

XAM330

Xamarin.Forms Effects

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. Customize control appearance
2. Apply effects to controls
3. Create an effect
4. Add configurable properties





Customize control appearance

Tasks

1. Change element properties
2. Use platform themes to update native control appearance



Strive for app elegance

- ❖ Our goal, as mobile developers, should be to build **useful**, **elegant** and **beautiful** applications
- ❖ Our applications should **look and feel natural** on each platform, taking advantage of the platform's unique style and patterns



Recall: Xamarin.Forms elements

- ❖ Xamarin.Forms allows you to define your UI using a set of **elements** that are common across all platforms

Button is
available
everywhere



```
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; }
    ...
}
```

Xamarin.Forms elements are models

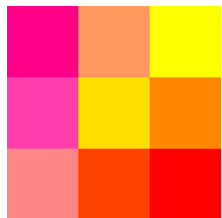
- ❖ Elements provide a *representation* of the UI we want to create and display

Properties let
you 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; }
    ...
}
```


Customizing elements

- ❖ Changing the properties of Xamarin.Forms elements allows for **limited customization** - which may or may not be sufficient for your needs



Can change
most colors



Can adjust
position +
width/height




Can add background
images into views



Can control
fonts

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!

What is a platform renderer?

- ❖ The *platform renderer* is the code that translates the Xamarin.Forms object 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

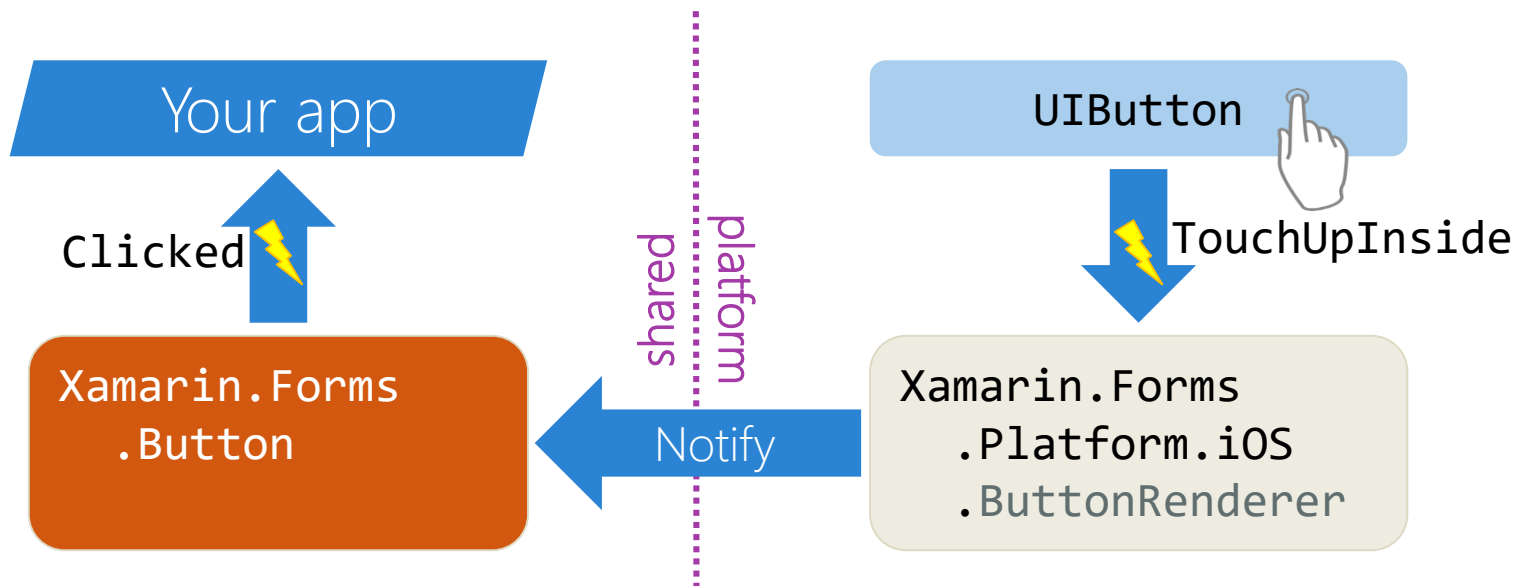
Click Me, I Dare You!

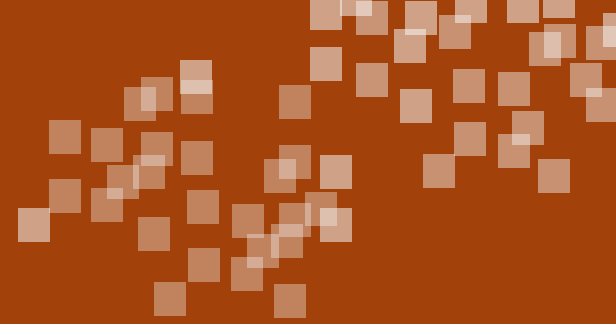
Xamarin.Forms
.Platform.WinRT
.ButtonRenderer

Click Me, I Dare You!

From Visual to Element

- ❖ The renderer is responsible for **watching** the native control notifications and **forwarding** them to the Xamarin.Forms **Element**





Demonstration

Explore the rendering architecture of Xamarin.Forms



Going beyond Forms

- ❖ There are several ways to influence the native control rendering

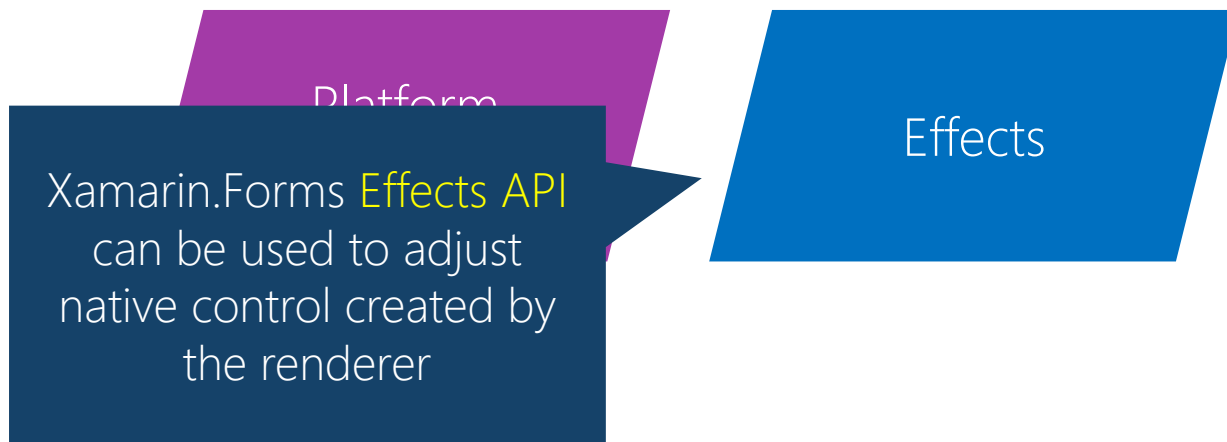
A diagram consisting of a purple parallelogram on the left and a dark blue rectangle on the right. A white arrow points from the right side of the purple parallelogram to the left side of the dark blue rectangle.

Platform
Themes

Can use the platform's
style/theme/appearance APIs to
adjust most visual properties

Going beyond Forms

- ❖ There are several ways to influence the native control rendering



Going beyond Forms

- ❖ There are several ways to influence the native control rendering

A purple parallelogram with the text 'Platform Themes' inside.

Platform
Themes

A blue parallelogram with the text 'Effects' inside.

Effects

A green parallelogram with the text 'Embedded Native controls' inside.

Embedded
Native controls

A dark blue rectangular callout box with a white arrow pointing to the green parallelogram. It contains the text 'Can embed native controls directly into a Xamarin.Forms UI'.

Can embed native controls
directly into a Xamarin.Forms UI

Going beyond Forms

- ❖ There are several ways to influence the native control rendering

A diagram illustrating three ways to influence native control rendering. It consists of three parallelogram-shaped boxes: a purple one at the top left, a blue one at the top right, and a light blue one at the bottom right. A dark blue speech bubble points from the bottom left towards the light blue box. The entire diagram is set against a white background with a colorful horizontal bar at the bottom.

Platform
Themes

Effects

Finally, you can *replace* the
renderer and generate a
completely different
visualization!

Custom
Renderers

Going beyond Forms

- ❖ There are several ways to influence the native control rendering

A purple parallelogram shape, tilted to the right, containing the text 'Platform Themes' in white.

Platform
Themes

A blue parallelogram shape, tilted to the right, containing the text 'Effects' in white.

Effects

Platform Themes

- ❖ Each platform has an API you can use to control the native visual appearance of your app



Style +
ControlTemplate

iOS

UIAppearance



android:theme

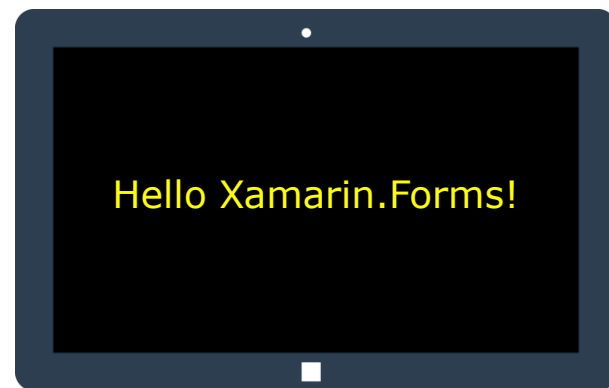


Styles and control templates [Windows]

- ❖ Each Windows XAML control has a default style and control template – these can be modified to customize appearance and behavior

```
<Application.Resources>  
  <Style TargetType="TextBlock">  
    <Setter Property="Foreground" Value="Yellow" />  
    <Setter Property="FontFamily" Value="Verdana" />  
    <Setter Property="FontSize" Value="96" />  
  </Style>  
</Application.Resources/>
```

Native Windows **Styles** will
affect controls created by the
Xamarin.Forms renderer

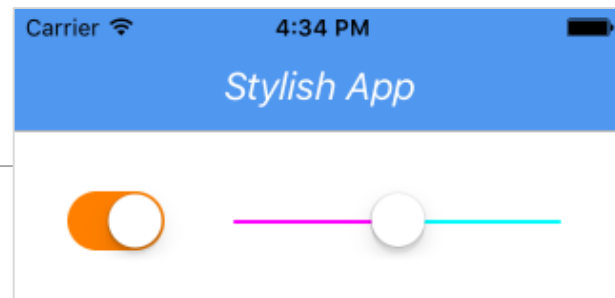


Appearance API [iOS]

- ❖ The iOS Appearance API lets you define visual settings at a class level that apply to all instances of that type

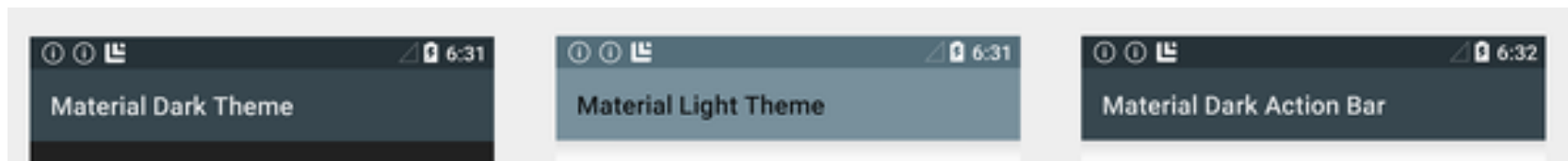
```
public override bool FinishedLaunching(...)
{
    UISwitch.Appearance.OnTintColor = UIColor.Orange;
    UISlider.Appearance.MinimumTrackTintColor = UIColor.Magenta;
    UISlider.Appearance.MaximumTrackTintColor = UIColor.Cyan;

    UINavigationController.Appearance.BarTintColor = UIColor.FromRGB(51, 134, 238);
    UINavigationController.Appearance.SetTitleTextAttributes(new UITextAttributes()
    { TextColor = UIColor.White, Font = UIFont.ItalicSystemFontOfSize(20)});
}
```



Themes [Android]

- ❖ Android Themes determine the look and feel of views and activities; there are built in themes and you can create custom themes



```
[Activity(Label = "DroidThemes",
    Theme = "@android:style/Theme.Material.Light",
    MainLauncher = true, Icon = "@mipmap/icon")]
public class MainActivity : Activity
{
    ...
}
```


Exercise

Use platform-specific themes



Xamarin
University

Summary

1. Change element properties
2. Use platform themes to update native control appearance





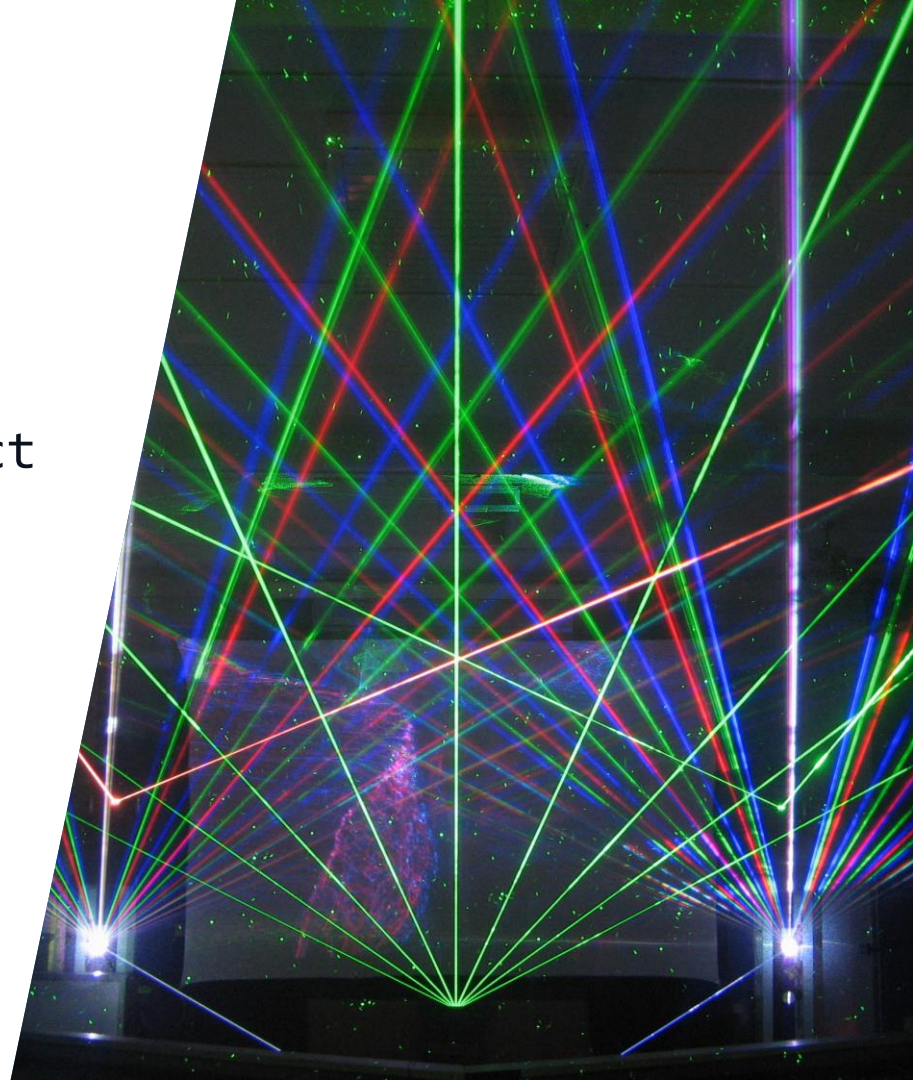
Apply effects to controls



Xamarin
University

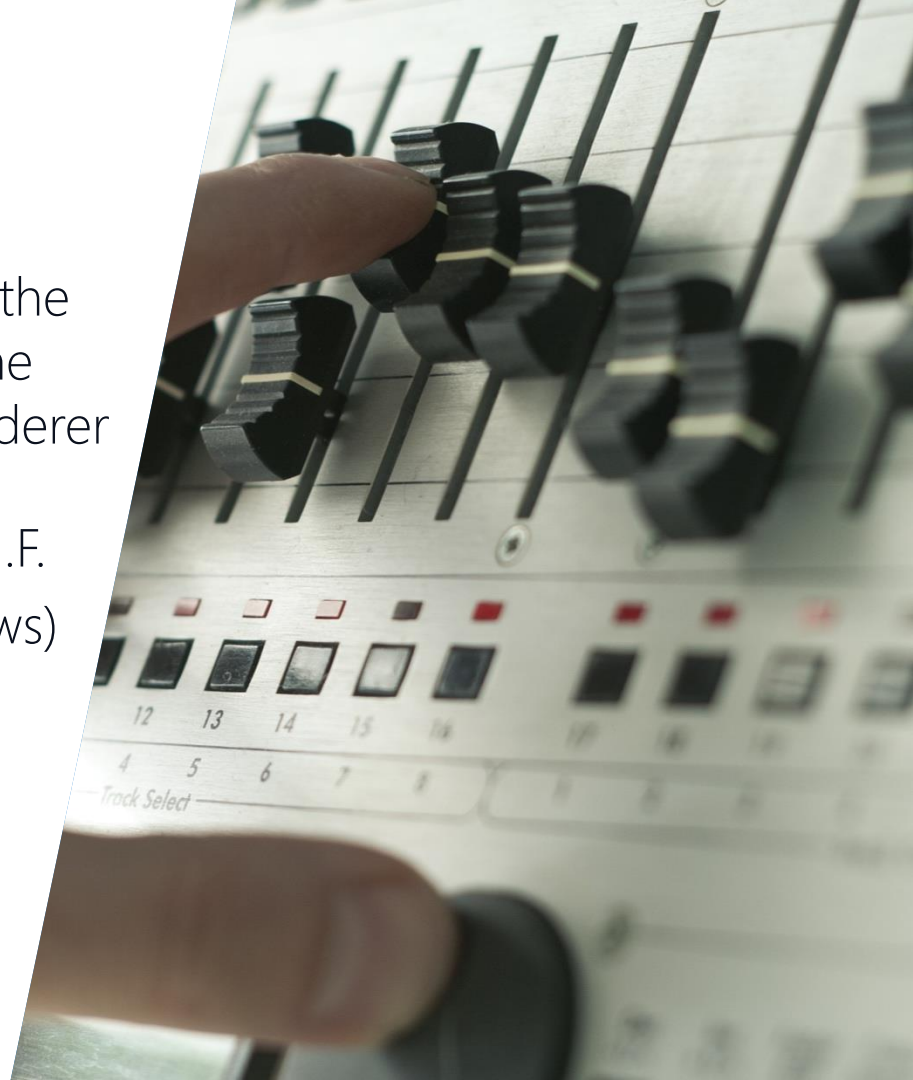
Tasks

1. Resolve an effect
2. Apply an effect programmatically
3. Wrap an effect with **RoutingEffect**
4. Apply an effect in XAML



Xamarin.Forms Effects

- ❖ The **Effects API** lets your code *tweak* the visual appearance and behavior of the native controls generated by the renderer
 - Change properties not exposed by X.F.
 - Access platform features (e.g. shadows)
 - Handle native control notifications
 - Add or remove visual children



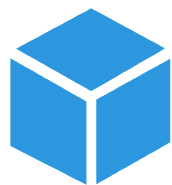
What is an effect?

- ❖ An *effect* is a platform-specific class that uses the native APIs to change the appearance and behavior of the native control that underlies a Xamarin.Forms **Element**



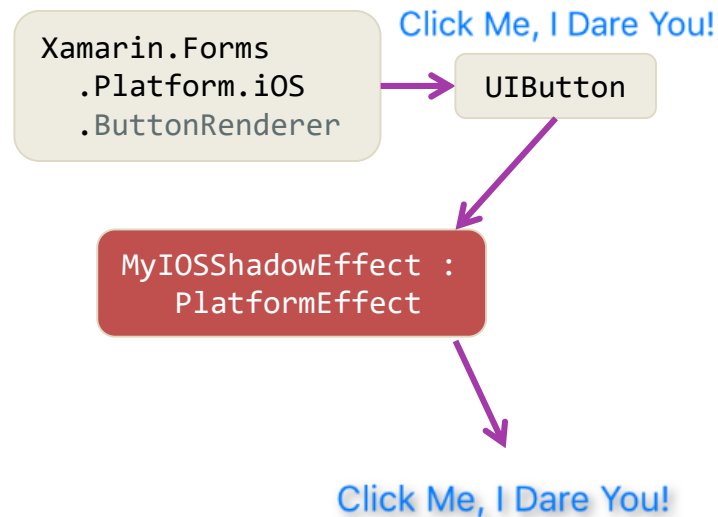
Xamarin.Forms Effects API

- ❖ The Effects API allows you to interact with and change properties on the controls created by the native renderers



Element
(Button)

shared
platform



One effect per platform

- ❖ The author of an effect implements one class for each platform they choose to support

The implementation
has access to the
native APIs

```
class MyAndroidShadowEffect : ...  
{ ...  
}
```

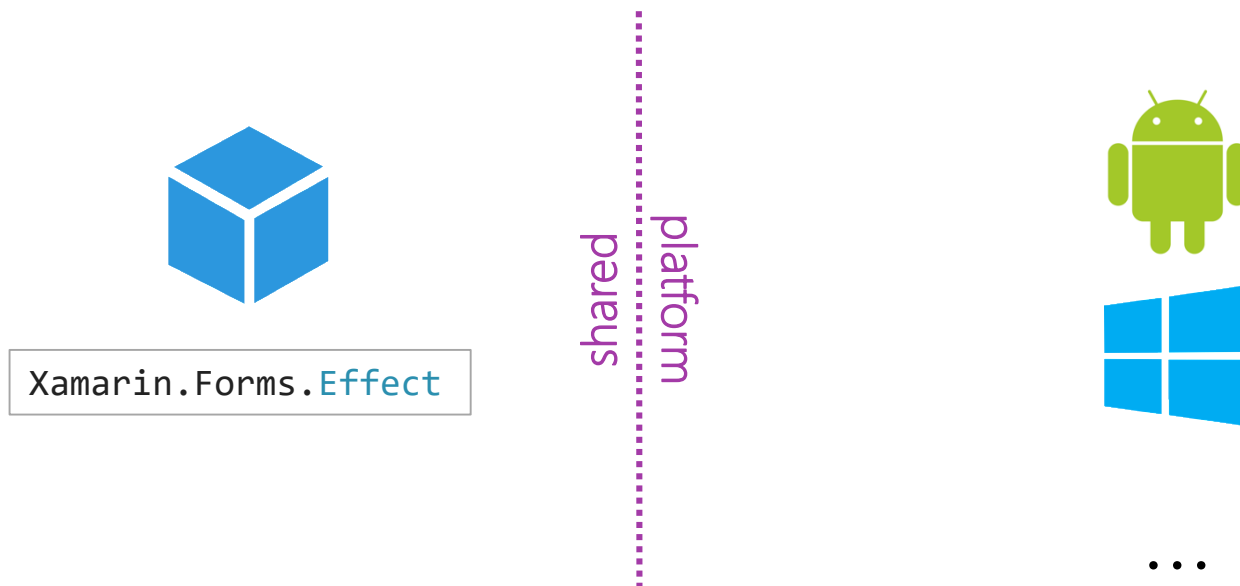
```
class MyIOSShadowEffect : ...  
{ ...  
}
```

```
class MyUWPShadowEffect : ...  
{ ...  
}
```

shared
platform


The Effect class

- ❖ In the shared code, effects are represented by the abstract **Effect** class which is the shared representation of the native platform-specific effect



Effects collection

- ❖ Every Xamarin.Forms **Element** has a collection to hold applied effects called **Effects**

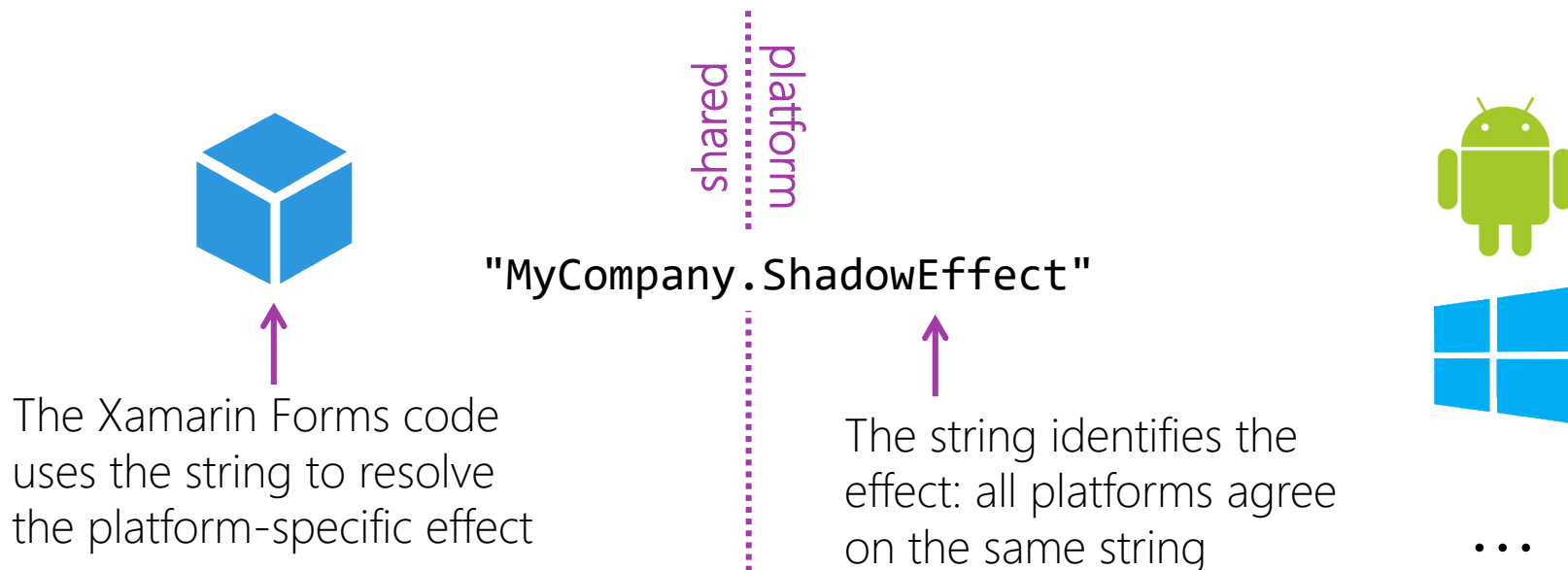


```
public class Element : ...  
{  
    public IList<Effect> Effects { get; }
```

Used to add and remove
effects to an element

The effect identifier

- ❖ The effect author chooses a string to identify the effect



Resolve an effect

- ❖ Effects are resolved at runtime by the identifier using the static **Effect.Resolve** method

```
Button dareButton = new Button();  
...  
Effect eff = Effect.Resolve("MyCompany.ShadowEffect");  
dareButton.Effects.Add(eff); // ok
```

↑
Add the effect
to the collection

↑
Resolve by name

shared
platform

Effect instances

- ❖ Each element must have a **unique instance** of the effect

```
Effect eff = Effect.Resolve("MyCompany.ShadowEffect"));  
  
dareButton.Effects.Add(eff); // ok  
  
anotherButton.Effects.Add(eff); ❌ // runtime error
```

Removing Effects

- ❖ **Effects** can be removed dynamically; the comparison is done using reference equality

```
var dareButton = new Button();  
...  
dareButton.Effects.Remove(eff);  
dareButton.Effects.Clear();
```

Remove
the effect
instance

platform
shared

Individual Exercise

Add and use an existing Effect



Xamarin
University

The RoutingEffect class

- ❖ **RoutingEffect** is a class that wraps a call to the **Effect.Resolve** static method – which makes it easier to apply effects in code and XAML

```
public class RoutingEffect : Effect
{
    internal readonly Effect Inner;
    ...
    protected RoutingEffect(string effectId)
    {
        Inner = Resolve(effectId);
    }
}
```



Implementing a RoutingEffect

- ❖ Create a **RoutingEffect** derived class for each effect you want to resolve

```
public class ShadowEffect : RoutingEffect
{
    public ShadowEffect()
        : base("MyCompany.ShadowEffect")
    {
    }
}
```

Supply effect identifier to the
RoutingEffect constructor

platform
shared

Using RoutingEffect in code

- ❖ Instantiate your **RoutingEffect** derived type and add it to an element's **Effects** collection – this is a type safe way to add effects

```
public class ShadowEffect : RoutingEffect
{
    public ShadowEffect()
        : base("MyCompany.ShadowEffect")
    {
    }
}
```

```
dareButton.Effects.Add(new ShadowEffect());
```

Using RoutingEffect in XAML

- ❖ **RoutingEffect** makes it easy to apply an effect in XAML

```
xmlns:ef="clr-namespace:MyEffects"  
...  
<Button ...>  
    <Button.Effects>  
        <ef:MyShadowEffect />  
    </Button.Effects>  
</Button>
```

Create an instance and add it
to the **Effects** collection

Individual Exercise

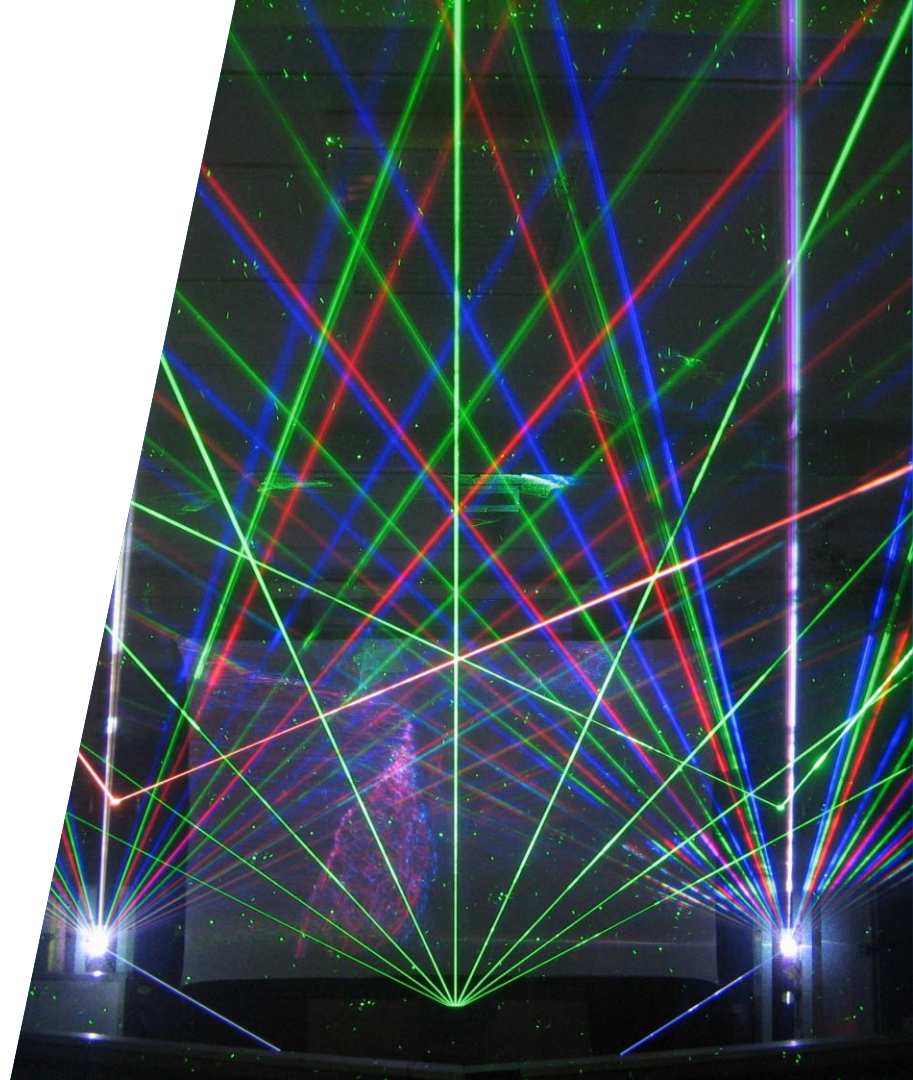
Create a RoutingEffect



Xamarin
University

Summary

1. Resolve an effect
2. Apply an effect programmatically
3. Wrap an effect with RoutingEffect
4. Apply an effect in XAML



Create an Effect

Tasks

1. Create an effect
2. Update platform specific UI
3. Respond to UI changes and notifications
4. Export an effect



Creating a new Effect

- ❖ Effects are platform-specific classes located in a platform-specific project (iOS, Android, or Windows)

shared
platform

```

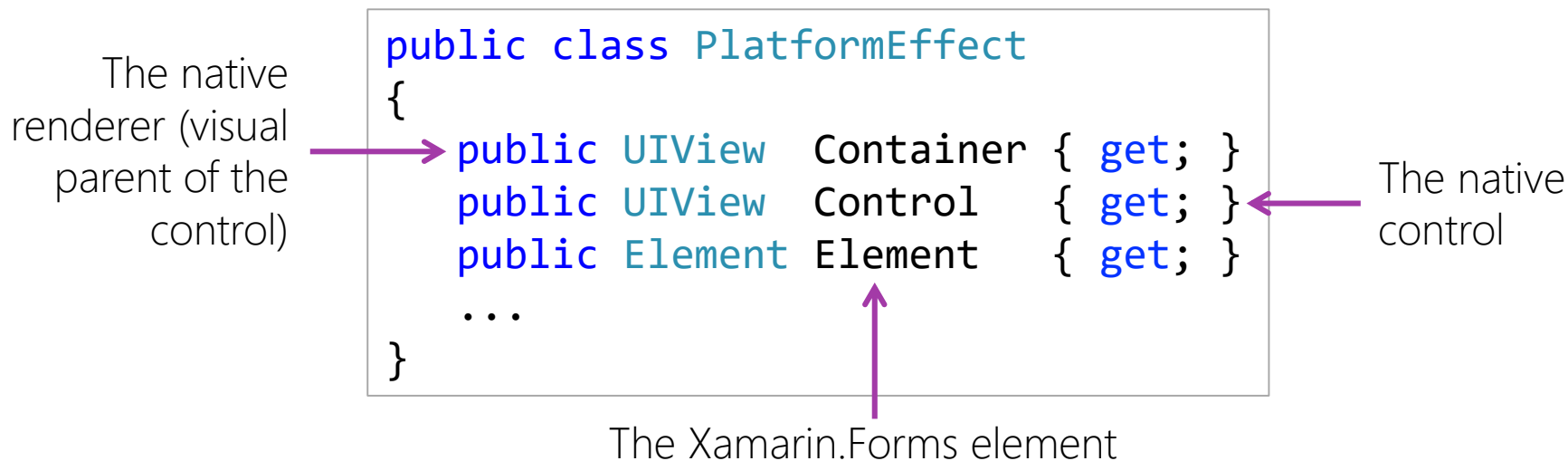
internal class MyIOSShadowEffect : PlatformEffect
{
    ...
}
            
```

Effect can be internal to the platform project

Derive from **PlatformEffect** which provides access to the native control

PlatformEffect properties

- ❖ **PlatformEffect** base class provides properties to interact with and manipulate both sides of the visual: the element and the native control



PlatformEffect property types

- ❖ Each platform has platform-specific types for the **Container** and **Control** properties

```
public class PlatformEffect : ...  
{  
    ...  
    public UIView Container { get; }  
    public UIView Control { get; }  
}
```

```
public class PlatformEffect : ...  
{  
    ...  
    public ViewGroup Container { get; }  
    public View Control { get; }  
}
```

```
public class PlatformEffect : ...  
{  
    ...  
    public FrameworkElement Container { get; }  
    public FrameworkElement Control { get; }  
}
```

PlatformEffect methods

- ❖ **PlatformEffect** base class has lifecycle methods that you override to do your work

```
public class MyPlatformEffect : PlatformEffect
{
    ...
    protected override void OnAttached() {...}

    protected override void OnDetached() {...}

    protected override void OnElementPropertyChanged(...) {...}
}
```

shared
platform

Adjusting visual properties

- ❖ **OnAttached** is called when the effect is added to a control; should adjust visual properties, update visual tree, wire up native events, etc.

```
class MyAndroidShadowEffect : PlatformEffect
{
    protected override void OnAttached()
    {
        ...
        Control.SetShadowLayer (5, 2, 2, Color.Black);
    }
}
```

The control is the
Android View

Use the native
android APIs

shared
platform

Restricting Effects to specific types

- ❖ Effects are intended to be used with any element type, but you can restrict to specific elements deliberately when necessary

```
class MyButtonEffect : PlatformEffect
{
    protected override void OnAttached()
    {
        Button button = Element as Button;
        if (button == null)
            return;
        ...
    }
}
```

shared
platform

Detaching from the native control

- ❖ **OnDetached** indicates the effect is being removed from the control, should reverse any visual changes, remove event handlers, etc.

```
class ShadowEffect : PlatformEffect
{
    ...
    protected override void OnDetached()
    {
        // Remove drop shadow
        ...
        Control.ClearShadowLayer();
    }
}
```

shared
platform

Monitoring runtime changes

- ❖ Effects can also monitor changes to the Xamarin.Forms **Element** by overriding the **OnElementPropertyChanged** method

```
class DisabledOpacityEffect : PlatformEffect
{
    protected override void OnElementPropertyChanged(PropertyChangedEventArgs e)
    {
        base.OnElementPropertyChanged(e);
        if (e.PropertyName == VisualElement.IsEnabledProperty.PropertyName) {
            if (((VisualElement)Element).IsEnabled)
                Control.Layer.Opacity = 1;
            else
                Control.Layer.Opacity = 0.5f;
        }
    }
}
```

Can cast the **Element** to more specific classes to check current properties

shared
platform

Setting an effect identifier

- ❖ The effect identifier is a combination of two values (group name and effect name) which are set using two assembly-level attributes applied in the platform assembly

```
[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(MyIOSShadowEffect), "ShadowEffect")]
```

platform
shared

```
Effect.Resolve("MyCompany.ShadowEffect")
```

Export an effect

- ❖ The **ExportEffect** attribute ties the effect name to the platform-specific class that implements the effect

```
[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(MyIOSShadowEffect), "ShadowEffect")]
```

```
class MyIOSShadowEffect : PlatformEffect
{
    ...
}
```

shared
platform

Individual Exercise

Create a PlatformEffect



Xamarin
University

Summary

1. Create an effect
2. Update platform specific UI
3. Respond to UI changes and notifications
4. Export an effect





Add configurable properties



Xamarin
University

Tasks

1. Add attached properties to an effect
2. Use and update attached properties



Creating flexible effects

- ❖ You may want to provide additional properties to customize your visual behavior

Drop Shadow Text



Shadow properties could include:
size, color, angle, transparency, etc.

How to add properties?

- ❖ You add properties to your **RoutingEffect** derived class to pass data to the platform effects – there are two ways to do this

A purple parallelogram shape containing the text 'Regular C# Properties' in white.

Regular C#
Properties

Easy to code but
does not support
dynamic update

A blue parallelogram shape containing the text 'Attached Properties' in white.

Attached
Properties

More work to code,
automatically supports
dynamic update

Defining regular properties

- ❖ Public properties can be added to **RoutingEffect** derived classes and consumed from the platform-specific effect

```
public class ShadowEffect : RoutingEffect
{
    public Color ShadowColor { get; set; }
}
```

```
<Button ...>
  <Button.Effects>
    <ef:ShadowEffect ShadowColor="Blue" />
  </Button.Effects>
</Button>
```

shared
platform

Consuming regular properties

- ❖ **PlatformEffect** derived classes can consume properties defined on the **RoutingEffect**

```
protected override void OnAttached ()
{
    var effect = (ShadowEffect)Element.Effects.
        FirstOrDefault (e => e is ShadowEffect);

    if (effect != null)
        Control.Layer.ShadowColor = effect.Color.ToCGColor ();
}
```

Access the
RoutingEffect
from the **Effects**
collection on the
Element

shared
platform

These properties will not respond to runtime property changes

Adding attached properties

- ❖ Can use *attached properties* on the **RoutingEffect** definition to provide custom updatable data to the platform effect

```
public class ShadowEffect : RoutingEffect
{
    ...
    public static readonly BindableProperty ColorProperty
        = BindableProperty.CreateAttached("Color",
            typeof(Color), typeof(ShadowEffect), Color.Black);

    public static Color GetColor(View view) {
        return (Color)view.GetValue(ColorProperty);
    }
    public static void SetColor(View view, Color color) {
        view.SetValue(ColorProperty, color);
    }

    public ShadowEffect() : base ("MyCompany.ShadowEffect") { }
}
```

What are attached properties?

- ❖ Attached properties are global properties that are settable on any object – allows external properties to be "added" to an object



Creating attached properties

❖ Creating attached properties is done in 3 steps:

A green parallelogram with a white border, tilted slightly to the right.

BindableProperty

A blue parallelogram with a white border, tilted slightly to the right.

Getter

A purple parallelogram with a white border, tilted slightly to the right.

Setter

Creating an attached property

- ❖ Attached properties are identified through a static BindableProperty

```
public static readonly BindableProperty ColorProperty
    = BindableProperty.CreateAttached(
        propertyName: "Color",
        returnType: typeof(Color),
        declaringType: typeof(ShadowEffect),
        defaultValue: Color.Black);
```

Use the factory **CreateAttached** method and specify the name, declaring type and return type

Read and write attached properties

- ❖ Define static Get and Set methods to read and write the attached property

```
public static void SetColor(View view, Color color)
{
    view.SetValue(ColorProperty, color);
}

public static Color GetColor(View view)
{
    return (Color)view.GetValue(ColorProperty);
}
```


Setting property values

- ❖ Set values onto the element using the static Set method – either in code or XAML

```
BoxView box = new BoxView() { ... }  
  
ShadowEffect.SetColor(box, Color.Blue);
```

```
<BoxView local:ShadowEffect.Color="Blue" ...>
```

Responding to property changes

- ❖ Platform effect can watch for attached property changes in the **OnElementPropertyChanged** method and update control properties

```
protected override void OnElementPropertyChanged(PropertyChangedEventArgs e)
{
    base.OnElementPropertyChanged(e);

    if (e.PropertyName == MyApp.ShadowEffect.ColorProperty.PropertyName)
    {
        Container.Layer.ShadowColor =
            MyApp.ShadowEffect.GetColor((View)Element).ToCGColor();
    }
}
```

shared
platform



Individual Exercise

Add an attached property to an Effect



Xamarin
University

Summary

1. Add attached properties to an effect
2. Use and update attached properties



Thank You!

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