Data Binding Basics

DEVELOPMENTOR

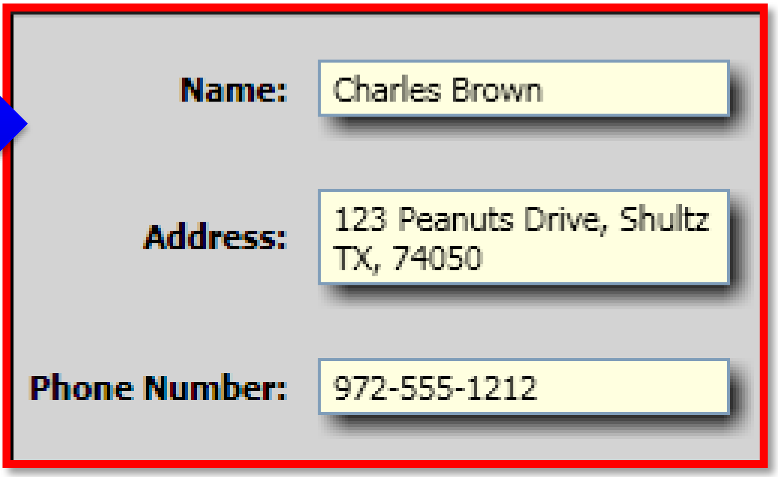
DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Importance [internal data]

- Most applications maintain internal data and map it to UI

```
Name    "Charles Brown"
Address "123 Peanuts..."
Phone   "972-555-1212"
```

```
Name    "Lucy"
Address "456 Peanuts..."
Phone   "972-555-2121"
```

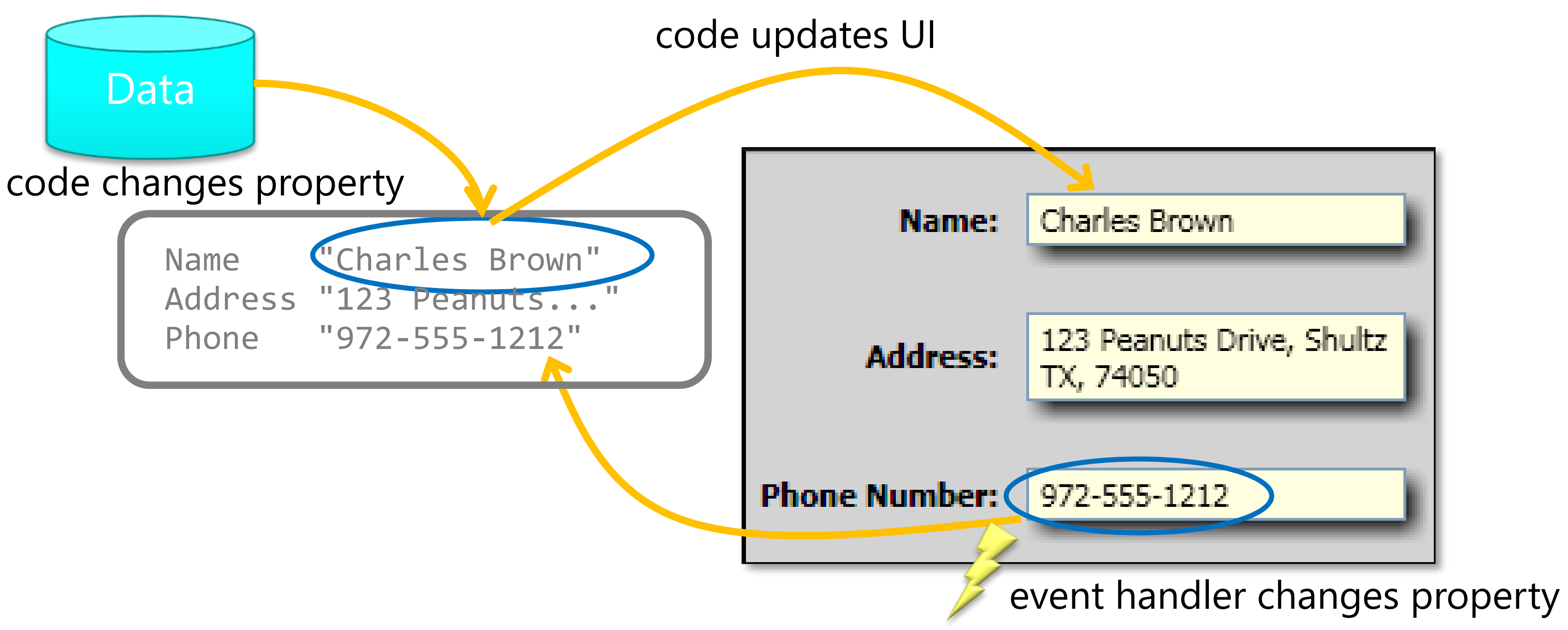


Name:	Charles Brown
Address:	123 Peanuts Drive, Shultz TX, 74050
Phone Number:	972-555-1212

```
public class Contact
{
   public string Name { get; set; }
   public string Address { get; set; }
   public string Phone { get; set; }
   ...
}
```

Importance [data change]

- Changes need to be propagated in both directions
  - typically done programmatically
  - tends to be error prone
  - tightly couples UI with data

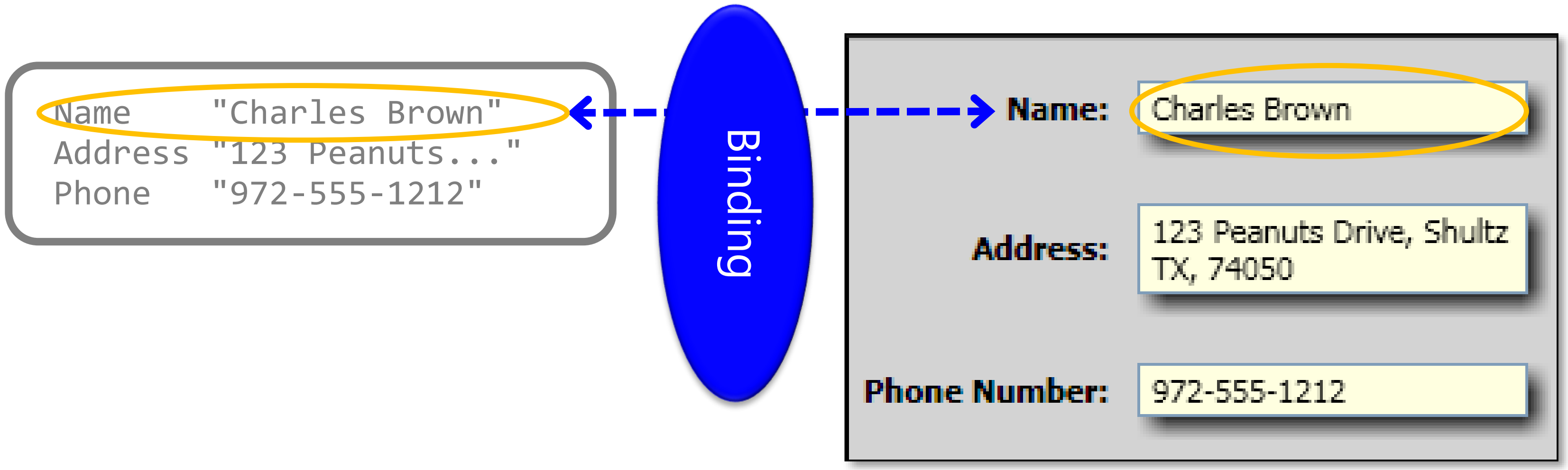


```
Name    "Charles Brown"
Address "123 Peanuts..."
Phone   "972-555-1212"
```

Name:	Charles Brown
Address:	123 Peanuts Drive, Shultz TX, 74050
Phone Number:	972-555-1212

Propagation the WPF way

- Binding object ties two properties together
  - automatically copies changed values back and forth
  - allows data and UI to be loosely coupled through binding



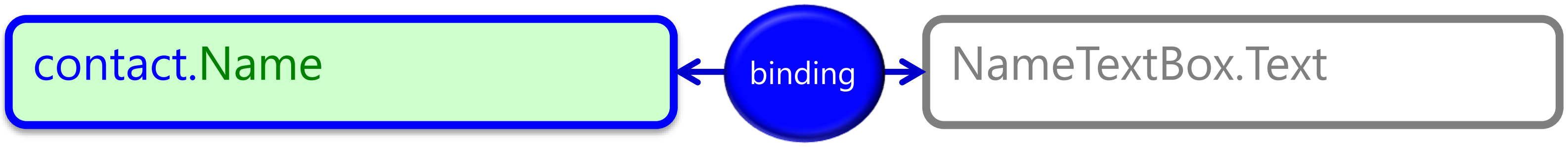
```
Name    "Charles Brown"
Address "123 Peanuts..."
Phone   "972-555-1212"
```

Name: Charles Brown
Address: 123 Peanuts Drive, Shultz TX, 74050
Phone Number: 972-555-1212

Creating bindings

- `System.Windows.Data.Binding` synchronizes two properties
  - source is the object where the data is coming from
  - path establishes the property to retrieve the value from
  - target identifies instance and property data is going to

```
Contact contact = new Contact("Charles Brown", ...);
...
Binding binding = new Binding();
binding.Source = contact;
binding.Path= new PropertyPath("Name");
NameTextBox.SetBinding(TextBox.TextProperty, binding);
...
```



Creating bindings in XAML

- Binding is placed on target DependencyProperty
  - source typically a resource or set in code-behind

```
<StackPanel>
   <StackPanel.Resources>
      <local:Contact x:Key="contact" Name="Charles Brown" ... />
   </StackPanel.Resources>

   <Label>Name:</Label>

   <TextBox x:Name="NameTextBox">
      <TextBox.Text>
         <Binding Source="{StaticResource contact}" Path="Name" />
      </TextBox.Text>
   </TextBox>

</StackPanel>
```

Using the Binding markup extension

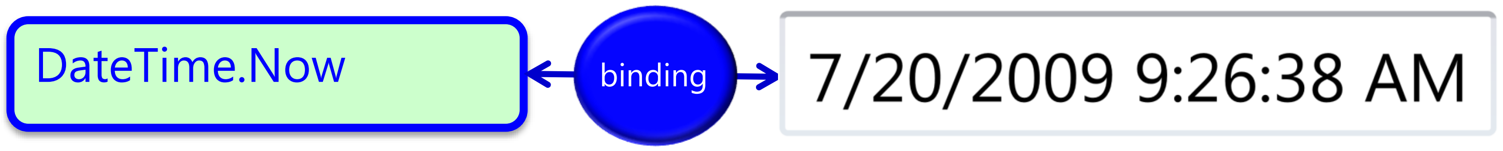
- {Binding} markup extension reduces typing in XAML
  - short-hand notation for the **Binding** object

```
<StackPanel>
   <StackPanel.Resources>
      <local:Contact x:Key="contact" Name="Charles Brown" ... />
   </StackPanel.Resources>

   <TextBox Text="{Binding Path=Address,
                       Source={StaticResource contact}}" />

</StackPanel>
```

allows binding to be applied to property inline

Controlling the flow of information

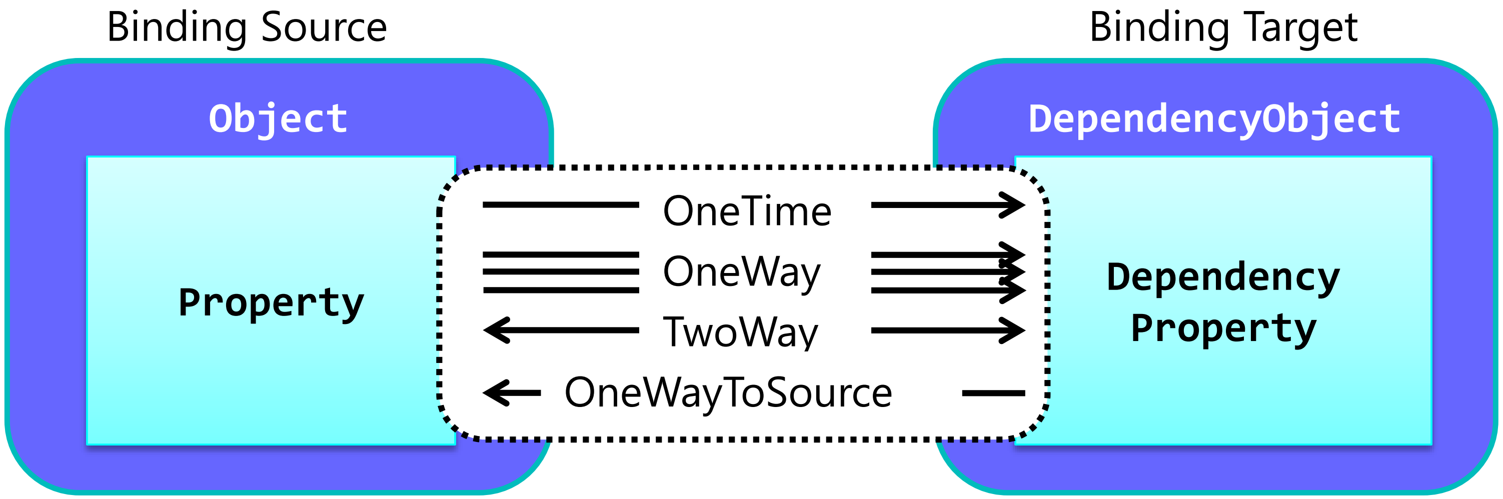
- Binding target DP decides how information should transfer
  - one direction or both directions
  - part of dependency property metadata
- Sometimes default choice is inappropriate
  - ex: **`TextBox.Text`** bound to a read-only property



TextBox changes cannot be propagated back to current time

Controlling the flow of information [2]

- Binding Mode determines data transfer direction

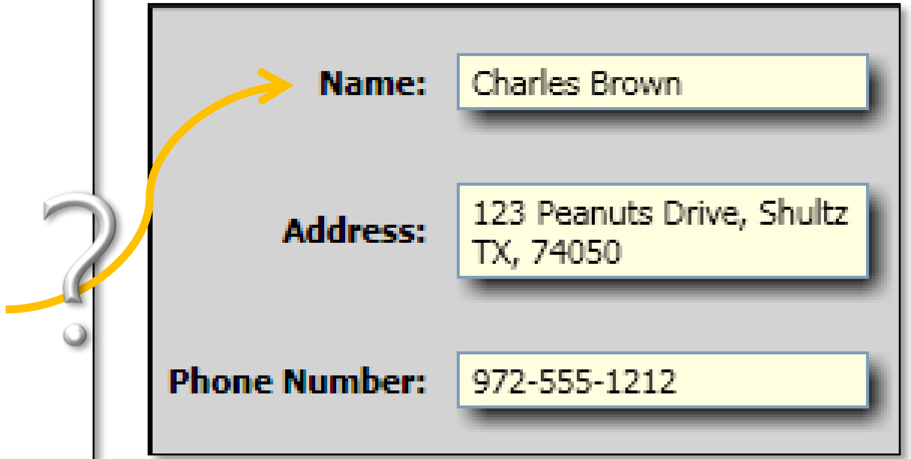


```
Binding binding = new Binding();
binding.Source = DateTime.Now,
binding.Mode = BindingMode.OneTime;
timeTextBox.SetBinding(TextBox.TextProperty, binding);
```

Going the other direction: source to target

- Target property is always `DependencyProperty`
  - WPF knows immediately when these are changed
- Source can be <u>any</u> object
  - property can be any path leading to value (e.g. **`Address.Zip`**)

```
public class Contact
{
   private string _name;
   public string Name
   {
      get { return _name; }
      set { _name = value; }
   }
   ...
}
```

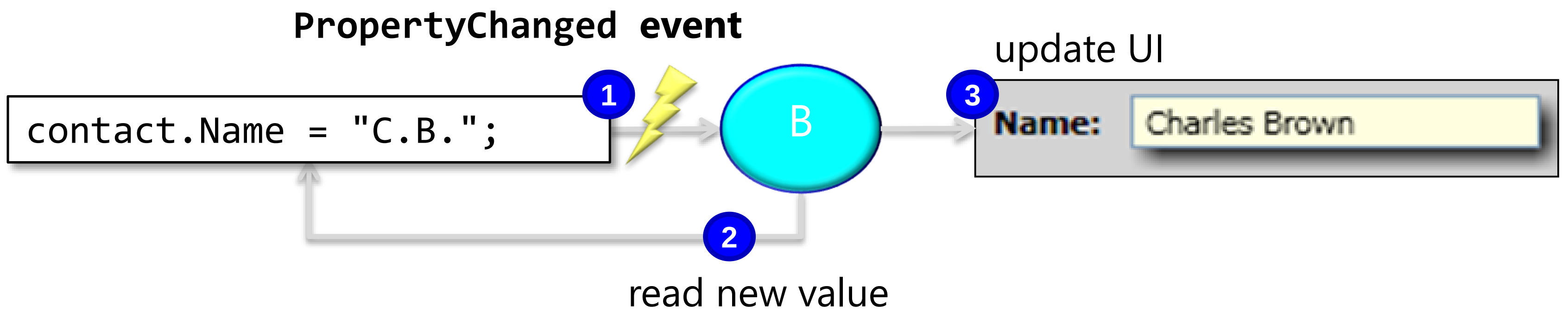


Name:	Charles Brown
Address:	123 Peanuts Drive, Shultz TX, 74050
Phone Number:	972-555-1212

How can the Binding know the underlying value has changed and needs to be moved to the target?

Making CLR objects binding friendly

- Source objects provide change notifications by:
  - implementing **INotifyPropertyChanged** (preferred)
  - or exposing **XXXChanged** event for each property (deprecated)
- WPF reads property value when event is raised and updates UI



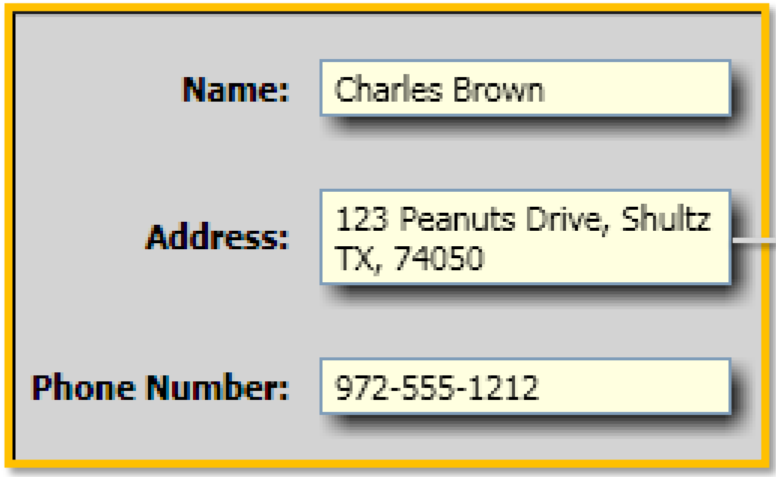
```
public interface INotifyPropertyChanged
{
   public PropertyChangedEvent PropertyChanged;
}
```

Implementing INotifyPropertyChanged

```
public class Contact : INotifyPropertyChanged
{
   private string _name;
   public string Name
   {
      get { return _name; }
      set { _name = value; OnPropertyChanged("Name"); }
   }

   public event PropertyChangedEventHandler PropertyChanged;
   private void OnPropertyChanged(string name)
   {
      Debug.Assert(string.IsNullOrEmpty(name) ||
                   GetType().GetProperty(name) != null);
      if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(name));
   }
   ...
}
```

Determining when the data exchange happens

- Source → Target always occurs on property change
- Target → Source varies depending on usage



Name:	Charles Brown
Address:	123 Peanuts Drive, Shultz TX, 74050
Phone Number:	972-555-1212

```
Name    "Charles Brown"
Address "123 Peanuts..."
Phone   "972-555-1212"
```

should the property setter be called for every character change?

what if the property setter writes to a database or web service?

Determining when the data exchange happens [2]

- `UpdateSourceTrigger` decides when change applied to source
  - **`LostFocus`** – copy when focus is lost on target
  - **`PropertyChanged`** – copy when target value changes
  - **`Explicit`** – copy only when asked[1]

```
<TextBox x:Name="NameTextBox"
         Text="{Binding Source={StaticResource contact},
                Path=Name, UpdateSourceTrigger=Explicit}" />
```

```
void UpdateContact()
{
  BindingExpression expr = NameTextBox.GetBindingExpression(
                                          TextBox.TextProperty);
  expr.UpdateSource();  // Transfers values here..
  CallWebServiceToUpdate(contact);
}
```

Responding to changes

- Bindings can notify code behind when it causes a change
  - **NotifyOnSourceUpdated** causes notification of source
  - **NotifyOnTargetUpdated** causes notification of target
- Events can then be wired on associated FrameworkElement
  - raises **SourceUpdated** and **TargetUpdated** events

```
<TextBox TargetUpdated="TextBox_AgeUpdated"
    Text="{Binding Age, NotifyOnTargetUpdated=true}" />
```

```
void TextBox_AgeUpdated(object sender, DataTransferEventArgs e)
{
   TextBox tb = (TextBox)e.TargetObject;
   tb.BorderBrush = Brushes.Gold;
}
```

Dealing with slow properties

- Sometimes the property being bound is expensive
  - requires processing or I/O time to calculate or retrieve value
  - bindings utilize UI thread and will block waiting for values
  - can increase performance by using asynchronous mode

binding will retrieve property on a background thread – UI will continue processing and display value when it returns

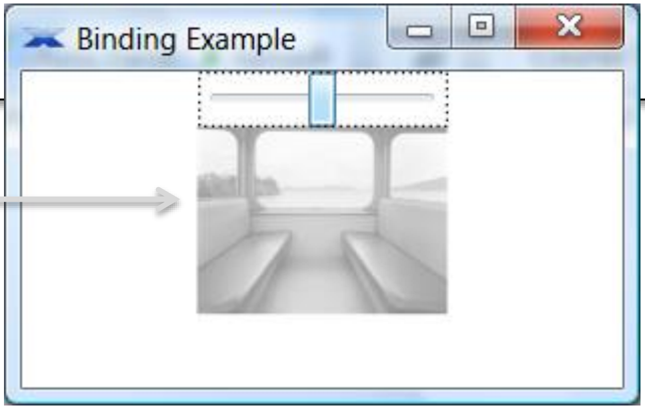
```
<TextBox Text="{Binding Path=YTDSales, IsAsync=True,
             Source={StaticResource allContacts}}" />
```

Binding for designers [tying elements together]

- ElementName associates elements in the same XAML file
  - source located by **Name** or **x:Name** property
  - elements must be part of same name scope[1]

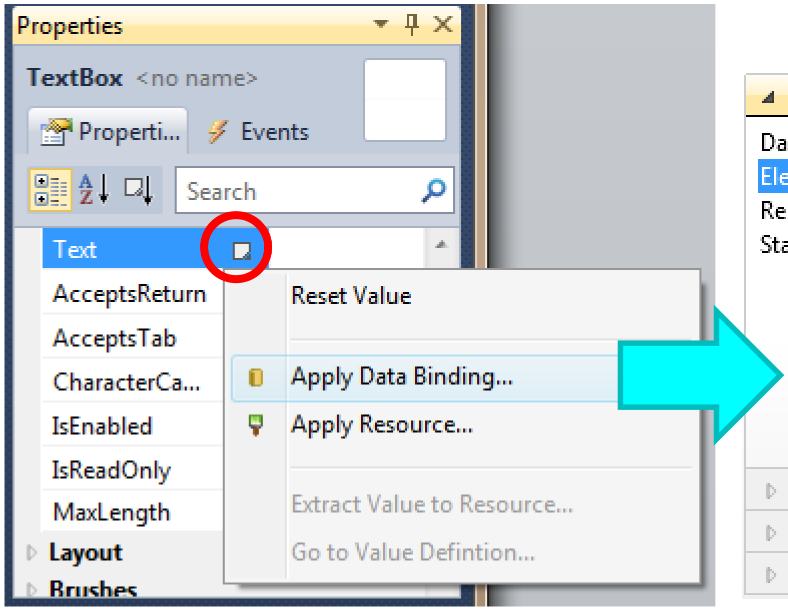
```
<StackPanel>
   <Slider x:Name="slider" Minimum="0" Maximum="1"
           Width="100" Value="1" />

   <Image Source="img1.jpg" Width="100"
          Opacity="{Binding ElementName=slider, Path=Value}" />

</StackPanel>
```

as slider is changed,
Opacity of image changes
automatically



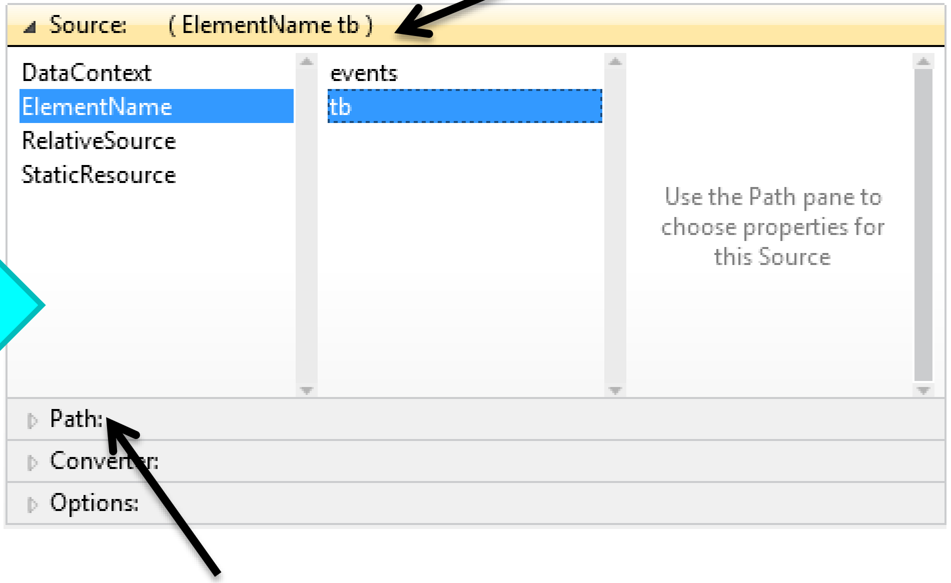
Creating Bindings in Visual Studio 2010

- Designer supports easy binding generation



select context menu next to any property and choose "Apply Data Binding..."

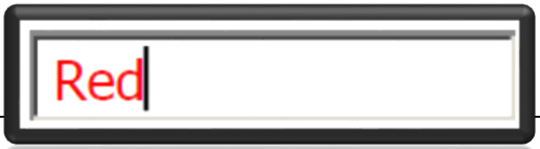
select binding source



... and then binding path

Binding to relative elements

- `RelativeSource` property allows `Binding` to identify elements relative to current element
  - **`FindAncestor`** – some ancestor of data-bound element
  - **`PreviousData`** – previous data item in list
  - **`TemplatedParent`** – parent where template is applied
  - **`Self`** – element where binding is applied



```
<TextBox Foreground="{Binding Path=Text,
         RelativeSource={RelativeSource Self}}" />
```

most common form is Self, but others are useful in templates which we will discuss later

Binding to properties on the source

- Binding source Path can navigate sub-properties[1]
  - uses reflection at runtime

```
<TextBlock Text="{Binding ElementName=contactList,
                          Path=SelectedItems.Count}" />
```

...can indicate array indexes – even multi-dimensional

```
<TextBox Text="{Binding ElementName=contactList,
                        Path=Items[2].Name}" />
```

...even supports casting to get to explicit properties

```
<TextBox Text="{Binding ElementName=contactList,
                        Path=SelectedItem.(local:IDbEntity.ID)}" />
```

Binding to attached properties in XAML

- Attached properties may be used as the source property
  - in that case, reflection cannot be used to lookup the value
  - instead, special syntax **`(Type.Property)`** used on **`Path`**

```
<Canvas Height="200" Width="300" Background="AliceBlue">

   <Rectangle x:Name="r1" Width="50" Height="50"
              Canvas.Left="20" Canvas.Top="20" Fill="Blue" />

   <Rectangle Canvas.Left="100" Fill="Gold"
      Canvas.Top="{Binding ElementName=r1, Path=(Canvas.Top)}"
      Width="{Binding ElementName=r1, Path=Width}"
      Height="{Binding ElementName=r1, Path=Height}" />

</Canvas>
```

When the binding fails

- Failed data binding will output results to debug console
  - any resulting exceptions automatically caught
  - can control diagnostic output [**Low|Medium|High**]

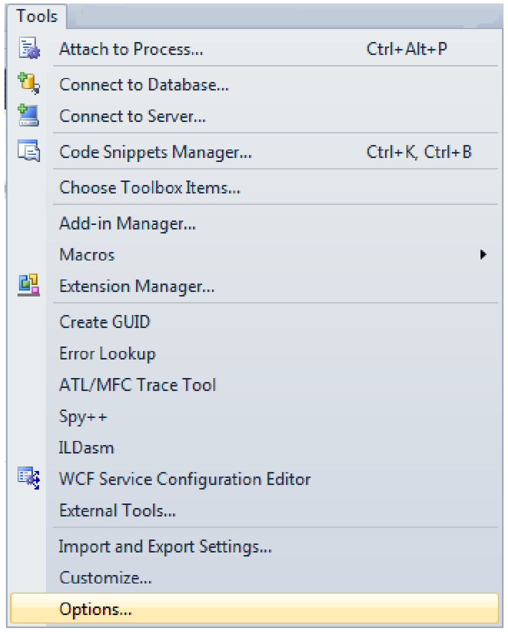
3.5 SP1

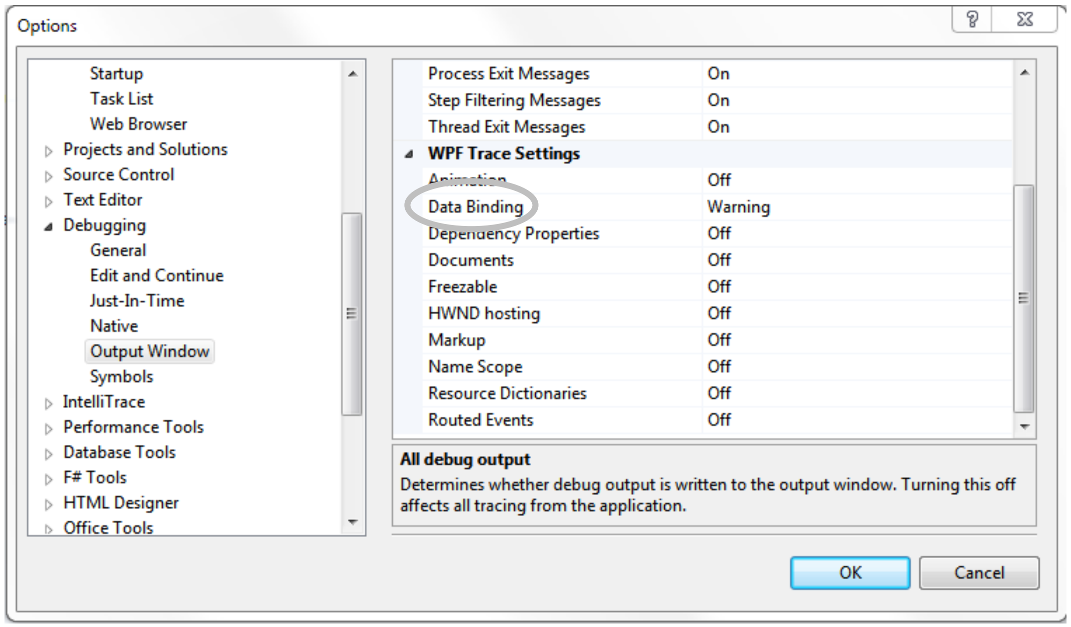
- Supply a default value when you expect binding to fail
  - e.g. value is not always available
  - can also supply a value to be used if the source is null

```
<Grid>
   <Rectangle
      Fill="{Binding Source={StaticResource selectedBrush},
      PresentationTraceSources.TraceLevel=High,
      FallbackValue=Red, TargetNullValue=Blue}" />
</Grid>
```

Controlling binding details in Visual Studio 2010

- Visual Studio 2010 has new Output Window tracing options
  - limits output to specified types (defaults bindings to **errors**)





Process Exit Messages	On
Step Filtering Messages	On
Thread Exit Messages	On
WPF Trace Settings	
Animation	Off
Data Binding	Warning
Dependency Properties	Off
Documents	Off
Freezable	Off
HWND hosting	Off
Markup	Off
Name Scope	Off
Resource Dictionaries	Off
Routed Events	Off

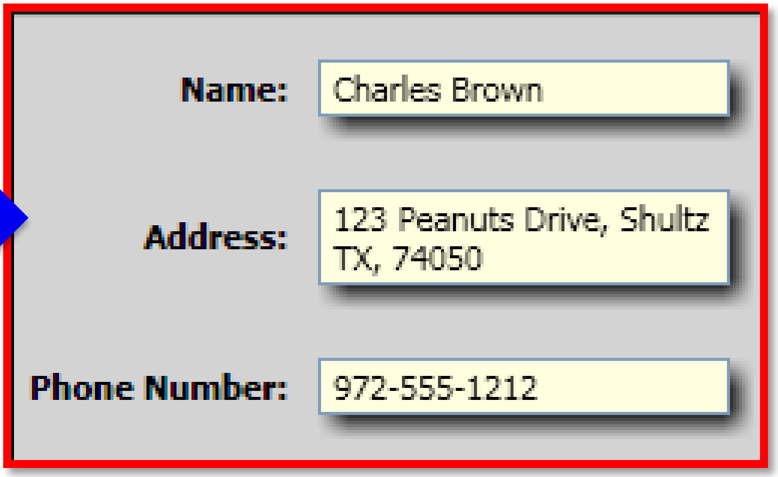
All debug output
Determines whether debug output is written to the output window. Turning this off affects all tracing from the application.

change to warning to see possible failures

Consolidating binding sources

- Often the same binding source is shared with many elements

```
Name    "Charles Brown"
Address "123 Peanuts..."
Phone   "972-555-1212"
```



Name: Charles Brown

Address: 123 Peanuts Drive, Shultz TX, 74050

Phone Number: 972-555-1212

same contact object is bound to three different fields – each looking at a different property

tedious and error-prone to specify same source on every element .. what if we change it?

Sharing the binding source across elements

- `DataContext` property provides a *default* binding source
  - inherited through visual tree from parent to child
  - typically set in code-behind

```
<Grid>
   <Grid.DataContext>
      <local:Contact Name="Charles Brown" />
   </Grid.DataContext>

   <Label Content="Name:" />
   <TextBox Text="{Binding Path=Name}" Grid.Column="1" />
   <Label Content="Address:" Grid.Row="1" />
   <TextBox Text="{Binding Path=Address}"
            Grid.Column="1" Grid.Row="1" />
</Grid>
```

`Binding.Source` is unnecessary on child controls
as it is inherited from the `Grid` parent

Formatting the target value

3.5 SP1

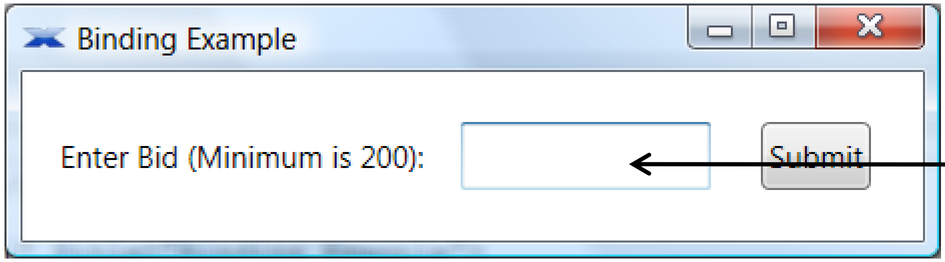
- `Binding.StringFormat` can be used for simple formatting
  - does a **`String.Format`** on the source value just before transferring it to the target property

Bid.Price is a double – would like to format it as currency when we display it..

```
<StackPanel>
   <StackPanel.Resources>
      <local:Bid x:Key="bid" Price="30.25" Symbol="MSFT" />
   </StackPanel.Resources>

   <TextBlock>Current Bid Price is </TextBlock>
   <TextBlock Text="{Binding Source={StaticResource bid}
                    Path=Price, StringFormat=C}" />
   ...
</StackPanel>
```

Going beyond simple formatting …

- Data binding cannot coerce between incompatible types
  - only simple textual conversions are valid (numeric to string)



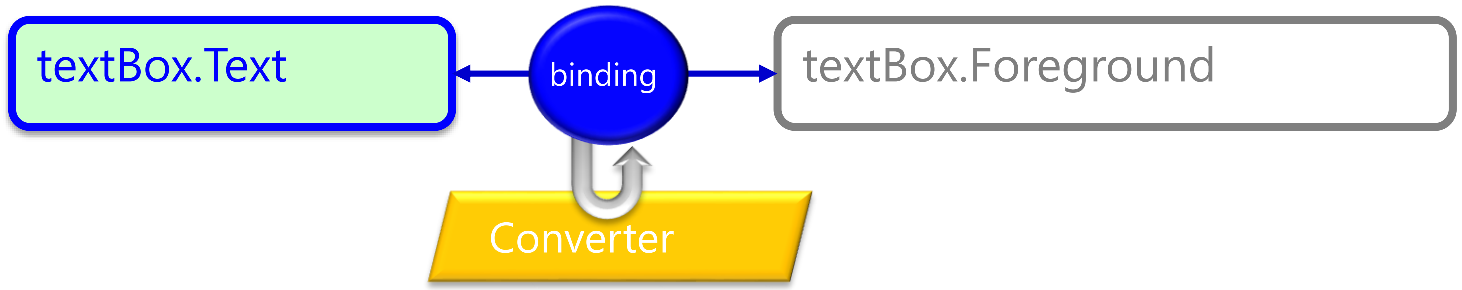
desire to change text color to red if below minimum bid

```
<StackPanel>
   <Label>Enter Bid (Minimum is 200):</Label>
   <TextBox FontSize="36pt"
        Foreground="{Binding Price,
                      Source={StaticResource bid}}" />
   ...
</StackPanel>
```

…but Price is a double, Foreground is a Brush

Converting data-bound values

- Converters may be placed onto Bindings
  - public class that implements **`IValueConverter`**



- Converters are applied to/from the raw value *before* it is transferred from the target to the data source
- WPF comes a few built-in converters
  - **`BooleanToVisibilityConverter`** is the most useful

Steps to create Value Converters

1. Create a public class that implements `IValueConverter`
   – pass configuration through properties
2. Implement the two methods defined on interface:
   – **`Convert`** changes value from source to target type
   – **`ConvertBack`** changes value from target to source type
3. Define an instance of your converter in XAML
   – and add it to the binding

Step 1: Create IValueConverter class

optional attribute indicates usage for designer tools

```
[ValueConversion(typeof(double), typeof(Brush))]
public partial class BidToBrushConverter : IValueConverter
{
   public double MinimumBid { get; set; }

}
```

pass global configuration for converter through properties

Step 2: Implement Convert and ConvertBack

- ConvertBack only necessary in two-way bindings
  - can be stubbed out for one-way usage like this one

```
public partial class BidToBrushConverter
{
   public object ConvertBack(object value, Type targetType,
                object parameter, CultureInfo culture)
   {
      // Can only be used in one-way bindings
      throw new NotSupportedException();
   }
}
```

Step 2: Implement Convert and ConvertBack

- `BidToBrushConverter` needs to check Double bid price against minimum and return proper Brush

```
partial class BidToBrushConverter
{
   public object Convert(object value, Type targetType,
           object parameter, CultureInfo culture)
   {
      double bid = Double.Parse(value.ToString());
      return (bid < MinimumBid)
           ? Brushes.Red
           : Brushes.Black;
   }
}
```

Do you see anything wrong with this code?

What about exceptions?

- Value converters are called during the binding process
  - unhandled exceptions in designer will kill designer view[1]
  - unhandled exceptions at runtime will terminate app
- Must anticipate failures and return appropriate values
  - **`DependencyProperty.UnsetValue`** for no value produced
  - **`Binding.DoNothing`** to ignore the binding altogether

```
public object Convert(object value, Type targetType,
           object parameter, CultureInfo culture)
{
   try { ... Do conversion ... }
   catch { return DependencyProperty.UnsetValue; }
}
```

Dealing with the designer surface

- Design surface instantiates controls and executes bindings
  - converters often need runtime values or assume proper input
  - can test for designer instantiation inside converter

```
public object Convert(object value, Type targetType,
          object parameter, CultureInfo culture)
{
   if ((bool)(DesignerProperties.IsInDesignModeProperty
          .GetMetadata(typeof(DependencyObject))
          .DefaultValue))
      return Brushes.Black;

   ...
}
```

returns true if the converter is currently running inside the Blend or Visual Studio designer ... code can then return designer data or default values

Step 3: Declare and apply the converter

- Converter property identifies a specific instance to use
  - typically stored in resources, but could be supplied inline

```
<StackPanel>
   <StackPanel.Resources>
      <local:BidToBrushConverter x:Key="bidCvt"
                                 MinimumBid="200" />
   </StackPanel.Resources>

   <Label>Enter Bid (Minimum is 200):</Label>
   <TextBox Foreground="{Binding Path=Price,
              Source={StaticResource bid},
              UpdateSourceTrigger=PropertyChanged,
              Converter={StaticResource bidCvt}}" />
   ...
</StackPanel>
```

Passing parameters to Converters

- Additional information may be necessary for the conversion
  - passed through **`Binding.ConverterParameter`** property
  - passed as a string for **`{Binding}`** markup extension
  - use full property-element syntax if you need complex value

```
<TextBox Foreground="{Binding Path=Text, ...
           Converter={StaticResource bidCvt},
           ConverterParameter=100}" />
```

```
public partial class BidConverter : IValueConverter
{
   public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture) { ... }
}
```

Common Bindings Cheat Sheet

Binding Syntax	**Description**
{Binding Source=source, Path=Text}	Bind to "Text" property of the object identified by “source".
{Binding ElementName=tb, Path=Text}	Bind to "Text" property of the the XAML element named "tb"
{Binding Text} or {Binding Path=Text}	Bind to the "Text" property of the current DataContext
{Binding Property.Count}	Bind to "Property" then to "Count" on that property
{Binding RelativeSource = {RelativeSource Self}}	Bind to the element where the binding is placed
{Binding RelativeSource = {RelativeSource FindAncestor, AncestorType={x:Type ListBox}}}	Walk up the visual tree, starting at the current element and locate the first element of type "ListBox".

Summary

- Data Binding associates UI elements with underlying data
  - two-way, automatic updates
  - in many cases it can replace simple procedural code
- Source can be any CLR object
  - including other elements in XAML
- Target must be DependencyProperty
- `DataContext` used to share common binding source
  - reduces markup and allows dynamic changes to source
- Converters may be used to translate values in binding
  - solves type mismatch between binding source and target