



XAM310

Data Binding in Xamarin.Forms

Download class materials from
university.xamarin.com



Xamarin University

Information in this document is subject to change without notice. The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user.

Microsoft or Xamarin may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any license agreement from Microsoft or Xamarin, the furnishing of this document does not give you any license to these patents, trademarks, or other intellectual property.

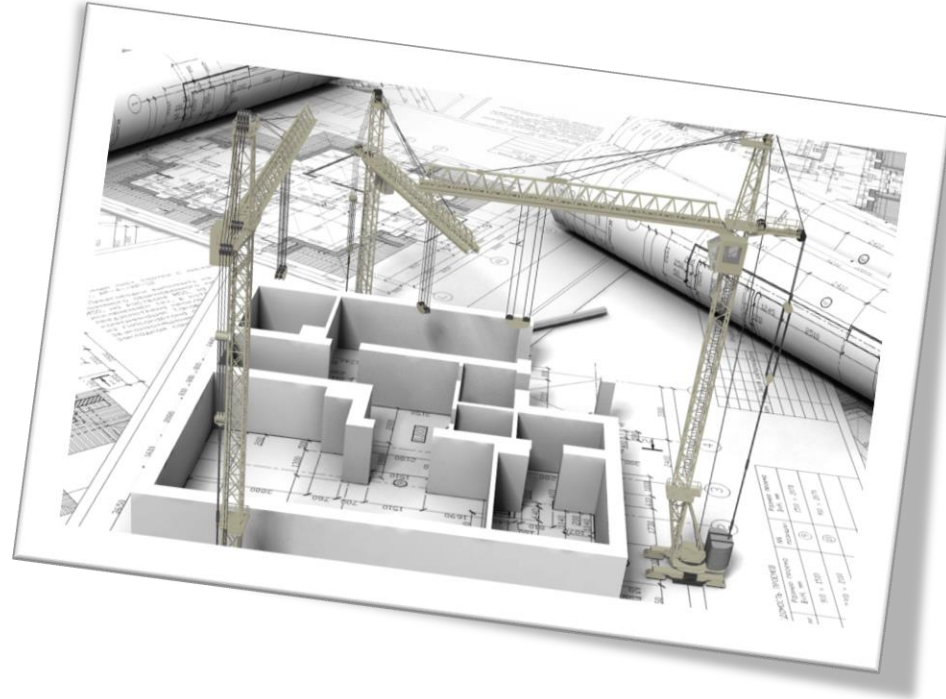
© 2014-2017 Xamarin Inc., Microsoft. All rights reserved.

Xamarin, MonoTouch, MonoDroid, Xamarin.iOS, Xamarin.Android, Xamarin Studio, and Visual Studio are either registered trademarks or trademarks of Microsoft in the U.S.A. and/or other countries.

Other product and company names herein may be the trademarks of their respective owners.

Objectives

1. Introduction to Data Binding
2. Dealing with Incompatible Types





Introduction to Data Binding

Tasks

1. Mapping Data to Visuals
2. Creating Bindings in Code
3. Creating Bindings in XAML
4. Working with Binding Context
5. Binding Modes
6. Property Change Notifications



Apps are driven by data

- ❖ Most applications display and manipulate data in some form
 - internally generated
 - read from an external source
- ❖ Classes created to represent data are often referred to as Models
 - can also refer to "entity" objects



Data > Views

- ❖ We use code to display internal data in our pages

```
headshot.Source = ...;  
nameEntry.Text = person.Name;  
emailEntry.Text = person.Email;  
birthday.Date = person.Dob;  
...
```

- ❖ ... and events to provide interactivity / behavior

```
nameEntry.TextChanged += (sender, e) =>  
    person.Name = nameEntry.Text;  
emailEntry.TextChanged += (sender, e) =>  
    person.Email = emailEntry.Text;
```



Data > Views in code

- ❖ This approach works, and for small-ish applications is perfectly adequate but it has disadvantages as the application grows in complexity



Updates to data are not centralized



Relationships in data or UI behavior is harder to manage



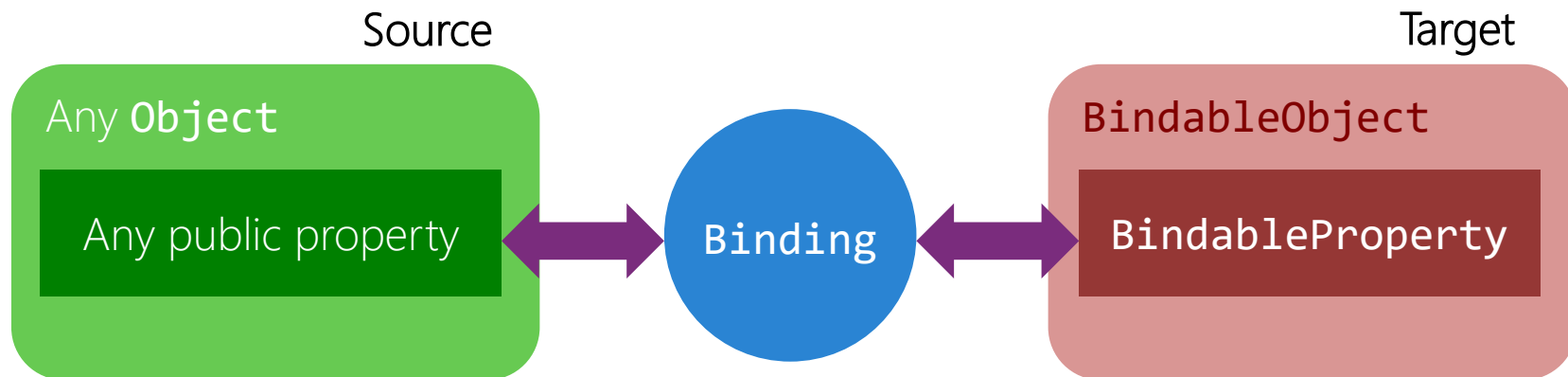
Hard to unit test



UI is tightly coupled to the code behind logic, changes ripple through code

Introducing: Data Binding

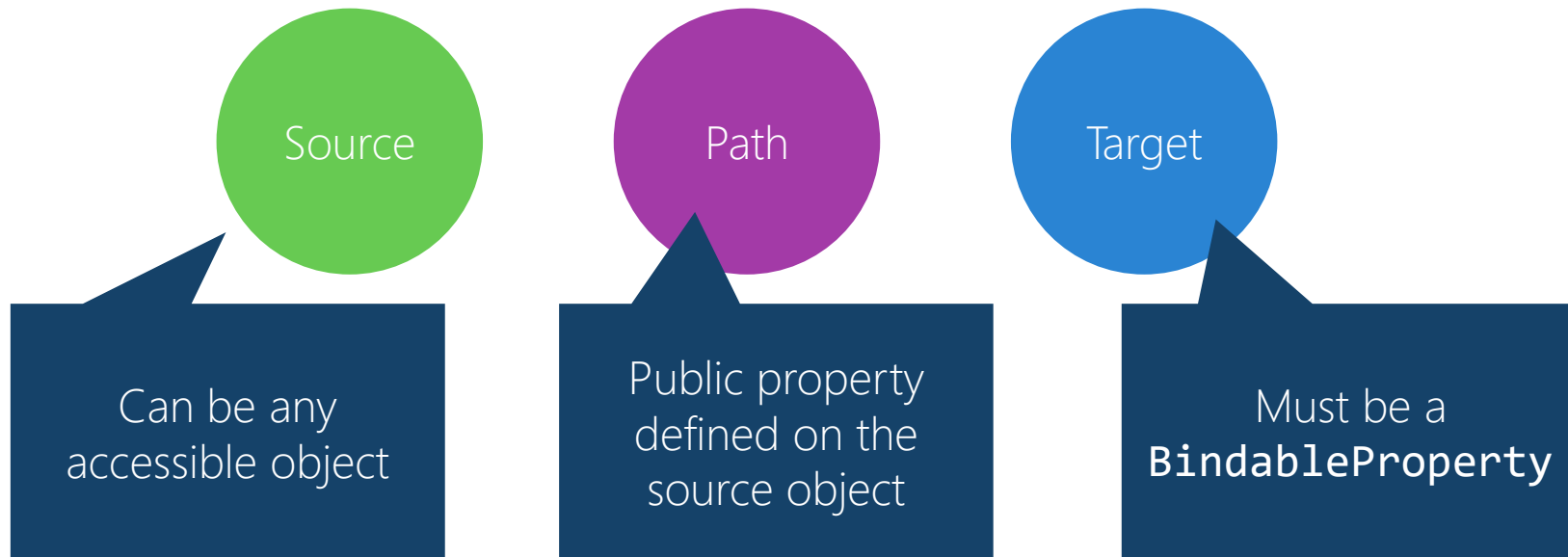
- ❖ Data Binding involves creating a loose **relationship** between a **source property** and a **target property** so that the source and target are **unaware of each other**



Binding acts as an *intermediary* – moving the data between the source and target

Creating Bindings in Xamarin.Forms

❖ Bindings require three pieces of information



Creating bindings [Source]

```
Person person = new Person() { Name = "Homer Simpson", ... };
```

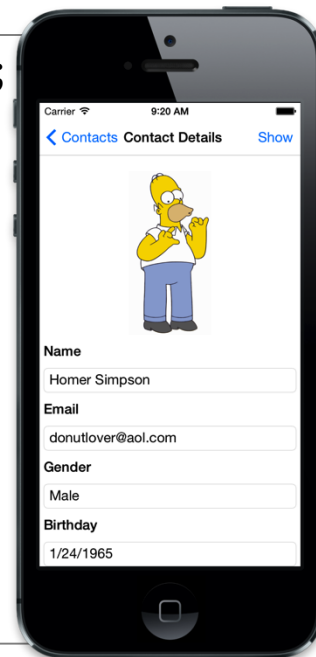
```
Entry nameEntry = new Entry();
```

1

```
Binding nameBinding = new Binding();  
nameBinding.Source = person;
```

...

Binding identifies the **source** of the binding data – this is where the data comes from, in this case it's a single person defined in our application



Creating bindings [Path]

```
Person person = new Person() { Name = "Homer Simpson", ... };
```

```
Entry nameEntry = new Entry();
```

```
Binding nameBinding = new Binding();
```

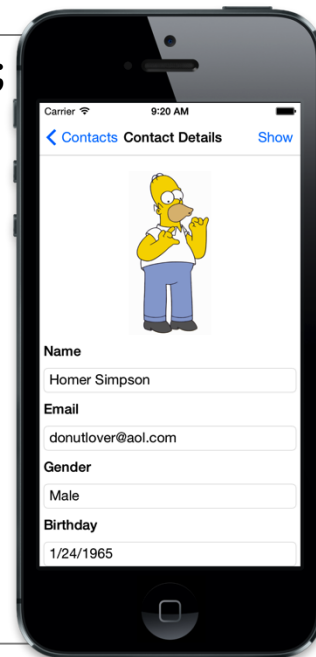
```
nameBinding.Source = person;
```

```
nameBinding.Path = "Name";
```

...

2

Binding identifies the *property path* which identifies a property on the source to get the data from, in this case we want to get the value from the **Person.Name** property



Creating bindings [Path]

```
Person person ... };
```

```
Entry nameEnt
```

```
Binding nameB
```

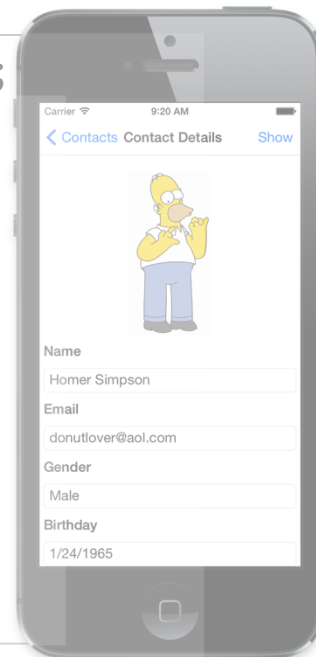
```
nameBinding.S
```

```
nameBinding.P
```

```
...
```

More Path Examples

```
new Binding("Property")  
new Binding("Property.Child")  
new Binding("Property[Key]")  
new Binding("Property[1]")  
new Binding("Item[Key]")  
new Binding(".")
```



Creating bindings [Target]

```
Person person = new Person() { Name = "Homer Simpson", ... };
```

```
Entry nameEntry = new Entry();
```

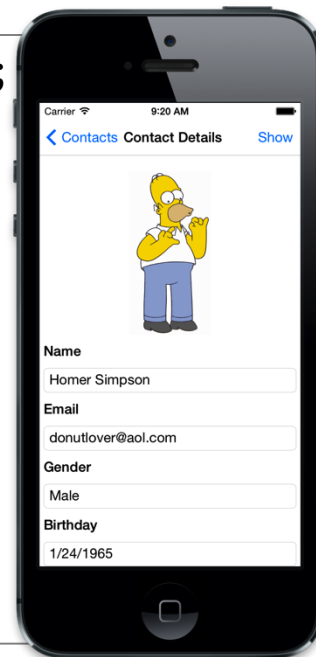
```
Binding nameBinding = new Binding();
```

```
nameBinding.Source = person;
```

```
nameBinding.Path = "Name";
```

3 `nameEntry.SetBinding(Entry.TextProperty, nameBinding);`

Binding is associated to the target property using the **BindableObject.SetBinding** method



Creating bindings [Target]

```
Person person = new Person() { Name = "Homer Simpson", ... };
```

```
Entry nameEntry = new Entry();
```

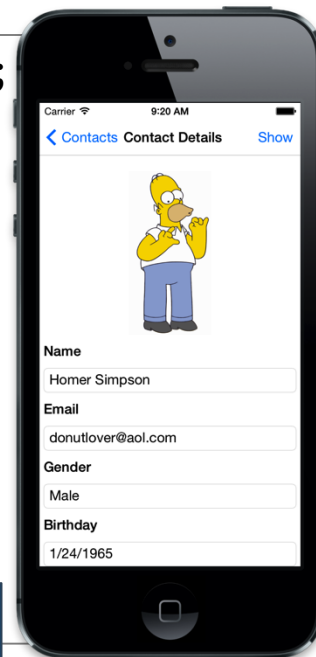
```
Binding nameBinding = new Binding();
```

```
nameBinding.Source = person;
```

```
nameBinding.Path = "Name";
```

3 nameEntry.SetBinding(Entry.TextProperty, nameBinding);

This is passed the specific target property the binding will work with – this must be a **BindableProperty**



Creating bindings [Target]

```
Person person = new Person() { Name = "Homer Simpson", ... };
```

```
Entry nameEntry = new Entry();
```

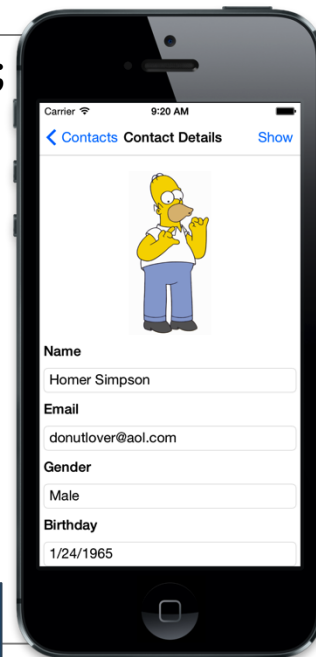
```
Binding nameBinding = new Binding();
```

```
nameBinding.Source = person;
```

```
nameBinding.Path = "Name";
```

3 nameEntry.SetBinding(Entry.TextProperty, nameBinding);

... and the binding which identifies the source and the property on the source to apply



Creating bindings [XAML]

- ❖ Create bindings in XAML with **{Binding}** markup extension

```
<StackLayout Padding="20" Spacing="20">
  <StackLayout.Resources>
    <ResourceDictionary>
      <Person x:Key="homer" Name="Homer Simpson" .../>
    </ResourceDictionary>
  </StackLayout.Resources>
  <Entry Text="{Binding Name,
    Source={StaticResource homer}}" />
  ...
</StackLayout>
```

{Binding} takes the Path as the first unnamed argument

Source supplied through resource

Assigned to Target property

Data binding source

- ❖ Pages often display properties from a small number of data objects
- ❖ Can set the binding source on each binding separately, or use the **BindingContext** as the *default* binding source

```
public class Person
{
    public string Name { get; set; }
    public string Email { get; set; }
    public Gender Gender { get; set; }
}
```

Name

Homer Simpson

Email

donutlover@aol.com

Gender


Male



Multiple Bindings

- ❖ **BindingContext** supplies the source for any binding associated with a view when the **Binding.Source** property is **not set**

```
Person person = new Person() { Name = "Homer Simpson", ... };  
...  
nameEntry.BindingContext = person;  
nameEntry.SetBinding<Person>(Entry.TextProperty,  
                             data => data.Name, BindingMode.TwoWay);
```



Useful to use a generic form of **SetBinding** to create bindings with typed properties when establishing bindings in code, notice we are *not* setting a source property on the binding – instead, it will use **BindingContext**

BindingContext inheritance

- ❖ **BindingContext** is automatically *inherited* from parent to child – can set it once on the root view and it will be used for all children

```
public partial class PersonDetailsPage : ContentPage
{
    public PersonDetailsPage (Person person)
    {
        BindingContext = person;
        InitializeComponent ();
    }
}
```


The diagram illustrates the concept of BindingContext inheritance. On the left, a code snippet shows the `PersonDetailsPage` class inheriting from `ContentPage`. The `InitializeComponent()` method is highlighted with a purple box. Four purple arrows originate from this box and point to the corresponding UI controls on the right: the `Name` text box, the `Email` text box, the `Gender` text box, and the `Birthday` text box. A fifth purple arrow points from the `BindingContext` property to the `Favorite` toggle switch, which is also highlighted with a purple box. The UI controls are arranged vertically and labeled: **Name** (Homer Simpson), **Email** (donutlover@aol.com), **Gender** (Male), **Birthday** (1/24/1965), and **Favorite** (a green toggle switch).

BindingContext inheritance

- ❖ **BindingContext** is automatically *inherited* from parent to child – can set it once on the root view and it will be used for all children

```
BindingContext = person;
```

```
<StackLayout Padding="20" Spacing="20">  
  <Entry Text="{Binding Name}" />  
  <Entry Text="{Binding Email}" />  
  <Label Text="{Binding Gender}" />  
  <Label Text="{Binding Dob}" />  
</StackLayout>
```



By setting the binding context to the **Person**, no explicit source is necessary in XAML

Group Exercise

Using Data Binding in a Xamarin.Forms Application

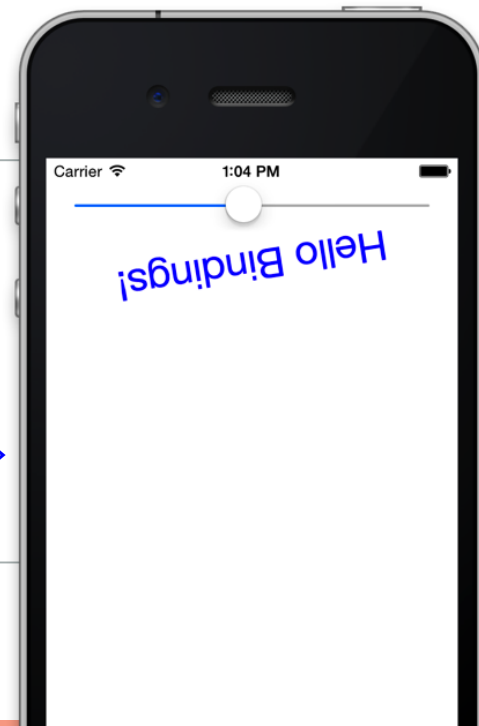


Xamarin
University

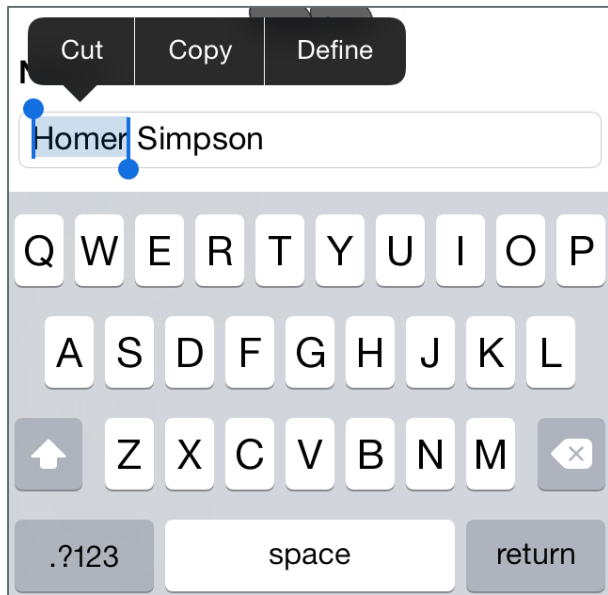
View-to-View Bindings

- ❖ **{x:Reference}** identifies named elements in the same XAML page – can use this to provide a source to a **Binding**

```
<StackLayout Padding="20" Spacing="20">  
  <Label Text="Hello, Bindings" TextColor="Blue" ...  
    Rotation="{Binding Source={x:Reference slider},  
      Path=Value}" />  
  ...  
  <Slider x:Name="slider" Minimum="0" Maximum="360" />  
</StackLayout>
```



Creating two-way bindings



- ❖ Typically want data to be bi-directional
 - source > target (always happens)
 - target > source (optional)

```
nameEntry.TextChanged += (sender, e)
=> person.Name = nameEntry.Text;
```


Binding Mode

- ❖ Binding **Mode** controls the direction of the data transfer, can set to "TwoWay" to enable bi-directional bindings

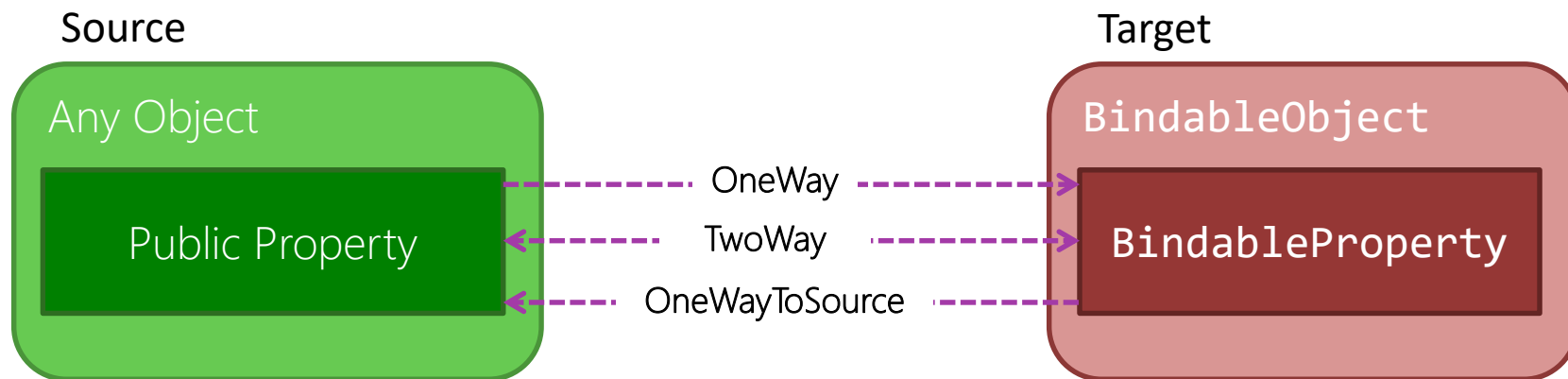
```
name.SetBinding(Entry.TextProperty,  
    new Binding("Name") {  
        Mode = BindingMode.TwoWay  
    });
```

```
<Entry  
    Text="{Binding Name, Mode=TwoWay}" />
```

Source Property must
have **public setter**

Manually controlled through the
Binding.Mode property

Available Binding Modes




BindingMode.Default is the default value and it decides the mode based on the target property preference – either **OneWay** or **TwoWay**

OneWayToSource binding

- ❖ OneWayToSource solves the limitation of having a single binding context

Text comes from the **Page's BindingContext**



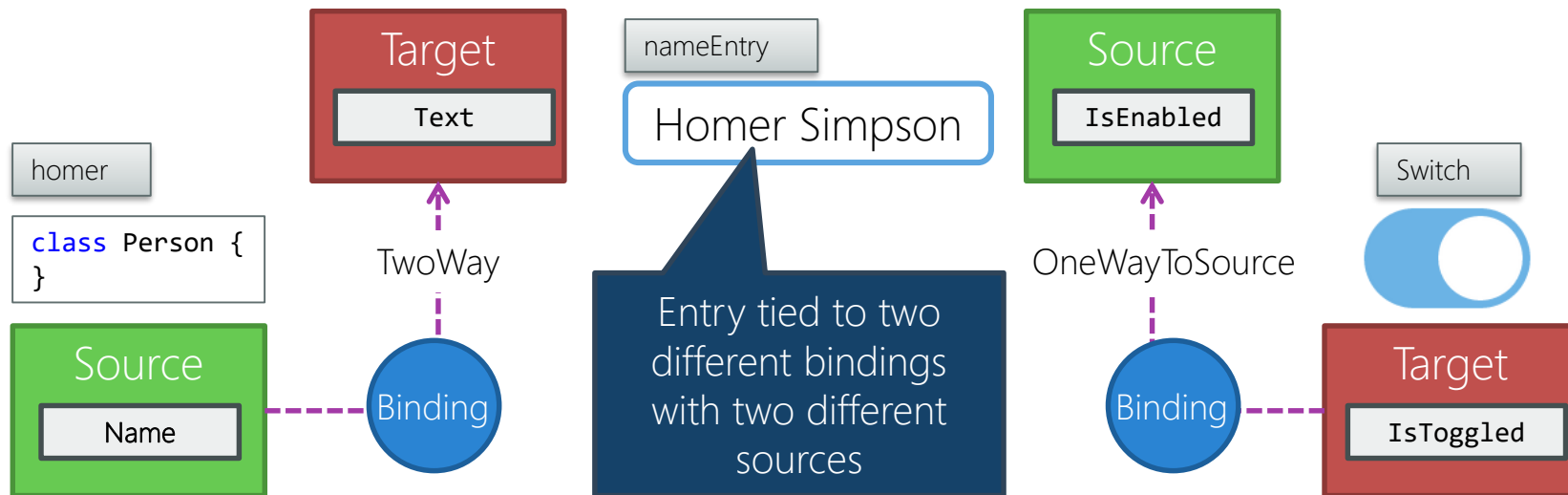
```
<StackLayout Padding="20" Spacing="20">
  <Label FontSize="36" Text="Name" />
  <Entry x:Name="nameEntry" Text="{Binding Name}" />

  <Switch IsToggled="{Binding IsEnabled, Mode=OneWayToSource}"
    BindingContext="{x:Reference nameEntry}" />
</StackLayout>
```

Switch becomes the source, identified **BindingContext** becomes the target by setting the mode to **OneWayToSource**

OneWayToSource binding

- ❖ **OneWayToSource** solves the limitation of having a single binding context



Default Binding Mode

- ❖ Default binding mode is *property-specific*, most are one-way by default with a few exceptions that default to two-way

`DatePicker.Date`

`SearchBar.Text`

`Entry.Text`

`Stepper.Value`

`ListView.SelectedItem`

`Switch.IsToggled`

`MultiPage<T>.SelectedItem`

`TimePicker.Time`

`Picker.SelectedIndex`



XAML platforms handle **binding modes differently**, best practice to get in the habit of explicitly setting the mode if it's not one-way – even if it defaults to what you want


Pushing changes to the UI

- ❖ One-Way and Two-Way bindings always update the UI when the source property is changed

```
person.Email = "homer@dunkin.com";
```

```
public class Person
{
    public string Name { get; set; }
    public string Email { get; set; }
    public Gender Gender { get; set; }
    public DateTime Dob { get; set; }
    public bool IsFavorite { get; set; }
}
```

Q: How does Xamarin.Forms know **Email** has changed?



Name	<input type="text" value="Homer Simpson"/>
Email	<input type="text" value="donutlover@aol.com"/>
Gender	<input type="text" value="Male"/>
Birthday	<input type="text" value="1/24/1965"/>
Favorite	<input checked="" type="checkbox"/>

INotifyPropertyChanged

- ❖ **INotifyPropertyChanged** provides change notification contract, should be implemented by any modifiable model object you bind to

```
namespace System.ComponentModel
{
    public interface INotifyPropertyChanged
    {
        event PropertyChangedEventHandler PropertyChanged;
    }
}
```

Implementing INotifyPropertyChanged

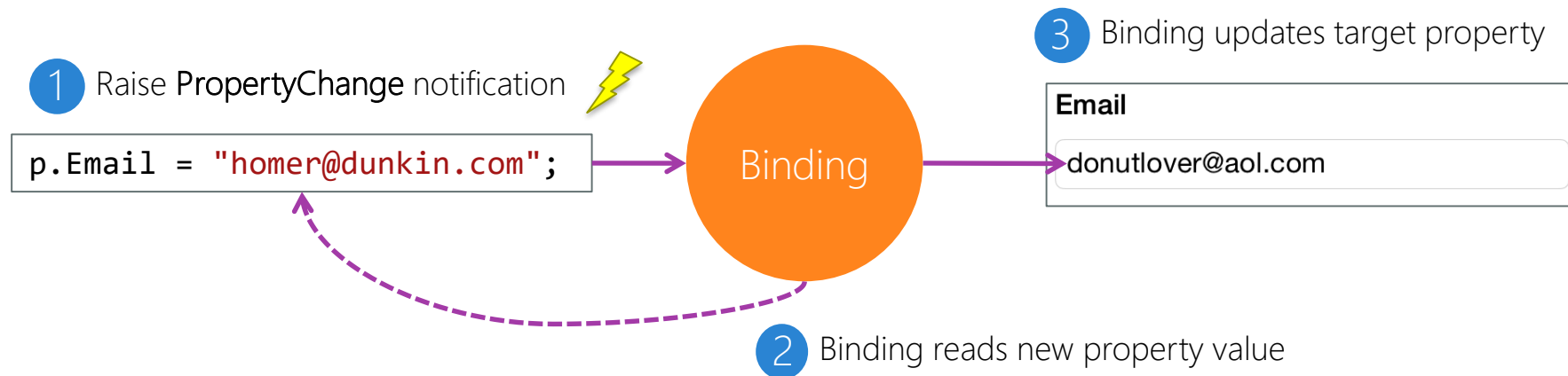
```
public class Person : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged = delegate {};

    string email;
    public string Email {
        get { return email; }
        set {
            if (email != value) {
                email = value;
                PropertyChanged(this,
                    new PropertyChangedEventArgs("Email"));
            }
        }
    }
}
```

Must raise the **PropertyChanged** event when any property is changed – otherwise the UI will not update

INPC + Bindings

- ❖ Binding will subscribe to the **PropertyChanged** event and update the target property when it sees the source property notification





Individual Exercise

Working with Two-Way Bindings



Xamarin
University

Flash Quiz

Flash Quiz

- ① The source data is supplied through _____ (Select all that apply).
- a) DataContext property
 - b) Binding.Source property
 - c) BindingContext property
 - d) None of the above

Flash Quiz

- ① The source data is supplied through _____ (Select all that apply).
- a) DataContext property
 - b) Binding.Source property
 - c) BindingContext property
 - d) None of the above

Flash Quiz

- ② The source can be any object
- a) True
 - b) False

Flash Quiz

- ② The source can be any object
- a) True
 - b) False

Flash Quiz

- ③ The target can be any object
- a) True
 - b) False

Flash Quiz

- ③ The target can be any object
- a) True
 - b) False

Flash Quiz

- ④ Model objects should perform the following steps when a property setter is called (pick the best answer):
- a) Change the property and raise the PropertyChanged event
 - b) Check if the property is different, change the property and raise the PropertyChanged event
 - c) Check if the property is different, raise the PropertyChanged event and then change the property
 - d) None of these are correct

Flash Quiz

- ④ Model objects should perform the following steps when a property setter is called (pick the best answer):
- a) Change the property and raise the PropertyChanged event
 - b) Check if the property is different, change the property and raise the PropertyChanged event
 - c) Check if the property is different, raise the PropertyChanged event and then change the property
 - d) None of these are correct

Summary

1. Mapping Data to Visuals
2. Creating Bindings in Code
3. Creating Bindings in XAML
4. Working with Binding Context
5. Binding Modes
6. Property Change Notifications





Dealing with Incompatible Types

Tasks

1. Simple text conversions
2. Working with Incompatible Types
3. Value Converters
4. Simple Textual Conversions



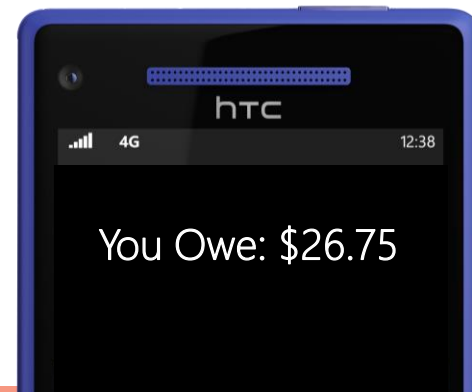
Simple Textual Conversions

- ❖ Binding can do simple, text formatting when going from Source > Target

```
public double BillAmount { get; set; }
```

```
<Label Text="{Binding BillAmount,  
                        StringFormat='You Owe: {0:C}'}"/>
```

Binding calls a **String.Format** passing the specified format string and the source value before assigning it to the target



Going beyond textual formatting

- ❖ Bindings attempt to **automatically coerce data** when C# would allow it, but sometimes the data available isn't quite what the UI needs to display

Enter Password

Invalid

Enter Password

Excellent

Want the text color to change based on the password strength

```
<Label Text="{Binding PasswordStrength}"
      TextColor="{Binding PasswordStrength}"
      FontSize="24" />
```


Value Converters

- ❖ Value Converters enable type coercion and formatting
- ❖ Assigned to **Converter** property of Binding
- ❖ Supports optional parameter (**Binding.ConverterParameter**)

```
public interface IValueConverter
{
    object Convert(object value,
                   Type targetType,
                   object parameter,
                   CultureInfo culture);

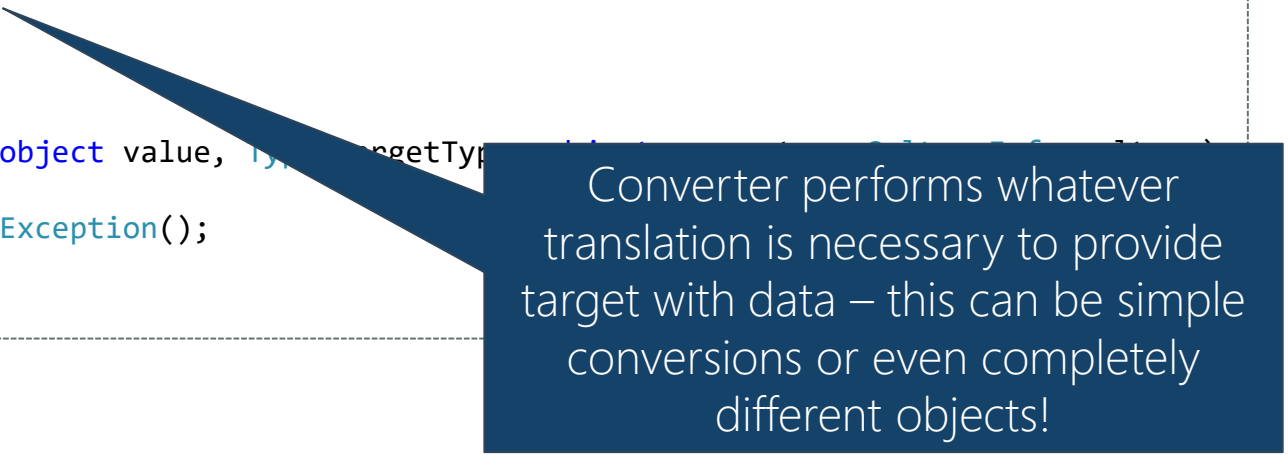
    object ConvertBack(object value,
                      Type targetType,
                      object parameter,
                      CultureInfo culture);
}
```

Convert used for source ➡ target
ConvertBack used for target ➡ source

Creating a Value Converter

```
public class PWStrengthConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        PasswordStrength pwdstr = (PasswordStrength) value;
        ...
        return Color.Red;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

A dark blue callout box with a white arrow pointing from the 'Convert' method in the code to the text. The text explains the role of the converter.

Converter performs whatever translation is necessary to provide target with data – this can be simple conversions or even completely different objects!

Creating a Value Converter

Provides backwards conversion for two-way binding, or can throw exception if this is not supported – this will cause a runtime failure

```
public class PWStrengthConverter :
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        PasswordStrength pwdstr =
        ...
        return Color.Red;
    }

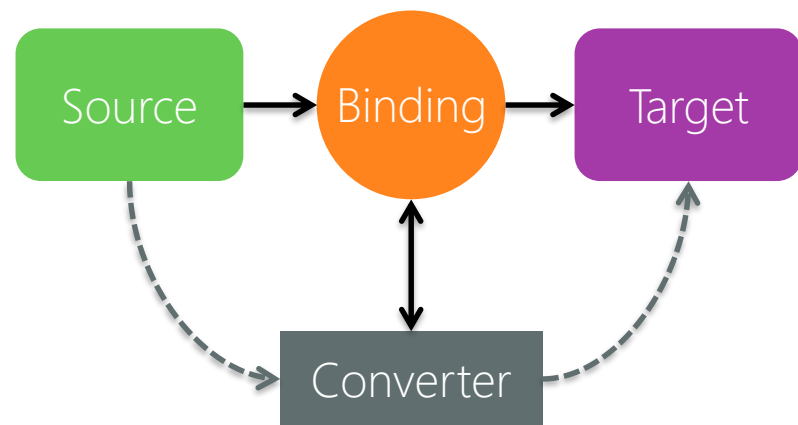
    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotSupportedException();
    }
}
```

Using a Value Converter

- ❖ Value Converter is assigned to the binding **Converter** property

```
var binding = new Binding("PasswordStrength"){  
    Converter = new PWStrengthConverter()  
};
```

```
<ContentPage.Resources>  
    <ResourceDictionary>  
        <c:PWStrengthConverter x:Key="pwsCvt"/>  
    </ResourceDictionary>  
</ContentPage.Resources>  
  
<Label TextColor="{Binding PasswordStrength,  
    Converter={StaticResource pwsCvt}}" />
```



Binding passes values through converter

Debugging Bindings

- ❖ Can use dummy converter to debug data bindings – gets called during the data transfer and provides for a convenient breakpoint location

```
10 public class DummyConverter : IValueConverter
11 {
12     public object Convert(object value, Type targetType,
13         object parameter, CultureInfo culture) {
14         return value;
15     }
16
17     public object ConvertBack(object value, Type targetType,
18         object parameter, CultureInfo culture) {
19         return value;
20     }
21 }
```

Individual Exercise

Using Value Converters



Xamarin
University

Flash Quiz

Flash Quiz

- ① **IValueConverter.Convert** is called when going from ____ to ____
- a) Source > Target
 - b) Target > Source

Flash Quiz

- ① **IValueConverter.Convert** is called when going from ____ to ____
- a) Source > Target
 - b) Target > Source

Flash Quiz

- ② To pass a binding-specific parameter to a value converter, you can set the _____ property.
- a. Parameter
 - b. ConversionParameter
 - c. ConverterParameter
 - d. BindingParameter

Flash Quiz

- ② To pass a binding-specific parameter to a value converter, you can set the _____ property.
- a. Parameter
 - b. ConversionParameter
 - c. ConverterParameter
 - d. BindingParameter

Flash Quiz

- ③ **Binding.StringFormat** can be used to convert an integer type to a double type
- a. True
 - b. False

Flash Quiz

- ③ **Binding.StringFormat** can be used to convert an integer type to a double type
- a. True
 - b. False

Summary

1. Working with Incompatible Types
2. Value Converters
3. Simple Textual Conversions



Thank You!

Please complete the class survey in your profile:
university.xamarin.com/profile

