4. WPF - Data Binding and MVVM

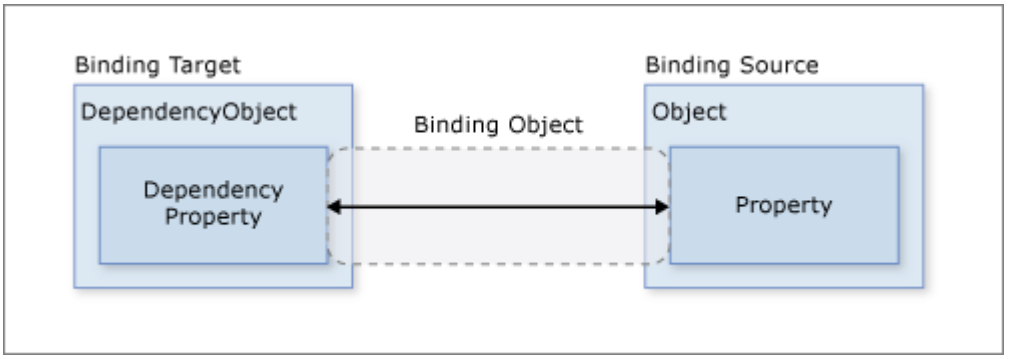
Data binding in Windows Presentation Foundation (WPF) provides a simple and consistent way for apps to present and interact with data. Elements can be bound to data from different kinds of data sources in the form of .NET objects. Any ContentControl such as Button and any ItemsControl, such as ListBox and ListView, have built-in functionality to enable flexible styling of single data items or collections of data items.

4.1 What is data binding?

Data binding is the process that establishes a **connection between** the app **UI and** the **data it displays**. If the binding has the correct settings and the data provides the proper notifications, when the data changes its value, the elements that are bound to the data, reflect changes automatically. Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a TextBox element, the underlying data value is automatically updated to reflect that change.

4.1.1 Basic data binding concepts

Regardless of what element you're binding and the nature of your data source, each binding always follows the model illustrated by the following figure.



As the figure shows, data binding is essentially the bridge between your binding target and your binding source. The figure demonstrates the following fundamental WPF data binding concepts:

- Typically, each binding has **four components**:
  - A binding **target object** (a UI element)
  - A **target property** (of the UI element)
  - A **binding source** (instance of a C# class that holds the data)

    - A **path to the value** in the binding source to use. (property of the object that holds the data)
- For example, if you bound the content of a `TextBox` to the `Employee.Name` property, you would set up your binding like the following table:

Setting	**Value**
Target	TextBox
Target property	Text
Source object	`Employee`
Source object value path	`Name`

To establish a binding, you use the Binding object.

The XAML would look something like this:

```
<TextBox Text="{Binding Path=Name}"></TextBox>
```

The source object can be specified in different ways (see next sections). One way to set the source object is to set a DataContext property in the code behind:

```
DataContext = new Employee { Name = "John Doe" };
```

4.1.2 Data context

When data binding is declared on XAML elements, they resolve data binding by looking at their immediate DataContext property. The data context is typically the binding source object for the binding source value path evaluation. If the `DataContext` property for the object hosting the binding isn't set, the parent element's `DataContext` property is checked, and so on, up until the root of the XAML object tree. In short, the data context used to resolve binding is inherited from the parent unless explicitly set on the object.

When the `DataContext` property changes, all bindings that could be affected by the data context are reevaluated.

4.1.3 Specifying the binding source

Sometimes it may be more appropriate to specify the binding source on individual binding declarations instead of relying on the DataContext.

One way to specify the source object of a binding is to use the Binding.Source property. The source object could, for example, be an application resource (see later on in this course)

```
<TextBlock
  Text="{Binding Source={StaticResource myRes}}"/>
```

You can also use the Binding.ElementName property or the Binding.RelativeSource property to specify the binding source.

The ElementName property is useful when you're binding to other elements in your app, such as when you're using a slider to adjust the width of a button. In the example below, the Text of the TextBox is bound to the SelectedItem.Content property of the ListView.

```
  <ListView x:Name="customerListView" Grid.Row="1" Margin="10 0 10 10">
    <ListViewItem>Julia</ListViewItem>
    <ListViewItem>Alex</ListViewItem>
    <ListViewItem>Thomas</ListViewItem>
  </ListView>
</Grid>

<!-- Customer detail -->
<StackPanel Grid.Column="1" Margin="10">
  <Label>Firstname:</Label>
  <TextBox Text="{Binding ElementName=customerListView,Path=SelectedItem.Content}"/>
```

The RelativeSource property is useful when you need to refer to elements relative to the target being bound. This makes it possible, for example, to use another property of the target as the binding source:

```
<Rectangle Fill="Red" Width="300"
  Height="{Binding RelativeSource=
             {RelativeSource Self},Path=Width}"/>
```

For more information, see Binding sources overview.

4.1.4 Specifying the path to the value

If your binding source is an object, you use the Binding.Path property to specify the value to use for your binding.

Although we have emphasized that the Path to the value to use is one of the four necessary components of a binding, in the scenarios that you want to bind to an entire object, the value to use would be the same as the binding source object. In those cases, it's applicable to not specify a Path. Consider the following example:

```
<ListBox ItemsSource="{Binding}"
         IsSynchronizedWithCurrentItem="true"/>
```

The above example uses the empty binding syntax: {Binding}. In this case, the ListBox inherits the DataContext from a parent DockPanel element (not shown in this example). When the path isn't specified, the default is to bind to the entire object. In other words, in this example, the path has been left out because we are binding the ItemsSource property to the entire object.

The Path property is used very often in bindings. That is why it is also allowed to omit the 'Path=' and only specify the target property name. In the example below, the Path property is set to 'Name'.

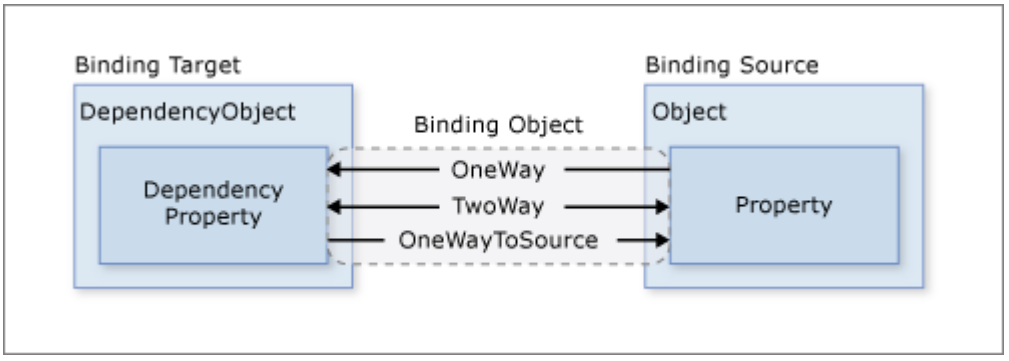
```
<TextBox Text="{Binding Name}"></TextBox>
```

4.1.5 Direction of the data flow

As indicated by the arrow in the previous figure, the data flow of a binding can go from the binding target to the binding source (for example, the source value changes when a user edits the value of a `TextBox`) and/or from the binding source to the binding target (for example, your `TextBox` content is updated with changes in the binding source) if the binding source provides the proper notifications.

You may want your app to enable users to change the data and propagate it back to the source object. Or you may not want to enable users to update the source data. You can control the flow of data by setting the Binding.Mode.

This figure illustrates the different types of data flow:



- OneWay binding causes changes to the source property to automatically update the target property, but changes to the target property are not propagated back to the source property. This type of binding is appropriate if the control being bound is implicitly read-only. For instance, you may bind to a source such as a stock ticker, or perhaps your target property has no control interface provided for making changes, such as a data-bound background color of a table. If there's no need to monitor the changes of the target property, using the OneWay binding mode avoids the overhead of the TwoWay binding mode.
- TwoWay binding causes changes to either the source property or the target property to automatically update the other. This type of binding is appropriate for editable forms or other fully interactive UI scenarios. Most properties default to OneWay binding, but some dependency properties (typically properties of user-editable controls such as the TextBox.Text and CheckBox.IsChecked default to TwoWay binding.
- OneWayToSource is the reverse of OneWay binding; it updates the source property when the target property changes. One example scenario is if you only need to reevaluate the source value from the UI.
- Not illustrated in the figure is OneTime binding, which causes the source property to initialize the target property but doesn't propagate subsequent changes. If the data context changes or the object in the data context changes, the change is *not* reflected in the target property. This type of binding is appropriate if either a snapshot of the current state is appropriate or the data is truly staticThis mode is essentially a simpler form of OneWay binding that provides better performance in cases where the source value doesn't change.

To detect source changes (applicable to OneWay and TwoWay bindings), the source must implement a suitable property change notification mechanism such as INotifyPropertyChanged.

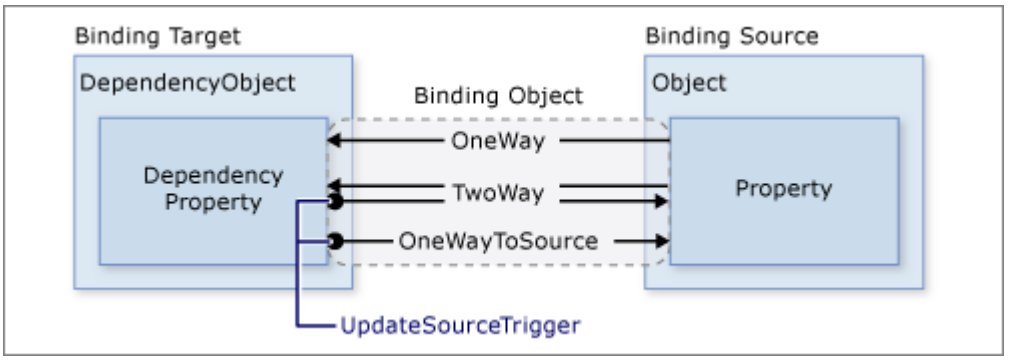
The Binding.Mode property provides more information about binding modes and an example of how to specify the direction of a binding.

4.1.6 What triggers source updates

Bindings that are TwoWay (or OneWayToSource) listen for changes in the target property and propagate them back to the source, known as updating the source. For example, you may edit the text of a TextBox to change the underlying source value.

However, is your source value updated while you're editing the text or after you finish editing the text and the control loses focus? The Binding.UpdateSourceTrigger property determines what triggers the update of the source. The dots of the right

arrows in the following figure illustrate the role of the Binding.UpdateSourceTrigger property.



If the `UpdateSourceTrigger` value is UpdateSourceTrigger.PropertyChanged, then the value pointed to by the right arrow of TwoWay or the OneWayToSource bindings is updated as soon as the target property changes. However, if the `UpdateSourceTrigger` value is LostFocus, then that value only is updated with the new value when the target property loses focus.

Similar to the Mode property, different dependency properties have different default UpdateSourceTrigger values. The default value for most dependency properties is PropertyChanged, which causes the source property's value to instantly change when the target property value is changed. Instant changes are fine for CheckBox and other simple controls. However, for text fields, updating after every keystroke can diminish performance and denies the user the usual opportunity to backspace and fix typing errors before committing to the new value. For example, the `TextBox.Text` property defaults to the `UpdateSourceTrigger` value of LostFocus, which causes the source value to change only when the control element loses focus, not when the `TextBox.Text` property is changed. See the UpdateSourceTrigger property page for information about how to find the default value of a dependency property.

The following table provides an example scenario for each UpdateSourceTrigger value using the TextBox as an example.

UpdateSourceTrigger value	**When the source value is updated**	**Example scenario for TextBox**
`LostFocus` (default for TextBox.Text)	When the TextBox control loses focus.	A TextBox that is associated with validation logic

`PropertyChanged`	As you type into the TextBox.	TextBox controls in a chat room window.
`Explicit`	When the app calls UpdateSource.	TextBox controls in an editable form (updates the source values only when the user presses the submit button).

The example below shows how the UpdateSourceTrigger of a binding on the Text property of a TextBox can be changed to the PropertyChanged value:

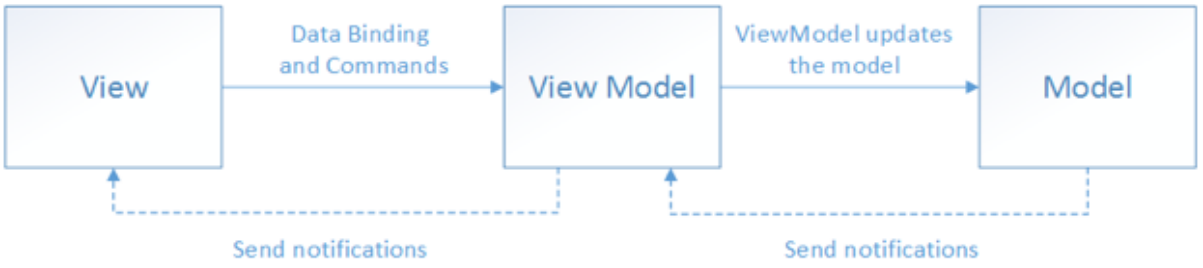
```
<TextBox Text="{Binding ElementName=customerListView,Path=SelectedItem.Content,
  Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"/>
```

4.2 MVVM

The WPF developer experience typically involves creating a user interface in XAML, and then adding code-behind that operates on the user interface. As applications are modified, and grow in size and scope, complex maintenance issues can arise. These issues include the tight coupling between the UI controls and the business logic, which increases the cost of making UI modifications, and the difficulty of unit testing such code.

The Model-View-ViewModel (MVVM) pattern helps to cleanly separate the business and presentation logic of an application from its user interface (UI). Maintaining a clean separation between application logic and the UI helps to address numerous development issues and can make an application easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of an application.

There are three core components in the MVVM pattern: the model, the view, and the view model. Each serves a distinct purpose. The figure below shows the relationships between the three components:



In addition to understanding the responsibilities of each component, it's also important to understand how they interact with each other. At a high level, the view "knows about" the view model, and the view model "knows about" the model, but the model is unaware of the

view model, and the view model is unaware of the view. Therefore, the view model isolates the view from the model, and allows the model to evolve independently of the view.
The benefits of using the MVVM pattern are as follows:

- If there's an existing model implementation that encapsulates existing business logic, it can be difficult or risky to change it. In this scenario, the view model acts as an adapter for the model classes and enables you to avoid making any major changes to the model code.
- Developers can create unit tests for the view model and the model, without using the view. The unit tests for the view model can exercise exactly the same functionality as used by the view.
- The app UI can be redesigned without touching the code, provided that the view is implemented entirely in XAML. Therefore, a new version of the view should work with the existing view model.
- Designers and developers can work independently and concurrently on their components during the development process. Designers can focus on the view, while developers can work on the view model and model components.

The key to using MVVM effectively lies in understanding how to factor app code into the correct classes, and in understanding how the classes interact. The following sections discuss the responsibilities of each of the classes in the MVVM pattern.

4.2.1 View

The view is responsible for defining the structure, layout, and appearance of what the user sees on screen. Ideally, each view is defined in XAML, with a limited code-behind that **does not contain business logic**. There are several options for executing code on the view model in response to interactions on the view, such as a button click or item selection. If a control supports commands, the control's Command property can be data-bound to an ICommand property on the view model. When the control's command is invoked, the code in the view model will be executed.

4.2.2 ViewModel

The view model implements **properties and commands to which the view can data bind to**, and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be displayed.

The view model is also responsible for coordinating the view's interactions with any model classes that are required. There's typically a one-to-many relationship between the view model and the model classes. The view model might choose to expose model classes directly to the view so that controls in the view can data bind directly to them. In this case, the model classes will need to be designed to support data binding and change notification events.
Each view model provides data from a model in a form that the view can easily consume. To accomplish this, the view model sometimes performs data conversion. Placing this data conversion in the view model is a good idea because it provides properties that the view can bind to. For example, the view model might combine the values of two properties to make it easier for display by the view.

In order for the view model to participate in two-way data binding with the view, its properties must raise the PropertyChanged event. View models satisfy this requirement by implementing the INotifyPropertyChanged interface, and raising the PropertyChanged event when a property is changed.
For collections, the view-friendly ObservableCollection<T> is provided. This collection implements collection changed notification, relieving the developer from having to implement the INotifyCollectionChanged interface on collections.

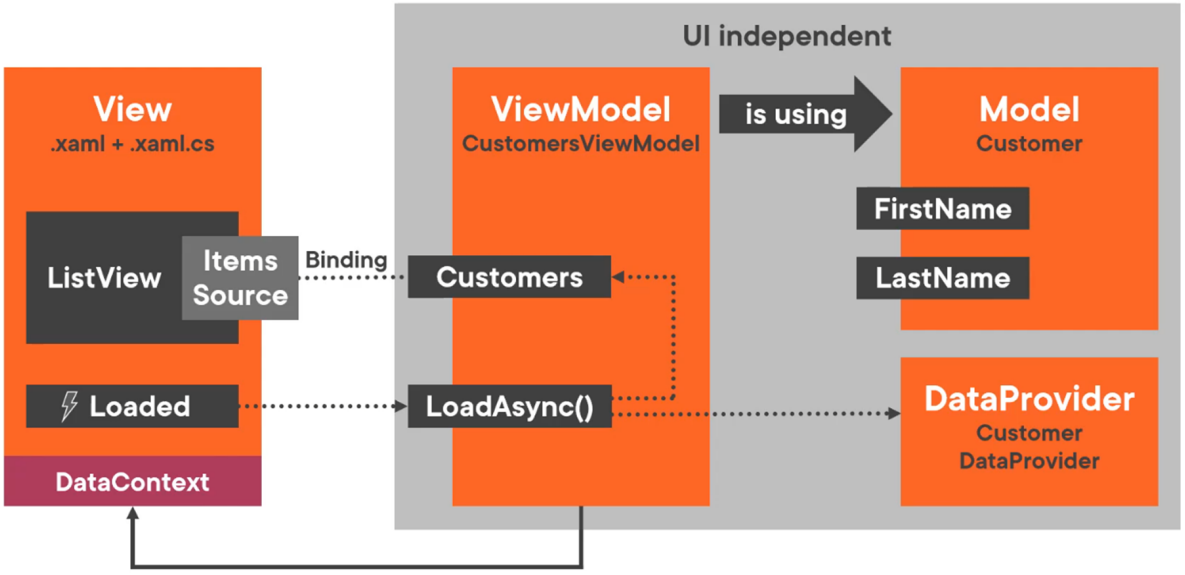
4.2.3 Model

Model classes are non-visual classes that encapsulate the app's data. Therefore, the model can be thought of as representing the app's domain model, which usually includes a data model along with business and validation logic.
Model classes are typically used in conjunction with services or repositories that encapsulate data access and caching.

4.2.4 How is everything connected?

You could perfectly build aWPF application without ViewModels.
That would mean that you would implement the user interface logic directly in the view, respectively, in the view's code-behind file with the file extension .xaml.cs.

But if you want to apply the MVVM pattern, you create, in addition, a ViewModel class that contains the user interface logic.



In the example above, the class is called CustomersViewModel, as it is linked to the CustomersView.
This CustomersViewModel class is using the model (Customer).
The ViewModel defines properties exactly in the way the view needs them. The view model is actually a model that is made for the view.
This is the reason why it is called a view model.
In the case of the Customers app, the view model can expose, for example, the Customers property (a list of Customer) for the view.

You connect the view to the view model with data binding.
To be able to bind to the view model, you assign a ViewModel instance to the view's DataContext property.
Then here, the ListView can bind its ItemsSource property to the Customers property of the view model.

To connect the View and the ViewModel an instance of the ViewModel class is set as the DataContext of the View. The View then uses **data binding** to bind to the properties and commands of the ViewModel.

```
public partial class CustomersView : UserControl
{
  private CustomersViewModel _viewModel;

  public CustomersView()
  {
    InitializeComponent();
    _viewModel = new CustomersViewModel(new CustomerDataProvider());
    DataContext = _viewModel;
  }
```

```
<ListView x:Name="customerListView"
          ItemsSource="{Binding Path=Customers}"
          Grid.Row="1" Margin="10 0 10 10"/>
```

4.3 Notify about property changes

To support OneWay or TwoWay binding to enable your binding target properties to automatically reflect the dynamic changes of the binding source (for example, to have the preview pane updated automatically when the user edits a form), your class needs to provide the proper property changed notifications. This can be achieved by implementing the INotifyPropertyChanged interface in the source class (e.g. a ViewModel).

To implement INotifyPropertyChanged you need to declare the PropertyChanged event and create the OnPropertyChanged method. Then for each property you want change notifications for, you call OnPropertyChanged whenever the property is updated.

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace SDKSample
{
  // This class implements INotifyPropertyChanged
  // to support one-way and two-way bindings
  // (such that the UI element updates when the source
  // has been changed dynamically)
  public class Person : INotifyPropertyChanged
  {
      private string name;
      // Declare the event
      public event PropertyChangedEventHandler PropertyChanged;

      public Person()
      {
      }

      public Person(string value)
      {
          this.name = value;
      }

      public string PersonName
      {
          get { return name; }
          set
          {
              name = value;
              // Call OnPropertyChanged whenever the property is updated
              OnPropertyChanged();
          }
      }

      // Create the OnPropertyChanged method to raise the event
      // The calling member's name will be used as the parameter.
      protected void OnPropertyChanged([CallerMemberName] string name = null)
      {
          PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
      }
  }
}
```

When a binding is defined in XAML, the binding automatically subscribes to the PropertyChanged event of the source object (if it implements INotifyPropertyChanged). When the binding gets notified of a change in its source property, it will update the value of the target property.

4.3 Data conversion

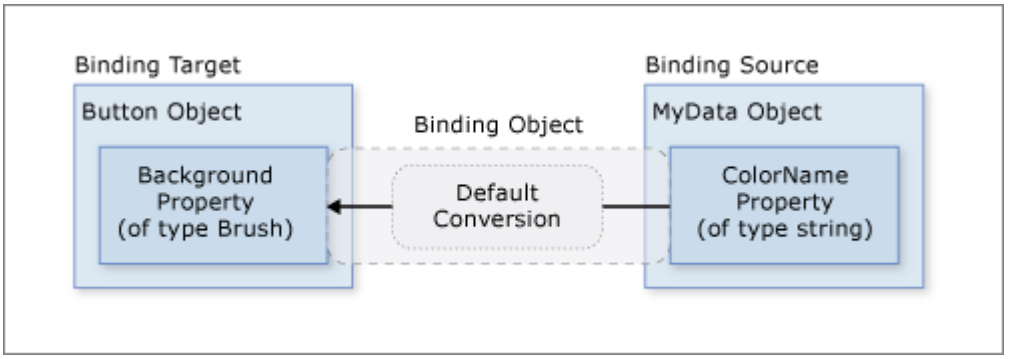
Consider the following example:

```
<StackPanel>
    <Button Background="{Binding Path=ColorName}">Click me</Button>
</StackPanel>
```

```
public partial class ConverterDemo : UserControl
{
    0 references
    public ConverterDemo()
    {
        InitializeComponent();
        DataContext = new MyData(colorName: "Red");
    }
}
```

```
2 references
public class MyData
{
    1 reference
    public string ColorName { get; set; }
    1 reference
    public MyData(string colorName)
    {
        ColorName = colorName;
    }
}
```

The button is red because its Background property is bound to a string property with the value "Red". This string value works because a type converter is present on the Brush type to convert the string value to a Brush.
Adding this information to the figure in the Create a binding section looks like this.



However, what if instead of having a property of type string your binding source object has a Color property of type Color? In that case, in order for the binding to work you would need to first turn the Color property value into something that the Background property accepts. You would need to create a custom converter.

If you want to associate a value converter with a binding, create a class that implements the IValueConverter interface and then implement the Convert and ConvertBack methods. Converters can change data from one type to another, translate data based on cultural information, or modify other aspects of the presentation.

Value converters are culture-aware. Both the Convert and ConvertBack methods have a culture parameter that indicates the cultural information. If cultural information is irrelevant to the conversion, then you can ignore that parameter in your custom converter.
The Convert and ConvertBack methods also have a parameter called parameter so that you can use the same instance of the converter with different parameters. For example, you can write a formatting converter that produces different formats of data based on the input parameter that you use. You can use the ConverterParameter of the Binding class to pass a parameter as an argument into the Convert and ConvertBack methods.

The converter below is an example of a converter that can convert instances of Color to SolidColorBrush.

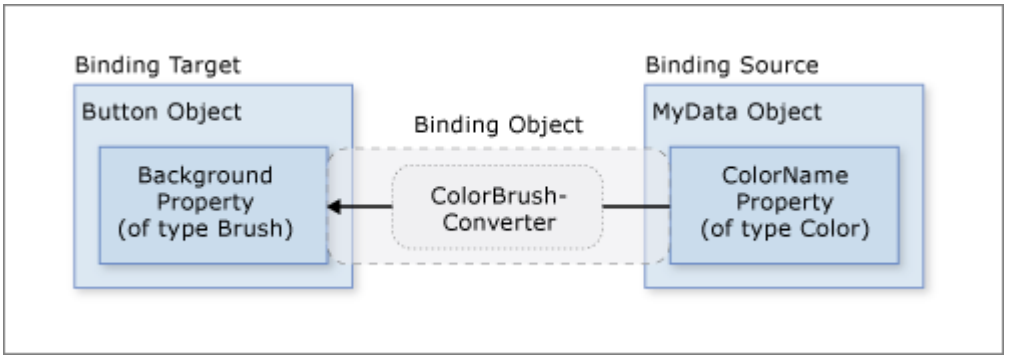
```
public class ColorBrushConverter : IValueConverter
{
    0 references
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

    0 references
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return null;
    }
}
```

To use the custom converter, it can be specified in the Converter property of the binding:

```
<UserControl x:Class="WiredBrainCoffee.CustomersApp.View.ConverterDemo"
             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"
             xmlns:local="clr-namespace:WiredBrainCoffee.CustomersApp.View"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
    <UserControl.Resources>
        <local:ColorBrushConverter x:Key="ColorBrushConverter"></local:ColorBrushConverter>
    </UserControl.Resources>
    <StackPanel>
        <Button Background="{Binding Path=Color, Converter={StaticResource ColorBrushConverter}}">Click me</Button>
    </StackPanel>
```

Using resources will be explained later on in the course.

Now the custom converter is used instead of default conversion, and our diagram looks like this.



To reiterate, default conversions may be available because of type converters that are present in the type being bound to. This behavior will depend on which type converters are available in the target. If in doubt, create your own converter.
The following are some typical scenarios where it makes sense to implement a data converter:

- Your data should be displayed differently, depending on culture. For instance, you might want to implement a currency converter or a calendar date/time converter based on the conventions used in a particular culture.
- The data being used isn't necessarily intended to change the text value of a property, but is instead intended to change some other value, such as the source for an image, or the color or style of the display text. Converters can be used in this instance by converting the binding of a property that might not seem to be appropriate, such as binding a text field to the Background property of a table cell.
- More than one control or multiple properties of controls are bound to the same data. In this case, the primary binding might just display the text, whereas other bindings handle specific display issues but still use the same binding as source information.

4.4 Commands

4.4.1 What are commands?

Commands have several purposes. The first purpose is to separate the semantics and the object that invokes a command from the logic that executes the command. This allows for multiple and disparate sources to invoke the same command logic, and it allows the command logic to be customized for different targets. For example, the editing operations Copy, Cut, and Paste, which are found in many applications, can be invoked by using different user actions if they are implemented by using commands. An application might allow a user to cut selected objects or text by either clicking a button, choosing an item in a menu, or using a key combination, such as CTRL+X. By using commands, you can bind each type of user action to the same logic.
Also Commands make it possible to use databinding to link UI interaction to command logic using data binding (MVVM).

Another purpose of commands is to indicate whether an action is available. To continue the example of cutting an object or text, the action only makes sense when something is selected. If a user tries to cut an object or text without having anything selected, nothing would happen. To indicate this to the user, many applications disable buttons and menu items so that the user knows whether it is possible to perform an action. A command can indicate whether an action is possible by implementing the CanExecute method. A button can subscribe to the CanExecuteChanged event and be disabled if CanExecute returns false or be enabled if CanExecute returns true.

Commands in WPF are created by implementing the ICommand interface. ICommand exposes two methods, Execute, and CanExecute, and an event, CanExecuteChanged. Execute performs the actions that are associated with the command. CanExecute determines whether the command can execute on the current command target. CanExecuteChanged is raised if the command manager that centralizes the commanding

operations detects a change in the command source that might invalidate a command that has been raised but not yet executed by the command binding.

The figure below shows an implementation of ICommand:

```
public class DelegateCommand : ICommand
{
  private readonly Action<object?> _execute;
  private readonly Func<object?, bool>? _canExecute;


  public DelegateCommand(Action<object?> execute, Func<object?,bool>? canExecute = null)
  {
    ArgumentNullException.ThrowIfNull(execute);
    _execute = execute;
    _canExecute = canExecute;
  }


  public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);

  public event EventHandler? CanExecuteChanged;


  public bool CanExecute(object? parameter) => _canExecute is null || _canExecute(parameter);


  public void Execute(object? parameter) => _execute(parameter);
}
```

In this implementation the constructor takes an *execute* parameter that is an Action delegate. This is the method that will be called when the Execute method of the command is invoked.
Constructor also takes a canExecute parameter that is an Action delegate. This is the method that will be called when the CanExecute method of the command is invoked.
Note that CanExecute always returns true if no canExecute action was provided in the constructor (it is optional)
There is a public method RaiseCanExecuteChanged that can be invoked (by e.g. a ViewModel class) to raise the CanExecuteChanged event.

4.4.2. Commands and MVVM

In MVVM the View and ViewModel are linked through data binding. The ViewModel is the source object to which the View data binds. The goal of MVVM is to separate the UI (XAML) and UI logic as much as possible. The ViewModel should contain all the application logic.
Now how can the logic that should be executed after some user interaction (e.g. button click) be defined in the ViewModel?
The answer is commands. The application logic is encapsulated in a command in the ViewModel and exposed as a property. The View can then databind to this exposed command.
A Button, for example, has a Command property that can be used (instead of the Click event).

```
<Button Margin="10" Width="75" Command="{Binding AddCommand}">
  <StackPanel Orientation="Horizontal">
    <Image Source="/Images/add.png" Height="18" Margin="0 0 5 0"/>
    <TextBlock Text="Add"/>
  </StackPanel>
</Button>
```

```
public CustomersViewModel(ICustomerDataProvider customerDataProvider)
{
  _customerDataProvider = customerDataProvider;
  AddCommand = new DelegateCommand(Add);
}

public ObservableCollection<CustomerItemViewModel> Customers { get; } = new();

public CustomerItemViewModel? SelectedCustomer...

public NavigationSide NavigationSide...

public DelegateCommand AddCommand { get; }
```

```
private void Add(object? parameter)
{
  var customer = new Customer { FirstName = "New" };
  var viewModel = new CustomerItemViewModel(customer);
  Customers.Add(viewModel);
  SelectedCustomer = viewModel;
}
```