WPF Data Binding

Objectives

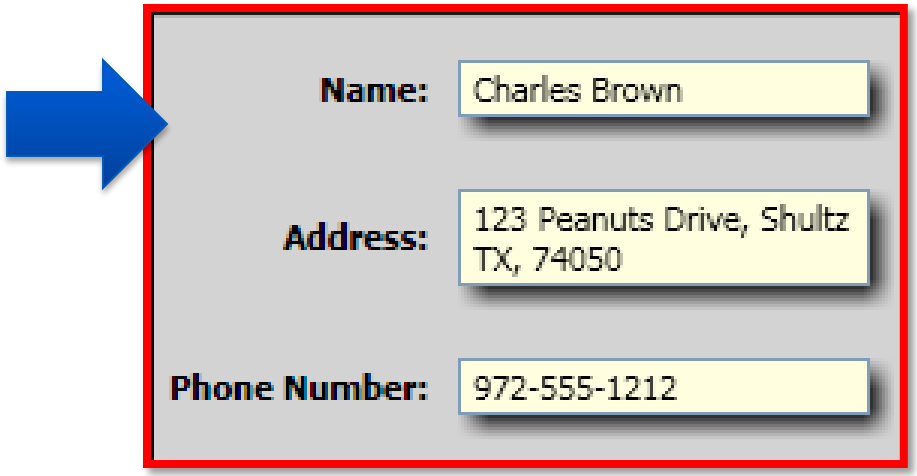
- **Introduce the Data Binding capabilities**
  - assigning a source/path and target
  - changing how and when the binding occurs
  - binding to simple objects and collections

Importance [internal data]

- **Most applications maintain internal data and map it to UI**

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

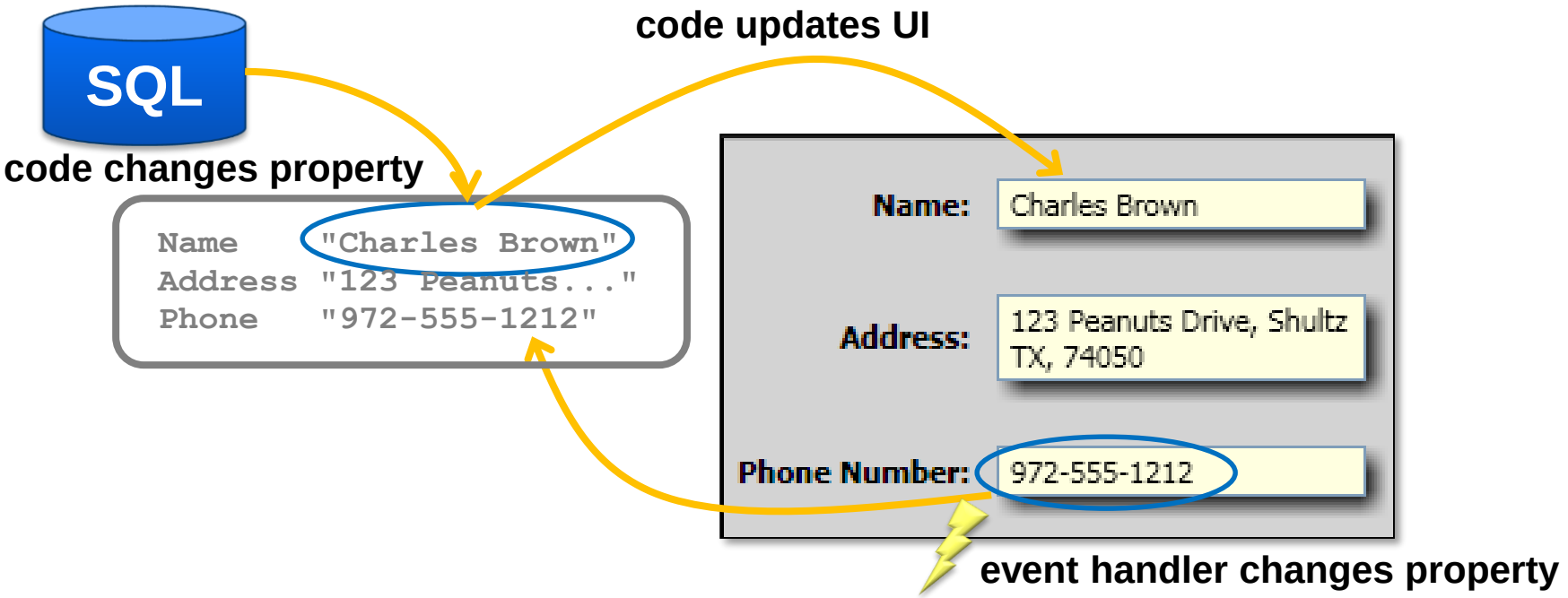
```
Name     "Lucy"
Address  "456 Peanuts..."
Phone    "972-555-2121"
```




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

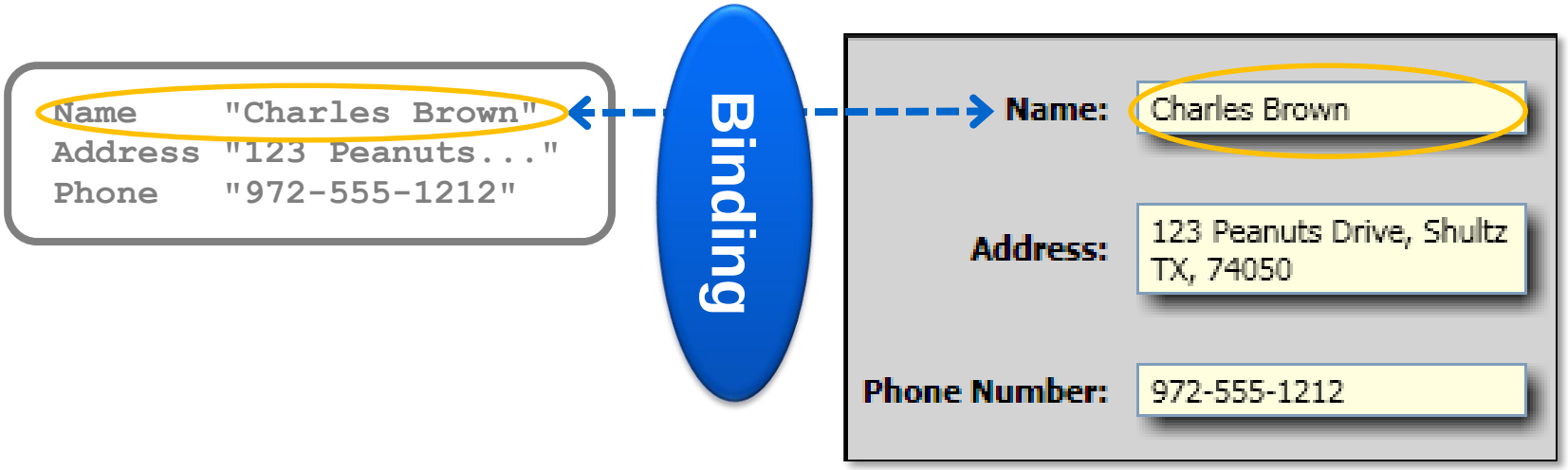

Importance [data change]

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



Propagation the WPF way

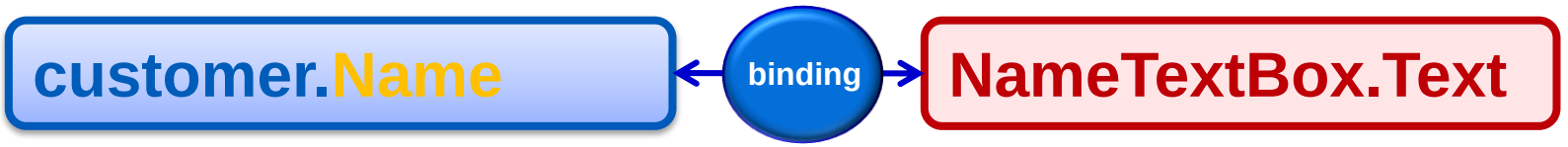
- **Binding object ties two properties together**
  - automatically copies changed values back and forth
  - allows data and UI to be loosely coupled through binding



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

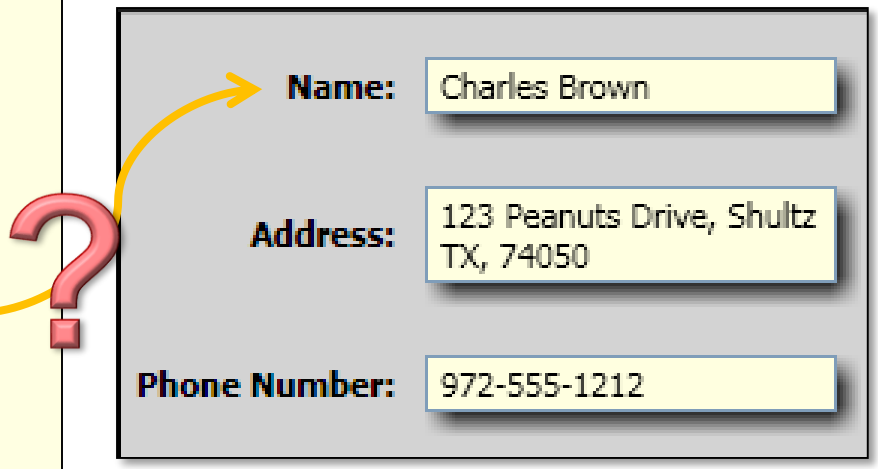
```
Customer customer = new Customer("Charles Brown", ...);
...
Binding binding = new Binding();
binding.Source = customer;
binding.Path= new PropertyPath("Name");
nameTextBox.SetBinding(TextBox.TextProperty, binding);
...
```



Binding rules

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

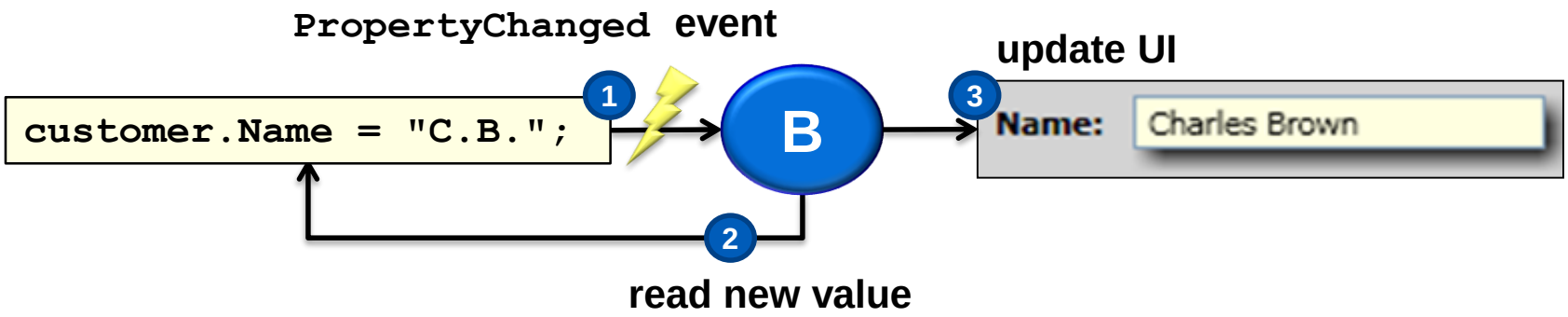
```
public class Customer
{
    private string _name;
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    ...
}
```



How can WPF determine property has changed?

Making CLR objects binding friendly

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



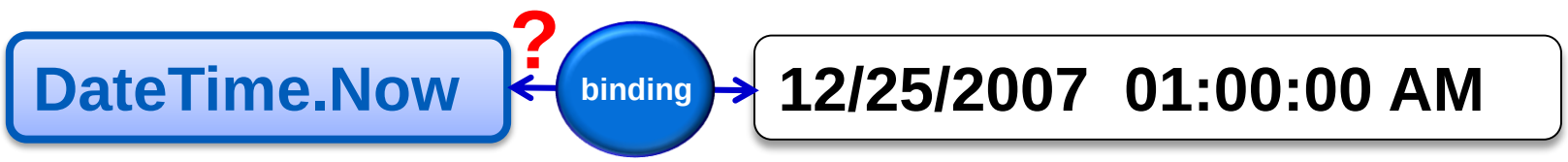

```
public interface INotifyPropertyChanged
{
    public PropertyChangedEvent PropertyChanged;
}
```

Implementing INotifyPropertyChanged

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


   public event PropertyChangedEventPropertyChanged;
   private void OnPropertyChanged(string propName)
   {
      if (PropertyChanged != null)
          PropertyChanged(this,
             new PropertyChangedEventArgs(propName));
      }
   }
   ...
}
```


Controlling the flow of information

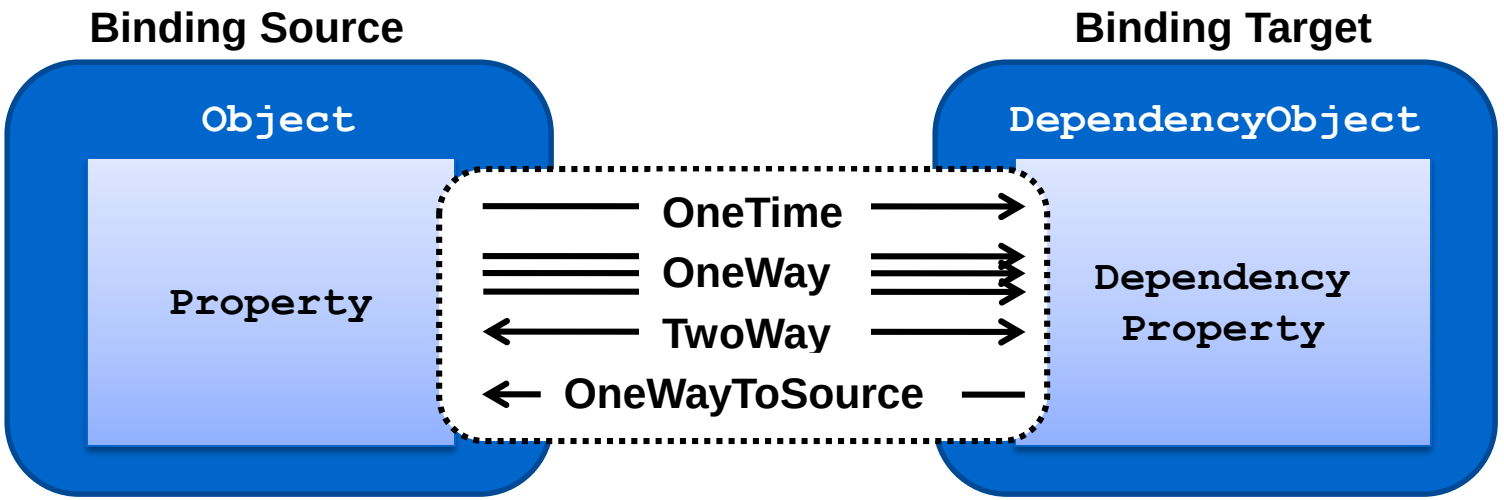
- **Common desire is to flow information both directions**
  - referred to as "two-way binding"
- **Sometimes information can only go one direction**
  - what if the property does not have a setter?
  - what if the UI element cannot be changed?



TextBox changes cannot be propagated back to current time!

Controlling the flow of information [2]

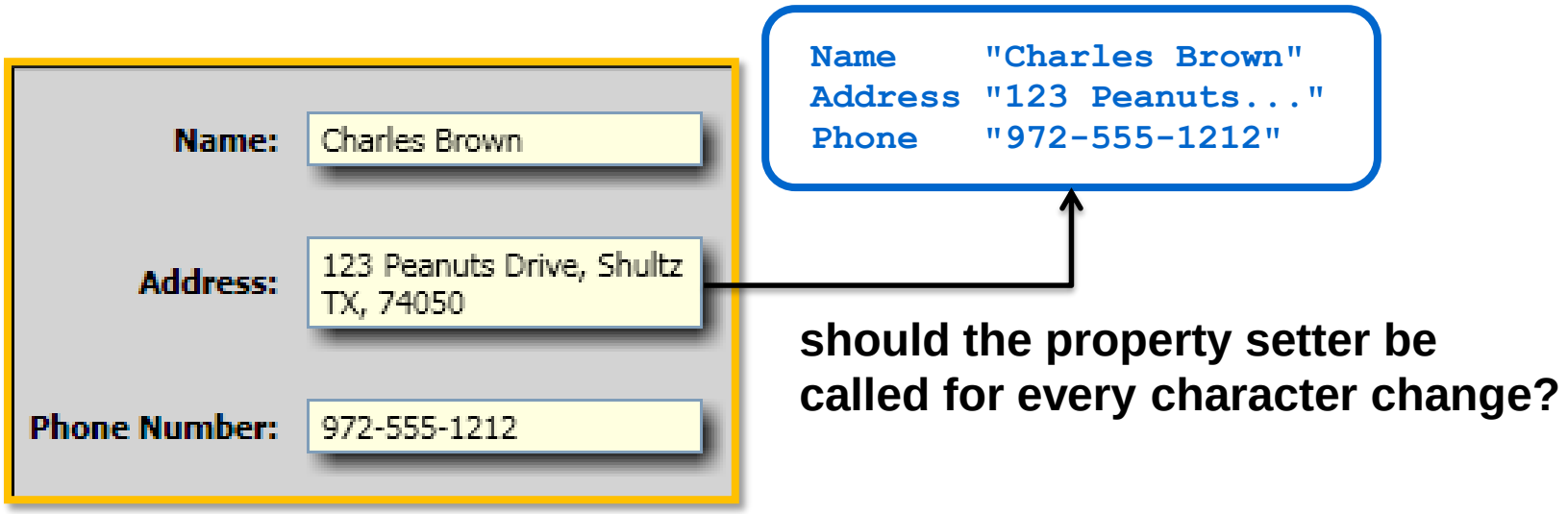
- **Binding `Mode` determines data transfer direction**
  - default decided by target **`DependencyProperty`**




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

Determining when the data exchange happens

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



what if a property change 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

```
Binding binding = new Binding();
binding.Source = customer;
binding.Path = new PropertyPath("Name");
binding.UpdateSourceTrigger = UpdateSourceTrigger.Explicit;
nameTextBox.SetBinding(TextBox.TextProperty, binding);
   ...
void UpdateCustomer()
{
   nameTextBox.GetBindingExpression(TextBox.TextProperty).
                                          UpdateSource();
   CallWebServiceToUpdate(customer);
}
```


Creating bindings in XAML

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

```
<StackPanel>
   <StackPanel.Resources>
      <me:Customer x:Key="customer" Name="Charles Brown" ... />
   </StackPanel.Resources>

   <Label>Name:</Label>

   <TextBox x:Name="nameTextBox">
      <TextBox.Text>
         <Binding Source="{StaticResource customer}"
                  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

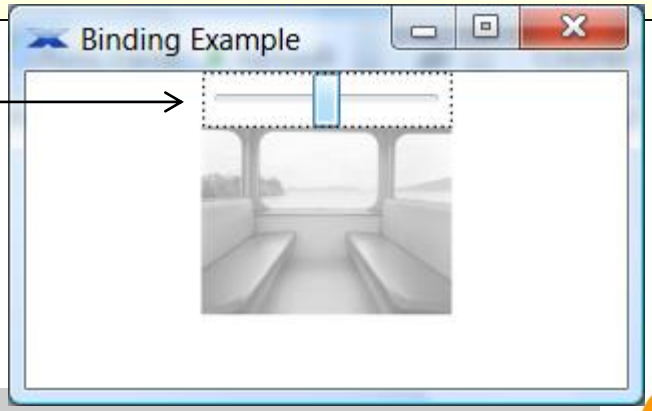
```
<StackPanel>
   <StackPanel.Resources>
      <me:Customer x:Key="customer" Name="Charles Brown" ... />
   </StackPanel.Resources>

   <TextBox x:Name="addressTextBox"
            Text="{Binding Source={StaticResource customer},
                  Path=Address}"/>
</StackPanel>
```

Binding in XAML: tying two elements together

- **`ElementName` associates elements in the same XAML file**
  - source located by `Name` or `x:Name` property

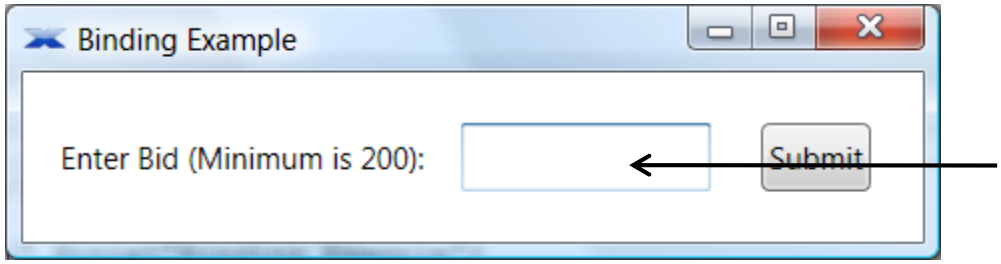
```
<StackPanel>
   <Slider Name="slider1" Minimum="0" Maximum="1"
           Width="100" Value="1" />

   <Image Source="img1.jpg" Width="100"
          Opacity="{Binding ElementName=slider1, Path=Value}" />

</StackPanel>
```

As slider is changed, Opacity of image changes automatically



What if the source and target are incompatible?

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



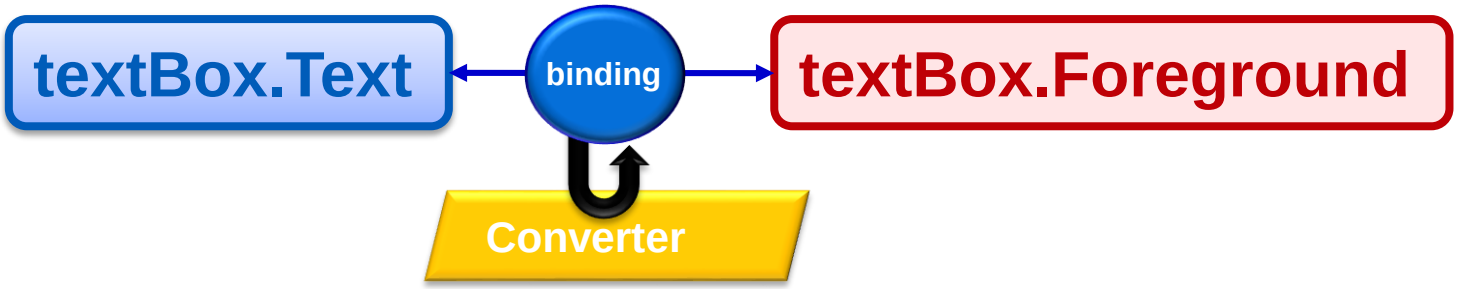
desire to change text color to Red if below minimum bid

```
<StackPanel>
   <Label>Enter Bid (Minimum is 200):</Label>
   <TextBox FontSize="36pt" Foreground="{Binding
     RelativeSource={RelativeSource Self}, Path=Text}" />
   ...
</StackPanel>
```

...but `Text` is a string, `Foreground` is a `Brush`

Converting data-bound values

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



- **Two methods defined on interface:**
  - `Convert` changes value from source to target type
  - `ConvertBack` changes value from target to source type

Example: Using a converter

- **`Converter` property identifies a specific instance to use**
  - can also specify **`ConverterParameter`** to pass to interface

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

   <Label>Enter Bid (Minimum is 200):</Label>
   <TextBox FontSize="36pt" Foreground="{Binding
         RelativeSource={RelativeSource Self}, Path=Text,
         Converter={StaticResource bidCvt}}" />
   ...
</StackPanel>
```

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>
      <me:Customer 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

Displaying non-Visuals in WPF

- **Elements render Content based on Type**
  - content which derives from **`UIElement`** calls **`OnRender()`**
  - non-UI content utilizes **`ToString()`**

```
public class Product
{
    public string Manufacturer {…}
    public string Name {…}
    public double Price {…}
}
```

```
<Button>
<me:Product
        Manufacturer="Apple"
        Name="iPod"
        Price="399" />
</Button>
```

override `ToString` to get custom *text* output

Introducing: Data Templates

- **Full visual representation specified through `DataTemplate`**
  - provides visual tree for the non-visual type
  - associated to control through **`ContentTemplate`** property

```
<Button ContentTemplate="{StaticResource ProductTemplate}">
    <me:Product
        Manufacturer="Apple"
        Name="iPod"
        Price="399" />
</Button>
```

`Button` now uses `DataTemplate` defined in resources to render `Product` type

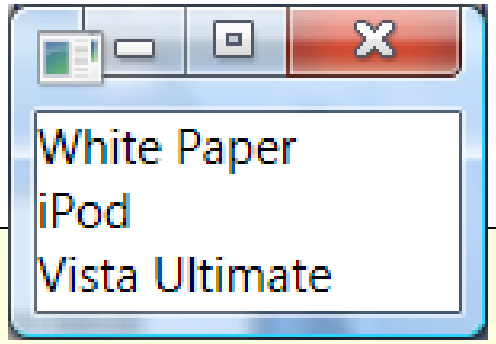
Creating a Data Template

1. Declare a `DataTemplate` object
2. Set the `DataType` property to the `System.Type` to represent
3. Add the visual representation as the content
4. Bind the properties of the visuals to the underlying type
   - `Source` is automatically set to the underlying type
5. Assign the data template to a control

```
<DataTemplate DataType="{x:Type me:Product}">
   <StackPanel>
      <TextBlock FontWeight="Bold" Text="{Binding Name}" />
      <TextBlock Text="{Binding Manufacturer}" />
      <TextBlock Foreground="Blue" Text="{Binding Price}" />
   </StackPanel>
</DataTemplate>
```

Displaying collections of objects

- **Collection oriented controls derive from `ItemsControl`**
  - `ItemsSource` property assigns data source
- **Data source can come from a variety of places**
  - **`IList`** (collections, arrays, generic collections)
  - **`IBindingList`** (**`DataTable`**, **`DataSet`**)
  - **`IEnumerable`** (LINQ, iterator methods)

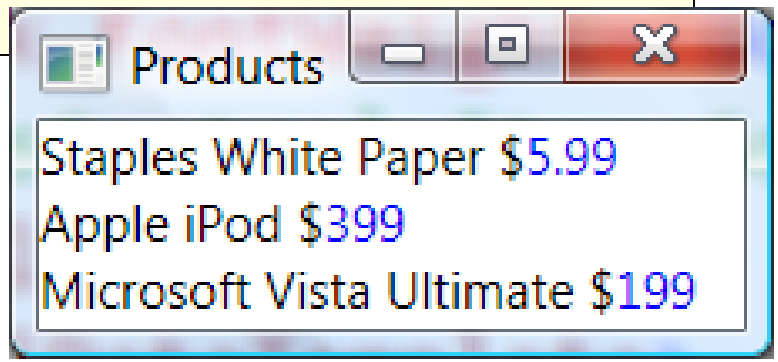



```
void InitializeComponent()
{
    string[] productList = GetProductListFromDatabase();
    itemList.ItemsSource = productList;
}
```

Using a DataTemplate with collections

- **Custom visual template can be assigned to `ItemTemplate`**
  - **`DataTemplate`** defined inline or in resources

```
<ItemsControl ItemsSource="{Binding}"
   <ItemsControl.ItemTemplate>
      <DataTemplate DataType="{x:Type Product}">
         <StackPanel>...</StackPanel>
      </DataTemplate>
   </ItemsControl.ItemTemplate>
</ItemsControl>
```

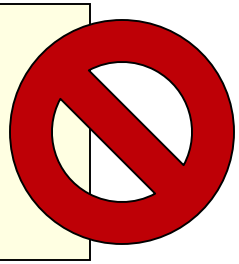


Changing the data

- **Cannot access `Items` collection of bound `ItemsControls`**
  - you must change the data *in the underlying collection*

```
<ListBox x:Name="productList"
         ItemsSource="{Binding}" />
```

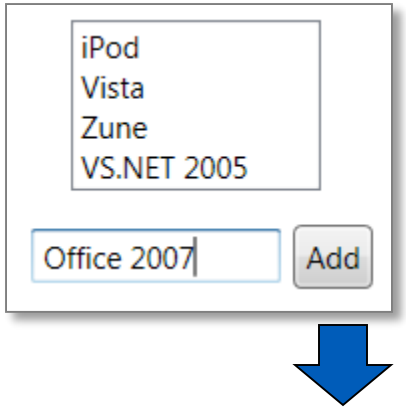
```
void OnAdd(Product newProduct)
{
   productList.Items.Add(newProduct);
}
```



```
List<Product> productList;
void OnAdd(Product newProduct) {
   productList.Add(newProduct);
}
```

How does WPF know the collection changed?

- **Collections in .NET have no built-in change notification**
  - WPF cannot "see" changes made to underlying collection




```
<StackPanel>
   <ListBox ItemsSource="{Binding}" />
   <StackPanel Orientation="Horizontal">
      <TextBox Name="textBoxProduct" />
      <Button Click="OnAdd">Add</Button>
   </StackPanel>
</StackPanel>
```

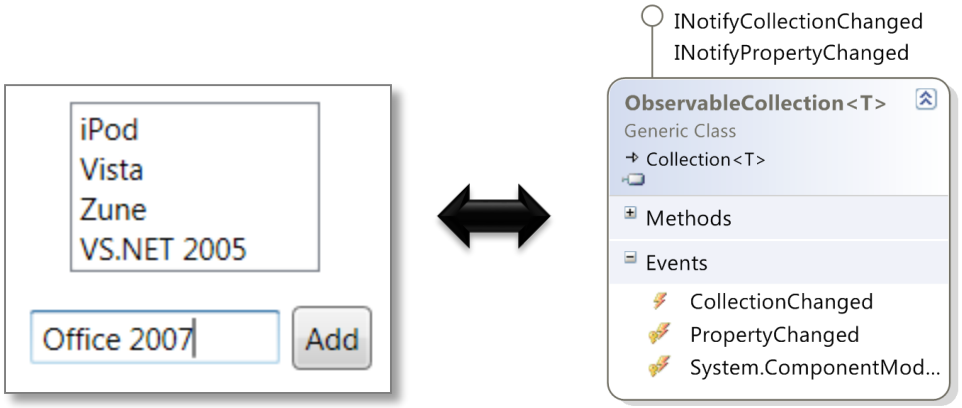
```
List<Product> productList;
void OnAdd(object sender, RoutedEventArgse) {
   productList.Add(new Product(textBoxProduct.Text));
}
```

Item is added to collection, but not ListBox

Two-way binding to collections

- **`ObservableCollection<T>` provides notification support**
  - implements **`INotifyCollectionChanged`**
  - keeps collection and bound target synchronized

```
ObservableCollection<Product> productList;
void OnAdd(object sender, RoutedEventArgs e) {
   productList.Add(new Product(textBoxProduct.Text));
}
```



Debugging data bindings

- **Failed data binding will output results to debug console**
  - any resulting exceptions automatically caught
  - can use pass-through data converter to place breakpoints
- **WPF 3.5 adds attached property to control level of output**
  - **`PresentationTraceSources.TraceLevel`**
  - value "**`Low|Medium|High`**" set on binding or data provider

```
<StackPanel xmlns:d="clr-namespace:System.Diagnostics;
                     assembly=WindowsBase">
   <Rectangle Width="100" Height="50">
      <Rectangle.Fill>
         <Binding Source="{StaticResource someBrush}"
                  d:PresentationTraceSources.TraceLevel="High" />
      </Rectangle.Fill>
   </Rectangle>
</StackPanel>
```

Summary

- **Goal of data binding is to separate visuals from logic**
  - XAML has all the visual information
  - code behind deals with PODOs
  - data binding enables two-way, automatic updates
- **DataContext can be used to share a common binding source**
  - reduce markup and allow dynamic change to source
- **Converters allow runtime conversions**
  - so code behind objects do not require WPF knowledge