XAM335

Xamarin.Forms Renderers

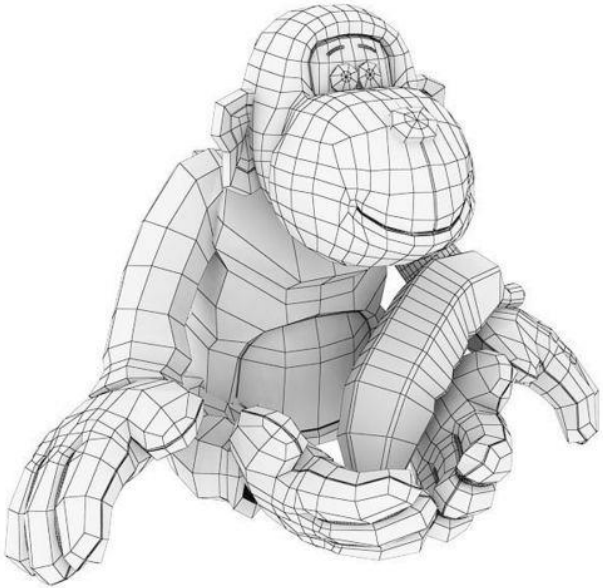
Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Embed native controls into Xamarin.Forms
2. Customize a renderer for an existing control
3. Create a renderer for a custom control
4. Send notifications between renderer and element



Embed native controls into Xamarin.Forms

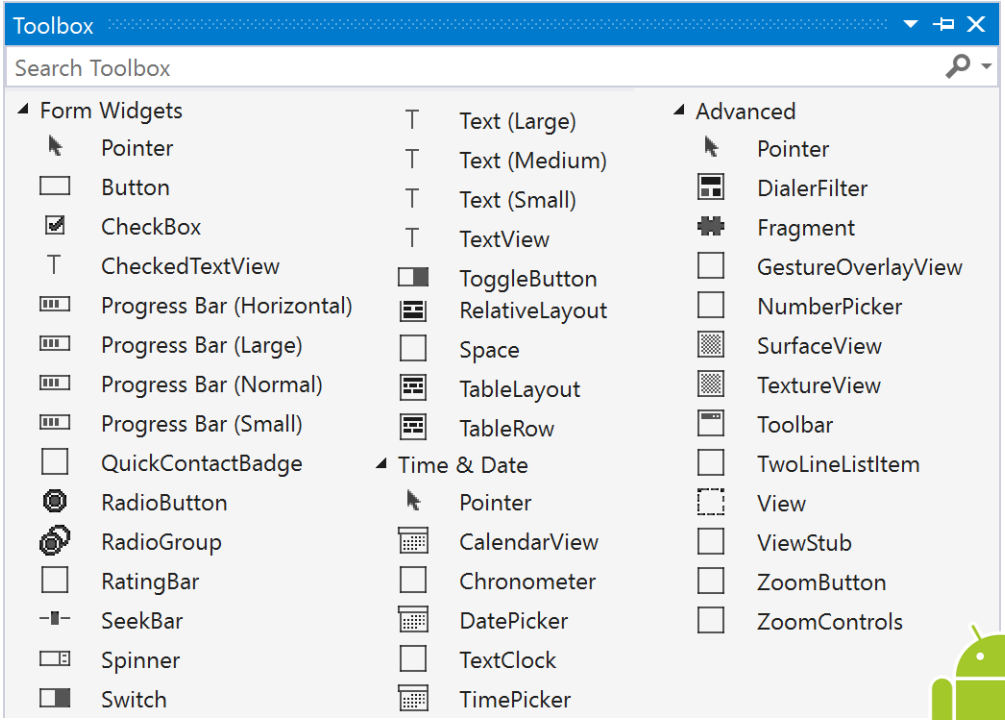
Xamarin University

Tasks

1. Define a native control
2. Add a native control to a Xamarin.Forms layout

What is a native control?

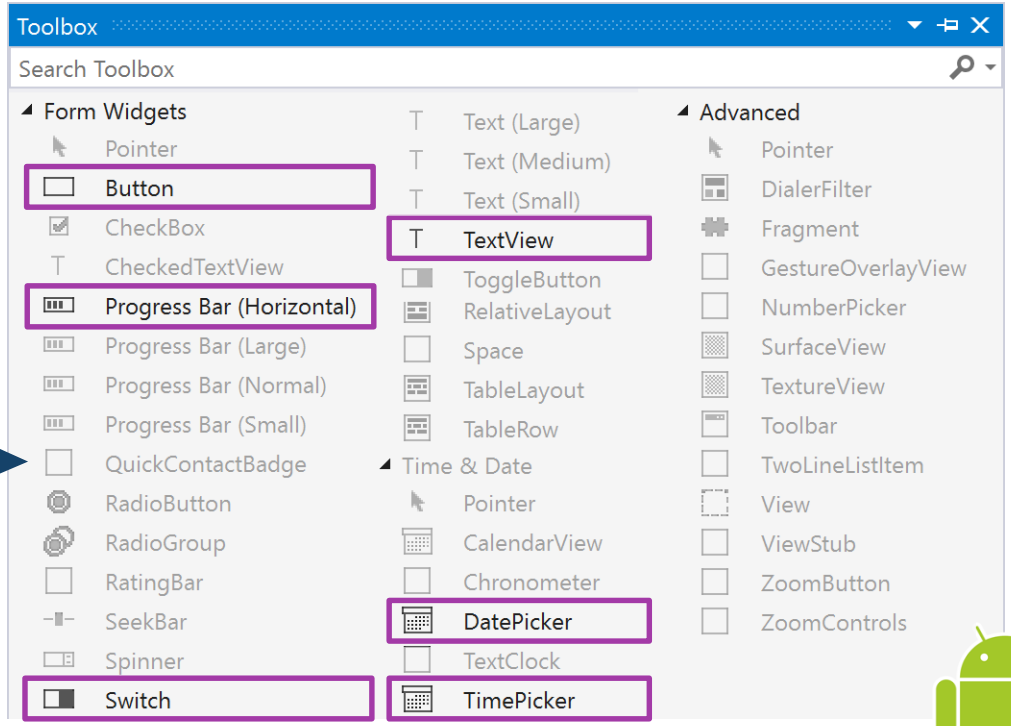
- Each platform has a rich selection of controls – many of which are unique to the individual OS
- Native controls are what actually present UI in your application



Control selection in Xamarin.Forms

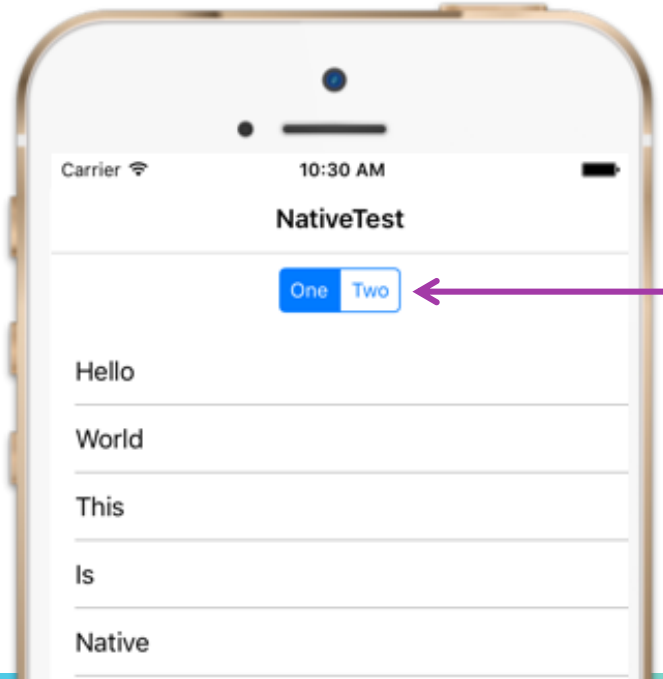
- Xamarin.Forms exposes a common set of controls across all platforms

Many of the native controls do not have Xamarin.Forms versions



Native controls in Xamarin.Forms

- Xamarin.Forms supports native controls in your Xamarin.Forms UI – this lets you use controls that are not directly provided



A native control like the iOS **`UISegmentedControl`** can be embedded into a Xamarin.Forms layout

Type incompatibility

- Xamarin.Forms containers hold types derived from **`Xamarin.Forms.View`** – the native controls are not compatible

```
namespace Xamarin.Forms
{
    public class ContentView : TemplatedView
    {
        public View Content { get; set; }
    }

    public class StackLayout : Layout<View>
    {
        ...
    }
```

Containers store Xamarin.Forms **`Views`**

shared

platform

NSObject

UIResponder

UIView

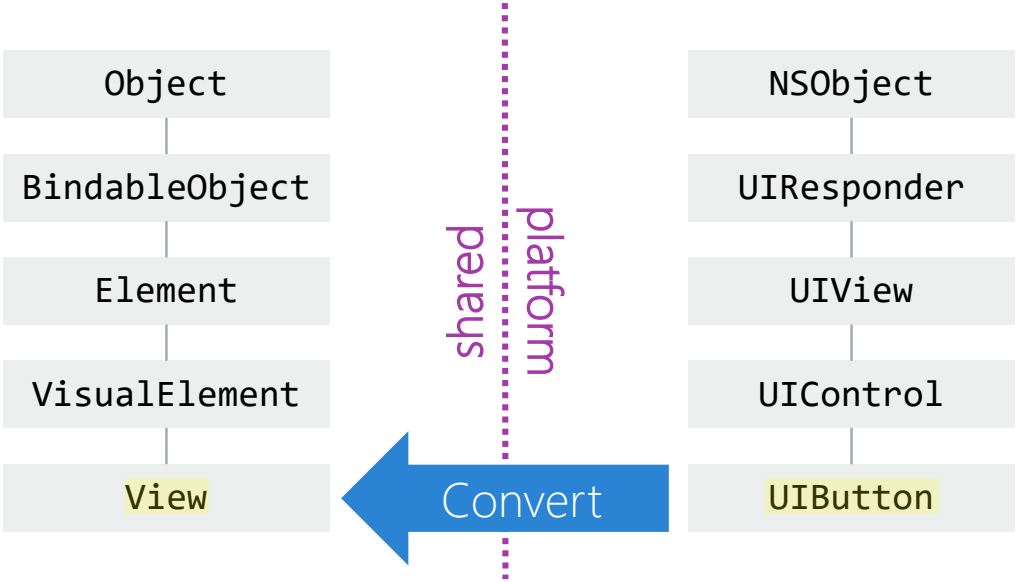
UIControl

UIButton

`UIButton` is not a Xamarin.Forms **`View`**

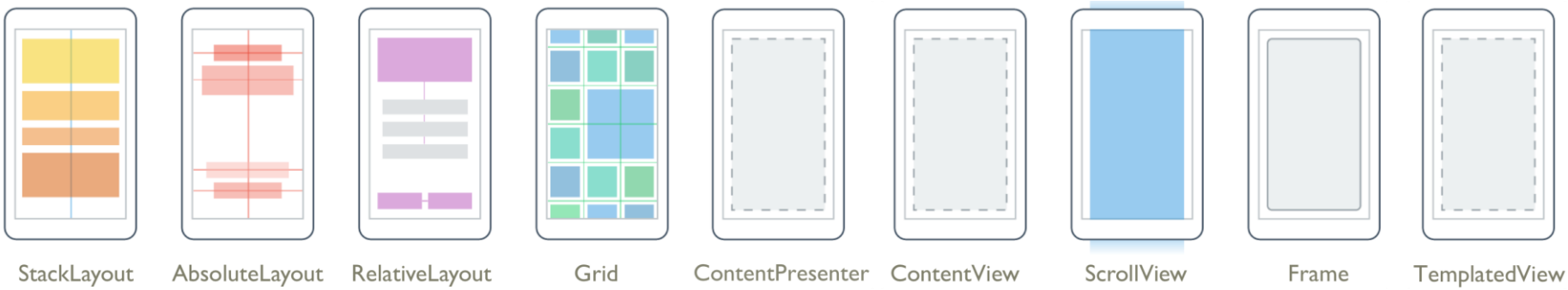
Type conversion

- Native controls must be converted to Xamarin.Forms **`View`**s before they can be added to Xamarin.Forms containers



How to add native controls

- You can add native controls to both Xamarin.Forms layouts and content controls; however, the techniques you use are slightly different



Use the **`Add`** extension method to convert and add all at once

Use the **`ToView`** extension method to convert, then load manually

Layout [extension methods]

- Xamarin.Forms provides extension methods on each platform to add native controls to a Xamarin.Forms layout

```
namespace Xamarin.Forms.Platform.Android
{
  public static class LayoutExtensions
  {
    public static
      void Add(this IList<Xamarin.Forms.View> children, Android.Views.View view, ...)
      {
        ...
      }
  }
}
```

The methods extend **IList** because that is the type of a layout's **Children** collection

Add an Android **View** to a collection of Xamarin.Forms **Views**

Layout [add]

- The **Add** extension method allows native controls to be added to layouts with a **Children** collection

```
var xfStack = new Xamarin.Forms.StackLayout();

var uwpButton = new Windows.UI.Xaml.Controls.Primitives.RepeatButton();

xfStack.Children.Add(uwpButton);
```

Xamarin.Forms
StackLayout

Native UWP
RepeatButton

Conversion [extension methods]

- Xamarin.Forms provides **`ToView`** extension methods that convert a native control to a Xamarin.Forms **`View`**

```
namespace Xamarin.Forms.Platform.Android
{
  public static class LayoutExtensions
  {
    public static Xamarin.Forms.View ToView(this Android.Views.View view, ...)
    {
      ...
    }
  }
}
```

Return a Xamarin.Forms **`View`**

Defined on each platform's native base visual type

Conversion [add]

- First convert the native view using **ToView**, then load the result into the Xamarin.Forms container

```
var iOSButton = UIButton.FromType(UIButtonType.DetailDisclosure);
iOSButton.TouchUpInside += () => { ... };

View xfView = iOSButton.ToView();

var xfContentView = new ContentView();
xfContentView.Content = xfView;
```

Get a **View** that can be used in the Xamarin.Forms visual tree

The Xamarin.Forms **View** can be assigned to the content property

Embedded controls in shared projects

- Can add native controls to your Xamarin.Forms UI from within a shared project by isolating the platform specific code with compiler directives

Add **using** statements inside guards

Add platform-specific views inside guards

```
#if __ANDROID__
using Android.Widget;
#endif

public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

#if __ANDROID__
        mainLayout.Children.Add(new CheckBox());
```

Embedded controls in PCLs

- When your Xamarin.Forms UI is defined in a PCL, the native controls must be added from the platform-specific projects using an abstraction

PCL defines the interface

```
public interface ICheckBoxFactory
{
    Xamarin.Forms.View GetCheckBox(string title, Action Checked);
}
```

Each native project implements it

```
class CheckBoxFactory : ICheckBoxFactory
{
    Context context;
    public Xamarin.Forms.View GetCheckBox(string title, Action Checked)
    {
        var cb = new Android.Widget.CheckBox(context) { Text = title };
        cb.CheckedChange += (s, e) => Checked();
        return cb.ToView();
    }
}
```

Use **ToView** to return a Xamarin Forms view

Exercise

Add a native control to a Xamarin.Forms layout

Xamarin University

Summary

1. Define a native control
2. Add a native control to a Xamarin.Forms layout

Customize a renderer for an existing control

Xamarin University

Tasks

1. Extend an existing renderer
2. Apply a customized renderer

Reminder: elements are models

- Xamarin.Forms Elements are platform-independent *representations* of the UI we want to create and display

Public properties are used to customize runtime visuals and behavior

```
public class Button : Element
{
    public Color  BorderColor  { get; set; }
    public int    BorderRadius { get; set; }
    public double BorderWidth  { get; set; }
    public string Text         { get; set; }
    public Color  TextColor    { get; set; }
    ...
```

shared

platform

Reminder: From Element to Visual

- At runtime, a platform-specific control is created to visualize each Xamarin.Forms Element

```
public class Button : Element
{
    public Color  BorderColor  { get; set; }
    public int    BorderRadius { get; set; }
    public double BorderWidth  { get; set; }
    public string Text         { get; set; }
    public Color  TextColor    { get; set; }
    ...
```

shared

platform

Android.Widget.Button

Click Me, I Dare You!

UIKit.UIButton

Click Me, I Dare You!

Windows.UI.Xaml.Controls.Button

Click Me, I Dare You!

Reminder: What is a platform renderer?

- A *platform renderer* is the code that translates a Xamarin.Forms Element to a platform-specific control

`Xamarin.Forms.Button`
Element

shared

platform

```
Xamarin.Forms
  .Platform.Android
  .ButtonRenderer
```

Click Me, I Dare You!

```
Xamarin.Forms
  .Platform.iOS
  .ButtonRenderer
```

Click Me, I Dare You!

```
Xamarin.Forms
  .Platform.WinRT
  .ButtonRenderer
```

Click Me, I Dare You!

Default renderers

- Xamarin.Forms provides platform-specific renderers for every visual element – naming is generally consistent across platforms

XF Element	Button	ContentPage	ContentView	EntryCell	…
iOS	ButtonRenderer	PageRenderer	ViewRenderer	EntryCellRenderer	…
Android	ButtonRenderer	PageRenderer	ViewRenderer	EntryCellRenderer	…
Windows	ButtonRenderer	PageRenderer	ViewRenderer	EntryCellRenderer	…

A complete list of renderers for each platform is available here:
https://developer.xamarin.com/guides/xamarin-forms/custom-renderer/renderers/

Motivation

- Xamarin.Forms provides limited APIs to change control appearance and behavior – custom renderers let you access all the native properties

```
public class Button : Element
{
    public Color BackgroundColor {...}
    public Color BorderColor {...}
    public Color TextColor {...}
    ...
```

Fewer customization options than the native peers

shared platform

```
public class Button : View ...
{
    public Color CurrentHintColor {...}
    public Color CurrentTextColor {...}
    public Color HighlightColor {...}
    public Color SolidColor {...}
    public ColorStateList HintTextColors {...}
    public ColorStateList LinkTextColors {...}
    public ColorStateList TextColors {...}
    public Drawable Background {...}
    ...
}
```

Customizing a platform renderer

- A platform-specific renderer lets us access properties on the native control that are not reachable through the Xamarin.Forms API

Click Me, I Dare You!

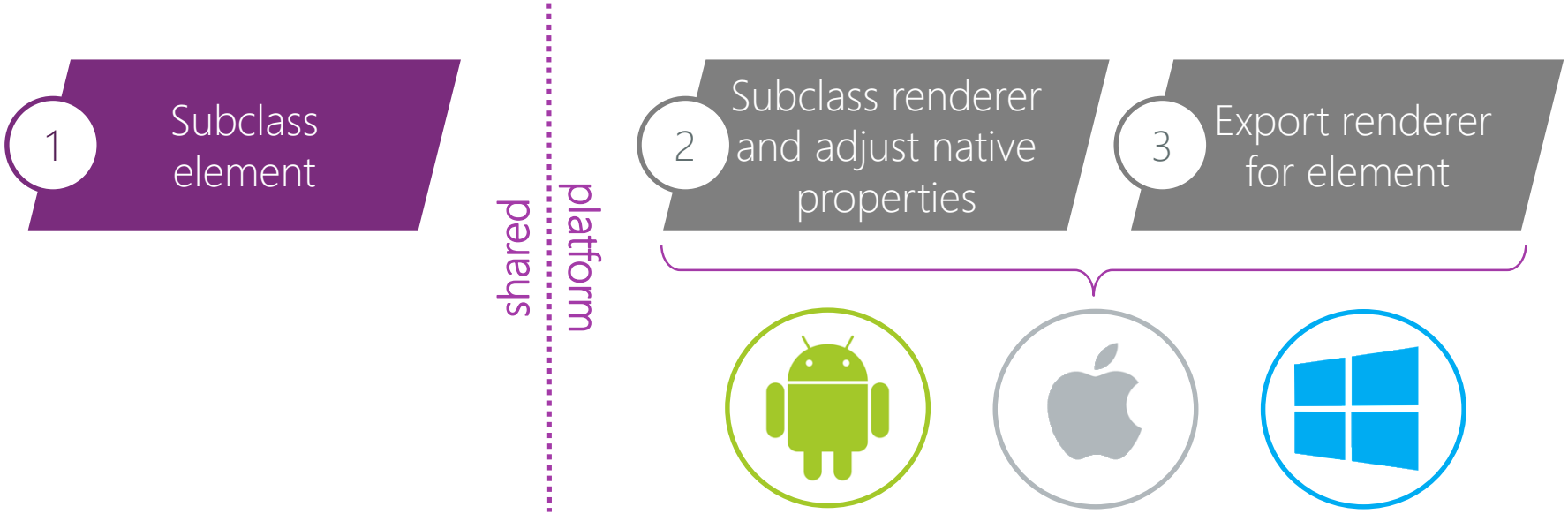
Default appearance from **`ButtonRenderer`** on iOS – cannot apply a shadow from Xamarin.Forms

Click Me, I Dare You!

Custom button renderer can use the UIKit APIs to apply a shadow directly to the native control

Steps to customize a renderer

- There are several steps required to create and apply a customized renderer



Subclass the element

- Create a subclass of the visual element to be customized

```
public class MyButton : Button
{
}
```

No properties, methods, or overrides are required

Optionally, you can add members to pass data to your custom renderer

shared

platform

Subclass the renderer

- Subclass the platform renderer for the element on each platform

shared

platform

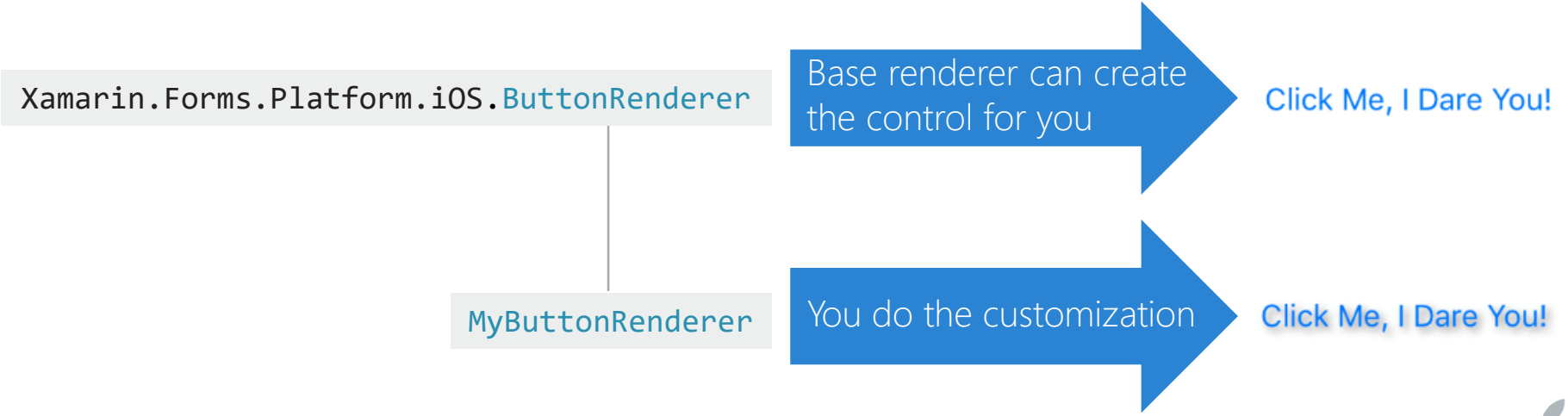
```
public class MyButtonRenderer : Xamarin.Forms.Platform.Android.ButtonRenderer
{
    ...
}
```

```
public class MyButtonRenderer : Xamarin.Forms.Platform.iOS.ButtonRenderer
{
    ...
}
```

```
public class MyButtonRenderer : Xamarin.Forms.Platform.UWP.ButtonRenderer
{
    ...
}
```

Renderer tasks

- Custom renderers have two main tasks: create the native control and then customize it using the native APIs



Renderer lifecycle

- The renderer's **`OnElementChanged`** method is called when the renderer receives the Xamarin.Forms element – this is where you do your work

shared

platform

```
public class MyButtonRenderer : ButtonRenderer
{
    protected override void OnElementChanged (...)
    {
        ...
    }
}
```

You create the native control and customize it

Your renderer overrides this method

Create the native control

- Call **`base.OnElementChanged`** and it will create the native control for you

shared

platform

```
public class MyButtonRenderer : ButtonRenderer
{
    protected override void OnElementChanged (...)
    {
        base.OnElementChanged(e);
        ...
    }
}
```

`ButtonRenderer` creates the native control

Accessing the native control

- Access to native control is provided via the platform renderer's **Control** property

shared

platform

```
public class MyButtonRenderer : ButtonRenderer
{
    protected override void OnElementChanged (...)
    {
        base.OnElementChanged(e);

        UIButton iOSButton = base.Control;
        ...
    }
}
```

Control property is strongly typed i.e. here it is a **UIButton**

ButtonRenderer assigns the native control it creates to the **Control** property

Customize the native control

- Use the native APIs on the **`Control`** property to do your customization

shared

platform

```
public class MyButtonRenderer : ButtonRenderer
{
    protected override void OnElementChanged (...)
    {   ...
        base.Control.Layer.ShadowOpacity = 1.0f;
    }
}
```

Use native APIs

Export a Renderer

- Use the assembly-level attribute to connect the Xamarin.Forms element to the platform-specific renderer

shared

platform

```
[assembly: ExportRenderer(typeof(MyButton), typeof(MyButtonRenderer))]
```

The Xamarin.Forms element type

The platform-specific renderer visualizing the element

It is possible to replace the renderer for a default element - but it's recommended to always create a derived element to allow access to the unmodified renderer

Consuming a custom renderer

- To use the custom renderer, create an instance of the element that the renderer is applied to and add it to your UI

Can use C#

```
var button = new MyButton();
myGrid.Children.Add(button);
```

Can use XAML

```
<ContentPage xmlns:local="clr-namespace:MyApp" ... />
   <local:MyButton Text="Click Me, I Dare You!" />
</ContentPage>
```

shared platform

Exercise

Create a hyperlink label renderer

Summary

1. Extend an existing renderer
2. Apply a customized renderer

Create a renderer for a custom control

Xamarin University

Tasks

1. Create a custom element
2. Create a renderer for a custom element

Motivation

- Some controls or visualizations don't match the elements provided by Xamarin.Forms

Which renderer would you derive from to produce this visualization?

What properties would you configure on this type of control?

Custom elements

- There are two ways to define custom elements in Xamarin.Forms

Derive from an existing element

Inherit properties and behavior from a known element

Derive from the `View` base class

You must create all properties and behavior

Create a Xamarin.Forms custom element

- Define a new Xamarin.Forms element by deriving from an existing type (such as **`Button`**) or directly from the **`View`** base class

```
public class MyGaugeView : View
{
    ...
}
```

shared

platform

Custom properties

- You can add properties to your custom element – you should define them as bindable properties to enable data-binding

```
public class MyGaugeView : View
{
    public static readonly BindableProperty NeedleColorProperty =
        BindableProperty.Create("NeedleColor", typeof(Color),
                                typeof(MyGaugeView), Color.Blue);

    public Color NeedleColor
    {
        get { return (Color)GetValue(NeedleColorProperty); }
        set { SetValue(NeedleColorProperty, value); }
    }
```

shared

platform

Exercise

Create a custom control

Xamarin University

Define the native controls

- You need to implement your custom control on each platform

shared

platform

```
public class MyAndroidGaugeView : Android.View.View
{
    ...
}
```

```
public class MyiOSGaugeView : UIKit.UIControl
{
    ...
}
```

```
public class MyWindowsGaugeView : Windows.UI.Xaml.Controls.Control
{
    ...
}
```

Define the renderers

- You need to subclass the base **`ViewRenderer`** class on each platform

Type of the Xamarin.Forms element

Type of the native control

shared

platform

```
public class MyGaugeRenderer : ViewRenderer<MyGaugeView, MyAndroidGaugeView>
{
    ...
}
```

```
public class MyGaugeRenderer : ViewRenderer<MyGaugeView, MyiOSGaugeView>
{
    ...
}
```

```
public class MyGaugeRenderer : ViewRenderer<MyGaugeView, MyWindowsGaugeView>
{
    ...
}
```

Assign the native control

- Your renderer must create the native control and then pass it to the **`SetNativeControl`** method

shared

platform

```
protected override void OnElementChanged(...)
{
    base.OnElementChanged(e);

    var gaugeView = new MyiOSGaugeView();

    base.SetNativeControl(gaugeView);
}
```

Create the native control

Tell Xamarin.Forms to add it to the native screen, this assigns the **`Control`** property

Multiple calls to OnElementChanged

- Xamarin.Forms may call **`OnElementChanged`** multiple times (this should be rare, but it is best practice to test for it so you only create the control once)

shared

platform

```
protected override void OnElementChanged(...)
{   ...
    if (Control == null)
    {
        var gaugeView = new MyiOSGaugeView();
        base.SetNativeControl(gaugeView);
    }
}
```

Have we already created the control?

Element access

- The base renderer makes the Xamarin.Forms element available in a property named **`Element`**

shared

platform

```
public class MyGaugeRenderer : ViewRenderer<MyGaugeView, ...>
{
    void OnGaugeTapped (...)
    {
        MyGaugeView myGauge = base.Element;
    }
}
```

It is strongly-typed, e.g. here the type is **`MyGaugeView`**

The Xamarin.Forms element associated with this renderer

Property change notifications

- Should monitor property changes on the element by overriding the **`OnElementPropertyChanged`** method

shared

platform

```
public class MyGaugeRenderer : ViewRenderer<MyGaugeView, MyiOSGaugeView>
{
  protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs e)
  {
    if (e.PropertyName == MyGaugeView.NeedleColorProperty.PropertyName)
      Control.NeedleForegroundColor = Element.NeedleColor;
  }
  ...
}
```

Called when properties change on the Xamarin.Forms element

Update the native control

Determine which property changed

Exercise

Create a renderer for a custom control

Xamarin University

Summary

1. Create a custom element
2. Create a renderer for a custom element

Send notifications between renderer and element

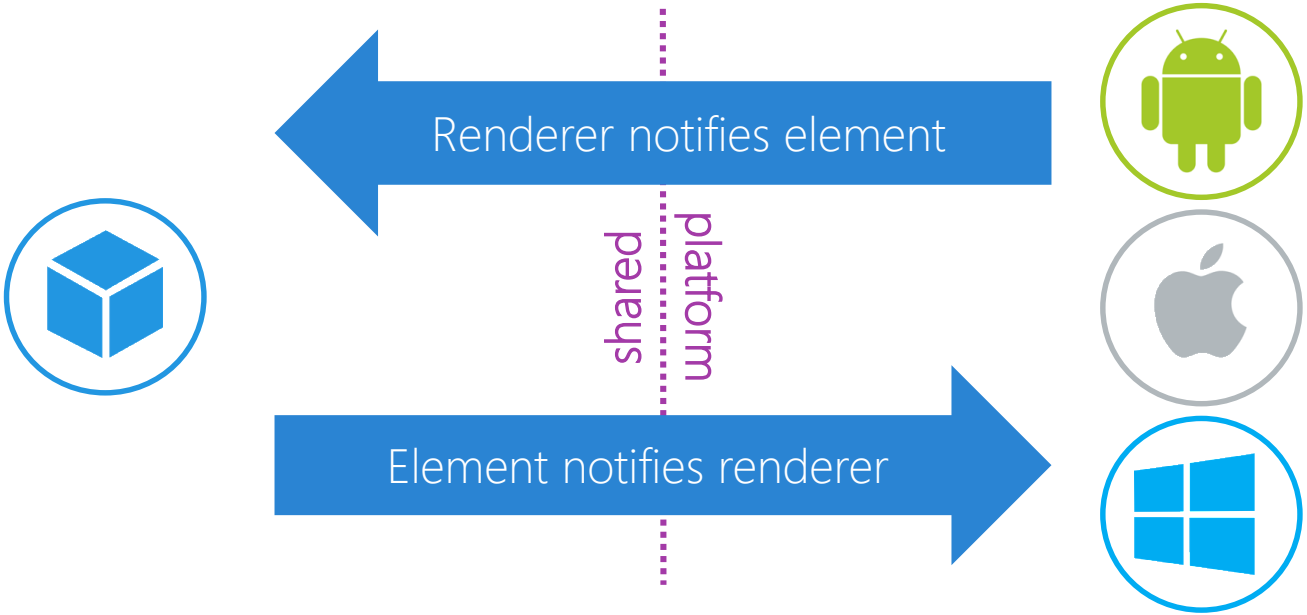
Xamarin University

Tasks

1. Send from renderer to element
2. Send from element to renderer

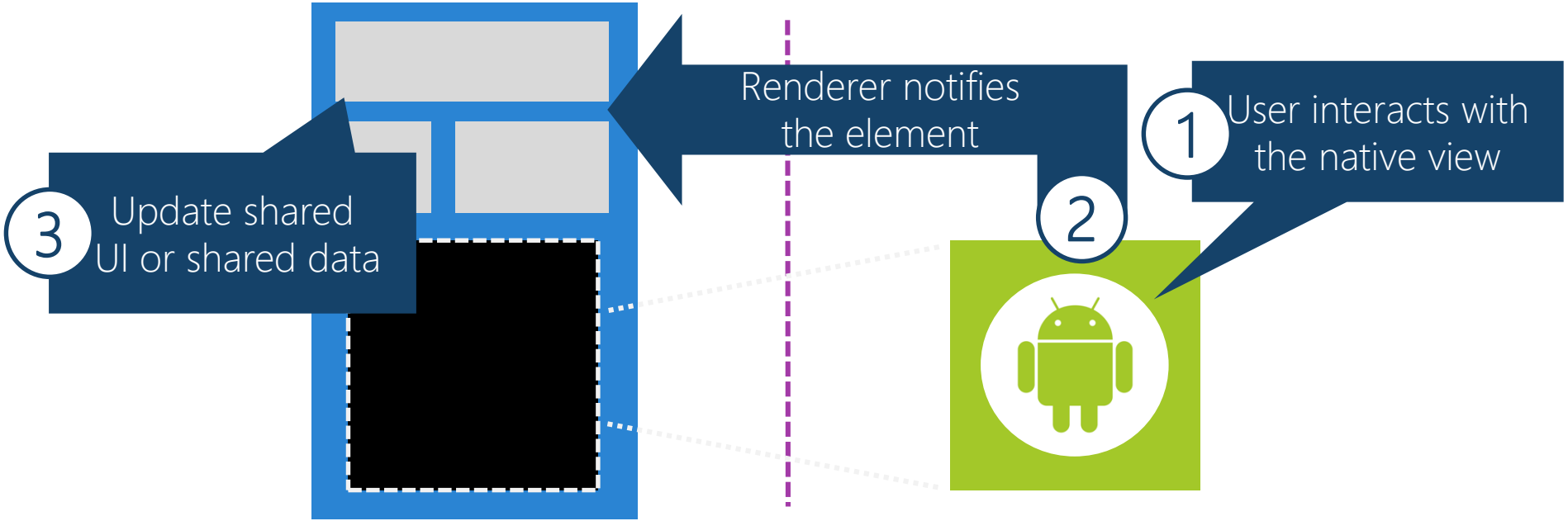
Motivation

- Custom controls often require communication between the renderer and the shared element – typically to report user actions



Renderer to element

- The renderer can notify the shared-code when a specific action takes place – e.g. allow the shared code to react to user input



Detect user interaction in the native view

- Subscribe to native events on the native control in the renderer to detect and respond to user input

shared

platform

```
public class MyGaugeRenderer : ViewRenderer<...>
{
   protected override void OnElementChanged (...)
   {
      var myGauge = new MyiOSGaugeView();
      myGauge.Tapped += OnGaugeTapped ();
   }

   void OnGaugeTapped (...)
   {
      // respond to input and notify shared code
   }
}
```

Notifying the element

- Can create public methods on your custom Xamarin.Forms element which can be called from your platform-specific renderer

```
class MyGaugeView : View
{
   public void SendReset()
   {
      //raise an event, etc.
   }
}
```

shared

platform

```
class MyGaugeRenderer : ViewRenderer<...>
{
   void OnGaugeTapped (...)
   {
      base.Element.SendReset();
   }
}
```

Notification method accessibility [concept]

- It's tricky to decide which accessibility level to use for the method

We don't want the method to be used from here

```
class MyGaugeView : View
{
    public void SendReset()
    {
        //raise an event, etc.
    }
}
```

The method we call in shared code must be reachable from here

Notification method accessibility [example]

- A public method can lead to confusing code – the notification method is not meant to be called from the shared code

public isn't ideal here…

```
class MyGaugeView : View
{
    public void SendReset() {...}
}
```

…it allows the method to be called from here when it is intended for use only by the renderer

```
public class MainPage
{
    public MainPage()
    {   ...
        var gauge = new MyGaugeView();
        gauge.SendReset(); //renderer only!
    }
}
```

shared

platform

Controller interface

- An interface can be used to define methods that should be reachable by the renderer but not easily discoverable from the shared code

```
interface IGaugeController
{
    void SendReset();
}
```

shared

platform

Obfuscate the method

- The element uses *explicit interface implementation* when it codes the notification method(s)

The **public** keyword is not allowed

```
class MyGaugeView : View, IGaugeController
{
    void IGaugeController.SendReset()
    {
        //raise an event, etc.
    }
}
```

shared

platform

Notify using the obfuscated method

- Call the element's explicitly defined method by casting to the interface type

```
class MyGaugeView :
  View, IGaugeController
{
  void IGaugeController.SendReset()
  {
    ...
  }
}
```

Method is explicitly implemented…

shared

platform

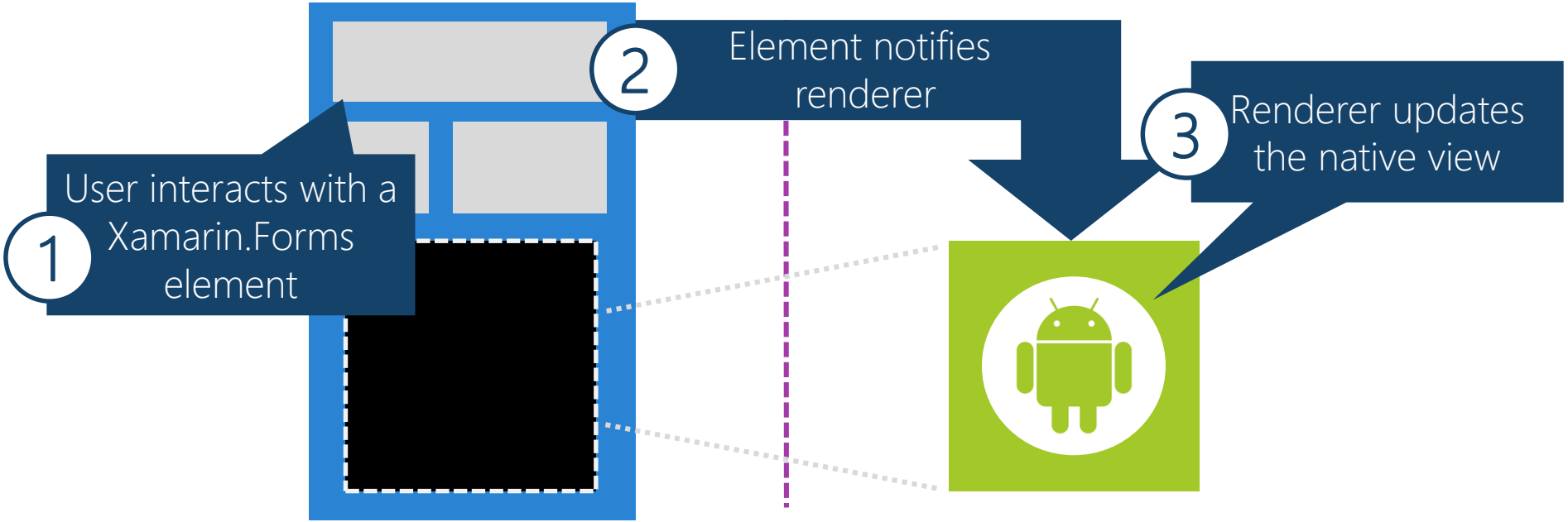
```
class MyGaugeRenderer : ViewRenderer<...>
{
  void OnGaugeTapped (...)
  {
    var igc = (IGaugeController)Element;

    igc?.SendReset();
  }
}
```

…must be called using an interface reference

Element to renderer

- The shared code can notify the renderer to update properties or set state



Communication options

- To notify the renderer, you can use bindable properties that do all the work for you or manually send a message

Bindable property

Detects change
Notifies renderer

Messaging service

More work, but useful for cases where properties are not ideal (e.g. reset, passing multiple parameters, etc.)

Bindable properties are the preferred solution unless a method call is required.

Messaging Service

- Use a messaging service to send notifications from the shared code to the platform specific code – in Xamarin.Forms we can use the built-in Messaging Center to create a loosely-coupled design

```
class MyGaugeView : View
{
    public void SetRaceMode()
    {
        MessagingCenter.Send(...);
    }
}
```

shared

platform

```
class MyGaugeRenderer ...
{
    ... void OnElementChanged(...)
    {
      MessagingCenter.Subscribe(...);
    }
    ...
}
```

Verifying the sender

- Verify the message source by comparing the message sender to the **`Element`** in the renderer

shared

platform

```
protected override void OnElementChanged(...)
{
   MessagingCenter.Subscribe<MyGaugeView>(this, "RaceMode",
                                          OnSetRaceMode);
}

void OnSetRaceMode(MyGaugeView sender)
{
   if(sender == Element)
      gaugeView.SetRaceMode();
}
```

Ensure the message is sent from our instance of the element

Cleanup

- Xamarin.Forms renderers use the Dispose pattern; override Dispose and perform cleanup, unsubscribe from events and messages

```
class MyGaugeRenderer ...
{
   protected override void Dispose(bool disposing)
   {
      MessagingCenter.Unsubscribe<MyGaugeView>(this, "RaceMode");

      Control.Tapped -= OnGaugeTapped ();
   }
}
```

Unsubscribe to messages

Unwire any event handler methods on the native control

Exercise

Interact with the renderer

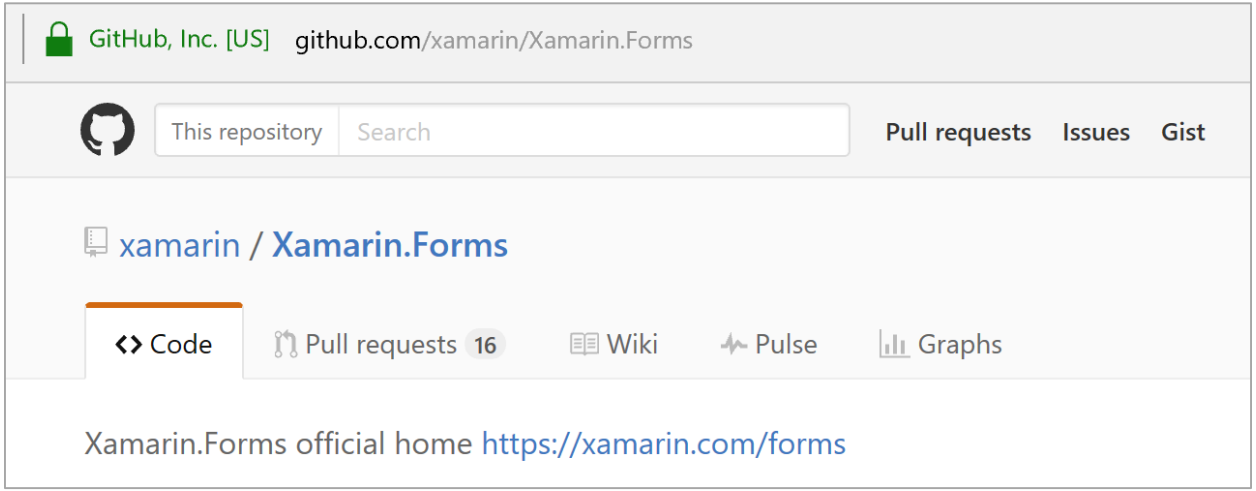
Xamarin University

Summary

1. Send from renderer to element
2. Send from element to renderer

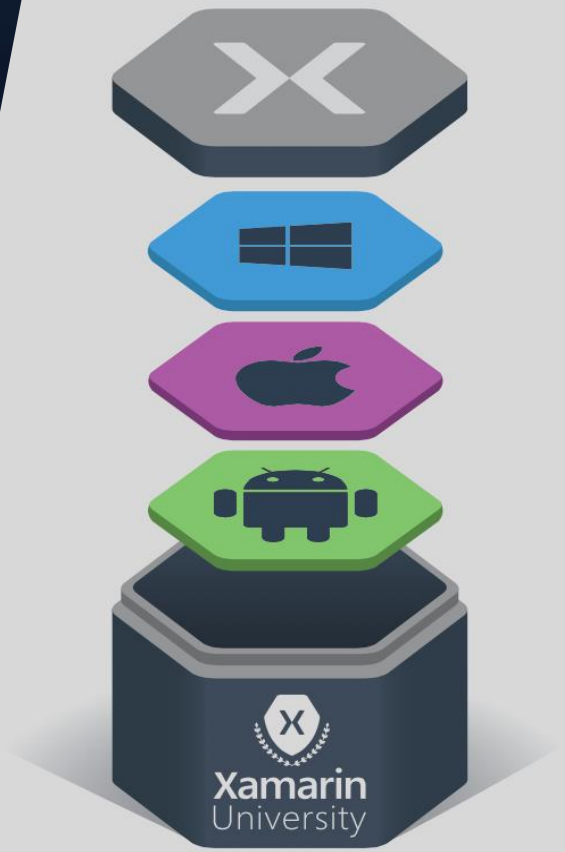
What's next?

- Take a look at the Xamarin.Forms source code to gain a deeper understanding on the existing architecture and patterns



Thank You!

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



Microsoft