Module 9: Designing the User Interface for a Graphical Application

Contents:

Module overview

An effective and easy-to-use user interface (UI) is essential for graphical applications. In this module, you will learn how to use Extensible Application Markup Language (XAML) and Windows Presentation Foundation (WPF) to create engaging UIs.

Objectives

After completing this module, you will be able to:

- Use XAML to design a UI.
- Bind a XAML control to data.
- Apply styles to a .

Lesson 1: Using XAML to Design a User Interface

XAML is a declarative, XML-based markup language that you can use to create UIs for .NET Framework applications. Defining a UI in declarative markup, rather than imperative code, makes your UI more flexible and portable, ensuring that the same application can be used on a variety of devices. Using XAML also helps to separate your application UI from its runtime logic.

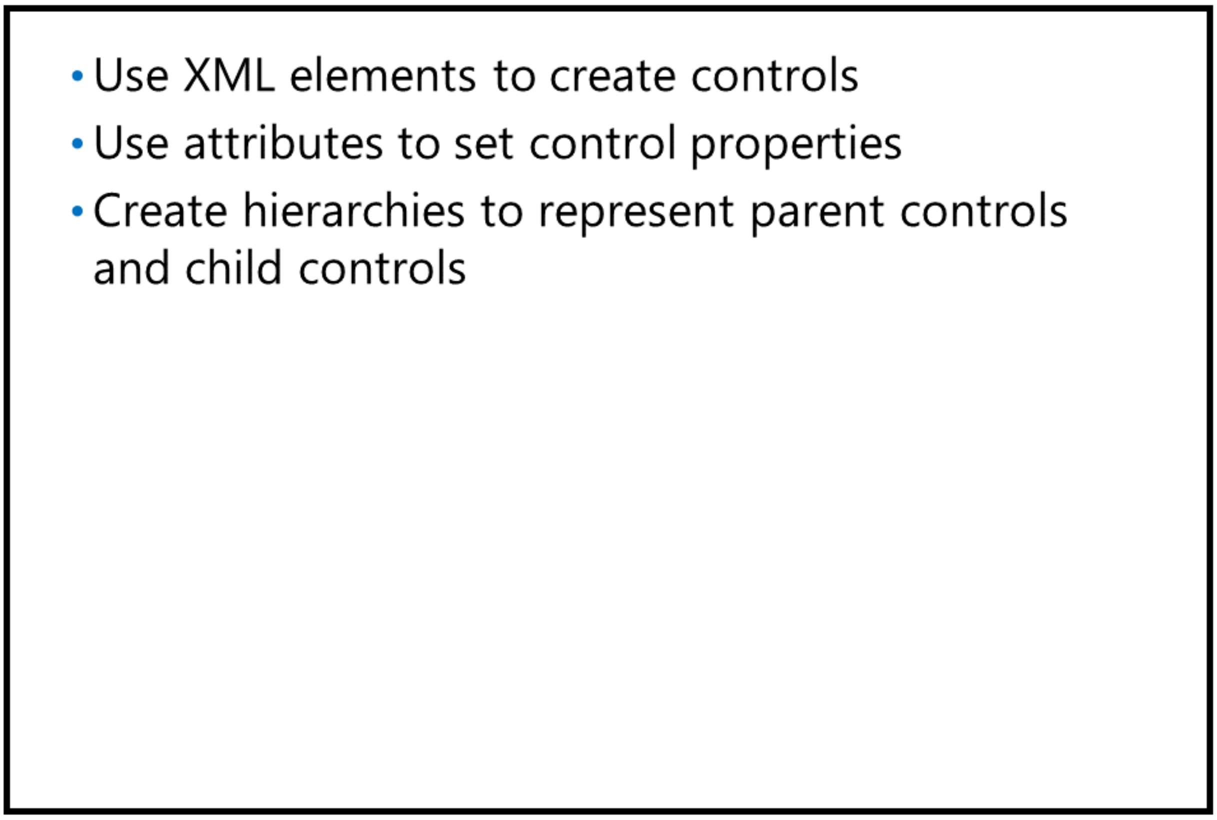
In this lesson, you will learn how to use XAML to create simple graphical UIs.

Lesson objectives

After completing this lesson, you will be able to:

- Explain how to use XAML to define the layout of a UI.
- Describe some of the common controls used by WPF applications.
- Create controls and set properties in XAML.
- Handle events for XAML controls.
- Use layout controls in XAML.
- Create user controls in XAML.

Introducing XAML



- Use XML elements to create controls
- Use attributes to set control properties
- Create hierarchies to represent parent controls and child controls

XAML enables you to define UI elements by using a declarative, XML-based syntax. When you use XAML in a WPF application, you use XML elements and attributes to represent controls and their properties. You can use a variety of tools to create XAML files, such as Visual Studio, the Microsoft Expression® suite, and text editors. When you build a WPF application, the build engine converts the declarative markup in your XAML file into classes and objects.

> **Note:** It's important to note that the XAML language itself is just a mapping of C# classes to declarative code. There's no special engine to treat XAML differently. Everything done with XAML in a declarative way can be achieved with normal C# code.

The following example shows the basic structure of a :

Defining a Button in XAML

```
<Window x:Class="MyNamespace.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                Title="Order Your Coffee Here" Height="350"
Width="525"
```

```
Width="525">
    <Grid>
        <Button Content="Get Me a Coffee!" />
    </Grid>
</Window>
```

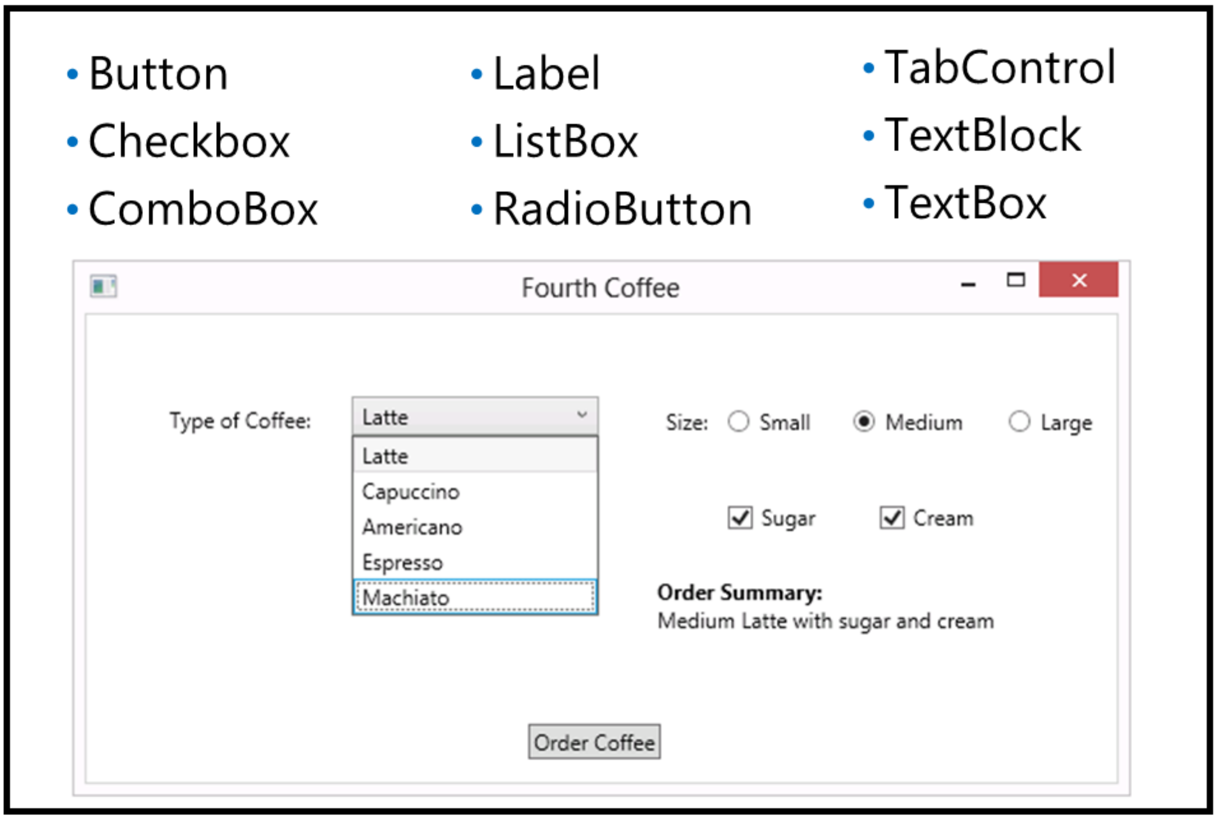
XAML uses a hierarchical approach to define a UI. The most common top-level element in a WPF XAML file is the **Window** element. The **Window** element can include several attributes. In the previous example, the **Window** element identifies various XML namespaces that make built-in controls available for use. It defines a title, which is displayed in the title bar of the application, and defines the initial height and width of the window.

A **Window** element can contain a single child element that defines the content of the UI. In most applications, a UI requires more than a single control, so WPF defines *container* controls that you can use to combine other lower-level controls together and lay them out. The most commonly used container control is the **Grid** control, and when you add a new **Window** to a WPF project in Visual Studio, the **Window** template automatically adds a **Grid** control to the **Window**.

The **Grid** control can contain multiple child controls that it lays out in a grid style. In the example shown above, the **Grid** control contains a single **Button** control. The **Grid** control defines a default layout that consists of a single row and a single column, but you can customize this layout by defining additional rows and columns as attributes of the **Grid**. This mechanism enables you to define cells in the **Grid**, and you can then place controls in specific cells. You can also nest a grid control inside a cell if you need to provide finer control over the layout of certain parts of a window.

Additional Reading: For more information about XAML, refer to the XAML Overview (WPF) page at **http://go.microsoft.com/fwlink/?LinkID=267821**.

Common Controls

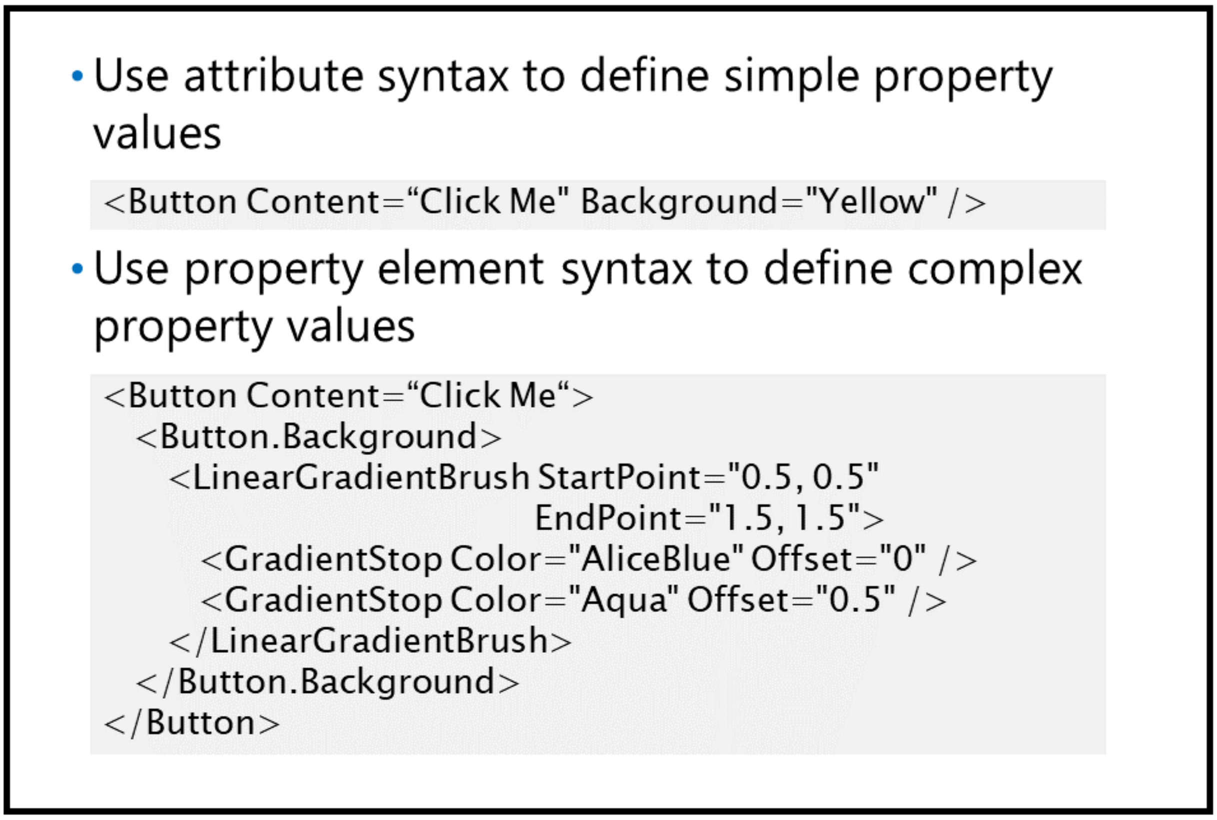


- Button
- Checkbox
- ComboBox
- Label
- ListBox
- RadioButton
- TabControl
- TextBlock
- TextBox

The .NET Framework provides a comprehensive collection of controls that you can use to implement a UI. You can find the complete set in the Toolbox that is available when you design a window. You can also define your own custom user controls, as described in this lesson. The following table summarizes some of the most commonly used 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.

Setting Control Properties



- Use attribute syntax to define simple property values

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

- Use property element syntax to define complex property values

```
<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>
```

When you add controls to a XAML window, you can define control properties in various ways. Most controls enable you to set simple property values by using attributes.

The following example shows how to set the properties of a button control by using attributes:

Using Attributes to Set Control Properties

```
<Button Content="Get Me a Coffee!"
              Background="Yellow"
              Foreground="Blue" />
```

This attribute syntax is easy to use and intuitive for developers with XML experience. However, this syntax does not provide the flexibility that you need to define more complex property values. Suppose that instead of setting the button background to a simple text value such as **Yellow**, you want to apply a more complex gradient effect to the button. You cannot define all the nuances of a gradient effect within an attribute value. Instead, XAML supports an alternative way of setting control properties called *property element syntax*. Rather than specifying the **Background** property as an inline attribute of the **Button** element, 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.

The following example shows how to set the properties of a button control by using property element syntax:

Using Property Element Syntax to Set Control Properties

```
<Button Content="Get Me a Coffee!">
   <Button.Background>
      <="0.5, 0.5" EndPoint="1.5, 1.5" >
         <GradientStop Color="AliceBlue" Offset="0" />
         <GradientStop Color="Aqua" Offset="0.5" />
      </LinearGradientBrush>
   </Button.Background>
   <Button.Foreground>
      <SolidColorBrush Color="" />
   </Button.Foreground>
</Button>
```

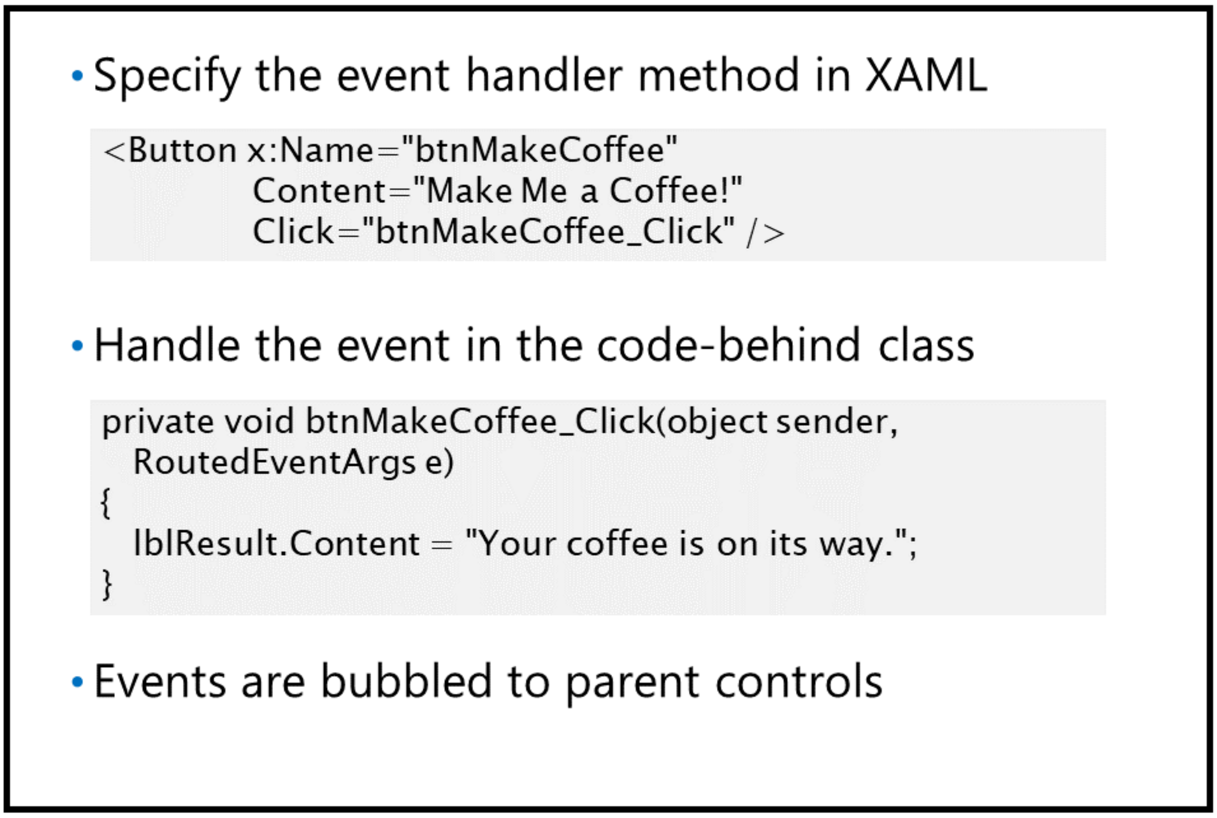
Many WPF controls include a **Content** property. The previous examples used an attribute to set the **Content** property of a button to a simple text value. However, the content of a control is rarely limited to text. Instead of specifying the **Content** property as an attribute, you can add the desired content between the opening and closing tags of the control. For example, you might want to replace the contents of the **Button** element with a picture of a cup of coffee.

The following example shows how to add content to a WPF control:

Adding Content to a WPF Control

```
<Button >
   <Image Source="Images/coffee.jpg" Stretch="Fill" />
</Button>
```

Handling Events



- Specify the event handler method in XAML

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

- 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

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.

Note: Visual Studio includes many features that make it easy to create handlers for events. For example, 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.

Suppose you create a simple application that consists of a button and a label. When you click the button, you want an event handler to add some text to the label. To do this, you need to do three things:

1. Make sure the button and the label include a **Name** property, so that you can reference the controls in your code.

2. Set the **Click** attribute of the button to the name of an event handler method. This method runs when the **Click** event occurs.

Note: The name of a control should be unique within the window that holds the control; no two controls in the same window should have the same name.

The following code example shows how to set the event handler method for the **Click** event of a **Button** control:

Handling Events in XAML

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

In the code-behind file, you can add logic to the **btnMakeCoffee_Click** method to define what should happen when a user clicks the button.

The following example shows how to create an event handler method for a WPF control:

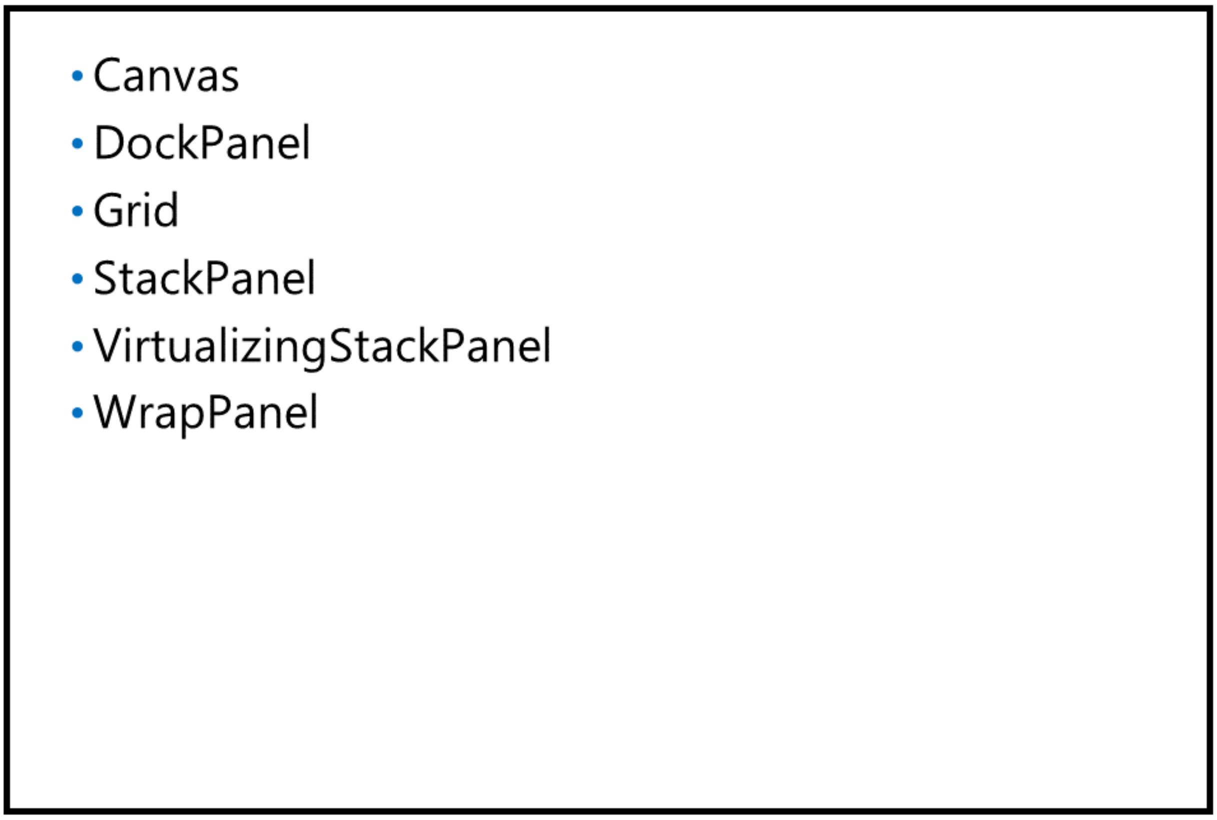
Creating Event Handler Methods

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

Note: WPF uses the concept of “routed events”. 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.

Additional Reading: For more information about how routed events work in WPF, refer to the Routed Events Overview page at **http://go.microsoft.com/fwlink/?LinkID=267822**.

Using Layout Controls


- Canvas
- DockPanel
- Grid
- StackPanel
- VirtualizingStackPanel
- WrapPanel

Support for relative positioning is one of the core principles of WPF. The idea is that your application should render correctly regardless of how the user positions or resizes the application window. WPF includes several layout, or container, controls that enable you to position and size your child controls in different ways. The following table shows the most common layout controls:

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.

Control	Description
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.

The following example shows how to define a grid with two rows and two columns:

Using a Grid Layout

```
<Grid>
   <Grid.RowDefinitions>
      <="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>
```

In the previous example, notice that you use **RowDefinition** and **ColumnDefinition** elements to define rows and columns respectively. For each row or column, you can specify a minimum and maximum height or width, or a fixed height or width. You can specify heights and widths in three ways:

- As numerical units. For example, **Width="200"** represents 200 units (where 1 unit equals 1/96th of an inch).

- 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 up 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.

To put a child control in a particular row or column, you add the **Grid.Row** and **Grid.Column** attributes to the child control element. Note that these properties are defined by the parent **Grid** element, rather than the child **Label** element. Properties such as **Grid.Row** and **Grid.Column** are known as attached properties—they are defined by a parent element to be specified by its child elements. Attached properties enable child elements to tell a parent element how they should be rendered.

Additional Reading: For more information about layout controls, refer:
- **https://aka.ms/moc-20483c-m9-pg1**.
- The DockPanel Class page at **https://aka.ms/moc-20483c-m9-pg2**.
- The Grid Class page at **https://aka.ms/moc-20483c-m9-pg3**.
- The StackPanel Class page at **https://aka.ms/moc-20483c-m9-pg4**.
- The VirtualizingStackPanel Class page at **https://aka.ms/moc-20483c-m9-pg5**.
- The WrapPanel Class page at **https://aka.ms/moc-20483c-m9-pg6**.

Demonstration: Using Design View to Create a XAML UI

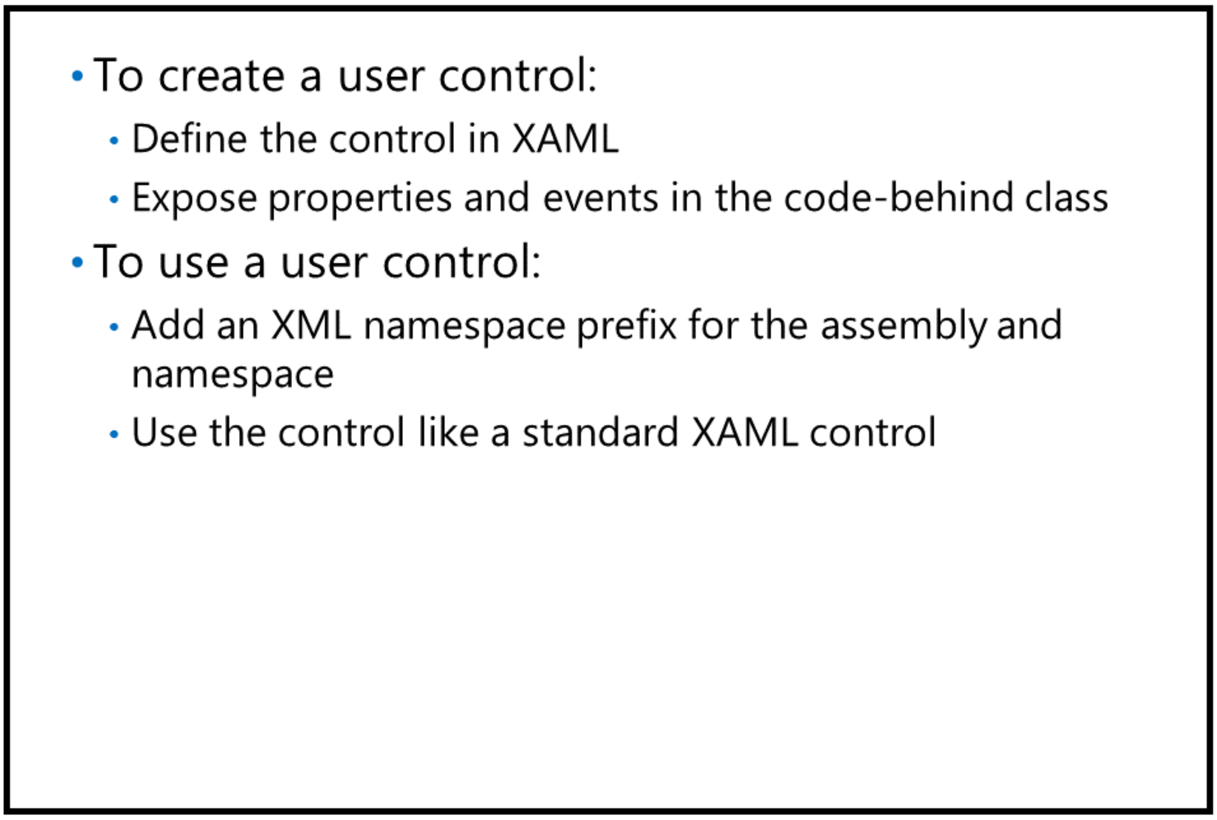
In this demonstration, you will create a simple WPF application that contains a button and a label. When you click the button, an event handler will update the text on the label.

The demonstration illustrates various ways in which you can create and configure XAML files in Visual Studio. It illustrates how you can add controls to the design surface by double-clicking a control in the toolbox. It shows how you can configure controls through a combination of using designer tools and editing XAML markup directly. It also illustrates how Visual Studio can automatically connect event handlers to your controls.

Demonstration steps

You will find the steps in the **Demonstration: Using Design View to Create a XAML UI** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_DEMO.md**.

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

When you work with WPF, you can create your own self-contained, reusable user controls in XAML. User controls are sometimes called composite controls, because they are a composite of other controls. For example, if you use the same combination of a text box and a button multiple times in a UI, it might be more convenient to create and use a user control that consists of the text box and the button. Alternatively, you might 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.

The following example shows a user control that enables people to select and order beverages:

Creating a User Control in XAML

```
<UserControl x:Class="DesignView.CoffeeSelector"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="200">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="2*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <StackPanel Grid.Row="0">
            <Label Content="Do you want coffee or tea?"/>
            <RadioButton x:Name="radCoffee" Content="Coffee"
HorizontalAlignment="Left"
                   VerticalAlignment="Top" Margin="5"
GroupName="Beverage"
                   IsChecked="True" Checked="radCoffee_Checked" />
            <RadioButton x:Name="radTea" Content="Tea"
HorizontalAlignment="Left"
                    VerticalAlignment="Top" Margin="5"
GroupName="Beverage"
                    Checked="radTea_Checked"/>
        </StackPanel>
        <StackPanel Grid.Row="1">
            <Label Content="Do you want milk?"/>
            <RadioButton x:Name="radMilk" Content="Milk"
HorizontalAlignment="Left"
                  VerticalAlignment="Top" Margin="5"
GroupName="Milk"
                  IsChecked="True" Checked="radMilk_Checked" />
            <RadioButton x:Name="radNoMilk" Content="No Milk"
HorizontalAlignment="Left"
                   VerticalAlignment="Top" Margin="5 "
GroupName="Milk"
                  Checked="radNoMilk_Checked"/>
        </StackPanel>
        <StackPanel Grid.Row="2">
            <Label Content="Do you want sugar?"/>
            <RadioButton x:Name="radSugar" Content="Sugar"
HorizontalAlignment="Left"
                    VerticalAlignment="Top" Margin="5"
GroupName="Sugar"
                    IsChecked="True" Checked="radSugar_Checked" />
            <RadioButton x:Name="radNoSugar" Content="No Sugar"
HorizontalAlignment="Left"
                    VerticalAlignment="Top" Margin="5"
GroupName="Sugar"
                    Checked="radNoSugar_Checked"/>
        </StackPanel>
        <Button x:Name="btnOrder" Content="Place Order" Margin="5"
Grid.Row="3"
              Click="btnOrder_Click"/>
    </Grid>
</UserControl>
```

As the previous example shows, creating the XAML for a user control is very similar to creating the XAML for a window. 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. When you edit the code-behind file, you should also:

- Define any required public properties. Creating public properties enables the consumers of your user control to get or set property values, either in XAML or in code.
- Define any required public events. Raising events enables consumers of your user control to respond in the same way that they would respond to other control events, such as the **Click** event of a button.

The following example shows the code-behind class for a user control:

Creating the Code-Behind Class for a User Control

```
public partial class CoffeeSelector : UserControl
{
   public CoffeeSelector()
   {
      InitializeComponent();
   }
   private string beverage;
   private string milk;
   private string sugar;
   public string Order
   {
      get
      {
         return $"{beverage}, {milk}, {sugar}";
      }
   }
   public event EventHandler<EventArgs> OrderPlaced;
   private void btnOrder_Click(object sender, RoutedEventArgs e)
   {
      if(OrderPlaced!=null)
```

```
            OrderPlaced(this, EventArgs.Empty);
        }
        private void radCoffee_Checked(object sender, RoutedEventArgs e)
    { beverage = "Coffee"; }
        private void radTea_Checked(object sender, RoutedEventArgs e) {
    beverage = "Tea"; }
        private void radMilk_Checked(object sender, RoutedEventArgs e) {
    milk = "Milk"; }
        private void radNoMilk_Checked(object sender, RoutedEventArgs e)
    { milk = "No Milk"; }
        private void radSugar_Checked(object sender, RoutedEventArgs e) {
    sugar = "Sugar"; }
        private void radNoSugar_Checked(object sender, RoutedEventArgs e)
    { sugar = "No Sugar"; }
    }
```

Note: In this example, you will notice a string usage—**$** strings, or string interpolation— that was not discussed before. Using **$** before a string allows you to use the string like a **String.Format** method, except that you don't specify the location of the values. Rather, you provide the values directly in the string itself. You can place anything in a string interpolation. Besides regular variables, you can use the results of calculations or even method return.
For more, refer: $ - string interpolation (C# Reference) **https://aka.ms/moc-20483c-m9-pg9**.

In the previous code example, the user control raises an **OrderPlaced** event when the user clicks the **Place Order** button. Applications that include the user control can subscribe to this event and take appropriate action.

To use your user control in a WPF application, you need to do two things:

1. 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.

2. Add the control to your application in the same way that you would add a built-in control, with the namespace prefix you defined.

The following example shows how to add a user control in XAML:

Adding a User Control to a WPF Application

```
<Window x:Class="DesignView.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:coffee="clr-namespace:DesignView"
        Title="Order Your Coffee Here" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="8*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <coffee:CoffeeSelector x:Name="coffeeSelector1" Grid.Row="0"
               OrderPlaced="coffeeSelector1_OrderPlaced"  />
        <Label x:Name="lblResult" Margin="5" Grid.Row="1" />
    </Grid>
</Window>
```

When you have added the user control, you can handle events and get or set property values in the same way that you would for a built-in control. In the previous example, the **OrderPlaced** event of the **CoffeeSelector** control is wired up to the **coffeeSelector1_OrderPlaced** method.

The following example shows how to interact with a user control in a code-behind class:

Programming with User Controls

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
   }
   private void coffeeSelector1_OrderPlaced(object sender, EventArgs
e)
   {
      lblResult.Content = coffeeSelector1.Order;
   }
}
```

In the previous code example, the **coffeeSelector1_OrderPlaced** method handles the **OrderPlaced** event of the **CoffeeSelector** control. The method then retrieves the order details from the control and writes them to a label.

Lesson 2: Binding Controls to Data

Most applications work with data in one form or another. The data that drives your application can come from a wide variety of sources, such as files, databases, or web services. Almost every graphical application needs to connect UI controls to an underlying data source so that users can retrieve, enter, or edit information.

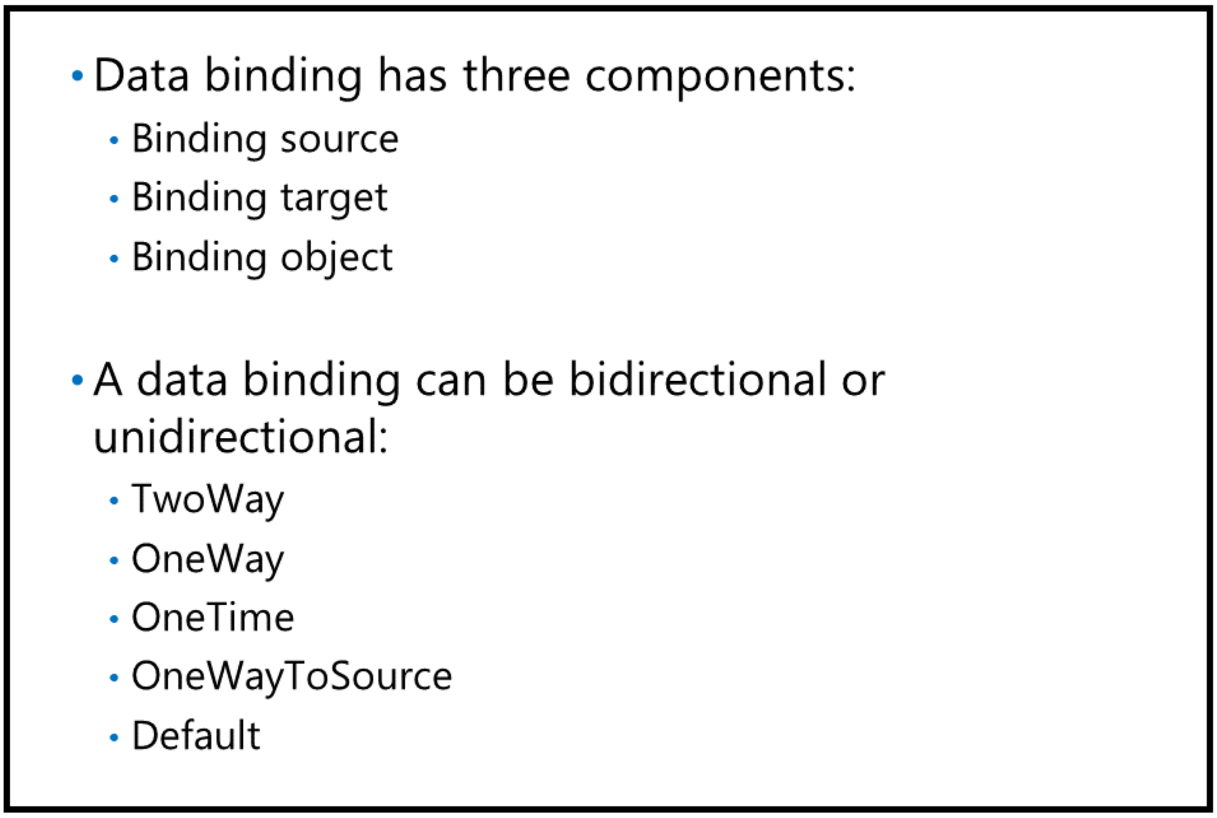
In this lesson, you will learn how to bind controls to data in WPF applications.

Lesson objectives

After completing this lesson, you will be able to:

- Describe how data binding works in WPF.
- Use XAML to bind controls to data.
- Use code to bind controls to data.
- Bind controls to collections of data.
- Create data templates to specify how data is rendered.

Introduction to Data Binding



- Data binding has three components:
  - Binding source
  - Binding target
  - Binding object
- A data binding can be bidirectional or unidirectional:
  - TwoWay
  - OneWay
  - OneTime
  - OneWayToSource
  - Default

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. Conceptually, data binding consists of three components:

- The *binding source*. This is the source of your data, and is typically a property of a custom .NET object. For example, you might bind a control to the **CountryOfOrigin** property of a **Coffee** object.
- The *binding target*. This is 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*. For example, you might bind data to the **Content** property of a **TextBox** element.

- The *binding object*. This is 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.

Note: 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. Most built-in UI elements implement dependency properties and will support data binding. For more information about dependency properties, refer to the Dependency Properties Overview page at **http://go.microsoft.com/fwlink/?LinkID=267829**.

The following example shows a simple data binding expression:

Simple Data Binding

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

In this example, the **Text** property of a **TextBlock** is set to a data binding expression. Note that:

- 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.

As a result of this expression, the **TextBlock** will always display the value of the **Bean** property of the **coffee1** object. If the value of the **Bean** property changes, the **TextBlock** will update automatically. In terms of the concepts described at the start of this topic:

- 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.

You can also configure the direction of the data binding. For example, you might want to update the UI when the source data is changed or you might want to update the source data when the user edits a value in the UI. To specify the direction of the binding, you set the **Mode** property of the **Binding** object. The **Mode** property can take the following values:

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.

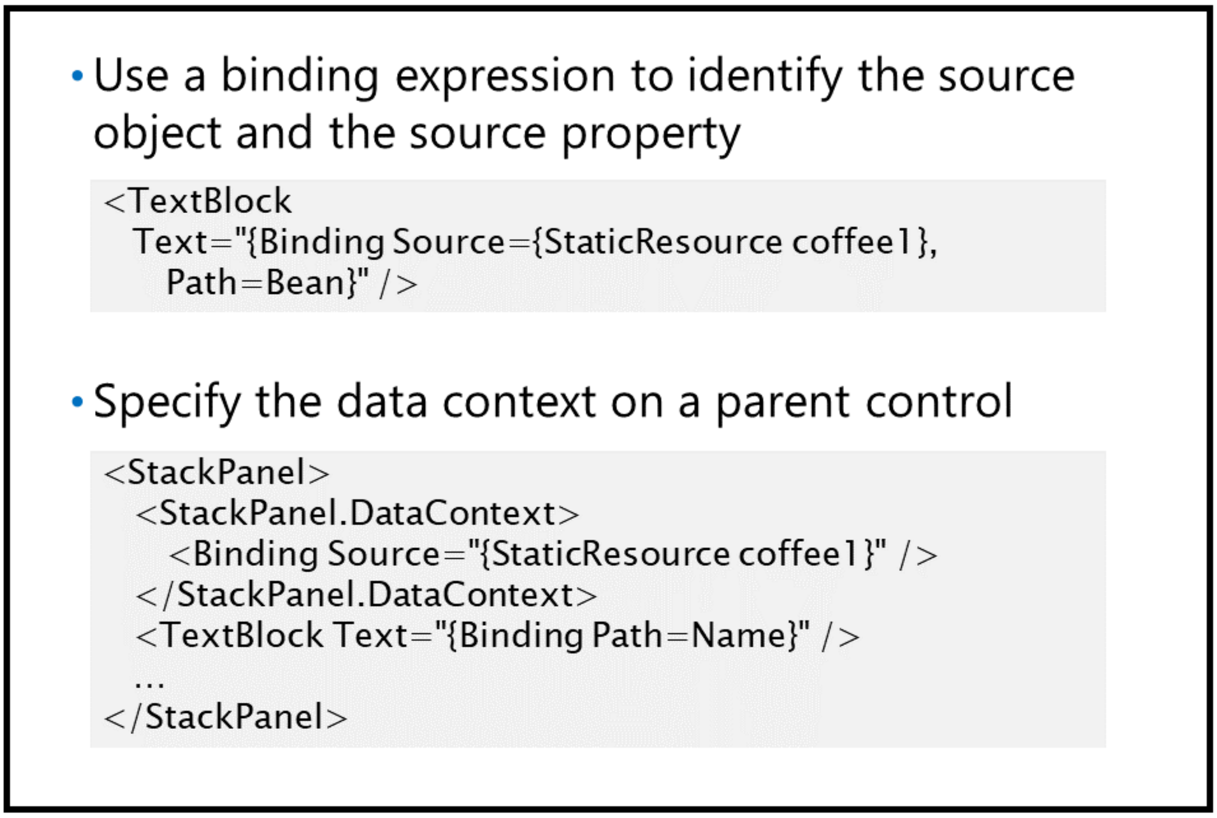
The following example shows how to configure a text box to use two-way data binding:

Specifying the Binding Direction

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

Additional Reading: For more information about the concepts of data binding, refer to the Data Binding Overview page at **http://go.microsoft.com/fwlink/?LinkID=267830**.

Binding Controls to Data in XAML


- Use a binding expression to identify the source object and the source property

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

- Specify the data context on a parent control

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

You can bind controls to data in various ways. 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. For example, if your assembly contains a class named **Coffee**, you can define a static resource to create an instance of **Coffee** and set properties on it. To create a static resource, you must:

- 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.

The following example shows how to create a static resource for data binding:

Creating a Static Resource

```
<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"
                         ="Arabica"
                         CountryOfOrigin="Brazil"
                         Strength="3" />
      …
   </Window.Resources>
   …
</Window>
```

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

The following example shows how to bind an individual item to a static resource:

Binding an Individual Item to a Static Resource

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

More commonly, you will want to bind multiple UI elements to different properties of a data source. In this case, you can set the **DataContext** property on a container object, such as a **Grid** or a **StackPanel**. Setting a **DataContext** property is similar to providing a partial data binding expression. It 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. This means that when you create data binding expressions for child controls, you can omit the source and simply specify the path to the property of interest.

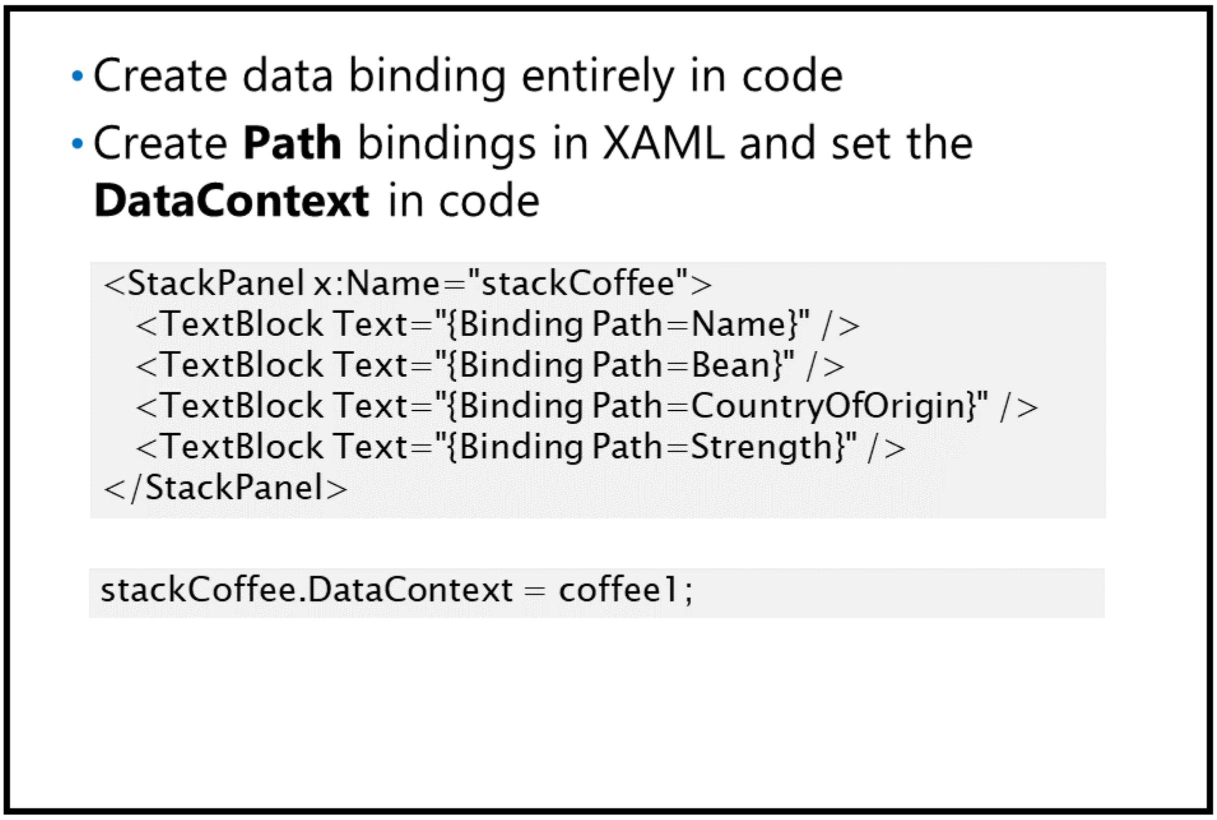
The following example shows how to set a data context for a set of child controls:

Specifying a Data Context

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

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.

Binding Controls to Data in Code



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

```
<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>
```

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

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.

In many cases you can use a mixture of XAML binding and code binding. For example, you might know that your UI elements will be bound to a Coffee instance at design time. In this case, you can specify the binding paths in XAML, and then set a **DataContext** property in code.

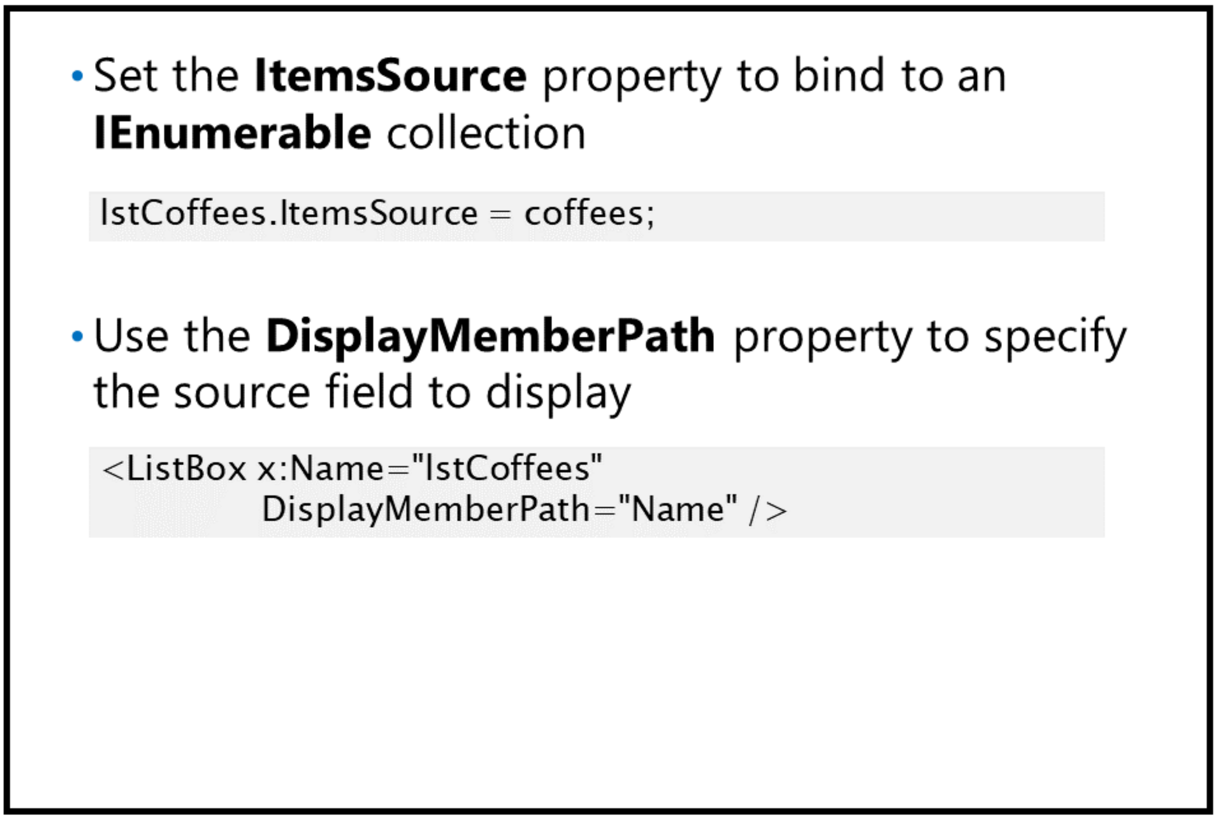
The following example shows how to specify binding paths in XAML:

Specifying Binding Paths in XAML

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

In this example, you have set the binding path for each individual text block in XAML. To complete the binding, all you need to do is to set the **DataContext** property of the parent **StackPanel** object to the **Coffee** instance that you want to display.

Binding Collections to Control



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

```
lstCoffees.ItemsSource = coffees;
```

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

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

In many scenarios, you will want to data bind a control to a collection of objects. WPF includes controls that are designed to render collections, such as the **ListBox** control, the **ListView** control, the **ComboBox** control, and the **TreeView** control. These controls all inherit from the **ItemsControl** class and, as such, support a common approach to data binding.

To bind a collection to an **ItemsControl** instance, you need to:

- Specify the source data collection in the **ItemsSource** property of the **ItemsControl** instance.
- Specify the source property you want to display in the **DisplayMemberPath** property of the **ItemsControl** instance.

You can bind an **ItemsControl** instance to any collection that implements the **IEnumerable** interface. You can set the **ItemsSource** and **DisplayMemberPath** properties in XAML or in code. One common approach is to define the **DisplayMemberPath** property (or a data template) in XAML, and then to set the **ItemsSource** programmatically at runtime.

The following code example shows how to set the **DisplayMemberPath** property in XAML:

Setting the DisplayMemberPath Property

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

Having set the **DisplayMemberPat**h property in XAML, you can now set the **ItemsSource** property in code to complete the data binding.

The following example shows how to set the **ItemsSource** property in code:

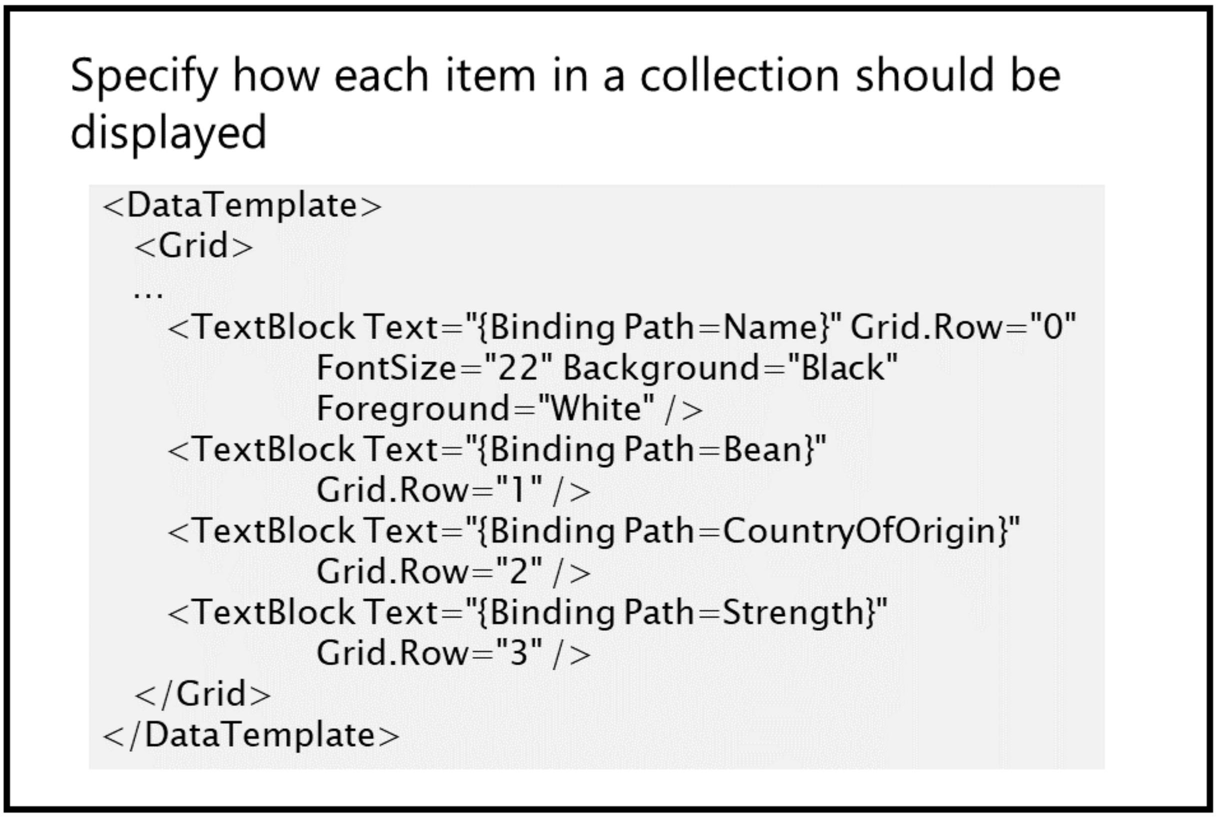
Setting the ItemsSource Property

```
// Create some Coffee instances.
var coffee1 = new Coffee("Fourth Coffee Quencher");
var coffee2 = new Coffee("Espresso Number Four");
var coffee3 = new Coffee("Fourth Refresher");
var coffee3 = new Coffee("Fourth Frenetic");
// Add the items to an observable collection.
var coffees = new ObservableCollection<Coffee>();
coffees.Add(coffee1);
coffees.Add(coffee2);
coffees.Add(coffee3);
coffees.Add(coffee4);
// Set the ItemsSource property of the ListBox to the coffees
collection.
lstCoffees.ItemsSource = coffees;
```

Note: 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**.
The control that binds to the collection receives the event, and can use it to refresh the data that it is displaying. Many of the WPF container controls catch and handle this event automatically.

Additional Reading: For more information about the **ObservableCollection<T>** class, refer to the ObservableCollection(T) class at **https://aka.ms/moc-20483c-m9-pg7**.
For more information about the **INotifyPropertyChanged** interface, refer to the INotifyPropertyChanged Interface page at **https://aka.ms/moc-20483c-m9-pg8**.

Creating Data Templates



Specify how each item in a collection should be displayed

```
<DataTemplate>
  <Grid>
  ...
    <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>
```

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. For example, suppose you want to use a **ListBox** control to display a collection of **Coffee** instances. Each **Coffee** instance includes several properties to represent the name of the coffee, the type of coffee bean, the country of origin, and the strength of the coffee. The data template specifies how each **Coffee** instance should be rendered, by mapping properties of the **Coffee** instance to child controls within the data template. Creating a data template gives you precise control over how your items are rendered and styled.

The following example shows how to define a data template for a **ListBox** control:

Creating a Data Template

```
<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=""
/>
            <TextBlock Text="{Binding Path=}" Grid.Row="1" />
            <TextBlock Text="{Binding Path=CountryOfOrigin}"
Grid.Row="2" />
            <TextBlock Text="{Binding Path=Strength}" Grid.Row="3"
/>
         </Grid>
      </DataTemplate>
   </ListBox.ItemTemplate>
</ListBox>
```

When you set the **ItemsSource** property of this **ListBox** to a collection of **Coffee** objects, the data template specifies how the **ListBox** should render each **Coffee** object. In this example, the data template uses a simple grid layout and displays the name of the coffee with a larger font and a contrasting background. However, you can make your data templates as complex and sophisticated as you want.

Additional Reading: For more information about data templates in WPF, refer to the Data Templating Overview page at **http://go.microsoft.com/fwlink/?LinkID=267833**.

Lesson 3: Styling a UI

If you had to style a graphical application by setting properties on one control at a time, the development process would quickly become laborious. It would also be difficult to maintain your application, as you would often need to make the same change in multiple locations. XAML enables you to define styles as reusable resources that you can apply to multiple controls.

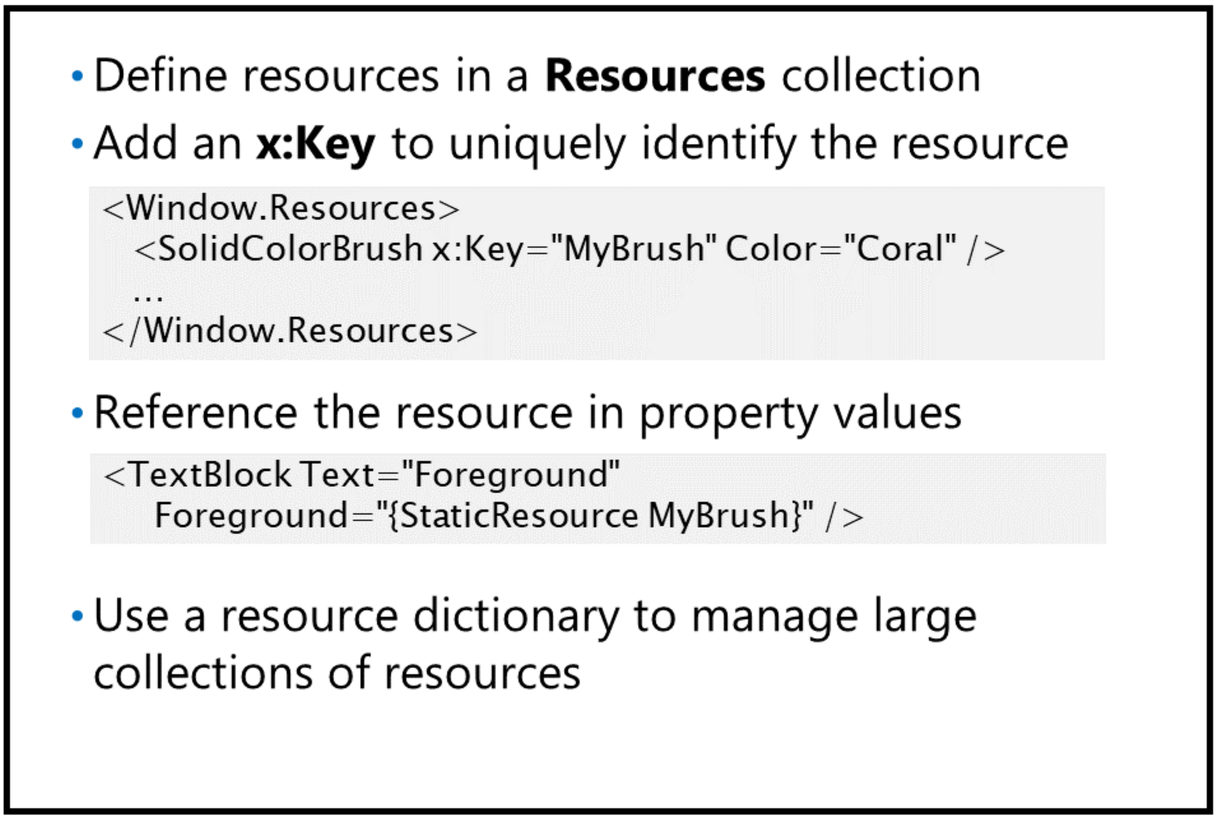
In this lesson, you will learn how to use styles and animations.

Lesson objectives

After completing this lesson, you will be able to:

- Create reusable resources in XAML.
- Define styles that apply to multiple controls.
- Use property triggers to apply styles when conditions are met.
- Use animations to create dynamic effects and transformations.

Creating Reusable Resources in XAML



- Define resources in a **Resources** collection
- Add an **x:Key** to uniquely identify the resource

```
<Window.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Coral" />
  ...
</Window.Resources>
```

- Reference the resource in property values

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

- Use a resource dictionary to manage large collections of resources

XAML enables you to create certain elements, such as data templates, styles, and brushes, as reusable resources. This has various advantages when you are developing a graphical application:

- You can define a resource once and use it in multiple places.
- You can edit a resource without editing every element that uses the resource.
- Your XAML files are shorter and easier to read.

Every WPF control has a **Resources** property to which you can add resources. This is because the **Resources** property is defined by the **FrameworkElement** class from which all WPF elements ultimately derive. 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.

Note: Every WPF application has an App.xaml file. It is a XAML file that 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 application starts.

Resources are stored in a dictionary collection of type **ResourceDictionary**. When you create a reusable resource, you must give it a unique key by providing a value for the **x:Key** attribute.

The following example shows how to create a brush as a window-level resource:

Creating and Using Resources

```
<Window x:Class="DataBinding.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                Title="Reusable Resources" Height="350" Width="525">
   <Window.Resources>
       <SolidColorBrush x:Key="MyBrush" Color="Coral" />
      …
   </Window.Resources>
   …
</Window>
```

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, provided that the resource is in scope. For example, if you create a brush as a resource, you can reference the brush in any property that accepts brush types, such as **Foreground**, **Background**, or **Fill**.

The following example shows how to reference a reusable resource in multiple places:

Referencing a Reusable Resource

```
<StackPanel>
   <Button Content="Click Me" Background="{}" />
   <TextBlock Text="Foreground" Foreground="{}" />
   <TextBlock Text="Background" Background="{}" />
   <Ellipse Height="50" Fill="{}" />
</StackPanel>
```

If you need to create several reusable resources, it can be useful to create your resources in a separate resource dictionary file. A resource dictionary is a XAML file with a top-level element of **ResourceDictionary**. You can add reusable resources within the **ResourceDictionary** element in the same way that you would add them to a **Window.Resources** element.

Note: You can create a WPF Resource Dictionary from the **Add New Item** menu in Visual Studio.

In most cases, you make a resource dictionary available for use in your application by referencing it in the application-scoped App.xaml file. The following example shows how to reference a resource dictionary in the App.xaml file:

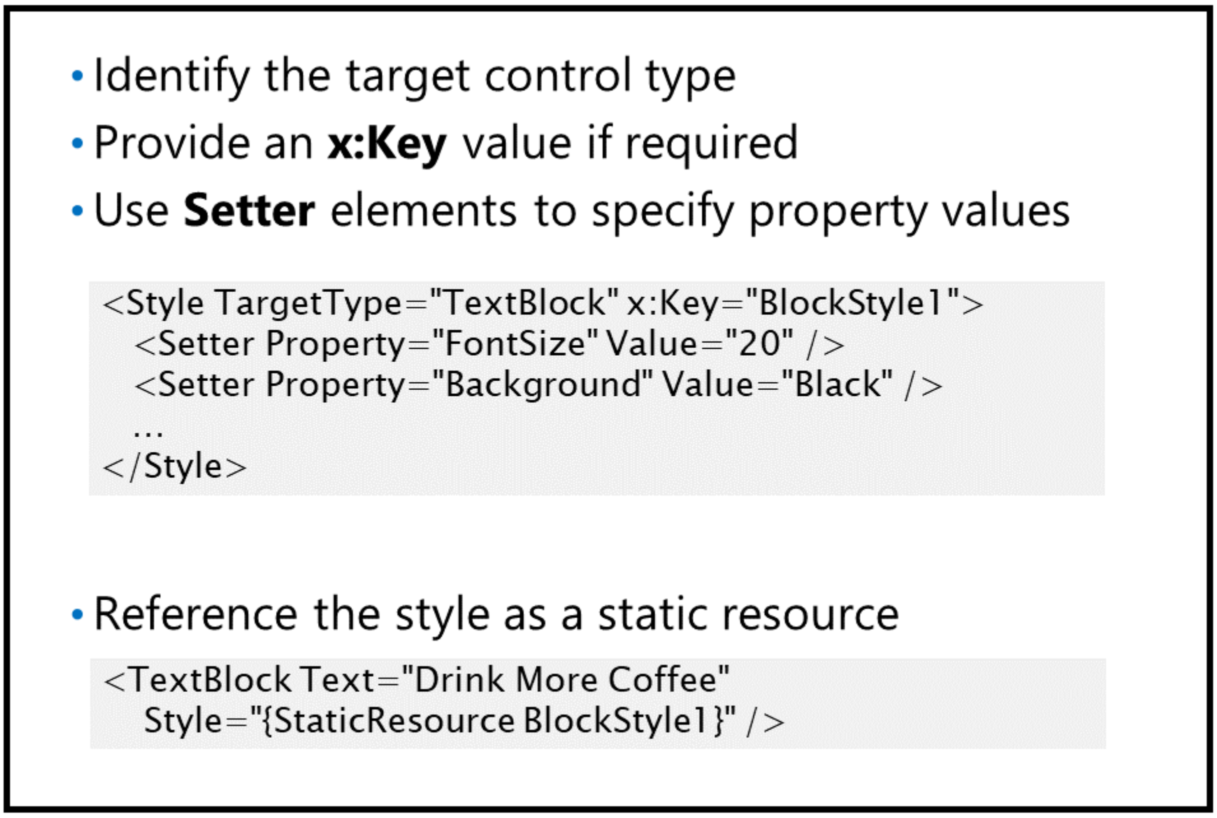
Referencing a Resource Dictionary

```
<Application x:Class="ReusableResources.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                     StartupUri="MainWindow.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary
Source="FourthCoffeeResources.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

Defining Styles as Resources


- Identify the target control type
- Provide an **x:Key** value if required
- Use **Setter** elements to specify property values

```
<Style TargetType="TextBlock" x:Key="BlockStyle1">
  <Setter Property="FontSize" Value="20" />
  <Setter Property="Background" Value="Black" />
  ...
</Style>
```

- Reference the style as a static resource

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

In many cases, you will want to apply the same property values to multiple controls of the same type within an application. For example, if a page contains five textboxes, you will probably want each textbox to have the same foreground color, background color, font size, and so on. To make this consistency easier to manage, you can create **Style** elements as resources in XAML. **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:

1. Add a **Style** element to a resource collection within your application (for example, the **Window.Resources** collection or a resource dictionary).

2. 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**).

3. Use the **x:Key** attribute of the **Style** element to enable controls to specify this

style. Alternatively, you can omit the **x:Key** attribute and your style will apply to all controls of the specified type.

4. Within the **Style** element, use **Setter** elements to apply specific values to specific properties.

The following example shows how to create a style that targets **TextBlock** controls:

Creating Styles

```
<Window.Resources>
    <Style TargetType="TextBlock" x:Key="BlockStyle1">
        <Setter Property="FontSize" Value="20" />
        <Setter Property="Background" Value="" />
        <Setter Property="Foreground">
            <Setter.Value>
                <="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>
```

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.

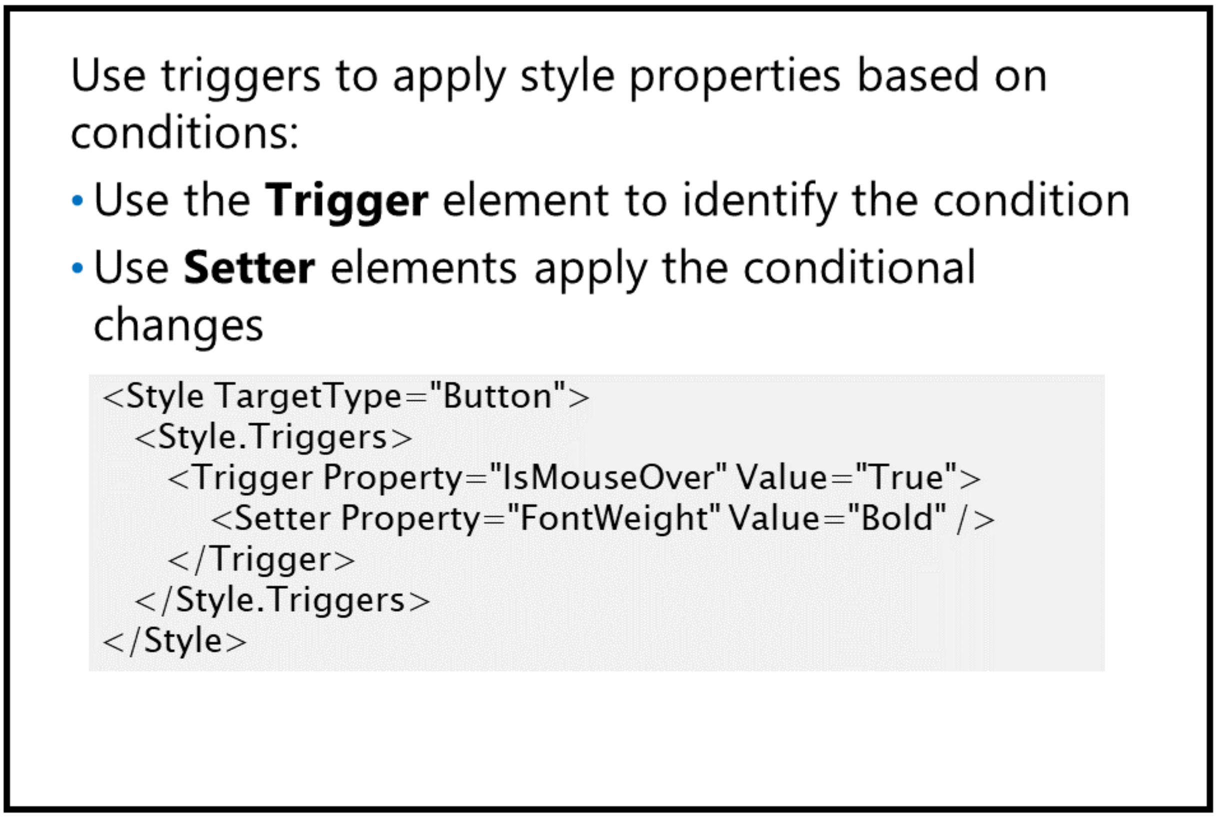
The following example shows how to apply a style to a control:

Applying a Style

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

Additional Reading: For more information about defining styles, refer to the Styling and Templating page at **http://go.microsoft.com/fwlink/?LinkID=267834**.

Using Property Triggers



Use triggers to apply style properties based on conditions:

- Use the **Trigger** element to identify the condition
- Use **Setter** elements apply the conditional changes

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

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.

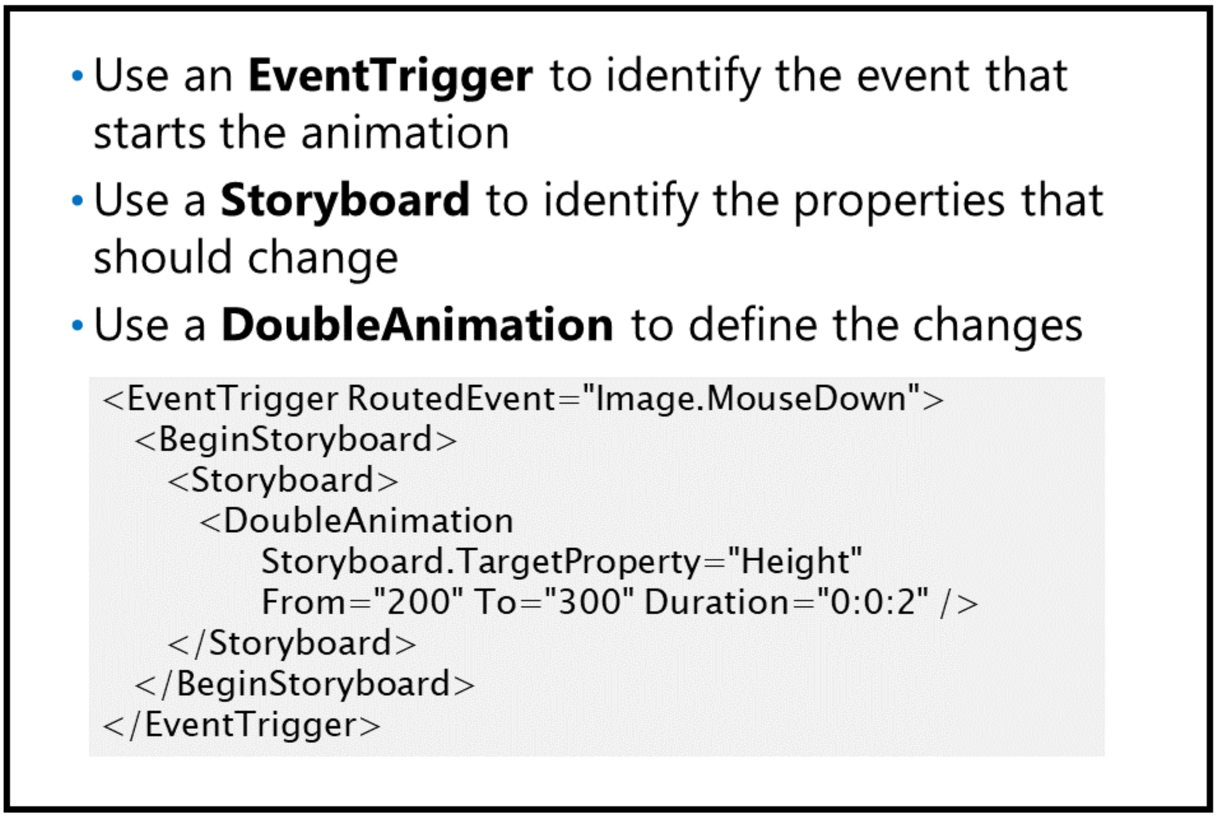
To apply style properties based on conditions, you add **Trigger** elements to your styles. The **Trigger** element identifies the property of interest and the value that should trigger the change. Within the **Trigger** element, you use **Setter** elements to apply changes to property values.

The following example shows how to make the text on a button bold while the user is hovering over the button:

Using a Property Trigger

```
<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



- Use an **EventTrigger** to identify the event that starts the animation
- Use a **Storyboard** to identify the properties that should change
- Use a **DoubleAnimation** to define the changes

```
<EventTrigger RoutedEvent="Image.MouseDown">
  <BeginStoryboard>
    <Storyboard>
      <DoubleAnimation
          Storyboard.TargetProperty="Height"
          From="200" To="300" Duration="0:0:2" />
    </Storyboard>
  </BeginStoryboard>
</EventTrigger>
```

Sophisticated graphical applications often use animations to make the user experience more engaging. 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, you need to do three things:

1. *Create an animation*.

   WPF includes various classes that you can use to create animations in XAML. The most commonly used animation element is **DoubleAnimation**. The **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.

2. *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.

3. *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.

The following example shows how to use an animation to rotate and expand an image when the user clicks on it:

Creating an Animation Effect

```
<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>
            <="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>
```

Additional Reading: For more information about animations, refer to the Animation Overview page at **http://go.microsoft.com/fwlink/?LinkID=267835**.

Demonstration: Customizing Student Photographs and Styling the Application Lab

In this demonstration, you will learn about the tasks that you will perform in the lab for this module.

Demonstration steps

You will find the steps in the **Demonstration: Customizing Student Photographs and Styling the Application Lab** section on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_DEMO.md**.

Lab: Customizing Student Photographs and Styling the Application

Scenario

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

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

Objectives

After completing this lab, you will be able to:

- Create and use user controls.
- Use styles and animations.

Lab setup

Estimated Time: 90 minutes

You will find the high-level steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_LAB_MANUAL.md**._

You will find the detailed steps on the following page: **https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD09_LAK.md**.

Exercise 1: Customizing the Appearance of Student Photographs

Scenario

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

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

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

Finally, you will run the application to verify that the student's image is displayed correctly on the **StudentsPage** view.

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

Exercise 2: Styling the Logon View

Scenario

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

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

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

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

Scenario

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

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

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

Module review and takeaways

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

Review Question(s)

Check Your Knowledge

Select the best answer

You want to use rows and columns to lay out a UI. Which container control should you use?

The Canvas control.

The DockPanel control.

The Grid control.

The StackPanel control.

The WrapPanel control.

Check answer Show solution Reset

Check Your Knowledge

Select the best answer

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?

Create a ListBox control. Add child controls to the ListBox control to represent each field.

Create a ListBox control. Use a DataTemplate to specify how each field is displayed within a list item.

Create a ListBox control. Create a custom control that inherits from ListBoxItem, and use this custom control to specify how each field is displayed.

Create a ListBox control. Use the DisplayMemberPath property to specify the fields you want to display in each list item.

Create a ListBox control. Use a Style to specify how each field is displayed within a list item.

Check answer Show solution Reset

Check Your Knowledge

Select the best answer

You want to apply a highlighting effect to selected items in a ListBox. How should you proceed?

Create a Style element and set the TargetType attribute to ListBox. Use a Setter element to apply the highlighting effect.

Create a Style element and set the TargetType attribute to ListBox. Use a Trigger element to apply the highlighting effect when a list box item is selected.

Create a Style element and set the TargetType attribute to ListBox. Use an EventTrigger element to apply the highlighting effect when a list box item is selected.

Create a Style element and set the TargetType attribute to ListBox. Use a Storyboard element to apply the highlighting effect when a list box item is selected.

Create a Style element and set the TargetType attribute to ListBox. Use a DoubleAnimation element to apply the highlighting effect when a list box item is selected.

Check answer Show solution Reset