

Microsoft® Official Course



Module 9

Designing the User Interface for a Graphical Application

Microsoft®

Module Overview

- Using XAML to Design a User Interface
 - Binding Controls to Data
 - Styling a UI
-
- In here, you will learn how to use Extensible Application Markup Language (XAML) and Windows Presentation Foundation (WPF) to create engaging UIs.
 - *Avoid going into more detail than the basic coverage provided in this module, but make sure that students have sufficient information to complete the lab.

Lesson 1: Using XAML to Design a User Interface

- Introducing XAML
- Common Controls
- Setting Control Properties
- Handling Events
- Using Layout Controls
- Demonstration: Using Design View to Create a XAML UI
- Creating User Controls

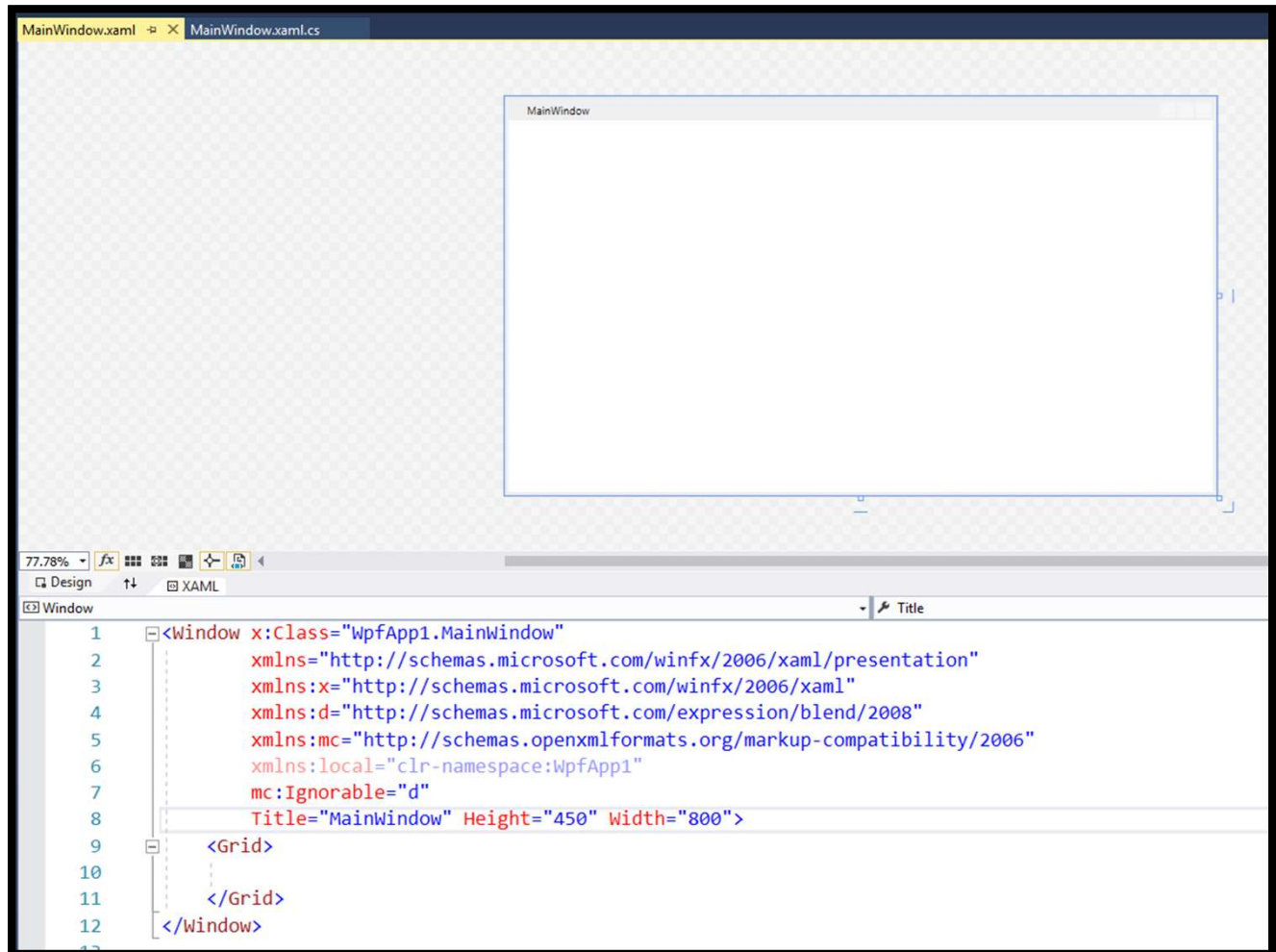
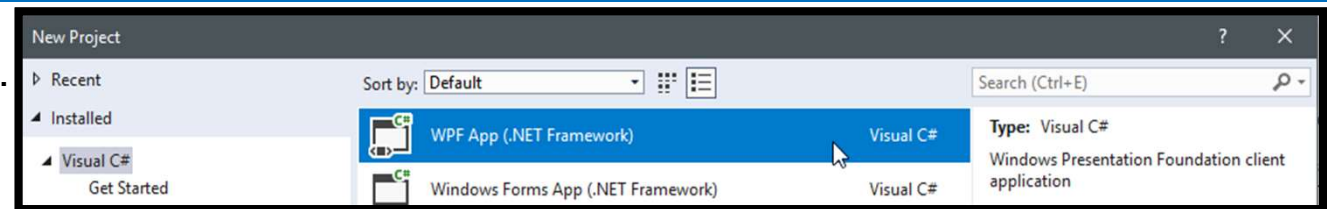
Introducing XAML

- Use XML elements to create controls
- Use attributes to set control properties
- Create hierarchies to represent parent controls and child controls
 - The most common top-level element in a WPF XAML file is the **Window** element. It can include several attributes (`xmlns` [xml namespaces], `title`, ...).
 - A **Window** element can contain a single child element that defines the content of the UI
 - WPF defines container controls that you can use to combine other lower-level controls together and lay them out - most commonly used container control is the **Grid** control
 - The **Window** template automatically adds a **Grid** control to the **Window**
 - The **Grid** control defines a default layout that consists of a single row and a single column, but you can customize
- When you build a **WPF application**, the build engine converts the declarative markup in your XAML file into classes and objects.

```
<Window x:Class="MyNamespace.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Order Your Coffee Here" Height="350" Width="525">
    <Grid>
        <Button Content="Get Me a Coffee!" />
    </Grid>
</Window>
```

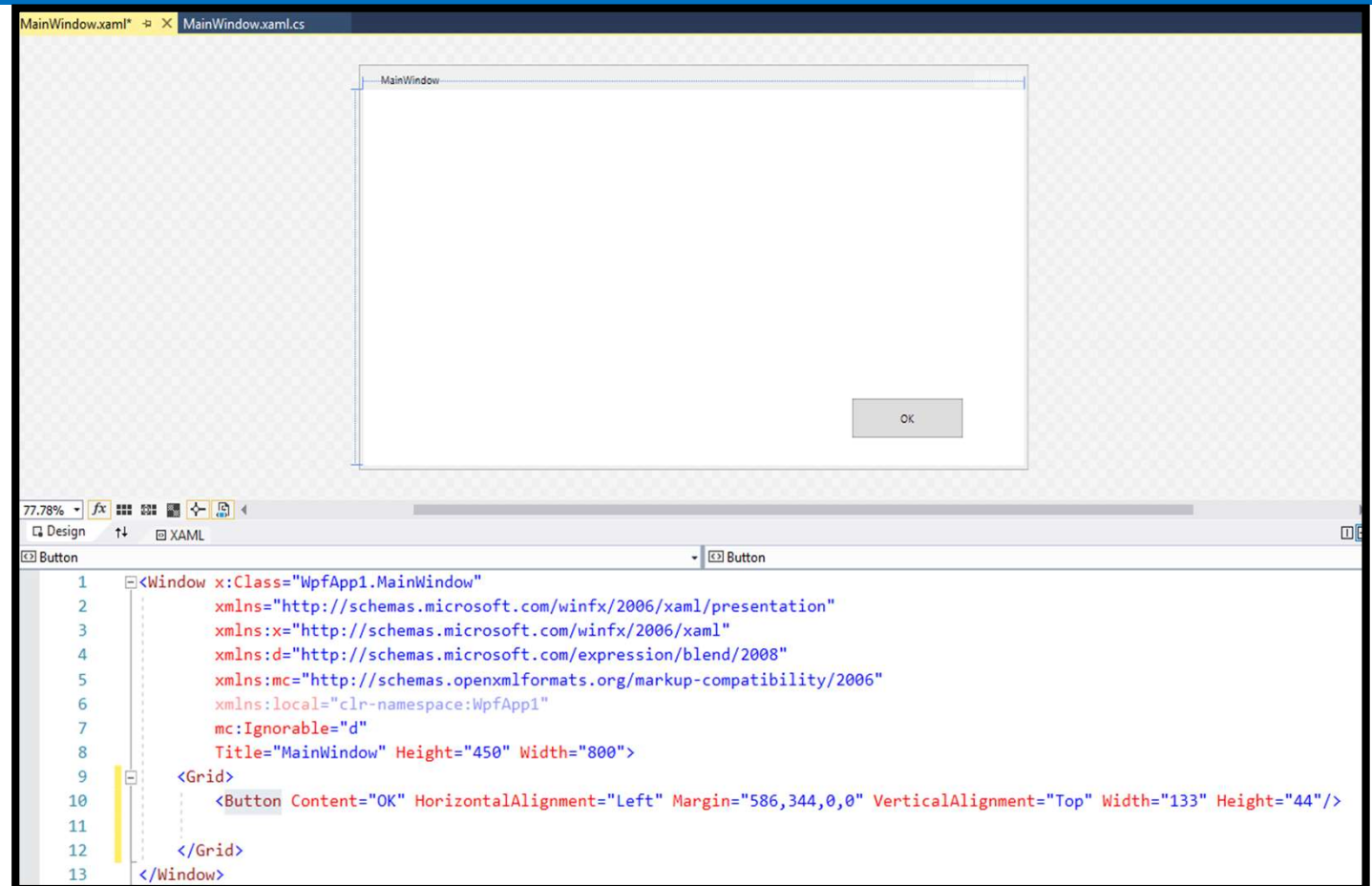
Extra Notes ... (example)

- Create a WPF App project ...



Extra Notes ... (example)

- Adding a button...
 - Give it a name
 - Use it in the cs file...
 - Program it
 - How does the application know where to position the button?
 - What are some elements of the Button control?



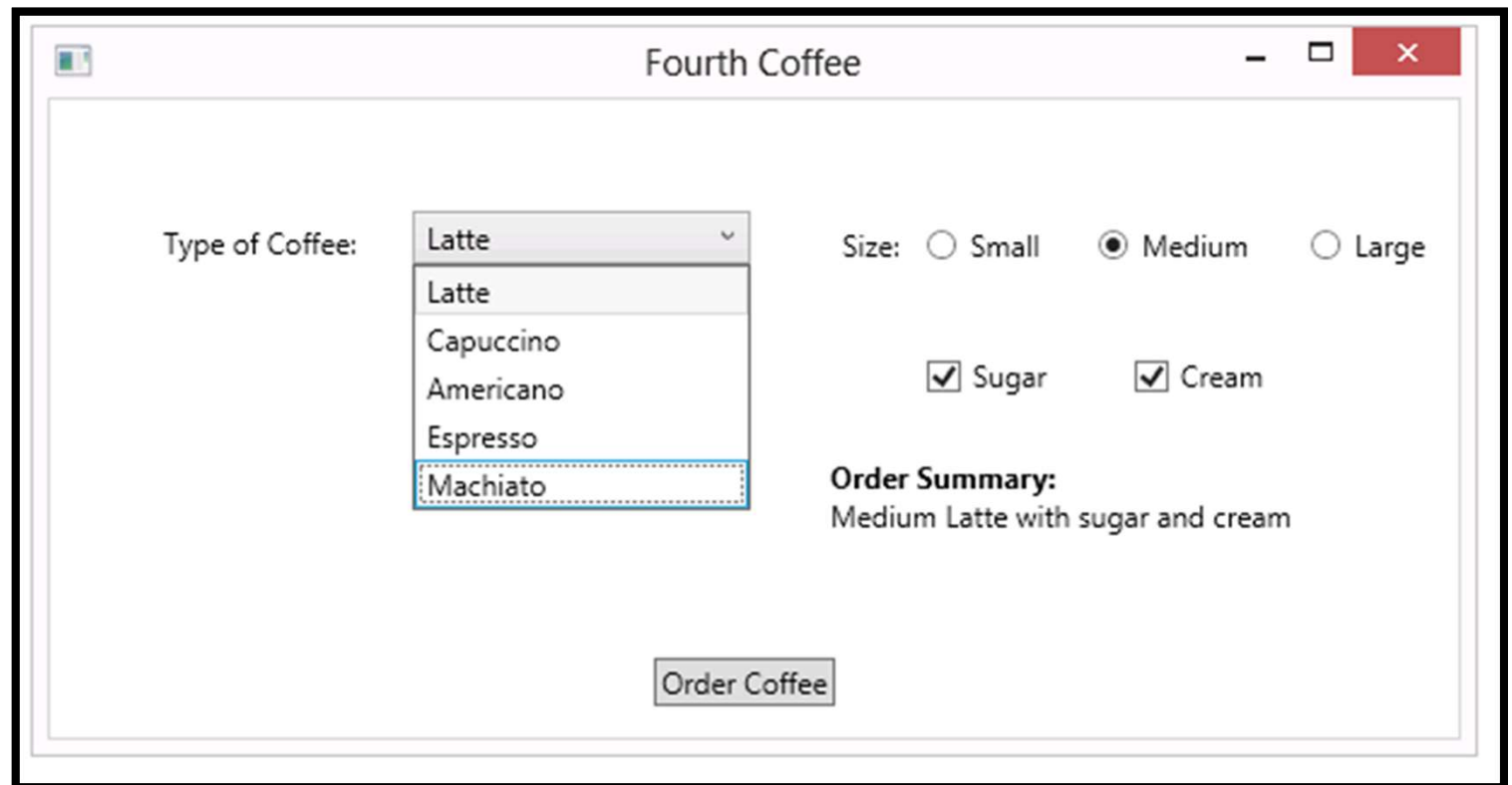
- See also
XAML overview (WPF)
<https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/xaml-overview-wpf>

Common Controls

Control	Description
Button	Displays a button that a user can click to perform an action.
CheckBox	Enables a user to indicate whether an item should be selected (checked) or not (blank).
ComboBox	Displays a drop-down list of items from which the user can make a selection.
Label	Displays a piece of static text.
ListBox	Displays a scrollable list of items from which the user can make a selection.
RadioButton	Enables the user to select from a range of mutually exclusive options.
TabControl	Organizes the controls in a UI into a set of tabbed pages.
TextBlock	Displays a read-only block of text.
TextBox	Enables the user to enter and edit text.

Common Controls

- What controls can you see in here?
 - **Label, ComboBox, RadioButton, CheckBox, TextBlock, and Button.**
- There are numerous other controls available
 - **ProgressBar, Slider, DatePicker, DataGrid, Toolbar, TreeViewer, and WebBrowser.**



The screenshot shows a Windows application window titled "Fourth Coffee". Inside the window, there is a form for ordering coffee. The form includes a "Type of Coffee:" label next to a ComboBox control. The ComboBox is open, showing a list of coffee types: "Latte", "Capuccino", "Americano", "Espresso", and "Machiato". The "Machiato" option is currently selected and highlighted. To the right of the ComboBox, there is a "Size:" label followed by three radio button options: "Small", "Medium", and "Large". The "Medium" radio button is selected. Below the size options, there are two checked checkboxes: "Sugar" and "Cream". At the bottom right of the form, there is an "Order Summary:" section that displays "Medium Latte with sugar and cream". At the bottom center of the window, there is a button labeled "Order Coffee".

Extra Notes ... (example)

- Adding a button...
 - Add two radio buttons
 - Group them (see GroupName)
 - IsChecked="True"

```
<Window x:Class="WpfApp1.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:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <Button x:Name="myOKButton" Content="OK" HorizontalAlignment="Left" Margin="586,344,0,0" VerticalAlignment="Top" Width="133" Height="44"/>
        <RadioButton x:Name="Option1" Content="Option1" HorizontalAlignment="Left" Margin="308,126,0,0" VerticalAlignment="Top" GroupName="MyOptions"/>
        <RadioButton x:Name="Option2" Content="Option2" HorizontalAlignment="Left" Margin="410,126,0,0" VerticalAlignment="Top" RenderTransformOrigin="0.229,1" GroupName="MyOptions"/>
    </Grid>
</Window>
```

Setting Control Properties

- Most **controls** enable you to **set simple property values** by using **attributes**.

```
<Button Content="Click Me" Background="Yellow" />
```

- Use **property element syntax** to define **complex property values**
 - E.g. you can add an element named **Button.Background** as a child element of the **Button** element, and then in the **Button.Background** element, you can add further child elements to define your gradient effect.

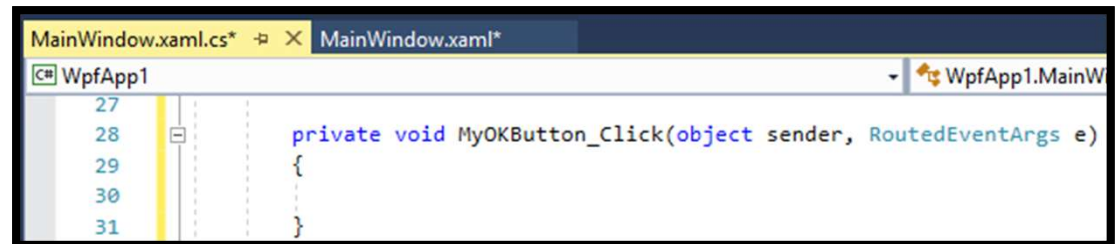
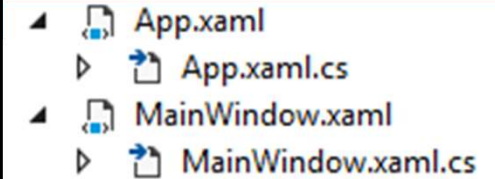
```
<Button Content="Click Me"
  <Button.Background>
    <LinearGradientBrush StartPoint="0.5, 0.5"
      EndPoint="1.5, 1.5">
      <GradientStop Color="AliceBlue" Offset="0" />
      <GradientStop Color="Aqua" Offset="0.5" />
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

- Above, the **Content** property was set to a text value. One can also use a picture for it ...

```
<Button Margin="150, 130, 150, 130">
  <Image Source="Images/coffee.jpg" Stretch="Fill" />
</Button>
```

Handling Events

- When you create a WPF application in Visual Studio, each XAML page has a corresponding code-behind file.
 - For example, the *MainWindow.xaml* file that Visual Studio creates by default has a code-behind file named *MainWindow.xaml.cs*.
 - You **subscribe to event handlers in your XAML markup** and then **define your event handler logic in the code-behind file**.
 - if you double-click a **Button** control at design time, Visual Studio will automatically create an event handler stub method in the code behind file.
 - It also automatically binds the **Click** event of the button to the event handler method.



```
<Window x:Class="WpfApp1.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:WpfApp1"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <Button x:Name="myOKButton" Content="OK" HorizontalAlignment="Left" Margin="586,344,0,0" VerticalAlignment="Top" Width="133" Height="44" Click="MyOKButton_Click"/>
        <RadioButton x:Name="Option1" Content="Option1" HorizontalAlignment="Left" Margin="308,126,0,0" VerticalAlignment="Top" GroupName="MyOptions" IsChecked="True" />
        <RadioButton x:Name="Option2" Content="Option2" HorizontalAlignment="Left" Margin="410,126,0,0" VerticalAlignment="Top" RenderTransformOrigin="0.229,1" GroupName="MyOptions"/>
    </Grid>
</Window>
```

Handling Events

- Specify the event handler method in XAML
 - Note: **x** prefix maps to the <http://schemas.microsoft.com/winfx/2006/xaml> namespace

```
<Button x:Name="btnMakeCoffee"
        Content="Make Me a Coffee!"
        Click="btnMakeCoffee_Click" />

<Label x:Name="lblResult"
        Content=""
        Margin="150, 186, 150, 75" />
```

- Handle the event in the code-behind class

```
private void btnMakeCoffee_Click(object sender,
    RoutedEventArgs e)
{
    lblResult.Content = "Your coffee is on its way.";
}
```

- Events are bubbled to parent controls
 - Next slide

Handling Events – the event-bubbling mechanism

- When an event is raised, WPF will attempt to run the corresponding event handler for the control that has the focus.
 - If there is no event handler available, then WPF will examine the parent of the control that has the focus, and if it has a handler for the event it will run. If the parent has no suitable event handler, WPF examines the parent of the parent, and so on right up to the top-level window. This process is known as bubbling, and it enables a container control to implement default event-handling logic for any events that are not handled by its children.
 - When a control handles an event, the event is still bubbled to the parent in case the parent needs to perform any additional processing.
- An event handler is passed information about the event in the *RoutedEventArgs* parameter.
 - An event handler can use the properties in this parameter to determine the source of the event (the control that had the focus when the event was raised).
 - The *RoutedEventArgs* parameter also includes a Boolean property called *Handled*. An event handler can set this property to *true* to indicate that the event has been processed. When the event bubbles, the value of this property can be used to prevent the event from being handled by a parent control.
- See also: <http://go.microsoft.com/fwlink/?LinkID=267822>

```
private void btnMakeCoffee_Click(object sender, RoutedEventArgs e)
{
    lblResult.Content = "Your coffee is on its way.";
}
```

Using Layout Controls

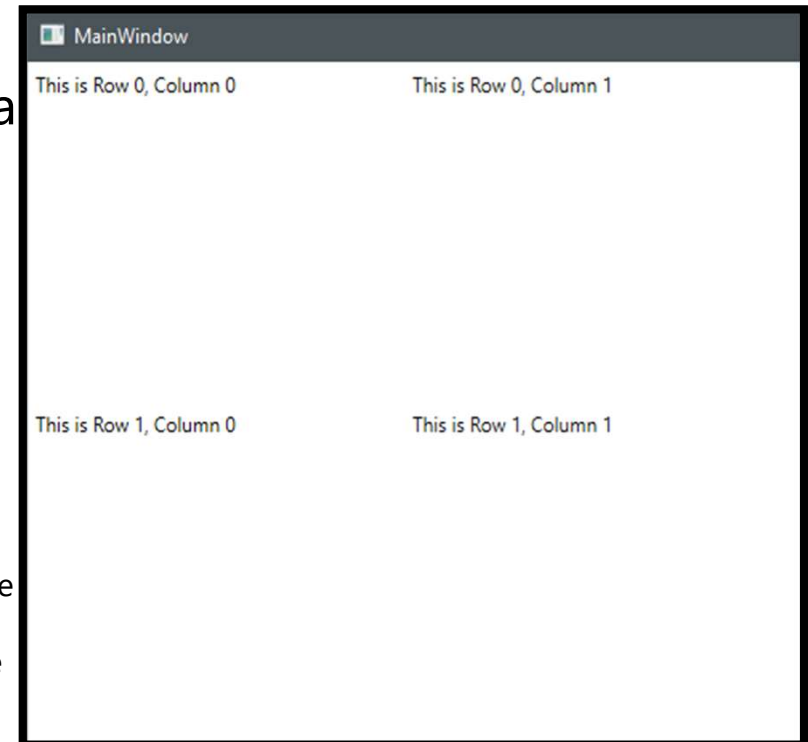
- WPF includes several layout, or container, controls that enable you to position and size your child controls in different ways.

Control	Description
Canvas	Child controls define their own layout by specifying canvas coordinates.
DockPanel	Child controls are attached to the edges of the DockPanel.
Grid	Child controls are added to rows and columns within the grid.
StackPanel	Child controls are stacked either vertically or horizontally.
VirtualizingStackPanel	Child controls are stacked either vertically or horizontally. At any one time, only child items that are visible on the screen are created.
WrapPanel	Child controls are added from left to right. If there are too many controls to fit on a single line, the controls wrap to the next line.

Using Layout Controls – Grid Control – SKIP

- The following example shows how to define a grid with two rows and two columns:
- notice the **RowDefinition** & **ColumnDefinition** elements to define rows and columns
- For each row or column, you can specify a **minimum** and **maximum** height or width, or a **fixed** height or width. Ways:
 - As **numerical units**.
 - For example, `Width="200"` represents 200 units
 - As **Auto**.
 - For example, `Width="Auto"` will set the column to the minimum width required to render all the child controls in the column.
 - As a **star** value.
 - For example, `Width="*"` will make the column use of any available space after fixed width columns and auto-width columns are allocated. If you create two columns with widths of 3* and 7*, the available space will be divided between the columns in the ratio 3:7.

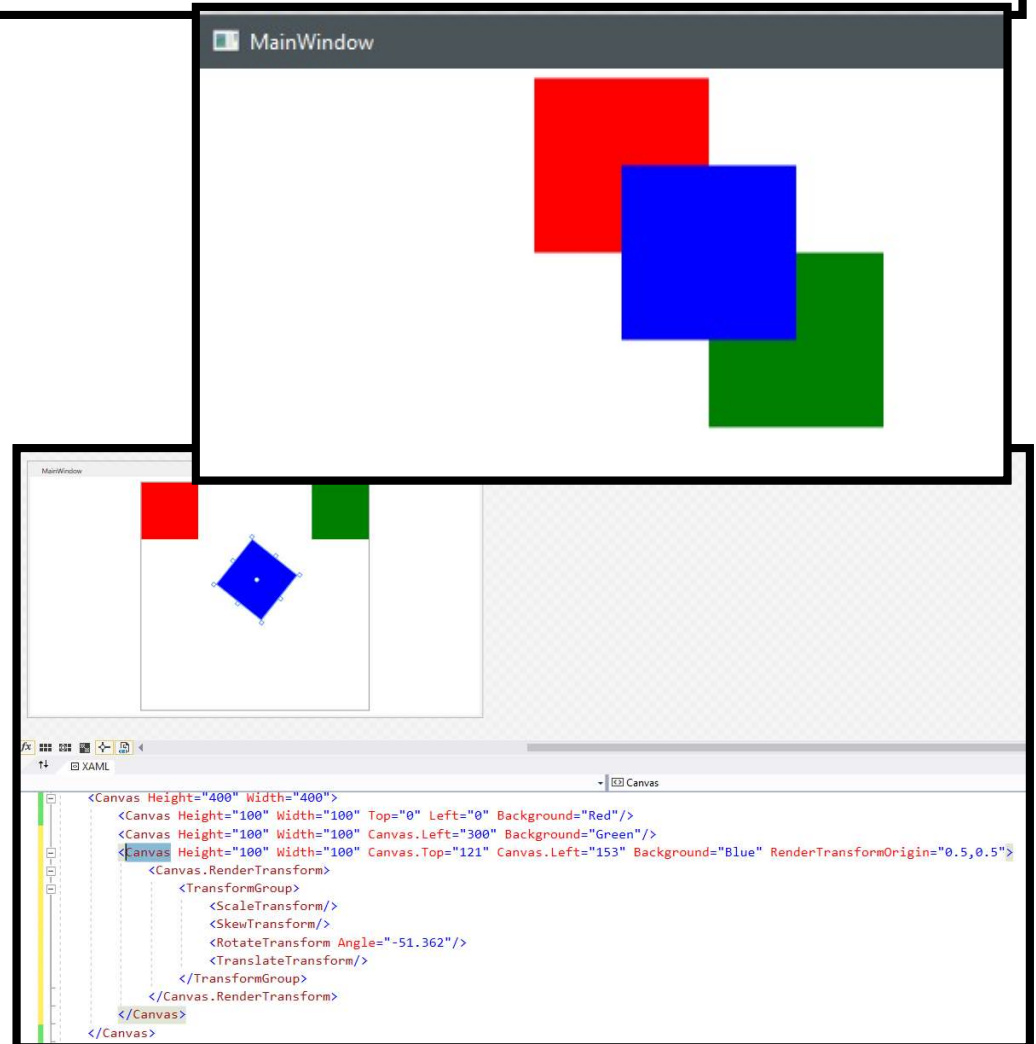
```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition MinHeight="100" MaxHeight="200" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="3*" />
    <ColumnDefinition Width="7*" />
  </Grid.ColumnDefinitions>
  <Label Content="This is Row 0, Column 0" Grid.Row="0" Grid.Column="0" />
  <Label Content="This is Row 0, Column 1" Grid.Row="0" Grid.Column="1" />
  <Label Content="This is Row 1, Column 0" Grid.Row="1" Grid.Column="0" />
  <Label Content="This is Row 1, Column 1" Grid.Row="1" Grid.Column="1" />
</Grid>
```



Using Layout Controls – Canvas Control – SKIP

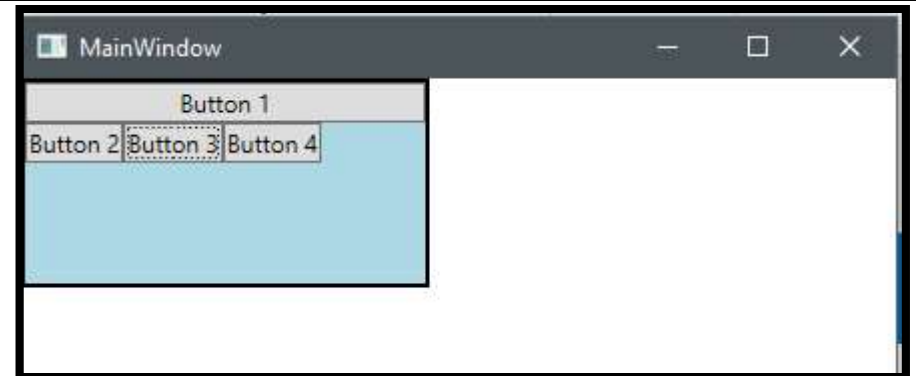
```
<Canvas Height="400" Width="400">
    <Canvas Height="100" Width="100" Top="0" Left="0" Background="Red"/>
    <Canvas Height="100" Width="100" Top="100" Left="100" Background="Green"/>
    <Canvas Height="100" Width="100" Top="50" Left="50" Background="Blue"/>
</Canvas>
```

- Defines an area within which you can explicitly position child elements by using coordinates that are relative to the Canvas area
 - Child elements of a Canvas are never resized, they are just positioned at their designated coordinates. This provides flexibility for situations in which inherent sizing constraints or alignment are not needed or wanted.
 - For cases in which you want child content to be automatically resized and aligned, it is usually best to use a Grid element..



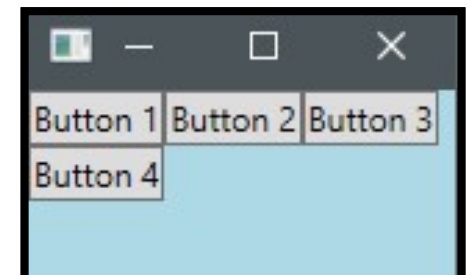
Using Layout Controls – WrapPanel Control – SKIP

```
<Border HorizontalAlignment="Left" VerticalAlignment="Top" BorderBrush="Black" BorderThickness="2">
  <WrapPanel Background="LightBlue" Width="200" Height="100">
    <Button Width="200">Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </WrapPanel>
</Border>
```



- Positions child elements in sequential position from left to right, breaking content to the next line at the edge of the containing box.
- Subsequent ordering happens sequentially from top to bottom or from right to left, depending on the value of the Orientation property.

```
<WrapPanel Background="LightBlue" >
  <Button>Button 1</Button>
  <Button>Button 2</Button>
  <Button>Button 3</Button>
  <Button>Button 4</Button>
</WrapPanel>
```



Creating User Controls

To **create** a user control:

- Define the control in XAML
- Expose properties and events in the code-behind class

To **use** a user control:

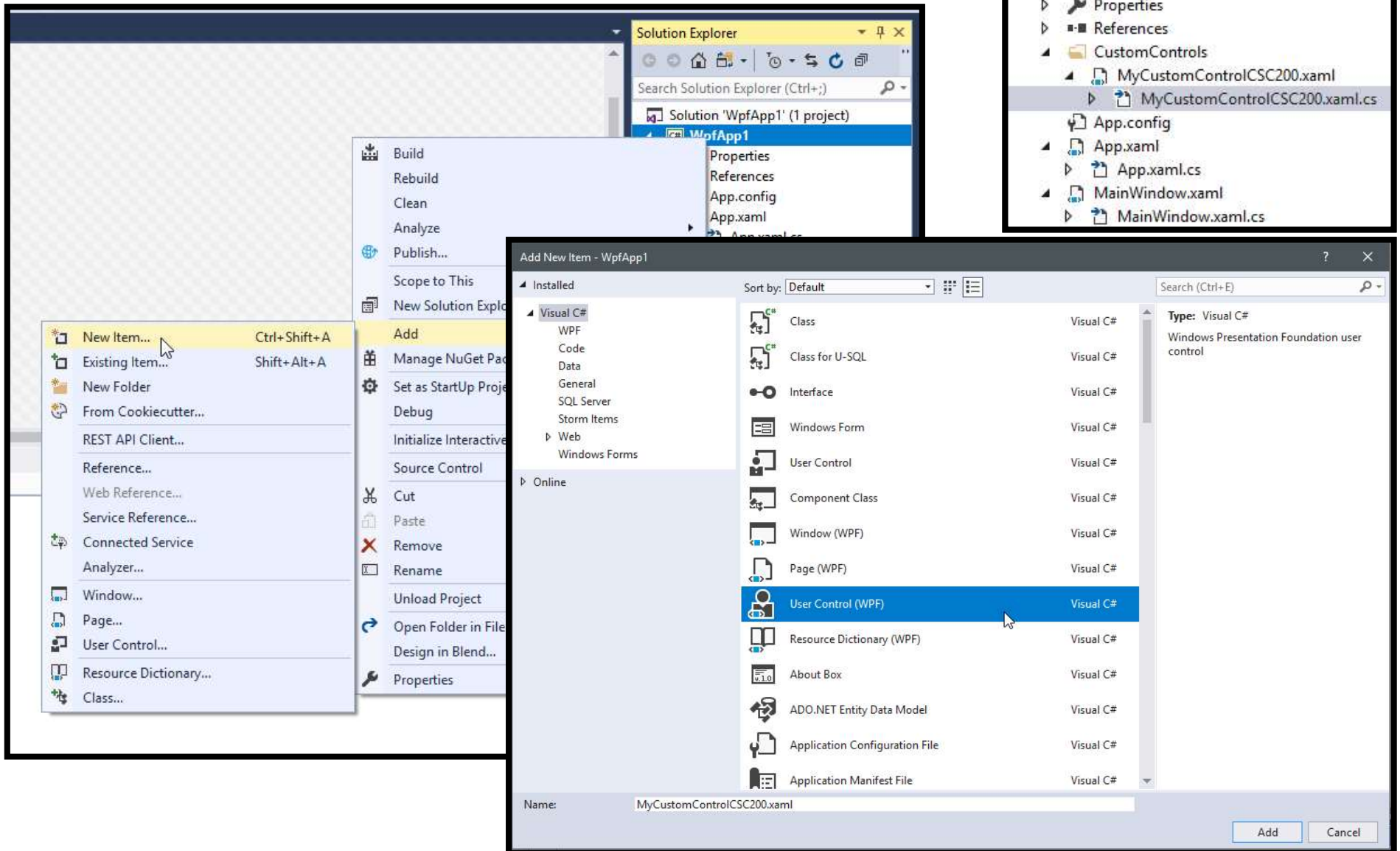
- Add an XML namespace prefix for the assembly and namespace
- Use the control like a standard XAML control

Creating User Controls

- **create a user control** to enable multiple developers to **share the same custom control across multiple assemblies**.
 - Like a regular XAML window, user controls consist of a **XAML file** and a **corresponding code-behind file**.
 - In the XAML file, the principal difference is that the top-level element is a **UserControl** element rather than a **Window** element.
 - In the code-behind file for the user control, you can create **event handler** methods in the same way that you would for regular XAML windows. You should:
 - **Define any required public properties** – so the consumers can get or set property values, in either the XAML or the code.
 - **Define any required public events**.
 - Use a template to create a WPF user control
- To **use your user control** in a WPF application, you need to do two things:
 - **Add a namespace prefix** for your user control namespace and assembly to the **Window** element. This should take the following form:
 - **xmlns:[your prefix]="clr-namespace:[your namespace],[your assembly name]"**
 - **Note:** If your application and your user control are in the same assembly, you can omit the assembly name from the namespace prefix declaration.
 - **Add the control to your application** in the same way that you would add a built-in control, with the namespace prefix you defined.

Creating User Controls

- Visual Studio includes a template for WPF user controls. This automatically creates the top-level element, the code-behind file, and the class constructor.



Lesson 2: Binding Controls to Data

- Introduction to Data Binding
- Binding Controls to Data in XAML
- Binding Controls to Data in Code
- Binding Controls to Collections
- Creating Data Templates

Introduction to Data Binding

- **Data binding** is the act of connecting a **data source** to a UI **element** in such a way that if one changes, the other must also change.
- Data binding has **three components**:
 - Binding **source**:
 - the **source of data**, typically a **property** of a custom .NET object.
 - Binding **target**:
 - the **XAML element** you want to bind to your data source, and is typically a UI **control**.
 - You must bind your data source to a property of your target object, and that property must be a **dependency property**. E.g. bind data to the **Content** property of a **TextBox** element.
 - Binding **object**
 - the **object that connects the source to the target**. The **Binding** object can also specify a converter, if the source property and the target property are of different data types.
- A **dependency property** is a special type of wrapper around a regular property.
 - Dependency properties are registered with the .NET Framework runtime, which enables the runtime to notify any interested parties when the value of the underlying property changes. This ability to notify changes is what makes data binding work.

Introduction to Data Binding - Example

- `<TextBlock Text="{Binding Source={StaticResource coffee1}, Path=Bean}" />`
- We have:
 - The binding **source** is the **Bean** property of the **coffee1** object.
 - The binding **target** is the **Text** property of the **TextBlock** element.
 - The **binding object** is defined by the expression in braces.
- **Text** property of a **TextBlock** is set to a **data binding expression**:
 - The **Binding** expression is enclosed in braces. This enables you to set properties on the **Binding** object before it is evaluated by the **TextBlock**.
 - The **Source** property of the **Binding** object is set to **{StaticResource coffee1}**. This is an **object instance** defined elsewhere in the solution.
 - The **Path** property of the **Binding** object is set to **Bean**. This indicates that you want to bind to the **Bean property of the **coffee1** object.**

Introduction to Data Binding

- A data binding can be **bidirectional** or **unidirectional**:
 - `<TextBox Text="{Binding Source={StaticResource coffee1}, Path=Bean, Mode=TwoWay}" />`

Mode Value	Details
TwoWay	Updates the target property when the source property changes, and updates the source property when the target property changes.
OneWay	Updates the target property when the source property changes.
OneTime	Updates the target property only when the application starts or when the DataContext property of the target is changed.
OneWayToSource	Updates the source property when the target property changes.
Default	Uses the default Mode value of the target property.

Binding Controls to Data in XAML – create static resource

- If your source data will not change during the execution of your application, you can create a **static resource** to represent your data in XAML
 - A **static resource** enables you to create instances of classes. To create a static resource:
 - Add an element to the **Resources** property of a container control, such as the top-level **Window**.
 - Set the name of the element to the name of the class you want to instantiate. For example, if you want to create an instance of the **Coffee** class, create an element named **Coffee**. Use a namespace prefix declaration to identify the namespace and assembly that contains your class.
 - Add an **x:Key** attribute to the element. This is the identifier by which you will specify the static resource in data binding expressions. You can create multiple instances of a resource in a window, but each instance should have a unique **x:Key** value.

```
<Window x:Class="DataBinding.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:loc="clr-namespace:DataBinding"
        Title="Data Binding Example" Height="350" Width="525">
    <Window.Resources>
        <loc:Coffee x:Key="coffee1"
                    Name="Fourth Coffee Quencher"
                    Bean="Arabica"
                    CountryOfOrigin="Brazil"
                    Strength="3" />
        ...
    </Window.Resources>
    ...
</Window>
```

Binding Controls to Data in XAML – bind item to static resource

- to bind an individual UI element to this static resource, you can use a binding statement that specifies both a source and a path.

- set the **Source** property to the static resource, and set the **Path** property to the specific property in the source object to which you want to bind.

```
<TextBlock Text="{Binding Source={StaticResource coffee1}, Path=Bean}" />
```

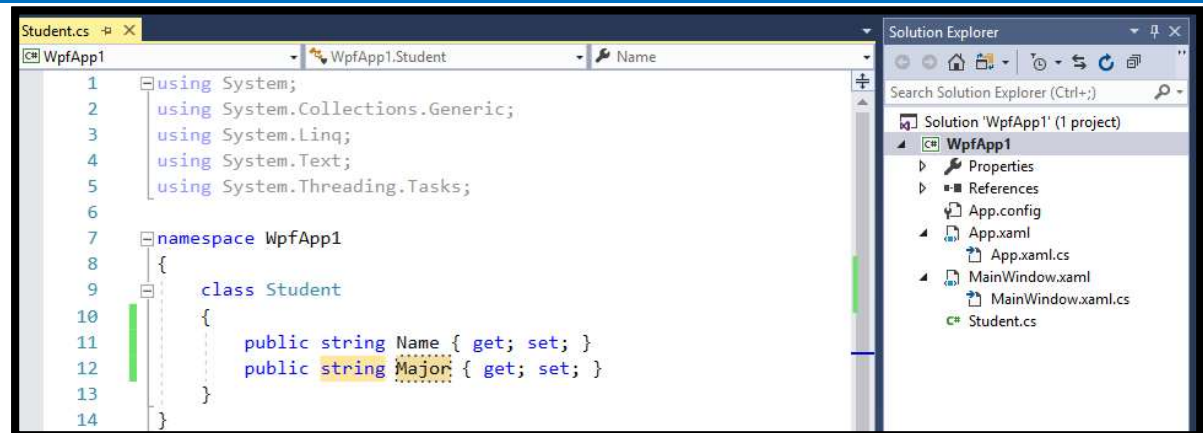
- to bind multiple UI elements to different properties of a data source

- set the **DataContext** property on a container object, such as a **Grid** or a **StackPanel**. This specifies the source object, but does not identify specific properties.
- Any child controls within the container object will inherit this data context, unless you override it. when you create data binding expressions for child controls, you can omit the source and simply specify the path to the property of interest.
 - While you could specify a full data binding expression for each individual UI element, specifying a **DataContext** property typically makes your XAML easier to write, easier to read, and easier to maintain.

```
<StackPanel>  
  <StackPanel.DataContext>  
    <Binding Source="{StaticResource coffee1}" />  
  </StackPanel.DataContext>  
  <TextBlock Text="{Binding Path=Name}" />  
  ...  
</StackPanel>
```

Binding Controls to Data in XAML – create static resource

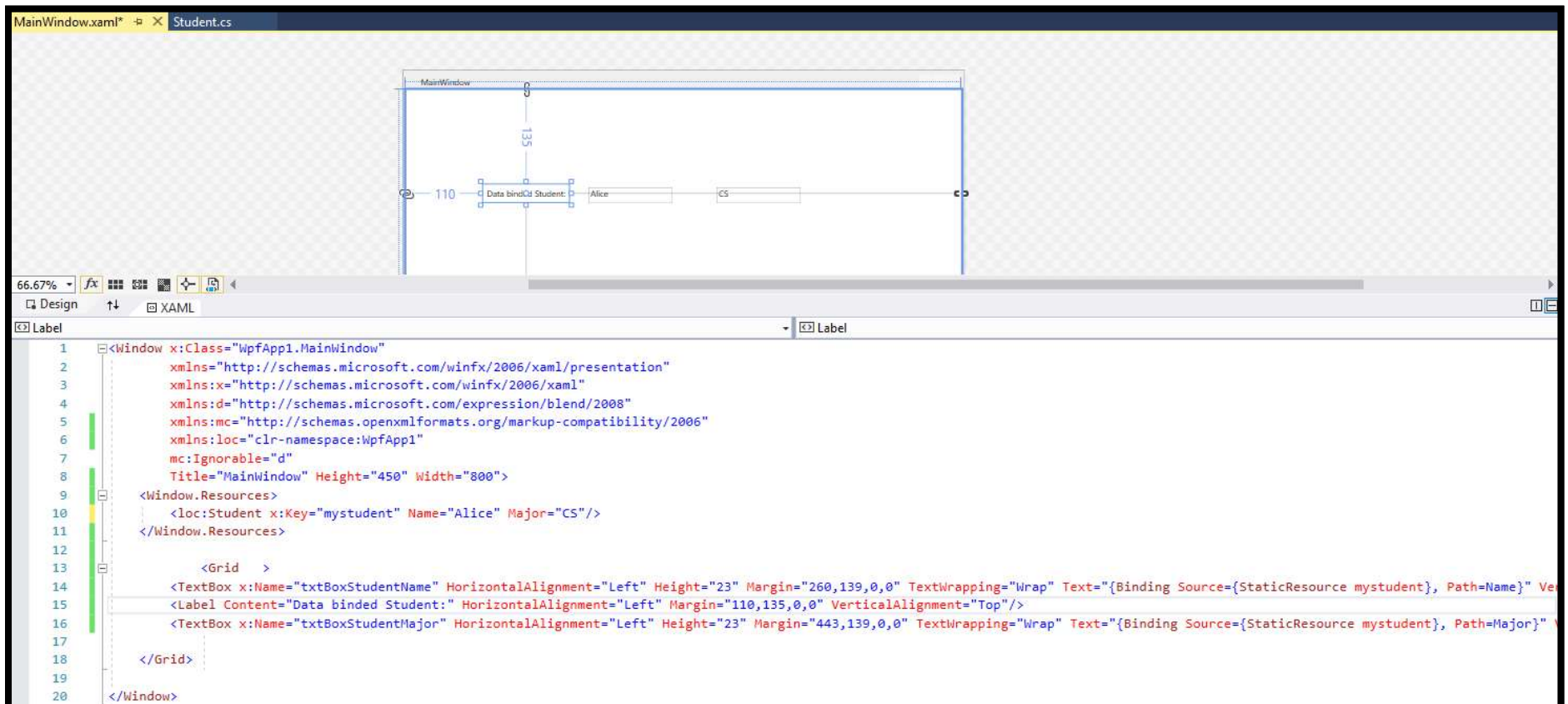
- Another example ...



The screenshot shows the Visual Studio IDE. The main window displays the `Student.cs` file with the following code:

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace WpfApp1
8 {
9     class Student
10     {
11         public string Name { get; set; }
12         public string Major { get; set; }
13     }
14 }
```

The Solution Explorer on the right shows the project structure for 'WpfApp1' (1 project), including files like `App.config`, `App.xaml`, `App.xaml.cs`, `MainWindow.xaml`, `MainWindow.xaml.cs`, and `Student.cs`.



The screenshot shows the Visual Studio IDE with the `MainWindow.xaml` file open. The top view is the Design view, showing a window titled 'MainWindow' with a grid layout. A label 'Data binded Student:' is visible, and a text box contains the value 'Alice'. The bottom view is the XAML view, showing the following code:

```
1 <Window x:Class="WpfApp1.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6       xmlns:loc="clr-namespace:WpfApp1"
7       mc:Ignorable="d"
8       Title="MainWindow" Height="450" Width="800">
9     <Window.Resources>
10         <loc:Student x:Key="mystudent" Name="Alice" Major="CS"/>
11     </Window.Resources>
12
13     <Grid >
14         <TextBox x:Name="textBoxStudentName" HorizontalAlignment="Left" Height="23" Margin="260,139,0,0" TextWrapping="Wrap" Text="{Binding Source={StaticResource mystudent}, Path=Name}" VerticalAlignment="Top"/>
15         <Label Content="Data binded Student:" HorizontalAlignment="Left" Margin="110,135,0,0" VerticalAlignment="Top"/>
16         <TextBox x:Name="textBoxStudentMajor" HorizontalAlignment="Left" Height="23" Margin="443,139,0,0" TextWrapping="Wrap" Text="{Binding Source={StaticResource mystudent}, Path=Major}" VerticalAlignment="Top"/>
17     </Grid>
18 </Window>
```

Binding Controls to Data in Code

- In real-world applications, it is unlikely that your source data will be static
 - It is far more likely that you will **retrieve data at runtime** from a database or a web service. In these scenarios, you **cannot use a static resource** to represent your data. Instead, you must **use code** to specify the data source for any UI bindings at runtime.
- Example: create data binding programmatically for a **TextBlock** element named **textblock1**:
 - Create data binding entirely in code

```
private void mainWindow_Loaded(object sender, RoutedEventArgs e)
{
    // Create a Coffee instance to use as a data source.
    Coffee coffee1 = new Coffee();
    coffee1.Name = "Fourth Coffee Quencher";
    coffee1.Bean = "Arabica";
    coffee1.CountryOfOrigin = "Venezuela";
    coffee1.Strength = 3;
    // Create a Binding object that references the Coffee instance
    Binding coffeeBinding = new Binding();
    coffeeBinding.Source = coffee1;
    coffeeBinding.Path = new PropertyPath("Name");
    // Add the binding to the Text property of the TextBlock.
    textblock1.SetBinding(TextBlock.TextProperty, coffeeBinding);
}
```

Binding Controls to Data in Code and XAML (mixed)

- Create **Path** bindings in **XAML** and set the **DataContext** in **code**

- **XAML:**

```
<StackPanel x:Name="stackCoffee">  
  <TextBlock Text="{Binding Path=Name}" />  
  <TextBlock Text="{Binding Path=Bean}" />  
  <TextBlock Text="{Binding Path=CountryOfOrigin}" />  
  <TextBlock Text="{Binding Path=Strength}" />  
</StackPanel>
```

- **Code**

```
stackCoffee.DataContext = coffee1;
```

Binding Controls to Data in XAML – create static resource

```
private void Grid_Loaded(object sender, RoutedEventArgs e)
{
    Student myStudent = new Student();
    myStudent.Name = "Bob";
    myStudent.Major = "MSSA";

    this.DataContext = myStudent;
}
```

Student.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace WpfApp1
8 {
9     class Student
10     {
11         public string Name { get; set; }
12         public string Major { get; set; }
13     }
14 }
```

Solution Explorer

- Solution 'WpfApp1' (1 project)
- WpfApp1
 - Properties
 - References
 - App.config
 - App.xaml
 - App.xaml.cs
 - MainWindow.xaml
 - MainWindow.xaml.cs
 - Student.cs

MainWindow.xaml

Student.cs

66.67%

Design

XAML

Window

```
1 <Window x:Class="WpfApp1.MainWindow"
2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4       xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5       xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6       xmlns:loc="clr-namespace:WpfApp1"
7       mc:Ignorable="d"
8       Title="MainWindow" Height="450" Width="800">
9     <Grid>
10        <TextBox x:Name="textBoxStudentName" HorizontalAlignment="Left" Height="23" Margin="260,139,0,0" TextWrapping="Wrap" Text="{Binding Path=Name}" VerticalAlignment="Top" Width="120"/>
11        <Label Content="Data bound Student:" HorizontalAlignment="Left" Margin="110,135,0,0" VerticalAlignment="Top"/>
12        <TextBox x:Name="textBoxStudentMajor" HorizontalAlignment="Left" Height="23" Margin="443,139,0,0" TextWrapping="Wrap" Text="{Binding Path=Major}" VerticalAlignment="Top" Width="120"/>
13    </Grid>
14 </Window>
```

MainWindow

Data bound Student: Bob MSSA

Binding Controls to Collections

- WPF includes controls that are designed to render **collections**, such as the **ListBox**, the **ListView**, the **ComboBox**, and the **TreeView** controls.
 - these controls all inherit from the **ItemsControl** hence ... a common approach to data binding
 - To **bind an IEnumerable collection to an ItemsControl instance**, you need to:
 - [in code] Specify the **source data collection** in the **ItemsSource** property of the **ItemsControl** instance.

```
IstCoffees.ItemsSource = coffees;
```
 - [in XAML] Specify the **source property** you want to display in **DisplayMemberPath** property of the **ItemsControl** instance.

```
<ListBox x:Name="IstCoffees" DisplayMemberPath="Name" />
```
- If you want **a control displaying data in a collection to be updated automatically** when items are added or removed, the collection must implement the **INotifyPropertyChanged** interface.
 - This interface defines an event called **PropertyChanged** that the collection can raise after making a change.
 - The .NET Framework includes a class named **ObservableCollection<T>** that provides a generic implementation of **INotifyPropertyChanged**.
- This topic has potential to engender lengthy discussion, but due to time constraints you should try to keep to the main points described in the notes and refer students who require more information to the [additional reading links](#). Subjects such as the **INotifyPropertyChanged** interface and the **ObservableCollection<T>** class are key to building Windows Store applications, and these items are covered in more detail in course 20484A.

Binding Controls to Collections

- Set the **ItemsSource** property to bind to an **IEnumerable** collection

```
IstCoffees.ItemsSource = coffees;
```

- Use the **DisplayMemberPath** property to specify the source field to display

```
<ListBox x:Name="IstCoffees"  
        DisplayMemberPath="Name" />
```


Creating Data Templates

- When you use controls that derive from the **ItemsControl** or **ContentControl** control, you can create a *data template* to specify how your items are rendered.
 - a data template gives you precise control over how your items are rendered and styled.
 - For example, for a **ListBox** control to display a collection of Coffee instances, each one containing several properties, one can use a template
 - This data template uses a simple grid layout. There are more complex templates:
 - <http://go.microsoft.com/fwlink/?LinkID=267833>

```
<ListBox x:Name="lstCoffees" Width="200">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.RowDefinitions>
          <RowDefinition Height="2*" />
          <RowDefinition Height="*" />
          <RowDefinition Height="*" />
          <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <TextBlock Text="{Binding Path=Name}" Grid.Row="0"
          FontSize="22" Background="Black" Foreground="White" />
        <TextBlock Text="{Binding Path=Bean}" Grid.Row="1" />
        <TextBlock Text="{Binding Path=CountryOfOrigin}" Grid.Row="2" />
        <TextBlock Text="{Binding Path=Strength}" Grid.Row="3" />
      </Grid>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Creating Data Templates - Example



```
////////////////////////////////////  
//List<Student> myList = new List<Student>(); //If you don't plan to add or remove items dynamically at runtime, you might as well use a List<T>  
ObservableCollection<Student> myList = new ObservableCollection<Student>();  
Student st1 = new Student();  
st1.Name = "Alice";  
st1.Major = "CS";  
  
Student st2 = new Student();  
st2.Name = "Bob";  
st2.Major = "MSSA";  
  
Student st3 = new Student();  
st3.Name = "Charlie";  
st3.Major = "undecided";  
  
myList.Add(st1);  
myList.Add(st2);  
myList.Add(st3);  
  
lstBoxStudents.ItemsSource = myList;
```

```
<Window x:Class="WpfApp1.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:loc="clr-namespace:WpfApp1"  
  mc:Ignorable="d"  
  Title="MainWindow" Height="450" Width="800">  
  <Grid Loaded="Grid_Loaded" >  
    <TextBox x:Name="txtBoxStudentName" HorizontalAlignment="Left" Height="23" Margin="260,139,0,0" TextWrapping="Wrap" Text="{Binding Path=Name}" VerticalAlignment="Top" ></TextBox>  
    <Label Content="Data binded Student:" HorizontalAlignment="Left" Margin="110,135,0,0" VerticalAlignment="Top"/>  
    <TextBox x:Name="txtBoxStudentMajor" HorizontalAlignment="Left" Height="23" Margin="443,139,0,0" TextWrapping="Wrap" Text="{Binding Path=Major}" VerticalAlignment="Top" ></TextBox>  
    <ListBox x:Name="lstBoxStudents" HorizontalAlignment="Left" Height="100" Margin="260,248,0,0" VerticalAlignment="Top" Width="100">  
      <ListBox.ItemTemplate>  
        <DataTemplate>  
          <Grid>  
            <TextBlock Text="{Binding Path = Name}" />  
          </Grid>  
        </DataTemplate>  
      </ListBox.ItemTemplate>  
    </ListBox>  
  </Grid>  
</Window>
```

Creating Data Templates - Example



```
////////////////////////////////////  
//List<Student> myList = new List<Student>(); //If you don't plan to add or remove items dynamically at runtime, you might as well use a List<T>  
ObservableCollection<Student> myList = new ObservableCollection<Student>();  
Student st1 = new Student();  
st1.Name = "Alice";  
st1.Major = "CS";  
  
Student st2 = new Student();  
st2.Name = "Bob";  
st2.Major = "MSSA";  
  
Student st3 = new Student();  
st3.Name = "Charlie";  
st3.Major = "undecided";  
  
myList.Add(st1);  
myList.Add(st2);  
myList.Add(st3);  
  
lstBoxStudents.ItemsSource = myList;
```

```
<ListBox x:Name="lstBoxStudents" HorizontalAlignment="Left" Height="100" Margin="260,248,0,0" VerticalAlignment="Top" Width="100">  
  <ListBox.ItemTemplate>  
    <DataTemplate>  
      <Grid>  
        <Grid.RowDefinitions>  
          <RowDefinition Height="2*" />  
          <RowDefinition Height="*" />  
        </Grid.RowDefinitions>  
        <TextBlock Text="{Binding Path = Name}" Grid.Row="0" Background="Black" Foreground="White"/>  
        <TextBlock Text="{Binding Path = Major}" Grid.Row="1" />  
      </Grid>  
    </DataTemplate>  
  </ListBox.ItemTemplate>  
</ListBox>
```

Lesson 3: Styling a UI

- Creating Reusable Resources in XAML
- Defining Styles as Resources
- Using Property Triggers
- Creating Dynamic Transformations
- Demonstration: Customizing Student Photographs and Styling the Application Lab

Creating Reusable Resources in XAML

- XAML enables you to create certain elements, such as data templates, styles, and brushes as **reusable resources** that you can apply to multiple controls.
 - Every WPF control has a **Resources** property to which you can add resources.
 - all WPF elements ultimately derive from the **FrameworkElement** class
 - In most cases, you **define resources on the root element in a file**, such as the **Window** element or the **UserControl** element.
 - The resources are then available to the root element and all of its descendants
 - You can also create resources for use across the entire application by **defining them in the App.xaml file**
 - Every WPF application has an **App.xaml** file. It can contain global resources used by all windows and controls in a WPF application. It is also the entry point for the application, and defines which window should appear when an appl. starts.
 - Resources are stored in a dictionary collection of type **ResourceDictionary**.
 - Add an **x:Key** to uniquely identify the resource
 - To **create** a brush as a windows-level resource →

```
<Window.Resources>  
  <SolidColorBrush x:Key="MyBrush" Color="Coral" />  
  ...  
</Window.Resources>
```
 - To **reference** the resource in property values → → →

```
<TextBlock Text="Foreground"  
  Foreground="{StaticResource MyBrush}" />
```

 - To reference a resource, you use the format **{StaticResource [resource key]}**.
 - You can use the resource in any property that accepts values of the same type as the resource
 - you can reference the brush in any property that accepts brush types, such as **Foreground**, **Background**, or **Fill**.
 - Use a resource dictionary to manage large collections of resources
 - A **resource dictionary** is a XAML file with a top-level element of **ResourceDictionary**.

Defining **Styles** as Resources

- **Style** elements enable you to apply a collection of property values to some or all controls of a particular type.
- To create a style, perform the following steps:
 - Add a **Style** element to a resource collection within your application (for example, the **Window.Resources** collection or a resource dictionary).
 - Use the **TargetType** attribute of the **Style** element to specify the type of control you want the style to target (for example, **TextBox** or **Button**).
 - Use the **x:Key** attribute of the **Style** element to enable controls to specify this style. Alternatively, if you omit the **x:Key** attribute your style will apply to all controls of the specified type.
 - Within the **Style** element, use **Setter** elements to apply specific values to specific properties.
- To apply this style to a **TextBlock** control, you need to set the **Style** attribute of the **TextBlock** to the **x:Key** value of the style resource.

```
<Window.Resources>
  <Style TargetType="TextBlock" x:Key="BlockStyle1">
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Background" Value="Black" />
    <Setter Property="Foreground">
      <Setter.Value>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
          <LinearGradientBrush.GradientStops>
            <GradientStop Offset="0.0" Color="Orange" />
            <GradientStop Offset="1.0" Color="Red" />
          </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
  </Style>
  ...
</Window.Resources>
```

```
<TextBlock Text="Drink More Coffee" Style="{StaticResource BlockStyle1}" />
```

Using Property Triggers

- When you create a style in XAML, you can specify **property values that are only applied when certain conditions are true**.
 - For example, you might want to change the font style of a button when the user hovers over it, or you might want to apply a highlighting effect to selected items in a list box.
- Use the **Trigger** element to **identify the condition**
- Use **Setter** elements **apply the conditional changes**
- example shows how to make the text on a button bold while the user is hovering over the button

```
<Window.Resources>
  <Style TargetType="Button">
    <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="FontWeight" Value="Bold" />
      </Trigger>
    </Style.Triggers>
  </Style>
  ...
</Window.Resources>
```


Creating Dynamic Transformations

- **Animations** are sometimes used to make transitions between states less abrupt.
 - For example, if you want to enlarge or rotate a picture, increasing the size or changing the orientation progressively over a short time period can look better than simply switching from one size or orientation to another.
- To create and apply an animation effect in XAML:

- **Create an animation.**

- WPF includes various classes that you can use to create animations in XAML.
- The most commonly used, **DoubleAnimation** element specifies how a value should change over time, by specifying the initial value, the final value, and the duration over which the value should change.

- **Create a storyboard.**

- To apply an animation to an object, you need to wrap your animation in a **Storyboard** element.
- The **Storyboard** element enables you to specify the object and the property you want to animate. It does this by providing the **TargetName** and **TargetProperty** attached properties, which you can set on child animation elements.

- **Create a trigger.**

- To trigger your animation in response to a property change, you need to wrap your storyboard in an **EventTrigger** element.
- The **EventTrigger** element specifies the control event that will trigger the animation. In the **EventTrigger** element, you can use a **BeginStoryboard** element to launch the animation.
- You can add an **EventTrigger** element containing your storyboards and animations to the **Triggers** collection of a Style element, or you can add it directly to the **Triggers** collection of an individual control

```
<Window.Resources>
  <Style TargetType="Image" x:Key="CoffeeImageStyle">
    <Setter Property="Height" Value="200" />
    <Setter Property="RenderTransformOrigin" Value="0.5,0.5" />
    <Setter Property="RenderTransform">
      <Setter.Value>
        <RotateTransform Angle="0" />
      </Setter.Value>
    </Setter>
    <Style.Triggers>
      <EventTrigger RoutedEvent="Image.MouseDown">
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="Height"
              From="200" To="300" Duration="0:0:2" />
            <DoubleAnimation
              Storyboard.TargetProperty="RenderTransform.Angle"
              From="0" To="30" Duration="0:0:2" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```


Module Review and Takeaways

- **Question:** You want to use rows and columns to lay out a UI. Which container control should you use?
 - () Option 1: The Canvas control.
 - () Option 2: The DockPanel control.
 - () Option 3: The Grid control.
 - () Option 4: The StackPanel control.
 - () Option 5: The WrapPanel control.
- **Question:** You are creating an application that enables users to place orders for coffees. The application should allow users to select the drink they want from a list. Each list item should display the name of the coffee, the description, the price, and an image of the coffee. How should you proceed?
 - () Option 1: Create a ListBox control. Add child controls to the ListBox control to represent each field.
 - () Option 2: Create a ListBox control. Use a DataTemplate to specify how each field is displayed within a list item.
 - () Option 3: Create a ListBox control. Create a custom control that inherits from ListBoxItem, and use this custom control to specify how each field is displayed.
 - () Option 4: Create a ListBox control. Use the DisplayMemberPath property to specify the fields you want to display in each list item.
 - () Option 5: Create a ListBox control. Use a Style to specify how each field is displayed within a list item.
- **Question:** You want to apply a highlighting effect to selected items in a ListBox. How should you proceed?
 - () Option 1: Create a Style element and set the TargetType attribute to ListBox. Use a Setter element to apply the highlighting effect.
 - () Option 2: Create a Style element and set the TargetType attribute to ListBox. Use a Trigger element to apply the highlighting effect when a list box item is selected.
 - () Option 3: Create a Style element and set the TargetType attribute to ListBox. Use an EventTrigger element to apply the highlighting effect when a list box item is selected.
 - () Option 4: Create a Style element and set the TargetType attribute to ListBox. Use a Storyboard element to apply the highlighting effect when a list box item is selected.
 - () Option 5: Create a Style element and set the TargetType attribute to ListBox. Use a DoubleAnimation element to apply the highlighting effect when a list box item is selected.