchar(50) OUT
     AS
     BEGIN
          SET NOCOUNT ON;
          SET @Dept = (SELECT DepartmentName FROM
                       tblDepartments WHERE ID =
     @IdDept)
     END
```

Make sure the stored procedure exists in your database and go back to the Visual Studio project. In the design mode of the form, double-click on the **Find** button and enter the following code:

```
private void btnFind_Click(object sender, EventArgs e)
        {
            uint searchCode;
            bool ok = uint.TryParse(txtCode.Text,
                                    out searchCode);
            if (ok)
            {
                try
                {
                    connection.Open();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
                finally
                {
                    connection.Close();
                }
            }
            else
            {
                MessageBox.Show("The ID must be an integer
                        greater than zero.", "Error",
                    MessageBoxButtons.OK,
MessageBoxIcon.Error);
                Reset();
                txtCode.Focus();
            }
        }
```

This code establishes the connection and makes sure the value entered is a positive integer. We used the **uint** data type here because we want to store a value that is greater than zero. The **u** in ***uint*** means "***unsigned***" so there is no place to store a positive or negative sign. The value itself can be between 0 and 4,294,967,295. You can already test this part if you want. Run the app and try to find a non integer value, you should get an error message. Now let's add the code for the stored procedure itself:

```
...
try
                {
                    connection.Open();
                    //Creates SqlCommand object
                    SqlCommand command = new

SqlCommand("spFindDepartment",connection);
                    //Specify that the command is a stored
                    // procedure
                    command.CommandType =
                            CommandType.StoredProcedure;
                    //Add parameters

command.Parameters.AddWithValue("@IdDept",
                        txtCode.Text);
                    SqlParameter ret= new
SqlParameter("Dept",
                        SqlDbType.NVarChar, 50);
                    ret.Direction =
ParameterDirection.Output;
                    command.Parameters.Add(ret);
                    command.ExecuteNonQuery();
                    string result = ret.Value.ToString();
                    txtDepartment.Text = result;
                }
                catch (Exception ex)
...
```

Let's examine that last bit of code. We first need to create a **SqlCommand** object to call the stored procedure. We also have to specify that this command will be of

stored procedure type. We than add the parameters. Regular parameters are easy to add, but the **OUT** parameter takes a little more effort because we have to specify that it is an ***OUT*** parameter by using the **Direction** property. Finally, we call the **ExecuteNonQuery** method to make the actual call to the stored procedure. The nice thing about stored procedures is that it lets the database server do all the heavy lifting so the transaction is much more efficient. Test the app to make sure it works. You should have no problem finding departments that exist.

We will now add a little more code to manage the case when no department is found. We will also prepare the form to update or delete a record once it is found. Make the following modifications:

```
...
                    command.ExecuteNonQuery();
                    string result = ret.Value.ToString();
                    if (result!=string.Empty)
                    {
                        txtDepartment.Text = result;
                        txtCode.ReadOnly = true;
                        btnFind.Enabled = false;
                        btnDelete.Enabled = true;
                        btnAdd.Text = "Update";
                        btnAdd.Enabled = true;
                    }
                    else
                    {
                        MessageBox.Show("No record has been
                         found.", "Error",
MessageBoxButtons.OK,
                          MessageBoxIcon.Error);
                        Reset();
                        txtCode.Focus();
                    }
                }
                catch (Exception ex)
...
```

You can now test the ***Find*** functionality, everything should work fine. Don't forget to try invalid values such as letters and negative numbers.

Writing records with stored procedures

It is now time to use our other stored procedures. If you open the file **AddUpdateDepartment.sql** with the editor of your choice, you should see the code to create a stored procedure that will insert a department in the **tblDepartments** table:

```
     USE TestSQL
     GO

     CREATE PROCEDURE [dbo].[spAddUpdateDepartment]
          -- Declare parameters
          @Mode nvarchar(10),
          @IdDept tinyint,
          @Dept nvarchar(50),
          @Return int OUT
     AS
     BEGIN
          SET NOCOUNT ON;
          if @Mode = 'Add'
          BEGIN
               if EXISTS (SELECT * FROM tblDepartments
     WHERE
                       ID = @IdDept)
               BEGIN
                    -- If code already exists
                    SET @Return = 0
               END
               ELSE IF EXISTS (SELECT * FROM
     tblDepartments
                      WHERE DepartmentName = @Dept)
               BEGIN
                    --If dept name already exists
                    SET @Return = 1
               END
               ELSE
               BEGIN
                    INSERT INTO tblDepartments
                           VALUES(@IdDept, @Dept)
                    --If everything is ok
                    SET @Return = 2
               END
          END
END
```

If you inspect the code, you will see that the procedure first checks the table to make sure that neither the ID nor the name are already used. If the ID is used, the procedure returns 0. If the name is used, it returns 1. Finally, if everything is ok, the INSERT statement is executed and the procedure returns 2. You will also note that there is a ***@Mode*** parameter used at the beginning of the code. Do not worry about this parameter yet, we will explain it later. You have executed this code at the beginning of the lesson so the stored procedure should already exist in the server. Let's get back to our Visual Studio project to use it.

In design mode of the form, double-click on the **Add** button to generate the **btnAdd_Click** method. Then enter the following code:

```
private void btnAdd_Click(object sender, EventArgs e)
        {
            uint searchCode;
            string deptName = string.Empty;
            bool ok = uint.TryParse(txtCode.Text, out
                                           searchCode);
            if (ok)
            {
                deptName =
ValidateDeptName(txtDepartment.Text);
                if (deptName!=string.Empty)
                {

                }
                else
                {
                    MessageBox.Show("You must enter a
department
                        name.", "Error",
MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
                    txtDepartment.Text = string.Empty;
                    //txtDepartment.Focus();
                    btnAdd.Enabled = false;
                }
            }
            else
            {
                MessageBox.Show("The ID must be an integer
                               greater than zero.",
"Error",
                    MessageBoxButtons.OK,
MessageBoxIcon.Error);
                Reset();
                txtCode.Focus();
            }

        }
```

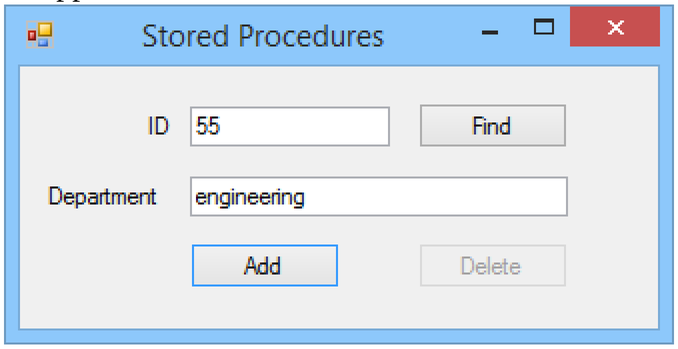
This code does all the usual validation to make sure the user input is correct. It also uses the **ValidateDeptName** method that we created earlier to format the name with the first letter in uppercase. We want this button to serve two functions: adding and updating. Right now, we are in "***add***" mode but we must keep in mind that the same button will be used to update records later. Because of the two modes, we will create two methods that will take care of the two different functionalities. Let's start by creating the **AddDepartment** method. Go at the end of the class and add the following method:

```
  private void AddDepartment(string deptName)
        {
            try
            {
                connection.Open();
                SqlCommand command = new
                SqlCommand("spAddUpdateDepartment",
connection);
                command.CommandType =
                   CommandType.StoredProcedure;
                command.Parameters.AddWithValue("@Mode",
"Add");
                command.Parameters.AddWithValue("@IdDept",
                   txtCode.Text);
                command.Parameters.AddWithValue("@Dept",
                   deptName);
                //OUT parameter
                SqlParameter ret = new
SqlParameter("@Return",
                   SqlDbType.Int);
                ret.Direction = ParameterDirection.Output;
                command.Parameters.Add(ret);
                command.ExecuteNonQuery();
                int result = (int)ret.Value;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            finally
            {
                connection.Close();
                }
        }
```

As you can see, calling a stored procedure that inserts a record is pretty much the same as calling a stored procedure that does a SELECT statement. Still, in the previous code, we can see three different ways to pass a parameter. The **@Mode** parameter is given the "***Add***" string value because we want to add a row and not update one. The **@IdDept** parameter takes its value from a textbox and the **@Dept** parameter takes its value from the **deptName** variable. Now let's test this code right away. To do so, we must add a call to this **AddDepartment** method in the click button event. Go back to the **btnAdd_Click** method and add the following code:

```
...
deptName = ValidateDeptName(txtDepartment.Text);
                if (deptName!=string.Empty)
                {
                        AddDepartment(deptName);
                }
...
```

Now we are ready to test. Run the application and type **55** in the **ID** TextBox. Then type ***engineering*** in the **Department** TextBox. Click the **Add** button, wait a second and then check the **tblDepartments** in SQL Server. You should see the new record added in the table, and you should even see that the first letter of ***engineering*** is in uppercase!



It works, but there is no message given to the user to confirm the action. Let's fix this by adding the following code in the **AddDepartment** method:

```
...
                    command.ExecuteNonQuery();
```

```
                int result = (int)ret.Value;
                //Display appropriate message
                switch(result)
                {
                    case 0:
                        MessageBox.Show("This ID is already 
used.",
                          "Error", MessageBoxButtons.OK,
                           MessageBoxIcon.Error);
                        txtCode.Text = string.Empty;
                        txtCode.Focus();
                        btnAdd.Enabled = false;
                        break;
                    case 1:
                        MessageBox.Show("This department name is
                          already used.", "Error",
                           MessageBoxButtons.OK, 
MessageBoxIcon.Error);
                        txtCode.Text = string.Empty;
                        txtCode.Focus();
                        btnAdd.Enabled = false;
                        break;
                    case 2:
                        MessageBox.Show("The department has been
                          added.", "Save", MessageBoxButtons.OK,
                           MessageBoxIcon.Information);
                        Reset();
                        break;
                }
            }
...
```

Test everything again and make sure you get the appropriate message. Add departments with duplicate IDs or names. Add valid departments with mixed up lowercase and uppercase letters and try to find them using the **Find** button. Everything should work well. Keep your IDs at 255 or lower because the ID data type (smallint) cannot hold numbers greater than 255.

We will now change the code a bit so that our **Add** button can become an **Update** button when the situation demands it. The idea is that once a department has been found using the **Find** button, we should be able to change the name of that department. This action is called ***update***, not ***add***. We will therefore modify our **AddUpdateDepartment** stored procedure so it can handle both actions.
Open the **ModifyAddUpdate.sql** file with the editor of your choice. You should see the following code:

```
      USE [TestSQL]
```

```
GO

ALTER PROCEDURE [dbo].[spAddUpdateDepartment]
     -- Declare parameters
     @Mode nvarchar(10),
     @IdDept tinyint,
     @Dept nvarchar(50),
     @Return int OUT
AS
BEGIN
     SET NOCOUNT ON;
     if @Mode = 'Add'
     BEGIN
          if EXISTS (SELECT * FROM tblDepartments
WHERE
                 ID = @IdDept)
          BEGIN
               --If code already exists
               SET @Return = 0
          END
          ELSE IF EXISTS (SELECT * FROM
tblDepartments
                     WHERE DepartmentName = @Dept)
          BEGIN
               --If dept name already exists
               SET @Return = 1
          END
          ELSE
          BEGIN
               INSERT INTO tblDepartments
                  VALUES(@IdDept, @Dept)
               --If everything is ok
               SET @Return = 2
          END
     END

     ELSE IF @Mode = 'Update'
     BEGIN
          UPDATE tblDepartments SET DepartmentName =
                    @Dept WHERE ID = @IdDept
          SET @Return = 2
     END
END

GO
```

This big chunk of code alters the already existing **AddUpdateDepartment** stored procedure by adding an ***UPDATE*** functionality. This is where the **@Mode** parameter becomes important. If the stored procedure receives a value of "***Add***" for **@Mode**, the INSERT part is executed. If the parameter's value is "***Update***", then the UPDATE part is executed. In both cases, the value **2** is returned if the action was successful. Run this code in SQL Server to alter the existing stored procedure and then return to the Visual Studio project.

We could now create an UpdateDepartment method in the class, but since most of it will be the same as the already existing **AddDepartment** method, we'll just modify this one instead. Go in the **AddDepartment** method and make the following changes:

```
...
command.Parameters.AddWithValue("@Mode", btnAdd.Text);
...

...
  case 2:
             if (btnAdd.Text=="Add")
             {
                   MessageBox.Show("The department has been
                     added.", "Save", MessageBoxButtons.OK,
                     MessageBoxIcon.Information);
             }
             else
             {
                   MessageBox.Show("The department has been
                     updated.", "Save",
MessageBoxButtons.OK,
                     MessageBoxIcon.Information);
             }

             Reset();
             break;
}
...
```

You can now test the ***update*** functionality. Run the application and find a department that exists. Then, change the name and click **Update**. Find the department again to make sure that the modifications were kept. You can also check the database table to confirm that the changes were made.

Challenge!

You will notice that, when you make an update, it is possible to change the name of the department to one that already exists. This problem was solved in the case of an ***Add***, but for an ***Update***. Try to fix that problem.

Deleting a record using a stored procedure

The deletion of a record is the simplest operation of them all. It is also the most dangerous since a deleted record is lost forever. Still, we will implement the delete functionality by using the **spDeleteDepartment** stored procedure that was created at the beginning of the lesson. Take a look at the **DeleteDepartment.sql** file with the editor of your choice:

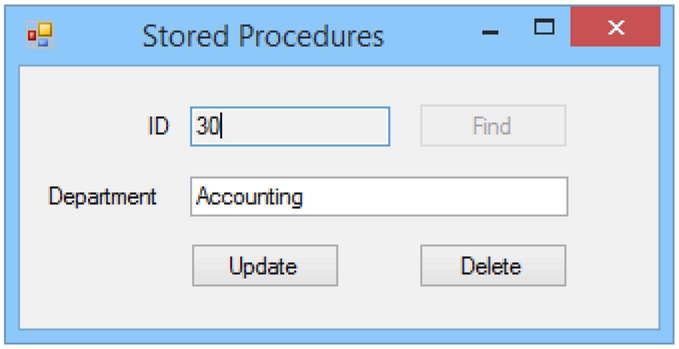
```
     USE TestSQL
     GO

     CREATE PROCEDURE spDeleteDepartment
          -- Declare parameters
          @IdDept tinyint
     AS
     BEGIN
          SET NOCOUNT ON;
          DELETE FROM tblDepartments WHERE ID = @IdDept
     END
```

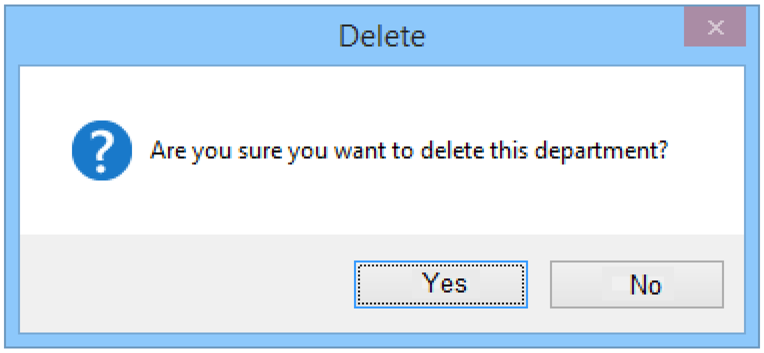
As you can see, the code is very simple. Only one parameter is required which is the department ID. Make sure the stored procedure exists in the database and return to the Visual Studio project.

In design mode, double-click the **Delete** button to generate the **btnDelete_Click** method. Then add the following code:

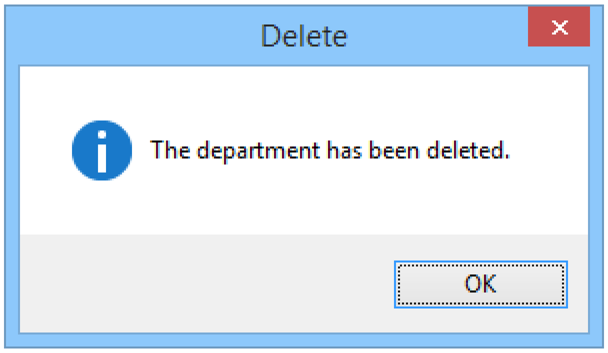
```
private void btnDelete_Click(object sender, EventArgs e)
        {
            if(MessageBox.Show("Are you sure you want to
delete
            this department?","Delete",
MessageBoxButtons.YesNo,
             MessageBoxIcon.Question)==DialogResult.Yes)
            {
                try
                {
                    connection.Open();
                    SqlCommand command = new
                   SqlCommand("spDeleteDepartment",
connection);
                    command.CommandType =
                       CommandType.StoredProcedure;

command.Parameters.AddWithValue("@IdDept",
                       txtCode.Text);
                    command.ExecuteNonQuery();
                    MessageBox.Show("The department has
been
                                       deleted.", "Delete",
                         MessageBoxButtons.OK,
                          MessageBoxIcon.Information);
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message);
                }
                finally
                {
                    connection.Close();
                }
            }
            Reset();
        }
```

By now, you should be able to understand everything that's going on in this code. Notice that the method starts with an ***if*** statement that makes sure the user is aware that he is deleting a record. After that, it's pretty much the same as the other stored procedure calls. Let's test the application to make sure it works. First, find department **30** which is ***Accounting***. Click the **Delete** button. Then, try to find department **30** again. It should not be there anymore.

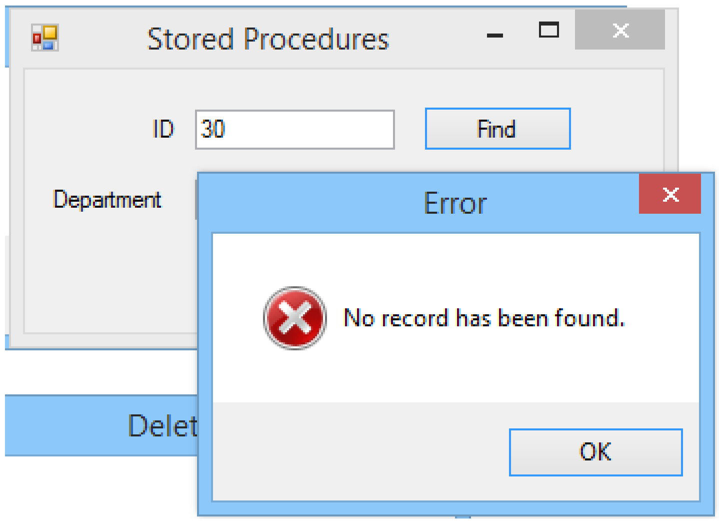




Are you sure you want to delete this department?



The department has been deleted.



No record has been found.

Hopefully you now understand how to call stored procedures from a C# application.

Progress check

Check your answer in Appendix A

Read carefully the statements and circle the letter that corresponds to the right answer.

1. What is SQL code injection?

- **a**. A technique used by programmers to generate SQL code
- **b**. A technique used by malicious individuals to access or harm a system
- **c.** Code generated by the Microsoft Framework
- **d.** None of the above

2. How can we add a parameter to a command?

- **a**. SqlCommand.Add()
- **b**. SqlConnection.Add()
- **c.** SqlCommand.Parameters.Add()
- **d.** SqlCommand.Parameters()

3. What property must be used to assign a value to a command's parameter?

- **a**. Value
- **b**. ExecuteScalar
- **c**. ExecuteNonQuery
- **d**. Add

4. True or false: a stored procedure doesn't have to exist in SQL Server before we can call it from C#. It will be automatically created.

- **a**. True
- **b**. False

5. Which SqlCommand property specifies that a command is a stored procedure?

- **a**. Command
- **b**. Procedure
- **c**. AddWithValue
- **d**. CommandType

Module 4

ADO.NET in disconnected mode

This lesson will examine the **System.Data** library which is part of the **.NET Framework**. More specifically, **System.Data** contains the classes that represent the **ADO.NET** architecture, the center piece of which being the **DataSet** class.

OBJECTIVES

- Use ADO.NET in disconnected mode

Up until now, we have worked with the database in ***connected*** mode. We are now going to see how we can interact with the database in ***disconnected*** mode. By "***disconnected"***, we mean that the data will first be brought from the database to the computer's memory. Then, all the manipulations (SELECT, INSERT, DELETE, UPDATE) will be made on that copied version of the data and not directly in the database. When we are done with the data, the modifications are all recopied back to the database. So the database is only solicited twice: when we first get the data and at the end when we copy it back.

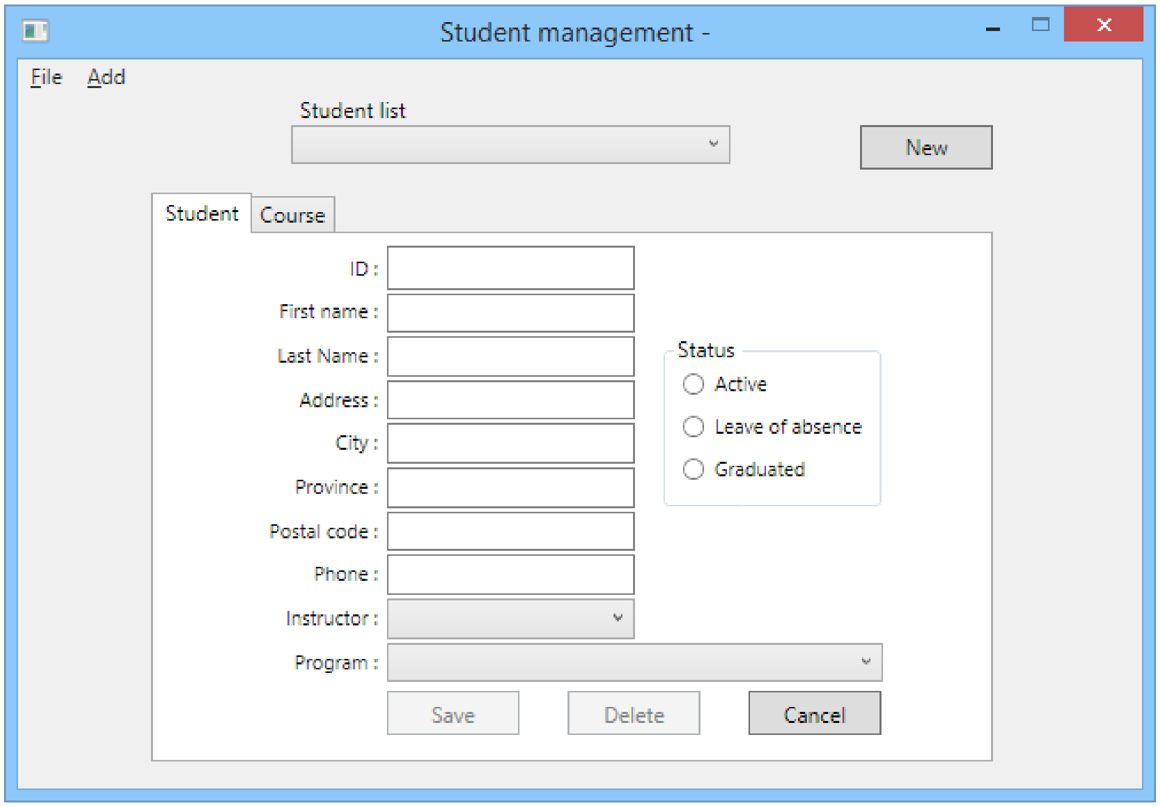
In ***disconnected*** mode, the following structure will be used:

- A **DataSet** is like a database, but resides in the computer's memory. It can contain many tables (called ***DataTables***).
- A **DataTable** is like a database table and can hold many rows and columns. Like a table, it can have a primary key constraint to help keep the **DataSet's** referential integrity.
- A **DataColumn** is like a table's column and has a name and a data type. It can also be used as a primary key or a foreign key in the ***Relations*** collection of the **DataSet**.
- A **DataRow** is like a row or a record. It's the equivalent of a line in a database table.

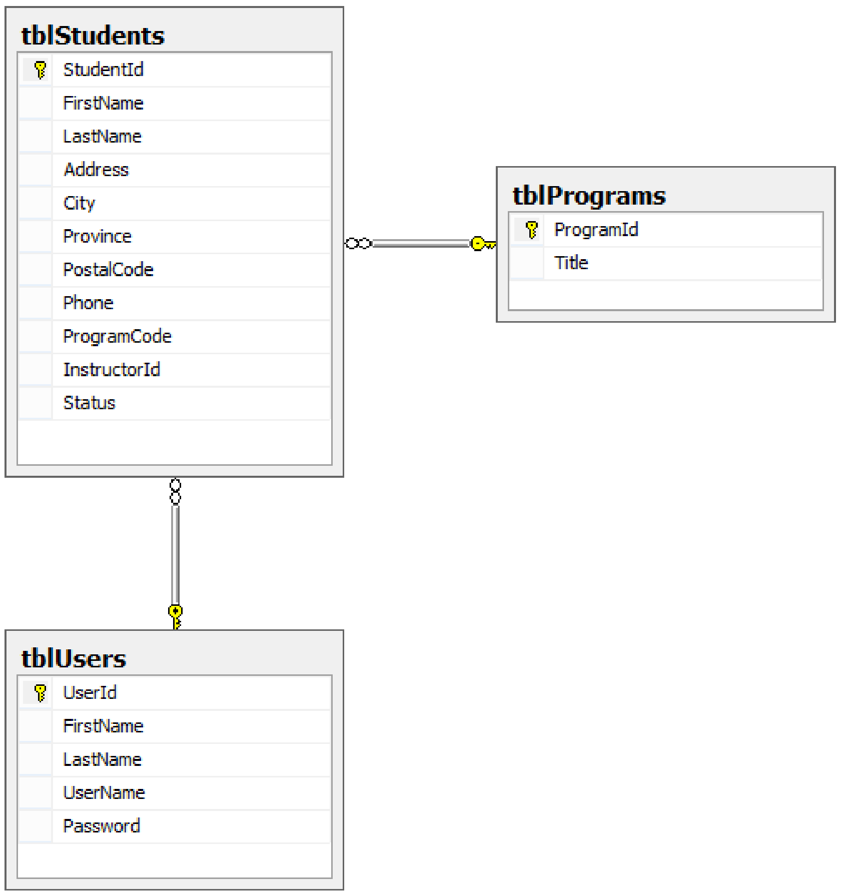
When we want to work in disconnected mode, the following steps are taking place:

- Creation of the **DataSet** object and at least one **DataTable**.
- The **DataRows** objects are generated and put in the ***DataTable's*** **Rows** collection.
- The **DataColumns** objects are generated in the ***DataTable's*** **Columns** collection.
- The ***DataTable*** is then added to the ***DataSet's*** **Tables** collection.
- If multiple tables exist and they have relations, we can apply these relations to the **DataTables** by adding **DataRelations** objects in the **Relations** collection of the **DataSet**.

We will now see how to manipulate data in a **DataSet**. To do so, we will use the **College** folder from the ***work files*** that contains a Visual Studio project and some SQL files. Open the **AppCollege** project in Visual Studio, run it and look at the main form:



This is the **WPF** form we will use to work with DataSets. We must now create the **College** database for our application to work. Go in SQL Server and delete the **College** database if it already exists. Then, execute the content of **CreateTablesCollege.sql** to create the database. This database should contain three tables: tblStudents, tblPrograms and tblUsers. The following diagram shows the relations between them:



Go back to the Visual Studio project. We will first adjust the connection string to the database. Open the **App.config** file and make the following change:

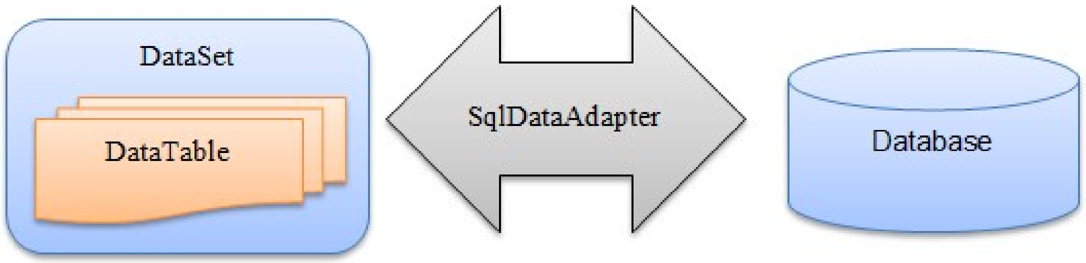
```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="constr"
         connectionString="server=.; initial catalog=College;
integrated
                                       security=True;"/>
  </connectionStrings>
  <startup>
      <supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.5.2" />
  </startup>
</configuration>
```

Now open the **frmMain.xaml.cs** file and make the following changes:

```
public partial class MainWindow : Window
     {
         SqlConnection connection;
```

```
        public MainWindow()
        {
            InitializeComponent();
            connection = new
SqlConnection(ConfigurationManager.

ConnectionStrings["constr"].ConnectionString);

        }
...
```

Up to that point, we have done the same thing has in the previous exercises. We created a connection and made sure that it points to the correct database on the correct server. Now things are going to change a bit. Keep in mind that we are going to need a **DataSet** to hold the data in the computer's memory; we do not want to work directly in the database as we did before. To do so, we are going to need a **SqlDataAdapter** that will act as a bridge between the physical database and the **DataSet** in our application's memory:



Indeed, it is the **SqlDataAdapter** that moves the data back and forth between the database and the DataSet. The nice thing about this is that a **DataAdapter** can be used to bridge a **DataSet** with other kinds of sources, not just a SQL Server database.

So let's proceed with the exercise. Before we do anything else, we will create three ***stored procedures*** in SQL Server to help us retrieve information. In the ***work files***, get the **spSelectStudents.sql**, s**pSelectPrograms.sql** and **spSelectUsers.sql** files and execute their content in SQL Server to create the stored procedures. Once you are done, go back to the Visual Studio project and open the **frmMain.xaml.cs** file. Add the following code:

```
public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter da;
        DataSet dsCollege = new DataSet();
```

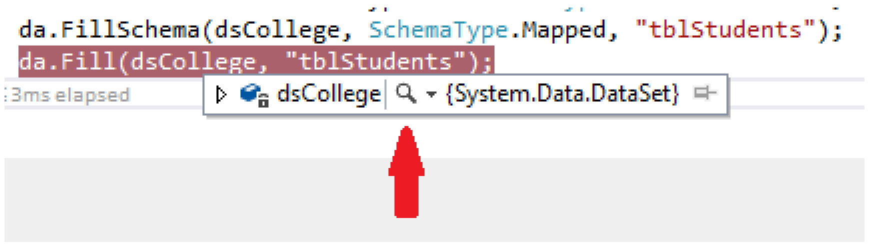
```
...

private void frmMain1_Loaded(object sender, RoutedEventArgs e)
        {
            da = new SqlDataAdapter("sp_SelectPrograms", connection);
            da.SelectCommand.CommandType = CommandType.StoredProcedure;
            da.FillSchema(dsCollege, SchemaType.Mapped, "tblPrograms");
            da.Fill(dsCollege, "tblPrograms");
        }
...
```

This code first creates the **SqlDataAdapter** and **DataSet** objects. Then, in the ***frmMain1_Loaded*** method, it connects the **DataAdapter** to the database by using the **spSelectPrograms** stored procedure. After that, with the help of the **FillSchema** method, the structure of the table **tblPrograms** is taken from the database and used to create a **DataTable** of the same name in the **DataSet**. This also includes the **primary key**. Finally, the **tblPrograms** DataTable is filled with the data that comes from the stored procedure. (NB: The ***frmMain1_Loaded*** method is automatically executed when the main form is loaded on the screen)

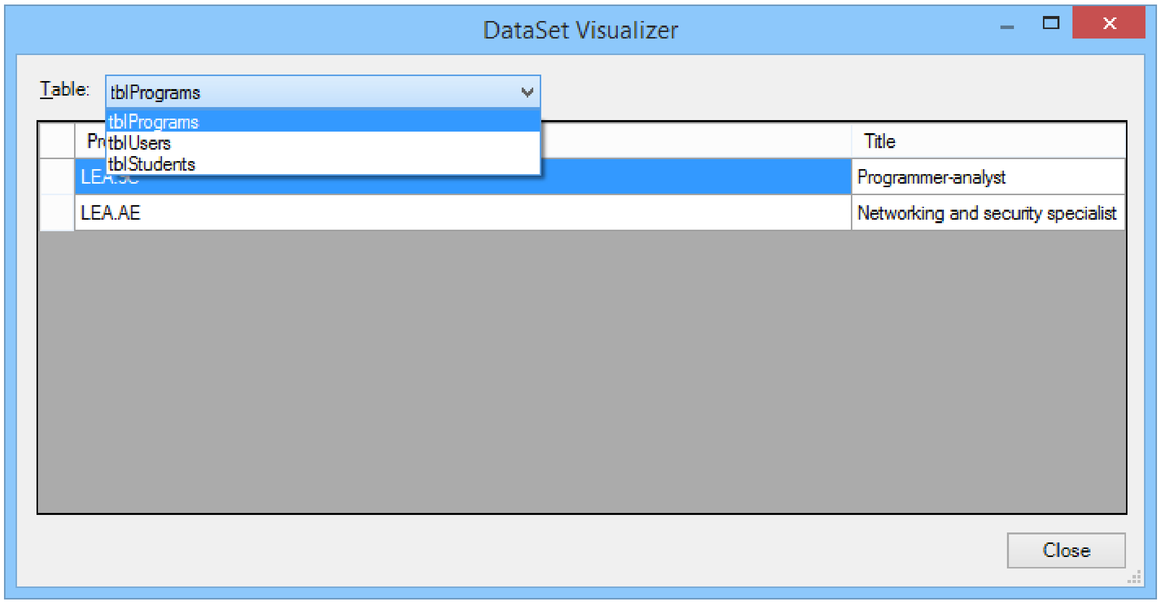
We will now do the same thing to generate the two other ***DataTables***: **tblStudents** and **tblUsers**:

```
private void frmMain1_Loaded(object sender, RoutedEventArgs e)
        {
            da = new SqlDataAdapter("sp_SelectPrograms", connection);
            da.SelectCommand.CommandType = CommandType.StoredProcedure;
            da.FillSchema(dsCollege, SchemaType.Mapped, "tblPrograms");
            da.Fill(dsCollege, "tblPrograms");

            da = new SqlDataAdapter("sp_SelectUsers", connection);
            da.SelectCommand.CommandType = CommandType.StoredProcedure;
            da.FillSchema(dsCollege, SchemaType.Mapped, "tblUsers");
            da.Fill(dsCollege, "tblUsers");

            da = new SqlDataAdapter("sp_SelectStudents", connection);
            da.SelectCommand.CommandType = CommandType.StoredProcedure;
            da.FillSchema(dsCollege, SchemaType.Mapped, "tblStudents");
            da.Fill(dsCollege, "tblStudents");
        }
```

To make sure that the **DataSet** and its **DataTables** have been generated properly, we will use the debugger and examine the result of our code. Put a *breakpoint* on the last line of the code you just added and run the application. When the program reaches the breakpoint, press **F10** to make a step. Then, put the mouse pointer on the "***dsCollege***" word and click the magnifying glass:


```
da.FillSchema(dsCollege, SchemaType.Mapped, "tblStudents");
da.Fill(dsCollege, "tblStudents");
```

You should then get the following window that shows you the entire content of the **DataSet**. You can select a table from the list on top to see its content:



Pr	Title
LEA.3C	Programmer-analyst
LEA.AE	Networking and security specialist

Examine the content of the three DataTables and make sure they match the content of the actual database.

We will now bind the content of these **DataTables** to the form's **ComboBoxes**. Add the following code at the end of the **frmMain1_Loaded** method:

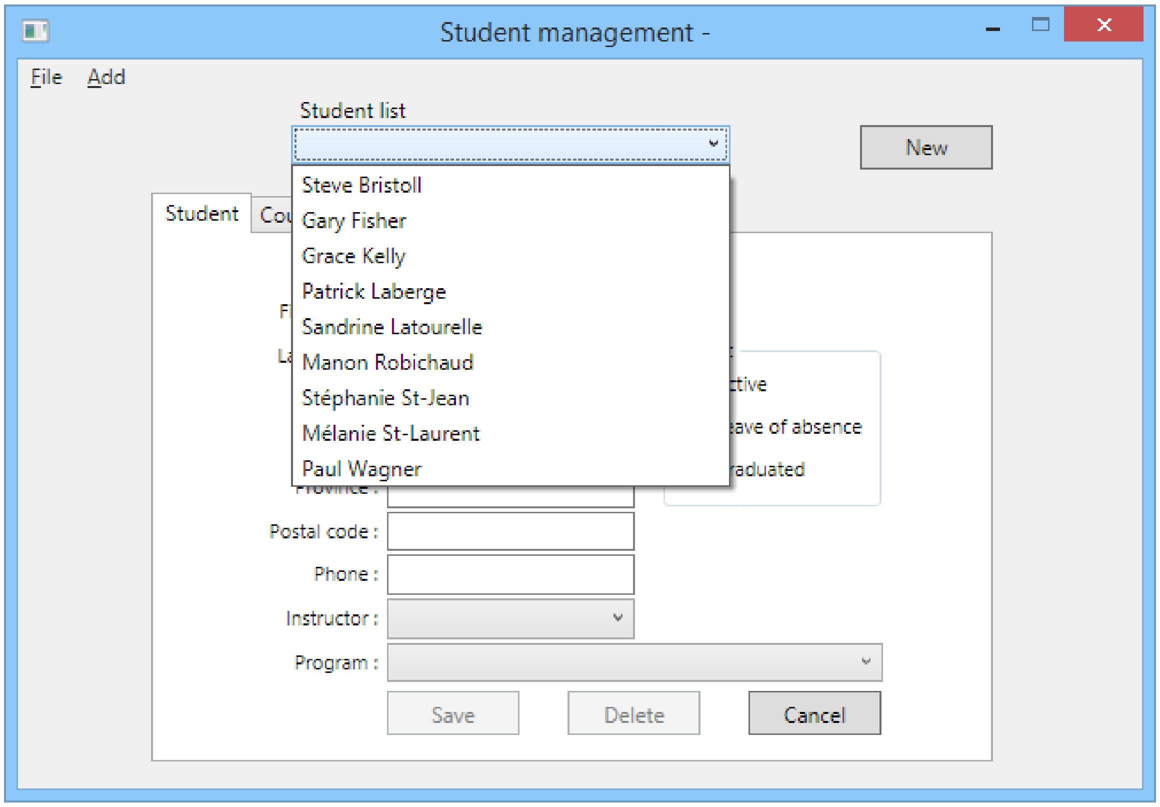
```
...

da = new SqlDataAdapter("sp_SelectStudents", connection);
            da.SelectCommand.CommandType = CommandType.StoredProcedure;
            da.FillSchema(dsCollege, SchemaType.Mapped, "tblStudents");
            da.Fill(dsCollege, "tblStudents");
            StudentList.ItemsSource =
                  dsCollege.Tables["tblStudents"].DefaultView;
            InstructorList.ItemsSource =
                  dsCollege.Tables["tblUsers"].DefaultView;
            ProgramList.ItemsSource =
                  dsCollege.Tables["tblPrograms"].DefaultView;

        }
```

Now that the data source for the comboboxes has been established in the C# code, we must also do the binding in the **XAML**. Go in the **frmMain.xaml** file and modify the three comboboxes like so (you can click on the ComboBox in design view to position the cursor on the correct XAML tag):

```
...
<ComboBox x:Name="StudentList" HorizontalAlignment="Left"
                  DisplayMemberPath="FullName"
SelectedValuePath="StudentId" Margin="154,37,0,0"
VerticalAlignment="Top" Width="248" Height="22" ></ComboBox>
  ...
<ComboBox x:Name="InstructorList" DisplayMemberPath="FullName"
SelectedValuePath="UserId" Grid.Column="2" Grid.Row="9" FontSize="11"
BorderBrush="Gray" BorderThickness="1" Margin="0,2,0,0"/>
...
<ComboBox x:Name="ProgramList" DisplayMemberPath="Title"
SelectedValuePath="ProgramId" Grid.Column="2" Grid.Row="10"
Grid.ColumnSpan="2" FontSize="11" BorderBrush="Gray"
BorderThickness="1" Margin="0,2,0,0"/>
...
```

DisplayMemberPath represents the field that will be shown in the ComboBox and ***SelectedValuePath*** represents what value the ComboBox will give us when we select an item.

You can now test the application. Execute the program and click on the three comboboxes to make sure they are correctly filled:



Steve Bristoll
Gary Fisher
Grace Kelly
Patrick Laberge
Sandrine Latourelle
Manon Robichaud
Stéphanie St-Jean
Mélanie St-Laurent
Paul Wagner

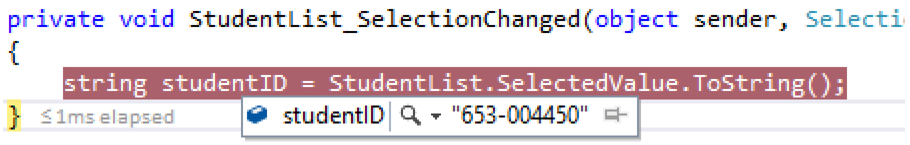
We can now proceed to the next step which is to fill the student form when we select a student in the list. First, we must update the XAML code for the **StudentList** ComboBox:

```
...
<ComboBox x:Name="StudentList" HorizontalAlignment="Left"
                  DisplayMemberPath="FullName"
SelectedValuePath="StudentId"
SelectionChanged="StudentList_SelectionChanged" Margin="154,37,0,0"
VerticalAlignment="Top" Width="248" Height="22" ></ComboBox>
...
```

Then, go back in the **frmMain.xaml.cs** and add the **StudentList_SelectionChanged** method at the end of the class: (you could also make Visual Studio generate some of this!)

```
private void StudentList_SelectionChanged(object sender,
                                    SelectionChangedEventArgs
e)
        {
            string studentID =
StudentList.SelectedValue.ToString();
        }
```

This method will be triggered when the user selects a student in the combobox. The selected value will then be put in the **studentID** variable. Let's make sure this code works by inserting a *breakpoint* on the line and running the app. Then, select a student in the list. This should bring you on the breakpoint. Make a step (**F10**) and put your pointer on the **studentID** variable. You should see the student ID being displayed:

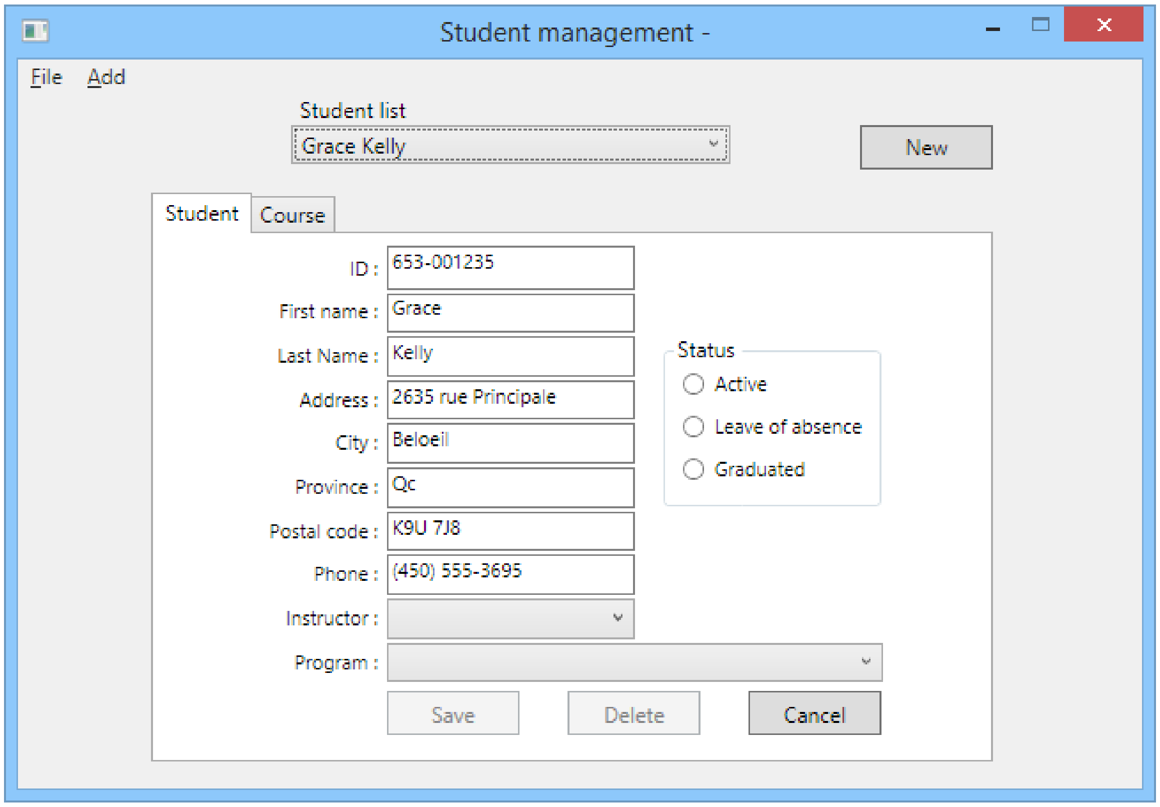


```
private void StudentList_SelectionChanged(object sender, Selecti
{
    string studentID = StudentList.SelectedValue.ToString();
}  ≤1ms elapsed        studentID  "653-004450"
```

Now that we know that the code works, we can find the appropriate record in the **DataSet** using the **Find** method of the **Rows** collection:

```
  public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter da;
        DataSet dsCollege = new DataSet();
        DataRow record;
...


  private void StudentList_SelectionChanged(object sender,

SelectionChangedEventArgs e)
        {
            if (StudentList.SelectedIndex == -1)
            {
                return;
            }
            string studentID =
StudentList.SelectedValue.ToString();
            //Get the student row
            record =

dsCollege.Tables["tblStudents"].Rows.Find(studentID);

            //Fill the form
            txtID.Text = record["StudentId"].ToString();
            txtFirstName.Text = record["FirstName"].ToString();
            txtLastName.Text = record["LastName"].ToString();
            txtAddress.Text = record["Address"].ToString();
            txtCity.Text = record["City"].ToString();
            txtProvince.Text = record["Province"].ToString();
            txtPostalCode.Text =
record["PostalCode"].ToString();
            txtPhone.Text = record["Phone"].ToString();
        }
```

First of all, notice that we have added an ***if*** statement that ends the method if the selected index of the combobox is -1. When **SelectedIndex** has a value of -1, it means that nothing is selected in the combobox. If that's the case, this method should be aborted or else the program will try to get an empty value from the combobox which will lead to errors. So once the **Find** method has done its job of locating the correct record in the **DataTable**, we use that record to fill all the TextBoxes on the form. Execute the program and select a student in the list. You should see its data fill the form:



Grace Kelly

ID : 653-001235
First name : Grace
Last Name : Kelly
Address : 2635 rue Principale
City : Beloeil
Province : Qc
Postal code : K9U 7J8
Phone : (450) 555-3695
Instructor :
Program :

Active
Leave of absence
Graduated

We will now try to make the **Instructor** and **Program** comboboxes respond to a student selection. There are two ways to make a selection in a combobox by code. The first is to set its **SelectedIndex** property to a value starting at zero. The second is to set the **SelectedValue** property to something that exists in the combobox. In our case, the values associated with our comboboxes are the instructor ID and the program code. We will use those to make the comboboxes selections. Add the following lines:

```
...
            txtPostalCode.Text = 
record["PostalCode"].ToString();
            txtPhone.Text = record["Phone"].ToString();

            //Comboboxes selections
            InstructorList.SelectedValue = 
record["InstructorId"];
            ProgramList.SelectedValue = record["ProgramCode"];
        }
```

Execute the app and make a selection in the student list. You should see that the ***Instructor*** and ***program*** comboboxes are now responsive.

All there is now left to fix in the form are the radio buttons that indicate the student status. In the **tblStudents** table, the status field can have 4 values: -1 means ***no status***, 0 means ***active***, 1 means ***on leave of absence*** and 2 means ***graduated***. This system makes it easy to activate the correct radio button using a ***switch case*** structure. Add the following lines at the top of the class and at the end of the **StudentList_SelectionChanged** method:

```
public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter da;
        DataSet dsCollege = new DataSet();
        DataRow record;
        int status = -1;

...
            ProgramList.SelectedValue = record["ProgramCode"];
            //RadioButton selection
            switch (Convert.ToInt16(record["Status"]))
            {
                case -1:
                    rbActive.IsChecked = false;
                    rbLeave.IsChecked = false;
                    rbGraduated.IsChecked = false;
                    break;
                case 0:
                    rbActive.IsChecked = true;
                    break;
                case 1:
                    rbLeave.IsChecked = true;
                    break;
                case 2:
                      rbGraduated.IsChecked = true;
                      break;
             }
        }
```

This should make the radio buttons responsive. Run the application and select students to make sure.

Adding a student to the DataSet

You may have noticed a ***New*** button on the right side of the form. This button will be used to prepare the form so that a user can create a new student. This means that all the textboxes must be emptied, the comboboxes and radiobuttons deselected, and the **ID** textbox must become editable. Using the property window in design mode of the form, generate the ***click*** event method for the **New** button (use the *lightning* symbol!). Then, add the following code:

```
public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter da;
        DataSet dsCollege = new DataSet();
        DataRow record;
        int status = -1;
        bool newStudent=false;
...

private void btnNew_Click(object sender, RoutedEventArgs e)
        {
            StudentList.SelectedIndex = -1;
            StudentList.IsEnabled = false;
            txtID.Text = txtFirstName.Text = "";
            txtLastName.Text = txtAddress.Text = "";
            txtCity.Text = txtProvince.Text = "";
            txtPostalCode.Text = txtPhone.Text = "";
            txtID.IsReadOnly = false;
            InstructorList.SelectedIndex = -1;
            ProgramList.SelectedIndex = -1;
            rbActive.IsChecked = rbGraduated.IsChecked =
rbLeave.IsChecked =
                                false;
            btnNew.IsEnabled = false;
            btnSave.IsEnabled = false;
            newStudent = true;
        }
```

Notice the **newStudent** variable at the top of the class that will be useful later. Test the program by selecting a student and then clicking on the **New** button. Everything is cleared, but then we are not able to select a new student. This is normal because we are now in ***new student*** mode. Let's make the **Cancel** button work to make it possible to exit the ***new student*** mode. In design mode, select the **Cancel** button and generate its ***click*** event method by using the property window. Then add the following code:

```
private void btnCancel_Click(object sender, RoutedEventArgs e)
        {
            Reset();
        }
```

Of course, we now need to create the **Reset** method. Go at the end of the class and add the following code:

```
private void Reset()
        {
            StudentList.SelectedIndex = -1;
            StudentList.IsEnabled = true;
            txtID.Text = txtFirstName.Text = "";
            txtLastName.Text = txtAddress.Text = "";
            txtCity.Text = txtProvince.Text = "";
            txtPostalCode.Text = txtPhone.Text = "";
            txtID.IsReadOnly = true;
            InstructorList.SelectedIndex = -1;
            ProgramList.SelectedIndex = -1;
            rbActive.IsChecked = rbGraduated.IsChecked =
                                 rbLeave.IsChecked = false;
            btnNew.IsEnabled = true;
            btnSave.IsEnabled = false;
            btnDelete.IsEnabled = false;
            newStudent = false;
            status = -1;
        }
```

Test the **Cancel** button by running the app and clicking the **New** button. Then, start to write some text in the textboxes and click the **Cancel** button. The form should be back to normal, canceling the ***add student*** action.

We are now ready to implement the actual ***add*** functionality. We will first add two variables at the top of the class that will become handy later:

```
public partial class MainWindow : Window
    {
        SqlConnection connection;
        SqlDataAdapter da;
        DataSet dsCollege = new DataSet();
        DataRow record;
        int status = -1;
        bool newStudent=false;
        bool textInfo = false;
        bool selection = false;
...
```

We'll now add a method called **Validate_Info** that will validate the user imput before the student is added. Go at the end of the class and add the following method:

```
private void Validate_Info(object sender,TextChangedEventArgs e)
        {
            if (newStudent)
            {
                if (!string.IsNullOrEmpty(txtID.Text.Trim())&&
                    !string.IsNullOrEmpty(txtFirstName.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtLastName.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtAddress.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtCity.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtProvince.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtPostalCode.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtPhone.Text.Trim()))
                {
                    textInfo = true;
                    if (selection)
                    {
                        btnSave.IsEnabled = true;
                    }
                }
            }
        }
```

This method is actually pretty simple. It first checks if we are in ***add new student*** mode. If we are, it then checks if all the textboxes are filled. If that's the case, the **textInfo** variable is set to true (for later use). Finally, the method makes sure that both comboboxes have selections (using the **selection** variable that we will set later) and enables the **Save** button.

Now it's time to create the method that will actually set the **selection** variable. At the end of the class, add the following method:

```
private void Validate_Selection(object sender,SelectionChangedEventArgs
e)
        {
            if (newStudent)
            {
                if (InstructorList.SelectedIndex!=-1 &&
                    ProgramList.SelectedIndex!=-1)
                {
                    selection = true;
                    if (textInfo)
                    {
                        btnSave.IsEnabled = true;
                    }
                }
            }
        }
```

Again, this method is pretty simple as its main goal is to activate the **Save** button if both comboboxes have a selection. We will now bind the two last methods with the XAML interface so that they are called when the **SelectionChanged** and **TextChanged** events are triggered. Go in the **frmMain.xaml** file and make the following modifications on all the textboxes and the **InstructorList** and **ProgramList** comboboxes:

```
<TextBox x:Name="txtID" TextChanged="Validate_Info"
Grid.Column="2" Grid.Row="1" FontSize="11" BorderBrush="Gray"
BorderThickness="1" IsReadOnly="True"/>
```

MAKE THE PREVIOUS MODIFICATION ON ALL THE TEXTBOXES!

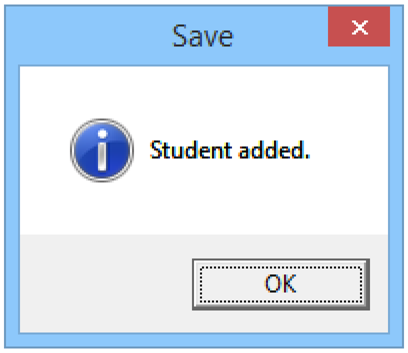
```
                    <ComboBox x:Name="InstructorList"
SelectionChanged="Validate_Selection" DisplayMemberPath="FullName"
SelectedValuePath="UserId" Grid.Column="2" Grid.Row="9" FontSize="11"
BorderBrush="Gray" BorderThickness="1" Margin="0,2,0,0"/>
..
                    <ComboBox x:Name="ProgramList"
SelectionChanged="Validate_Selection" DisplayMemberPath="Title"
SelectedValuePath="ProgramId" Grid.Column="2" Grid.Row="10"
Grid.ColumnSpan="2" FontSize="11" BorderBrush="Gray"
BorderThickness="1" Margin="0,2,0,0"/>
```

Test your application. If all goes well, the **Save** button should be activated once all the textboxes are filled and both comboboxes have selections.

Now let's put code in the ***click*** event method of the **Save** button to make it work. As usual, generate the **btnSave_Click** method using the property window and insert the following code:

```
private void btnSave_Click(object sender, RoutedEventArgs e)
        {
            if (newStudent)
            {
                try
                {
                    //Add a row
                    record = dsCollege.Tables["tblStudents"].NewRow();
                    record["StudentId"] = txtID.Text;
                    record["FirstName"] = txtFirstName.Text;
                    record["LastName"] = txtLastName.Text;
                    record["Address"] = txtAddress.Text;
                    record["City"] = txtCity.Text;
                    record["Province"] = txtProvince.Text;
                    record["PostalCode"] = txtPostalCode.Text;
                    record["Phone"] = txtPhone.Text;
                    record["InstructorId"] =
InstructorList.SelectedValue;
                    record["ProgramCode"] = ProgramList.SelectedValue;
                    record["Status"] = status;
                    record["FullName"] = txtFirstName.Text + " " +
                                           txtLastName.Text;
                    //add record to datatable
                    dsCollege.Tables["tblStudents"].Rows.Add(record);
                    MessageBox.Show("Student added.", "Save",
                       MessageBoxButton.OK,
MessageBoxImage.Information);
                    Reset();
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.Message, "Error",
MessageBoxButton.OK,
                                        MessageBoxImage.Error);
                }
            }
            else
            {

            }
        }
```

The main thing happening in that method is the use of the **NewRow** method at the beginning of the code. This method adds a new *empty* row in the **tblStudents** DataTable. The rest of the code is there to fill that row. Keep in mind that all of this is happening in the computer's memory. Nothing has happened in the database (yet). You can test this right now. Run the app, click the **New** button and fill the form. If you use a student ID that doesn't already exist, you should get that message:

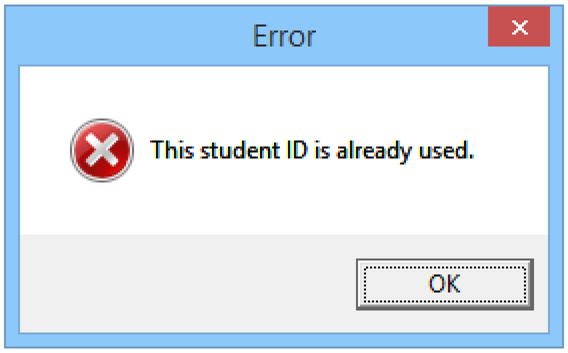


Student added.

But if you use a student ID already taken like ***653-001235***, you will get a system generated error message saying that the column value should be unique. This proves that the table's primary key constraint has been correctly transferred in the **DataSet**. However, the message can be a little confusing for the average user so we'll modify the message box in the **btnSave_Click** method to have a more user friendly message:

```
...
                    dsCollege.Tables["tblStudents"].Rows.Add(record);
                    MessageBox.Show("Student added.", "Save",
                       MessageBoxButton.OK,
MessageBoxImage.Information);
                    Reset();
                }
                catch (Exception ex)
                {
                    MessageBox.Show("This student ID is already used.",
                     "Error", MessageBoxButton.OK,
MessageBoxImage.Error);
                }
```

Do the test again and now you should get this:



Error

This student ID is already used.

If you create a new student with a valid ID, you should be able to see that student in the list. Test the app to make sure that you are able to select from the list a student you just created. Make sure that all his info is displayed properly. The only controls that don't save properly are the **RadioButtons**. We will fix that right now. Go at the end of the class and add this new method that will eventually be bonded to the **RadioButtons**:

```
  private void RadioButton_Checked(object sender,
RoutedEventArgs e)
        {
            switch (((RadioButton)sender).Name)
            {
                case "rbActive":
                    status = 0;
                    break;
                case "rbLeave":
                    status = 1;
                    break;
                case "rbGraduated":
                    status = 2;
                    break;

            }
        }
```

Then, let's bind this method with the **Checked** event of the three ***RadioButtons***. In the **frmMain.xaml** file, make the following adjustments:

```
...
<RadioButton x:Name="rbActive" Checked="RadioButton_Checked" Margin="5"
FontSize="11">Active</RadioButton>
<RadioButton x:Name="rbLeave" Checked="RadioButton_Checked" Margin="5"
FontSize="11">Leave of absence</RadioButton>
<RadioButton x:Name="rbGraduated" Checked="RadioButton_Checked"
Margin="5" FontSize="11">Graduated</RadioButton>
```

```
...
```

Run and test the application to make sure that the status of a student is recorded when you create it.

Updating a student in the DataSet

We will now add a new functionality to the app. We will make it possible to modify an existing student. The first thing to do is to enable the **Save** button once a selection has been made in the student list. We already have to methods that are triggered by the ***TextChanged*** and ***SelectionChanged*** events: **Validate_Info** and **Validate_Selection**. We will enable the **Save** button in these two methods:

```
private void Validate_Info(object sender,TextChangedEventArgs e)
        {
            if (newStudent)
            {
                if (!string.IsNullOrEmpty(txtID.Text.Trim())&&
                    !string.IsNullOrEmpty(txtFirstName.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtLastName.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtAddress.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtCity.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtProvince.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtPostalCode.Text.Trim()) &&
                    !string.IsNullOrEmpty(txtPhone.Text.Trim()))
                {
                    textInfo = true;
                    if (selection)
                    {
                        btnSave.IsEnabled = true;
                    }
                }
            }
            else
            {
                btnSave.IsEnabled = true;
            }
        }
```

```
  private void Validate_Selection(object
sender,SelectionChangedEventArgs e)
        {
            if (newStudent)
            {
                if (InstructorList.SelectedIndex!=-1 &&
                    ProgramList.SelectedIndex!=-1)
                {
                    selection = true;
                    if (textInfo)
                    {
                        btnSave.IsEnabled = true;
                    }
                }
            }
            else
            {
                btnSave.IsEnabled = true;
            }
        }
```

Test the application to make sure that the **Save** button is enabled when a student is selected. Now that the form responds correctly, we will add the actual code to update the record. In the **btnSave_Click** method, add the following code in the empty ***else*** at the end:

```
...

else
            {
                //Get the selected student
                record = 

dsCollege.Tables["tblStudents"].Rows.Find(txtID.Text);
                record["StudentId"] = txtID.Text;
                record["FirstName"] = txtFirstName.Text;
                record["LastName"] = txtLastName.Text;
                record["Address"] = txtAddress.Text;
                record["City"] = txtCity.Text;
                record["Province"] = txtProvince.Text;
                record["PostalCode"] = txtPostalCode.Text;
                record["Phone"] = txtPhone.Text;
                record["InstructorId"] = InstructorList.SelectedValue;
                record["ProgramCode"] = ProgramList.SelectedValue;
                record["Status"] = status;
                record["FullName"] = txtFirstName.Text + " " +
                                 txtLastName.Text;
            }
        }
```

This is very similar to what we did when we wanted to add a new student. The difference is that instead of using the **NewRow** method to add a new empty record in the **DataTable**, we use the **Find** method that ***locates*** an existing record. So no new row has been created. Test the program by selecting a student and trying to change one of the textboxes' content. Click on the **Save** button when you are done. Surprisingly, an error occurs! The system informs you that the **FullName** field is read only.

As you may remember, **FullName** is not a real column in our original database. It is a concatenation of the first and last names created in the stored procedure. To fix the problem, we will add the following line in the **frmMain1_Loaded** method:

```
...
da = new SqlDataAdapter("sp_SelectStudents", connection);
da.SelectCommand.CommandType = CommandType.StoredProcedure;
da.FillSchema(dsCollege, SchemaType.Mapped, "tblStudents");
da.Fill(dsCollege, "tblStudents");
dsCollege.Tables["tblStudents"].Columns["FullName"].ReadOnly =
false;
...
```

Now try it again. Select a student, change something and click **Save**. Then, select another student to change the values in the form. After that, go back to the student you modified. You should see that the modification is still there. Test with different fields to make sure everything is saved properly. All that is left to do is to display a confirmation message and reset the form when an update is made. Add the following code in **btnSave_Click**:

```
...
                record["Status"] = status;
                record["FullName"] = txtFirstName.Text + " " +
                                           txtLastName.Text;
                MessageBox.Show("Student updated.", "Save",
                       MessageBoxButton.OK,
MessageBoxImage.Information);
                Reset();
            }
...
```

Test the application to make sure the message appears.

Deleting a student in the DataSet

The last functionality we will add to the application is to be able to delete a student. Like we did with the **Save** button, we must enable the **Delete** button at the correct moment. This moment is right when a student has been selected in the list of the combobox. Since the **Validate_Selection** method is called when a student is selected in the list, we'll activate the button there. Add the following line of code in the **Validate_Selection** method:

```
...
                    if (textInfo)
                    {
                        btnSave.IsEnabled = true;
                    }
                }
            }
            else
            {
                btnSave.IsEnabled = true;
                btnDelete.IsEnabled = true;
            }
...
```

Note that although the **Validate_Selection** method is not triggered directly by the **SelectionChanged** event of the **StudentList** combobox, it is still triggered indirectly by the **SelectionChanged** event of the **ProgramList** combobox. We will also add a line of code in the **btnNew_Click** method for the situation where the **New** button is clicked after a student selection has been made. Add the following line in **btnNew_Click**:

```
  private void btnNew_Click(object sender, RoutedEventArgs e)
       {
            btnDelete.IsEnabled = false;
            StudentList.SelectedIndex = -1;
...
```

We will now make the **Delete** button work. Using the property window in design mode, generate the **btnDelete_Click** method and add the following code:

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
        {
            if (MessageBox.Show("Are you sure you want to delete this
                  student?","Warning",MessageBoxButton.YesNo,
                     MessageBoxImage.Question)==MessageBoxResult.Yes)
            {
                record =

dsCollege.Tables["tblStudents"].Rows.Find(txtID.Text);
                dsCollege.Tables["tblStudents"].Rows.Remove(record);
                MessageBox.Show("Student deleted", "Delete",
                      MessageBoxButton.OK,
MessageBoxImage.Information);
            }
            else
            {
                MessageBox.Show("Cancelling delete.", "Cancel",
                      MessageBoxButton.OK,
MessageBoxImage.Information);

            }
            Reset();
        }
```

This method uses the usual validations. For the actual deletion of the record, we first use the **Find** method to locate the record and then the **Remove** method to delete it. Test the application by trying to delete a student in the list. Notice that once the student is deleted, it doesn't appear in the list anymore.

Saving the changes to the database

The application works well, but none of our actions are reported to the database. This means that as soon as we close the applications, all our changes are lost. To fix this, we are going to create two methods: one that applies the **insert** and **update** changes, and one that applies the **delete** changes. Let's start with the first. Add the following method at the end of the class:

```
   private void SaveStudent()
        {
            string query = "";
            if (newStudent)
            {
                query = "sp_InsertStudent";
            }
            else
            {
                query = "sp_UpdateStudent";
            }
            try
            {
                da = new SqlDataAdapter();
                SqlCommand command = new SqlCommand(query, connection);
                command.CommandType = CommandType.StoredProcedure;
                command.Parameters.AddWithValue("@StudentId",
txtID.Text);
                command.Parameters.AddWithValue("@FirstName",
                       txtFirstName.Text);
                command.Parameters.AddWithValue("@LastName",
                       txtLastName.Text);
                command.Parameters.AddWithValue("@Address",
txtAddress.Text);
                command.Parameters.AddWithValue("@City", txtCity.Text);
                command.Parameters.AddWithValue("@Province",
                       txtProvince.Text);
                command.Parameters.AddWithValue("@PostalCode",
                       txtPostalCode.Text);
                command.Parameters.AddWithValue("@Phone",
txtPhone.Text);
```

```
                command.Parameters.AddWithValue("@InstructorId",
                       InstructorList.SelectedValue);
                command.Parameters.AddWithValue("@ProgramCode",
                       ProgramList.SelectedValue);
                command.Parameters.AddWithValue("@Status", status);
                da.InsertCommand = command;
                connection.Open();
                da.InsertCommand.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error",
MessageBoxButton.OK,
                                     MessageBoxImage.Error);
            }
            finally
            {
                connection.Close();
            }
        }
```

This method creates a command object that uses stored procedures to update the actual database. Then, this command is associated with the **DataAdapter** (the bridge to the real database) with the **InsertCommand** method. Finally, the **ExecuteNonQuery** method is called to write to the database.

Of course, for this to work, we need to create the stored procedures in the database. In the ***work files***, get the **sp_InsertStudent.sql** and **sp_UpdateStudent.sql** files and execute their content in SQL Server to create the stored procedures.

Once this is done, we need to make sure that the **SaveStudent** method is called when we need to write to the database. Make the following adjustments in the **btnSave_Click** method:

```
...

                    dsCollege.Tables["tblStudents"].Rows.Add(record);
                    MessageBox.Show("Student added.", "Save",
                       MessageBoxButton.OK,
MessageBoxImage.Information);
                    SaveStudent();
                    Reset();
...


             record["FullName"] = txtFirstName.Text + " " +
txtLastName.Text;
             MessageBox.Show("Student updated.", "Save",
MessageBoxButton.OK,
                        MessageBoxImage.Information);
                SaveStudent();
                Reset();
```

```
            }
```

If you test the program now, you will see that your new students and your modifications to existing students are saved to the database.

We must now do the same thing for the deletion of students. Add this new method at the end of the class:

```
private void DeleteStudent()
        {
            try
            {
                da = new SqlDataAdapter();
                SqlCommand command = new SqlCommand("sp_DeleteStudent",
                                                connection);
                command.CommandType = CommandType.StoredProcedure;
                command.Parameters.AddWithValue("@StudentId",
txtID.Text);
                da.DeleteCommand = command;
                connection.Open();
                da.DeleteCommand.ExecuteNonQuery();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message, "Error",
MessageBoxButton.OK,
                                    MessageBoxImage.Error);
            }
            finally
            {
                connection.Close();
            }
        }
```

And of course, we must also create the **sp_DeleteStudent** stored procedure in the database. In SQL Server, run the content of the **sp_DeleteStudent.sql** file. Make sure the **sp_DeleteStudent** stored procedure exists in the database. Then, the last step is to call the **DeleteStudent** method at the appropriate time. In the **btnDelete_Click** method, add the following line:

```
...
                dsCollege.Tables["tblStudents"].Rows.Remove(record);
                MessageBox.Show("Student deleted", "Delete",
                          MessageBoxButton.OK,
MessageBoxImage.Information);
                DeleteStudent();
            }

...
```

Run the application and make sure that the deletions are saved in the database.

We will leave this example here. The application is not perfect but the real goal was to learn how to use **DataSets** to interact with a database.

Progress check

Check your answer in Appendix A

Read carefully the statements and circle the letter that corresponds to the right answer.

1. True or false: the DataSet has a direct connection with the data source.

- **a**. True
- **b**. False

2. We can fill a Dataset from a database by using which SqlDataAdapter method?

- **a**. Add()
- **b**. Set()
- **c.** Data()
- **d.** Fill()

3. True or false: The DataTables that exist in a DataSet must have the same names as the original database tables.

- **a**. True
- **b**. False

4. How can we add a primary key to a DataTable object?

- **a**. .Columns.Add(PrimaryKey)
- **b**. .FillSchema
- **c.** .Add(PrimaryKey)
- **d.** .Add(PK)

5. A DataTable is the equivalent of a table in a database and a DataSet is the equivalent of...

- **a**. A record
- **b**. A field
- **c**. A database
- **d**. A view

Module 5

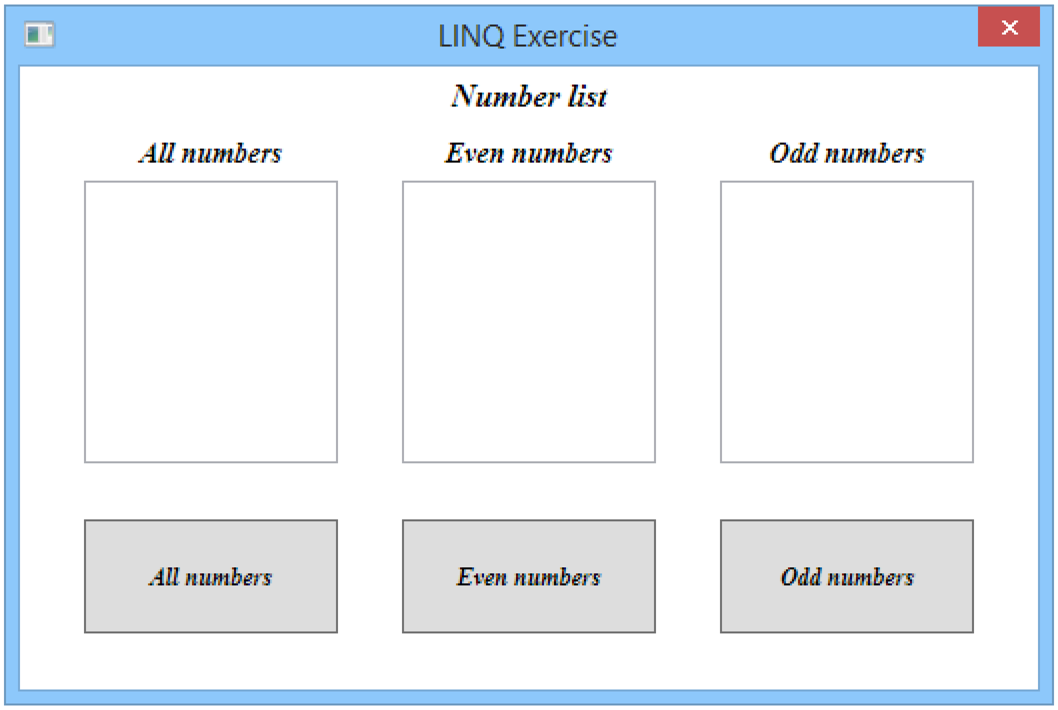
LINQ

LINQ is an acronym for ***Language Integrated Query*** and is pronounced *LIN-CUE*. It is a **Microsoft** owned technology that enables us to query data sources in C# much like the SQL language does for databases. These data sources can be databases, XML documents, lists or other objects stored in memory.

OBJECTIVES

- Understand the use of **LINQ**.

We will start by doing an exercise to become familiar with LINQ. In the ***work files***, get the **LINQ_Exercise_1** folder. In it, you will find a C# project that displays three list boxes. Execute it and you should get this window:



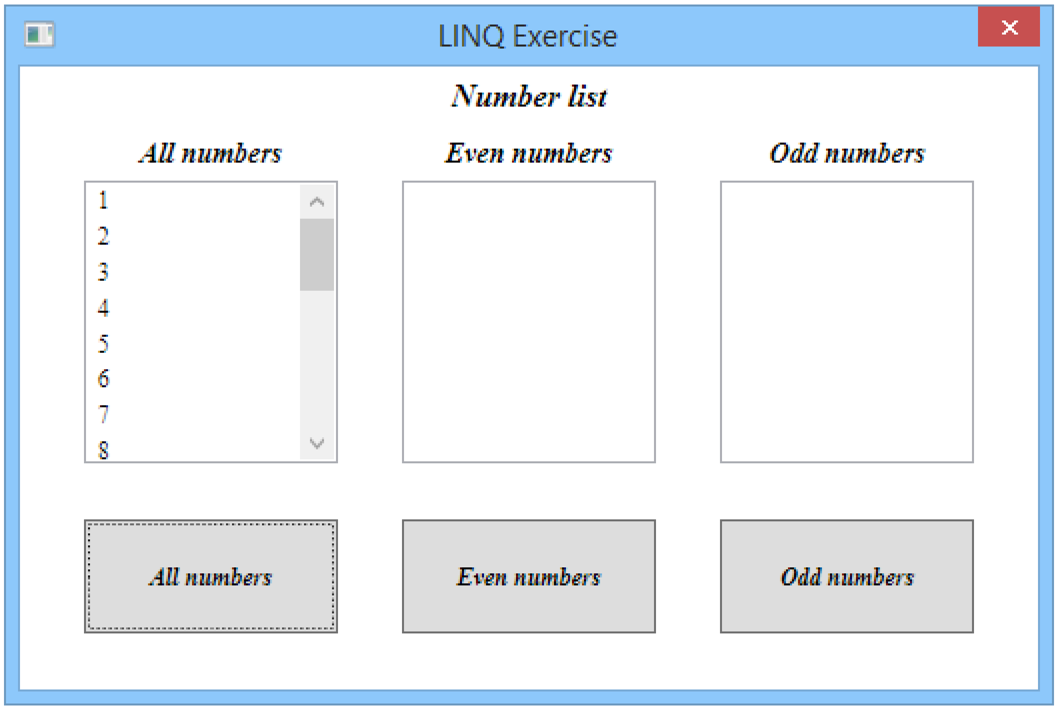
In this project, we have numbers stored in an array of integers. The C# code in the **MainWindow** class should look like this:

```
public partial class MainWindow : Window
    {
        int[] numbers =
{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};

        public MainWindow()
        {
            InitializeComponent();
        }

        private void DisplayAll_Click(object sender, RoutedEventArgs e)
        {

        }

        private void DisplayEven_Click(object sender, RoutedEventArgs
e)
        {

        }
        private void DisplayOdd_Click(object sender, RoutedEventArgs e)
        {

        }
    }
```

The goal of this exercise is to display all the numbers of the array in the first ListBox control. The second ListBox will display only the even numbers and the third, the odd numbers. We'll start by coding the **DisplayAll** button. In the **DisplayAll_Click** method, add the following code:

```
  private void DisplayAll_Click(object sender, RoutedEventArgs
e)
        {
            foreach(int number in numbers)
            {
                All.Items.Add(number);
            }
        }
```

This is a simple **foreach** loop that goes through all the numbers in the array and adds them, one by one, in the **All** ListBox. Run the application and click on the **All numbers** button to see the result:



This is nothing new, but it works. Let's now code the **Even numbers** button. Add the following code in the **DisplayEven_Click** method:

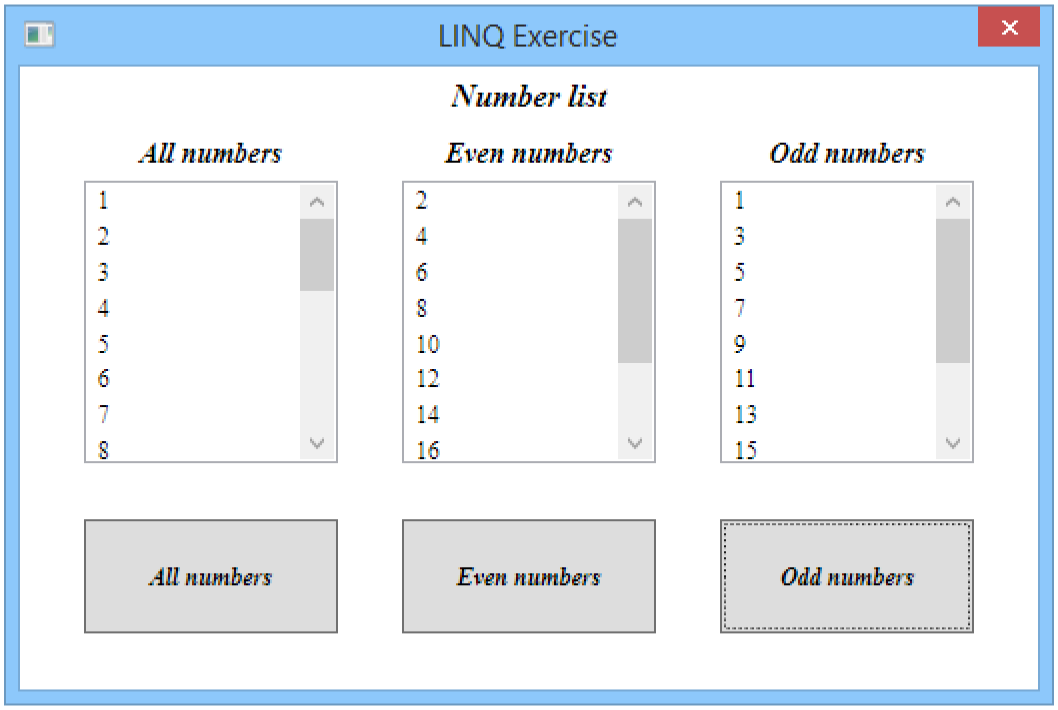
```
private void DisplayEven_Click(object sender, RoutedEventArgs
e)
        {
            foreach (int number in numbers)
            {
                if (number % 2 == 0)
                {
                    Even.Items.Add(number);
                }
            }
        }
```

This is pretty much the same logic as before, except for the ***if*** statement that checks if the number divides evenly by 2 using the ***modulus*** (%) operator (If the modulus operator is not clear to you, go see your instructor).

Test the app to make sure it works. We'll now do the same thing for the third button. Add the following code in the **DisplayOdd_Click** method:

```
private void DisplayOdd_Click(object sender, RoutedEventArgs e)
        {
            foreach (int number in numbers)
            {
                if (number % 2 != 0)
                {
                    Odd.Items.Add(number);
                }
            }
        }
```

Same logic as before, except we reversed the ***if*** statement to get those numbers that do not divide evenly. Test the app to make sure it works:



This is all good, but we haven't used LINQ yet. We'll modify the code so that we can take advantage of LINQ. We will now add an extra ***using*** statement at the top of the file and then modify the **DisplayAll_Click** method:

```
using System.Linq;
using System.Windows;
using System.Collections.Generic;
...

  private void DisplayAll_Click(object sender, RoutedEventArgs
e)
        {
            List<int> selection = from number in numbers
                                  select number;

            foreach(int number in selection)
            {
                All.Items.Add(number);
            }
        }
```

Yes, there is an error in that code, but we'll get to that in a second. First, examine the LINQ instruction itself. It starts with the keyword ***from*** followed by a variable that stores the record (row) returned by the query. The keyword ***in*** specifies the data source from which LINQ will get the data. Finally, the ***select*** keyword returns the selected column(s). When the ***select*** and the ***from*** are the same (as in our case), the entire row is returned. As you can see, this syntax shares a lot of similarities with the SQL language.

Now the C# compiler indicates that there is an error with that code. Notice that we are trying to put the result of the LINQ query in a variable of **List<int>** type. The error message tells us that this is impossible as an **IEnumerable<int>** type cannot be converted to a **List<int>** type.

So this means that LINQ returns **IEnumerable<T>** values. **List** and **IEnumerable** are two different kinds of collections, **IEnumerable** being a bit more optimized. We could use the **ToList()** method to convert the LINQ result into a **List** object, but instead, we'll just change the type of the ***selection*** variable to **IEnumerable<int>**. Make the following change in the code:

```
private void DisplayAll_Click(object sender, RoutedEventArgs e)
        {
            IEnumerable<int> selection = from number in numbers
                                  select number;

            foreach(int number in selection)
            {
                All.Items.Add(number);
            }
        }
```

Run the program and click the **All numbers** button. You should get the same result as before.

We'll now do the same thing to the Even numbers button. Make the following changes in the **DisplayEven_Click** method:

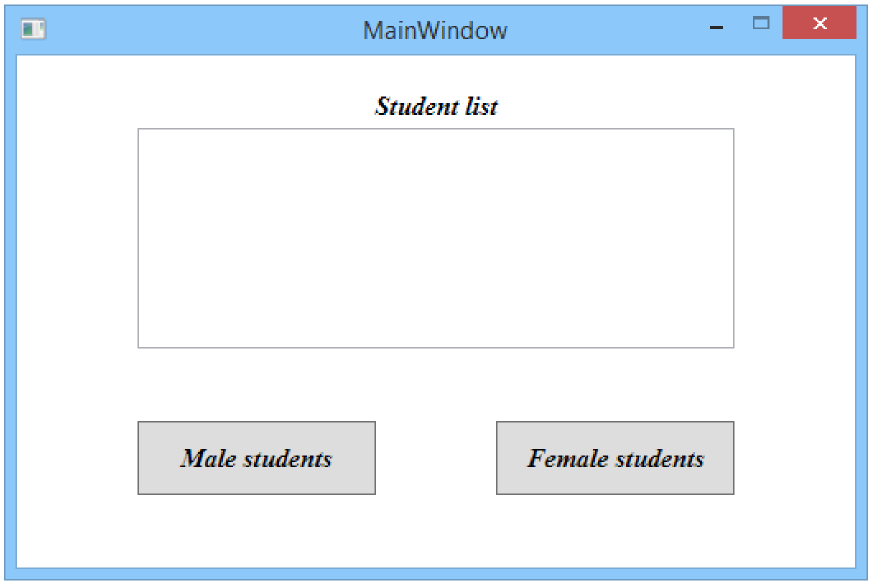
```
private void DisplayEven_Click(object sender, RoutedEventArgs
e)
        {
            IEnumerable<int> selection = from number in numbers
                                         where (number % 2 ==
0)

                                         select number;
            foreach (int number in selection)
            {
                Even.Items.Add(number);
            }
        }
```

We have now introduced the ***where*** keyword in the LINQ query. Again, this is very similar to the **where** clause of the SQL language and it serves the same purpose. It is basically a condition that filters the selected rows. The other change is that we do not need an ***if*** statement in the ***foreach*** loop anymore because the data has already been filtered by the LINQ query. Make the same kind of modification in the **DisplayOdd_Click** method:

```
private void DisplayOdd_Click(object sender, RoutedEventArgs e)
        {
            IEnumerable<int> selection = from number in numbers
                                         where (number % 2 !=
0)
                                         select number;
            foreach (int number in selection)
            {
                Odd.Items.Add(number);
            }
        }
```

Run the application to make sure everything executes the same as before. The goal of this exercise was to introduce you to the LINQ concept.

We will now go into another exercise which will go a little deeper. Get the **LINQ_Exercise_2** folder from the work files and open the project with Visual Studio. Run the app and you should get this screen:



There is a **Student** class already created in the project that will create a list full of students:

```
...
   List<Student> StudentList = new List<Student>();

   StudentList.Add(new Student { ID = 101, Name = "Joe", Sex =
false });
   StudentList.Add(new Student { ID = 102, Name = "Kim", Sex = true });
   StudentList.Add(new Student { ID = 103, Name = "Jim", Sex =
false });
   StudentList.Add(new Student { ID = 104, Name = "Karen", Sex =
true });
   StudentList.Add(new Student { ID = 105, Name = "Alexandra", Sex =
true });
   StudentList.Add(new Student { ID = 106, Name = "Yves", Sex =
false });
   StudentList.Add(new Student { ID = 107, Name = "francis", Sex =
false });
   StudentList.Add(new Student { ID = 108, Name = "Lounis", Sex = false
});
   StudentList.Add(new Student { ID = 109, Name = "Danielle", Sex =
true });
   StudentList.Add(new Student { ID = 110, Name = "John", Sex =
false });
...
```

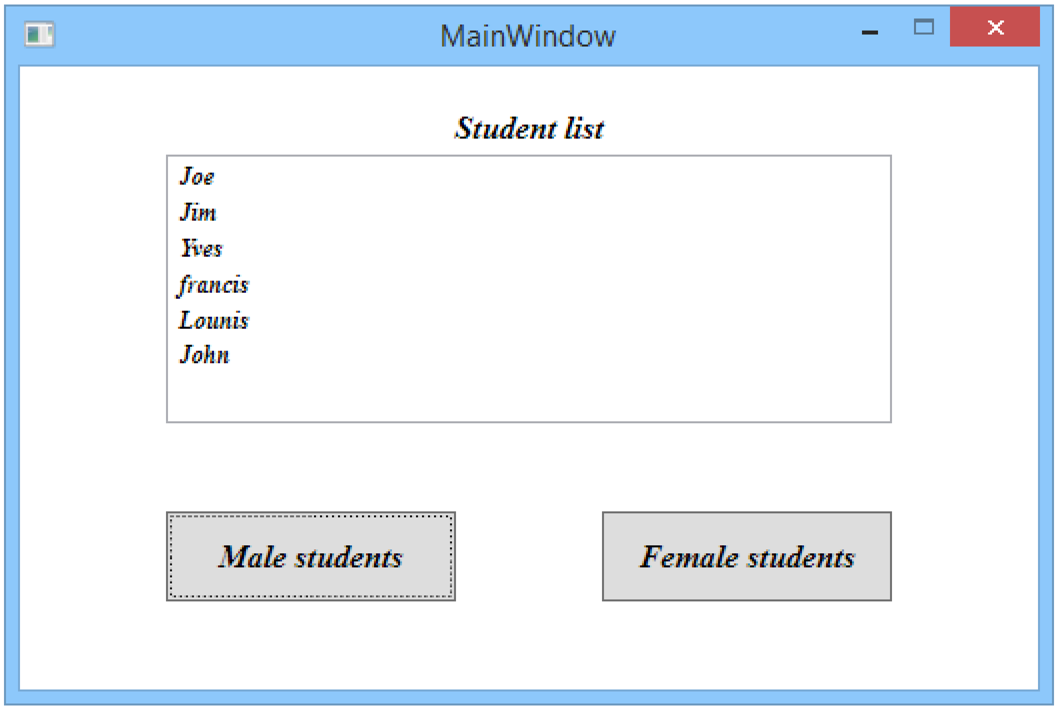
A value of **true** for the **sex** property means ***female*** and **false** means ***male***. Let's start by adding the code to fill the student ListBox. Add the following lines to the code of the main form class:

```
public partial class MainWindow : Window
    {
        List<Student> students = new List<Student>();
        public MainWindow()
        {
            InitializeComponent();
            // Get the student info
            students = Student.GetStudents();
        }
...
```

We will now add code to fill the list with male students. To do so, we will use a ***lambda*** expression in an ***extension*** method. While this might sound complicated, you will see that it's not that bad. A ***lambda expression***, which you have seen before, is basically a method with no name. And an ***extension method*** is a method that is added to an already existing data type. Go in the **btnMale_Click** and add the following code:

```
private void btnMale_Click(object sender, RoutedEventArgs e)
        {
            lstStudent.Items.Clear();
            IEnumerable<Student> selection =
                students.Where(stu => stu.Sex == false);
            foreach (Student student in selection)
            {
                lstStudent.Items.Add(student.Name);
            }
        }
```

So here we can see another way to use LINQ. The ***where*** statement is now an extension method to **students** which is of **List<Student>** type. Also, as a parameter to the ***where*** method, we use a lambda expression that returns a record only if its **Sex** property is set to false. Run the app and click the **Male students** button. You should get the following result:



```
Joe
Jim
Yves
francis
Lounis
John
```

We will now repeat the same process for the **Female students** button. Add the following code in the **btnFemale_Click** method:

```
private void btnFemale_Click(object sender, RoutedEventArgs e)
        {
            lstStudent.Items.Clear();
            IEnumerable<Student> selection =
                students.Where(stu => stu.Sex == true);
            foreach (Student student in selection)
            {
                lstStudent.Items.Add(student.Name);
            }
        }
```

The use of the ***where*** extension methods works on a **List<T>** type object because the **List<T>** class implements the **IEnumerable** interface. So this means that every class that implements the **IEnumerable** interface will be able to use its methods, and this includes the ***where*** method.

Now let's say we wish to display the students in alphabetical order. We can do that by using the **OrderBy** method which is also an extension method. Modify the **btnMale_Click** method like so:

```
private void btnMale_Click(object sender, RoutedEventArgs e)
        {
            lstStudent.Items.Clear();
            IEnumerable<Student> selection =
                students.Where(stu => stu.Sex == false)
                .OrderBy(stu => stu.Name);
            foreach (Student student in selection)
            {
                lstStudent.Items.Add(student.Name);
            }
        }
```

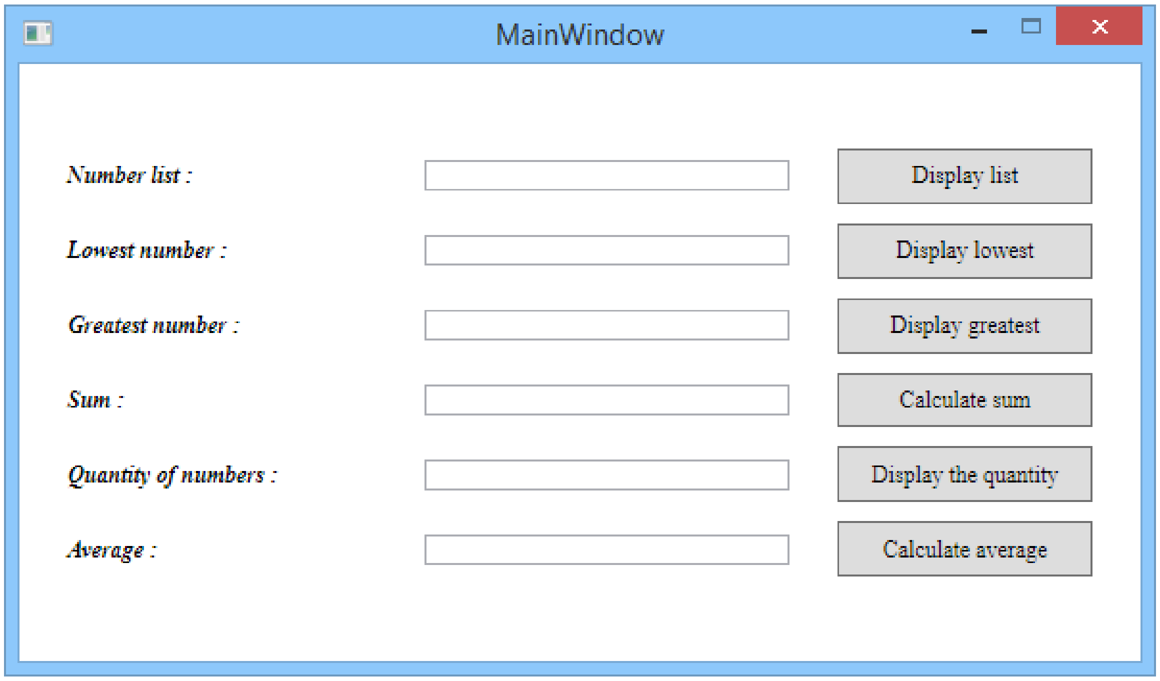
Here we can see that the **OrderBy** method is used after the **where** method call, so it will work with the result of the **where** method. Indeed, methods can be *chained* like that. We also notice that the **OrderBy** method has a lambda expression for its parameter.

Try this next: sort the female students by descending order using the **OrderByDescending** method:

```
private void btnFemale_Click(object sender, RoutedEventArgs e)
        {
            lstStudent.Items.Clear();
            IEnumerable<Student> selection =
                students.Where(stu => stu.Sex == true)
                .OrderByDescending(stu => stu.Name);
            foreach (Student student in selection)
            {
                lstStudent.Items.Add(student.Name);
            }
        }
```

Test the app to make sure it works.

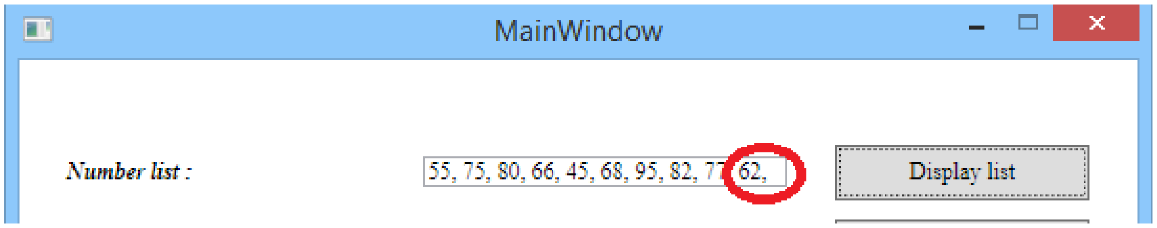
We will now take a look at the aggregate functions provided by LINQ. You have seen aggregate functions before when studying databases. Indeed, you might remember functions such as **Min**, **Max**, **Sum**, **Count** and **Average**. These methods work with groups of records, thus the ***aggregate*** term. To look at what LINQ can do with aggregate functions, get the **LINQ_Exercise_3** folder from the ***work files***. Open the Visual Studio project and run it. You should get this window:



This simple window contains six labels, six textboxes and six buttons. We will use these controls to show how easy it is to perform group operations with LINQ. If you look at the code in the main form, you will see that there is already a ***numbers*** array defined with 10 integers in it. We will start by displaying all the numbers in the array. Go in the **displayList_Click** method and enter the following code:

```
private void displayList_Click(object sender, RoutedEventArgs
e)
        {
            string result = "";
            foreach(int number in numbers)
            {
                result = result + number.ToString() + ", ";
            }
            List.Text = result;
        }
```

If you run the application and click the **Display list** button, you will see that it works, but there is an extra comma at the end of the string:



55, 75, 80, 66, 45, 68, 95, 82, 77, 62,

We will remove this extra comma with two extra lines of code. Make the following modifications:

```
  private void displayList_Click(object sender, RoutedEventArgs
e)
        {
            string result = "";
            foreach(int number in numbers)
            {
                result = result + number.ToString() + ", ";
            }
            //Get position of last comma
            int index = result.LastIndexOf(",");
            //Remove last comma
            result = result.Remove(index);
            List.Text = result;
        }
```

Test the app. You should see that the comma is now removed. Let's now code the **Display lowest** button. Enter this code in the **displayLow_Click** method:

```
private void displayLow_Click(object sender, RoutedEventArgs e)
        {
            int result = numbers[0];
            foreach (int number in numbers)
            {
                if (number < result)
                {
                    result = number;
                }
            }
            lowNumber.Text = result.ToString();
        }
```

Run the app and click on **Display lowest**. You should get **45**, which is the correct answer. Let's code the other buttons using the same kind of logic:

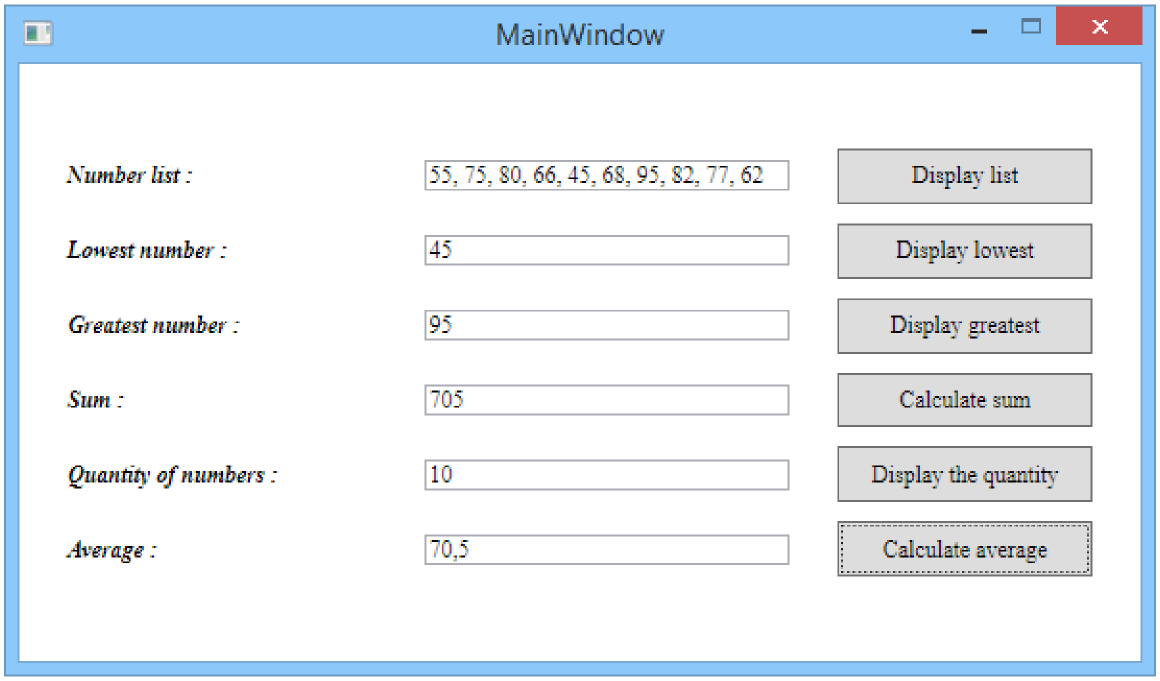
```
private void displayGreat_Click(object sender, RoutedEventArgs
e)
        {
            int result = 0;
            foreach(int number in numbers)
            {
                if(number > result)
                {
                    result = number;
                }
            }
            greatNumber.Text = result.ToString();
        }
```

```
  private void displaySum_Click(object sender, RoutedEventArgs
e)
        {
            int result = 0;
            foreach (int number in numbers)
            {
                result += number;
            }
            sumNumbers.Text = result.ToString();
        }
```

```
private void displayQuantity_Click(object sender,
RoutedEventArgs e)
        {
            int counter = 0;
            foreach (int number in numbers)
            {
                counter ++;
            }
            quantityNumbers.Text = counter.ToString();
        }
```

```
  private void displayAverage_Click(object sender,
RoutedEventArgs e)
        {
            double result = 0;
            double counter = 0;
            foreach (int number in numbers)
            {
                result += number;
                counter++;
            }
            double average = result / counter;
            avergaeNumbers.Text = average.ToString();
        }
```

Now we can run the application and test each button. We should get this result:



Number list : 55, 75, 80, 66, 45, 68, 95, 82, 77, 62
Lowest number : 45
Greatest number : 95
Sum : 705
Quantity of numbers : 10
Average : 70,5

This is all good. But now, we are going to change the code of all the buttons to use LINQ instead of the classic ***foreach*** loop. Make the following modifications in the **displayLow_Click** method:

```
private void displayLow_Click(object sender, RoutedEventArgs e)
        {
            int result = numbers.Min();
            lowNumber.Text = result.ToString();
        }
```

Wow! We've replaced almost the entire method with only one line of code. Test the app and you should see the same result as before when you click on the **Display lowest** button. As we have discussed before, LINQ has added an extension method called **Min** on the ***integer*** type. In fact, if you comment out the ***using System.Linq*** line at the top of your code, you will see that the **Min** extension method ceases to work and gives you an error. We will now change the rest of the code to use LINQ:

```
  private void displayGreat_Click(object sender, 
RoutedEventArgs e)
        {
            int result = numbers.Max();
            greatNumber.Text = result.ToString();
        }
```

```
private void displaySum_Click(object sender, RoutedEventArgs e)
        {
            int result = numbers.Sum();
            sumNumbers.Text = result.ToString();
        }
```

```
private void displayQuantity_Click(object sender, 
RoutedEventArgs e)
        {
            int counter = numbers.Count();
            quantityNumbers.Text = counter.ToString();
        }
```

```
private void displayAverage_Click(object sender, 
RoutedEventArgs e)
        {
            double average = numbers.Average();
            avergaeNumbers.Text = average.ToString();
        }
```

Run the application and test all the buttons. You will get the same results as before.

It is also possible to *chain* method calls. Let's say, for example, that we want the **Display the quantity** button to only count the even numbers. To do this, we might want to call the **Where** method in front of what we already did. Change the code of the **displayQuantity_Click** method like so:

```
private void displayQuantity_Click(object sender, 
RoutedEventArgs e)
```

```
        {
            int counter = numbers.Where(x=> x % 2==0).Count();
            quantityNumbers.Text = counter.ToString();
        }
```

If you run the app and click the **Display the quantity** button, you will get **5**, which is correct.

Now let's go back to the first button, **Display list**. It is also possible to simplify the code of that button with the help of the **Aggregate** method. Modify the code in the **displayList_Click** method for this:

```
private void displayList_Click(object sender, RoutedEventArgs e)
        {
            IEnumerable<string> selection = numbers.
                                         Select(num =>
num.ToString());
            string result = selection.Aggregate((a, b)
                                                     => a + ", " +
b);
            List.Text = result;
        }
```

We have seen the **Select** method before, so the first line is easy to understand. The second line takes the **selection** variable, which contains all the numbers of the array in string format, and applies the **Aggregate** method on it. The way **Aggregate** works is that it does the operation on the two first values (represented by **a** and **b**), takes the result and then continues to apply the operation with the third value, and then the fourth and so on. In our case, the operation is to concatenate **a** with **", "** and **b**. But the operation could be anything. Again, if we want to work only with even values, we could change the code like this:

```
private void displayList_Click(object sender, RoutedEventArgs
e)
        {
            IEnumerable<string> selection = numbers
            .Where(num => num % 2 == 0).Select(num =>
num.ToString());
            string result = selection.Aggregate((a, b)
                                                     => a + ", " +
b);
            List.Text = result;
```

```
          }
```

We hope these exercises gave you a good idea of what we can do with LINQ.

Progress check

Check your answer in Appendix A

Read carefully the statements and circle the letter that corresponds to the right answer.

1. What is a lambda expression?

- **a**. A Microsoft library
- **b**. An anonymous method
- **c.** A comparison function
- **d.** An arithmetic symbol

2. Which of the following are aggregate functions that can be done with LINQ?

- **a**. Calculate an average
- **b**. Calculate a sum
- **c.** Count items
- **d.** All of the above

3. Which LINQ method can find the highest value in a list?

- **a**. Greatest
- **b**. Maximum
- **c.** Max
- **d.** Total

4. True or false: LINQ can only work with SQL Server databases.

- **a**. True
- **b**. False

5. With LINQ, how can we filter a selection?

- **a**. Aggregate
- **b**. Where
- **c**. Filter
- **d**. None of the above

Module 6

Entity Framework

Entity Framework is an **O**bject **R**elational **M**apping framework, or **ORM**. This very powerful tool is able to create strongly typed classes based on database tables. Indeed, **Entity Framework** is the bridge between the OO class model used by C# and relational databases. We kept this for the end of the course because you will see that this framework does a lot of work for us, including generating the connection string.

OBJECTIVES

- Understand and use Entity Framework.

Watch the following videos

Get the following videos from the work files:

EntityFramework-Part1(English).mp4

EntityFramework-Part2(English).mp4

(NB: This module is not part of the exam. It will however help you for the project.)

You can now ask your instructor for the final exam and project.

APPENDIX A

SOLUTIONS (Test your newly acquired knowledge)

Module 1 **1-B** **2-C** **3-C** **4-D** **5-A** **Module 2** **1-A** **2-A** **3-C** **4-B** **5-C** **Module 3** **1-B** **2-C** **3-A** **4-B** **5-D**	**Module 4** **1-B** **2-D** **3-B** **4-B** **5-C** **Module 5** **1-B** **2-D** **3-C** **4-B** **5-B**