

XAM140

# Resources and Styles

Download class materials from  
[university.xamarin.com](https://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. Avoid duplicate XAML with Resources
2. Create consistent UI with Styles
3. Make your Resources and Styles available across your entire app
4. Apply the user's Accessibility choices with built-in Styles



# Avoid duplicate XAML with Resources



# Tasks

1. Use page-level Resources
2. Dynamically update Resources



# Motivation

- ❖ Duplicate XAML values are error prone and difficult to maintain

```
<StackLayout BackgroundColor="#FFFFFF">
  <Label      TextColor="#00FF00" FontSize="16.5" ... />
  <Entry      BackgroundColor="#FFFFFF" ... />
  <BoxView    BackgroundColor="#00FF00" ... />
  <Button     BackgroundColor="#00FF00" FontSize="16.5" ... />
</StackLayout>
```

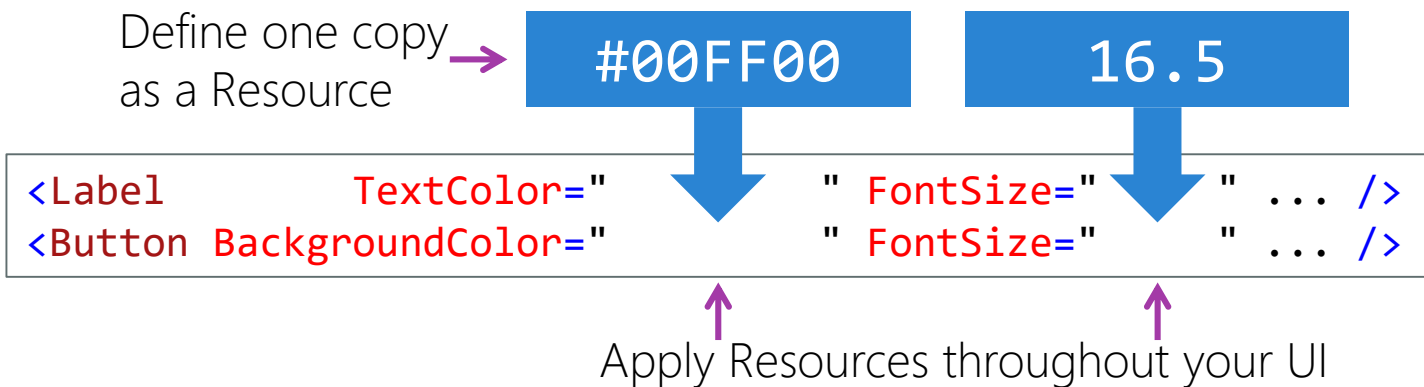
Common to use the same colors and sizes across the UI

# Group Exercise

Examine an app containing repeated code

# What is a Resource?

- ❖ A *Resource* is an object that can be used in multiple places in your UI





# What is a ResourceDictionary?

- ❖ **ResourceDictionary** is a key/value dictionary that is customized for use with UI Resources

Mostly has  
standard  
dictionary  
operations



```
public sealed class ResourceDictionary : ...  
{ ...  
    public object this[string index] { get; set; }  
  
    public void Add(string key, object value);  
    public void Add(Style implicitStyle);  
}
```



Some added UI-specific functionality

# Page-level Resources

- ❖ Every page can have a resource dictionary, must be set in code or XAML

You must create  
the dictionary

```
<ContentPage ...>

    <ContentPage.Resources>
        <ResourceDictionary>
            ...
        </ResourceDictionary>
    </ContentPage.Resources>

</ContentPage>
```

Assign the dictionary  
you create to the page's  
**Resources** property

# Creating Resources

- ❖ Resources created in XAML must use the XAML-language keyword **x:Key** to set the key

Create inside  
the page's  
dictionary

```
<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary>
      <Thickness x:Key="myKey">10,20,40,80</Thickness>
    </ResourceDictionary>
  </ContentPage.Resources>
</ContentPage>
```

Value

Key

Choose Resource names based on use, not value; e.g. use **bgColor**, not **redColor**.

# Using static Resources

- ❖ The **StaticResource** markup extension retrieves a resource, the value is applied once when the target object is created

Define

```
<ContentPage ...>

  <ContentPage.Resources>
    <ResourceDictionary>
      <Thickness x:Key="myKey">10,20,40,80</Thickness>
    </ResourceDictionary>
  </ContentPage.Resources>

```

Use

```
<StackLayout Padding="{StaticResource myKey}">
  ...
</StackLayout>

</ContentPage>

```

# XAML intrinsic types

- ❖ The XAML spec defines many types you can use for XAML Resources

**String** and **Double** are useful since many UI properties use those types

```
<ResourceDictionary>
  <x:String x:Key="...">Hello</x:String>
  <x:Char x:Key="...">X</x:Char>
  <x:Single x:Key="...">31.4</x:Single>
  <x:Double x:Key="...">27.1</x:Double>
  <x:Byte x:Key="...">8</x:Byte>
  <x:Int16 x:Key="...">16</x:Int16>
  <x:Int32 x:Key="...">32</x:Int32>
  <x:Int64 x:Key="...">64</x:Int64>
  <x:Decimal x:Key="...">12345</x:Decimal>
  <x:TimeSpan x:Key="...">1.23:5959</x:TimeSpan>
  <x:Boolean x:Key="...">True</x:Boolean>
</ResourceDictionary>
```

# Platform dependencies

- ❖ Can use **OnPlatform** objects in your resource dictionaries to handle platform-specific values

```
<ResourceDictionary>
  <OnPlatform x:Key="textColor"
    x:TypeArguments="Color"
    iOS="Silver"
    Android="Green"
    WinPhone="Blue" />
</ResourceDictionary>
```

```
<Label TextColor="{StaticResource textColor}" ... />
```



# Platform dependencies

- ❖ Can use **OnPlatform** objects in your resource dictionaries to handle platform-specific values

```
<ResourceDictionary>
  <OnPlatform x:Key="textColor"
    x:TypeArguments="Color">
    <OnPlatform="iOS" Value="Silver"/>
    <OnPlatform="Android" Value="Green"/>
    <OnPlatform="Windows" Value="Blue"/>
  </OnPlatform />
</ResourceDictionary>
```

```
<Label TextColor="{StaticResource textColor}" ... />
```

# Group Exercise

Use page-level Resources

# Motivation [delayed availability]

- ❖ You might download resource values after startup; however, resources applied with **StaticResource** will fail if the key is not in the dictionary

```
<ContentPage ...>

  <ContentPage.Resources>
    <ResourceDictionary>
      </ResourceDictionary>
    </ContentPage.Resources>

    → <StackLayout BackgroundColor="{StaticResource bg}">
      ...
    </StackLayout>

  </ContentPage>
```

Will throw an  
exception if  
key not found

# Motivation [change]

- ❖ Resource values might change over time; however, resources applied with **StaticResource** will not update in response to the change

Value applied  
once when the  
object is created

```
<ContentPage ...>

  <ContentPage.Resources>
    <ResourceDictionary>
      <Color x:Key="bg">Blue</Color>
    </ResourceDictionary>
  </ContentPage.Resources>

  <StackLayout BackgroundColor="{StaticResource bg}">
    ...
  </StackLayout>

</ContentPage>
```

# How to update Resources

- ❖ Can update resource values from code, useful when you download new values or let the user select preferred colors, font sizes, etc.

Define a  
default  
in XAML

```
<ResourceDictionary>
  → <Color x:Key="bg">Blue</Color>
</ResourceDictionary>
```

Update  
to new  
value

```
void OnChangeColor()
{
  → this.Resources["bg"] = Color.Green;
}
```

# Using dynamic Resources

- ❖ The **DynamicResource** markup extension retrieves a resource when the target object is created and updates it as the value changes

BackgroundColor  
set to **Blue** initially



```
<ResourceDictionary>
  <Color x:Key="bg">Blue</Color>
</ResourceDictionary>

<StackLayout BackgroundColor="{DynamicResource bg}">
  ...
</StackLayout>
```

BackgroundColor  
changes to **Green**



```
void OnChangeColor()
{
  this.Resources["bg"] = Color.Green;
}
```



# Key not found is OK

- ❖ **DynamicResource** leaves the property unset if the key is not found, it is not an error and no exception is generated

Key not  
defined

No value assigned to  
**BackgroundColor**

```
<ContentPage ...>

  <ContentPage.Resources>
    <ResourceDictionary>
    </ResourceDictionary>
  </ContentPage.Resources>

  <StackLayout BackgroundColor="{DynamicResource bg}">
    ...
  </StackLayout>

</ContentPage>
```

# Applying Resources in code

- ❖ Resources can be set in code using **SetDynamicResource**, allows logic to apply different resources based on runtime knowledge

```
var name = new Label { Text = "Name" };  
  
if (Device.OS == TargetPlatform.iOS)  
{  
    name.SetDynamicResource(Label.TextColorProperty, "hlColor");  
}
```

The **BindableProperty** to assign

The Resource key to apply

# Flash Quiz

# Flash Quiz

- ① If a ResourceDictionary value is updated at runtime, UI elements using it will update automatically:
- a) Always
  - b) Never
  - c) Only when using **DynamicResource**

# Flash Quiz

- ① If a ResourceDictionary value is updated at runtime, UI elements using it will update automatically:
- a) Always
  - b) Never
  - c) Only when using **DynamicResource**

# Flash Quiz

- ② To use a **DynamicResource** in code for a UI element named **myElement**, you would call:
- a) `myElement.DynamicResource.Add(...)`
  - b) `myElement.SetDynamicResource(...)`
  - c) `DynamicResource.SetOnElement(myElement, ...)`



# Flash Quiz

- ② To use a **DynamicResource** in code for a UI element named **myElement**, you would call:
- a) `myElement.DynamicResource.Add(...)`
  - b) `myElement.SetDynamicResource(...)`
  - c) `DynamicResource.SetOnElement(myElement, ...)`

# Individual Exercise

Dynamically update Resources



**Xamarin**  
University

# Summary

1. Use page-level Resources
2. Dynamically update Resources





# Create consistent UI with Styles

# Tasks

1. Create and apply a Style
2. Use Style inheritance to avoid repeated Setters



# Motivation [repeated code]

- ❖ Resources let you avoid duplicate values, but you still have to set each property individually which creates clutter and yields repeated code

The property settings must be repeated on each view

```
<Button
  BackgroundColor="{StaticResource highlightColor}"
  BorderColor    ="{StaticResource edgeColor}"
  BorderRadius   ="{StaticResource edgeRadius}"
  BorderWidth    ="{StaticResource edgeSize}"
  TextColor      ="{StaticResource textColor}"
  Text            ="OK" />
```

OK

```
<Button
  BackgroundColor="{StaticResource highlightColor}"
  BorderColor    ="{StaticResource edgeColor}"
  BorderRadius   ="{StaticResource edgeRadius}"
  BorderWidth    ="{StaticResource edgeSize}"
  TextColor      ="{StaticResource textColor}"
  Text            ="Cancel" />
```

Cancel



# Motivation [efficiency]

- ❖ Resource lookup can increase the startup time of your app since the lookup takes longer than assigning a literal value

```
<Button  
  TextColor="{StaticResource textColor}"  
  ... />
```

Slower

```
<Button  
  TextColor="White"  
  ... />
```

Faster

# What is a Setter?

- ❖ A **Setter** is a container for a property → value pair

```
<Setter Property="TextColor" Value="White" />
```



A bindable  
property

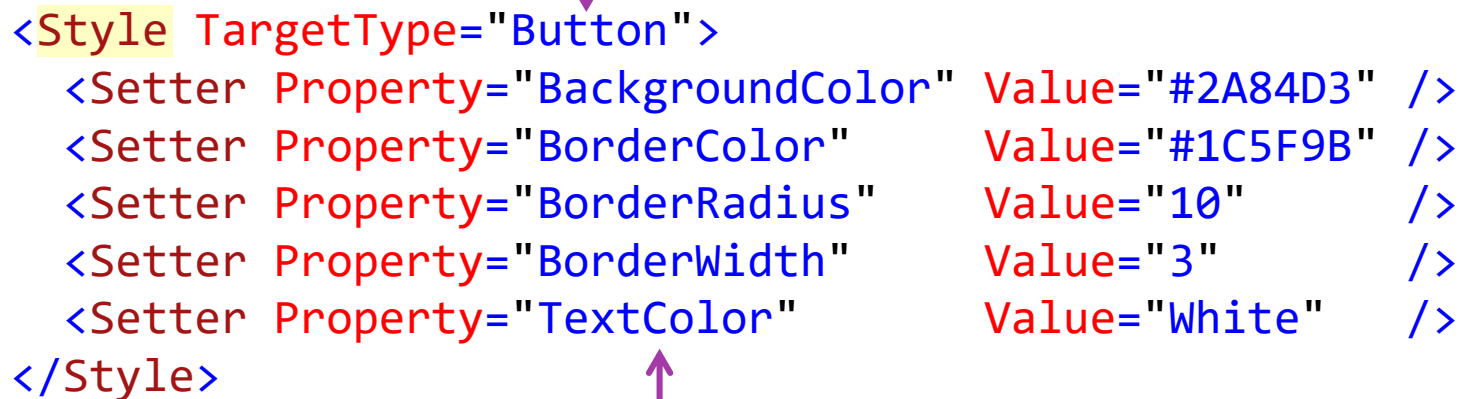


A value appropriate  
for the property

# What is a Style?

- ❖ A **Style** is a collection of setters for a particular type of view

**TargetType** must be set (or runtime exception)



```
<Style TargetType="Button">
  <Setter Property="BackgroundColor" Value="#2A84D3" />
  <Setter Property="BorderColor" Value="#1C5F9B" />
  <Setter Property="BorderRadius" Value="10" />
  <Setter Property="BorderWidth" Value="3" />
  <Setter Property="TextColor" Value="White" />
</Style>
```

The properties must be members of the  
**TargetType** class (or runtime exception)

# Styles as Resources

- ❖ Styles are shareable, so they are generally defined as Resources

Define in a  
dictionary

```
<ContentPage.Resources>
  <ResourceDictionary>
    <Style x:Key="MyButtonStyle" TargetType="Button">
      ...
    </Style>
  </ResourceDictionary>
</ContentPage.Resources>
```

# Using a Style

- ❖ Styles are set on a control through the **Style** property, this applies all the setters in the style to that control

```
<Button Text="OK" Style="{StaticResource MyButtonStyle}" />  
<Button Text="Cancel" Style="{StaticResource MyButtonStyle}" />
```



The **Style** property is defined in the **VisualElement** base class so it is available in all views

# Combining Styles and Resources

- ❖ Can use a resource as the **Value** for a setter, this lets it share a value with other styles

```
<Color x:Key="bgColor">White</Color>
<Color x:Key="fgColor">Black</Color>

<Style TargetType="Button" x:Key="AllButtons">
  <Setter Property="BackgroundColor" Value="{StaticResource bgColor}" />
  <Setter Property="TextColor" Value="{DynamicResource fgColor}" />
  ...
</Style>
```

Can use either static or dynamic lookup

# Implicit Styles

- ❖ Styles can be automatically applied to all controls of a target type by omitting **x:Key** and placing the style into an accessible dictionary

```
<ContentPage.Resources>
  <ResourceDictionary>
    → <Style TargetType="Button">
      <Setter Property="BackgroundColor" Value="Blue" />
      <Setter Property="BorderColor" Value="Navy" />
      ...
    </Style>
  </ResourceDictionary>
</ContentPage.Resources>
```

The target type is still specified and is matched exactly, this style will be applied to all buttons in this page

# Overriding a setter

- ❖ Styles provide the *default* values, explicit property values on the control are applied *after* the style and take precedence

```
<Style x:Key="MyButtonStyle" TargetType="Button">  
  <Setter Property="BackgroundColor" Value="Red" />  
</Style>
```

```
<Button  
  Style="{StaticResource MyButtonStyle}"  
  BackgroundColor="Blue" ✓  
  Text="Cancel"  
  ... />
```



Value set directly overrules the style value

Background is blue, not red



# Ancestor targeting

- ❖ A **Style** can target a base type of the object to which it is applied

This style targets **VisualElement**



```
<Style x:Key="MyVisualElementStyle" TargetType="VisualElement">  
  <Setter Property="BackgroundColor" Value="#2A84D3" />  
</Style>
```

```
<Button Style="{StaticResource MyVisualElementStyle}" ... />
```



Can apply to a button since the **Button** class is derived from **VisualElement**

# Creating a Style in code

- ❖ Styles can be created in code to allow runtime customizations

```
var s = new Style(typeof(Button));  
  
s.Setters.Add(new Setter {Property = Button.BackgroundColorProperty, Value = Color.Red});  
s.Setters.Add(new Setter {Property = Button.BorderRadiusProperty, Value = 4});
```



Can then apply **Style** to a **Button** directly, or add it to the resources to apply in XAML



# Individual Exercise

Create and apply a Style

# Motivation [repeated code]

- ❖ Styles often have duplicate Setters which are then hard to maintain

Repeated

```
<Style x:Key="MyButtonStyle" TargetType="Button">
  <Setter Property="BackgroundColor" Value="Blue" />
  <Setter Property="BorderColor" Value="Navy" />
  <Setter Property="BorderWidth" Value="5" />
</Style>

<Style x:Key="MyEntryStyle" TargetType="Entry">
  <Setter Property="BackgroundColor" Value="Blue" />
  <Setter Property="TextColor" Value="White" />
</Style>
```

# Motivation [customization]

- ❖ A provided Style might need some adjustment to meet your needs

```
<Style x:Key="MyButtonStyle" TargetType="Button">
  <Setter Property="BackgroundColor" Value="Blue" />
  ...
</Style>
```

← Color might not be right for current use

```
<Button Style="{StaticResource MyButtonStyle}" Text="OK" BackgroundColor="Purple" />
<Button Style="{StaticResource MyButtonStyle}" Text="Cancel" BackgroundColor="Purple" />
```



↑ It is tedious to manually set properties that don't fit the current situation

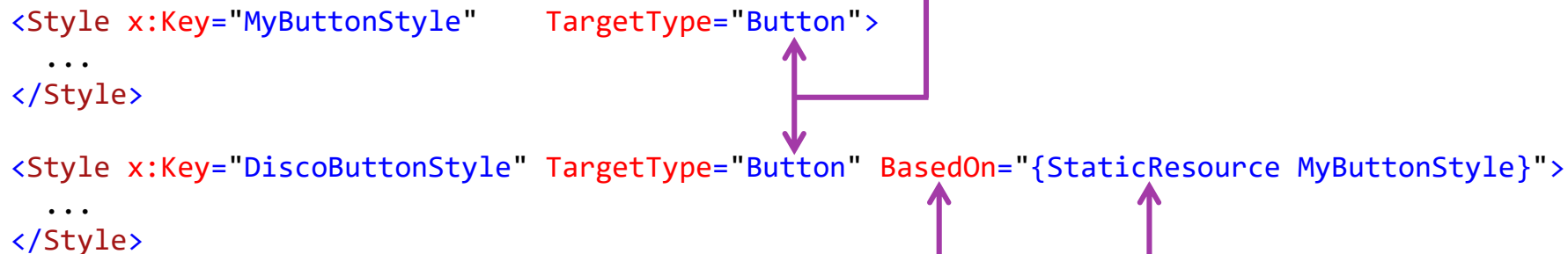
# Style inheritance

- ❖ A style can inherit from a base style

Base's **TargetType** must be the same or a base class

```
<Style x:Key="MyButtonStyle" TargetType="Button">
    ...
</Style>

<Style x:Key="DiscoButtonStyle" TargetType="Button" BasedOn="{StaticResource MyButtonStyle}">
    ...
</Style>
```



Indicates which style  
this will inherit from

Only **StaticResource** is  
allowed to set the base style

# Inherited properties

- ❖ The new style can modify existing property values and/or add new ones

```
<Style x:Key="MyButtonStyle" TargetType="Button">
  <Setter Property="BackgroundColor" Value="Blue" />
  <Setter Property="BorderColor" Value="Navy" />
</Style>

<Style x:Key="DiscoButtonStyle" TargetType="Button" BasedOn="{StaticResource MyButtonStyle}">
  <Setter Property="BackgroundColor" Value="Purple" />
  <Setter Property="Rotation" Value="30" />
</Style>
```

Add new setter

Replace inherited setter



# Individual Exercise

Use Style inheritance to refactor repeated code



# Summary

1. Create and apply a Style
2. Use Style inheritance to avoid repeated Setters





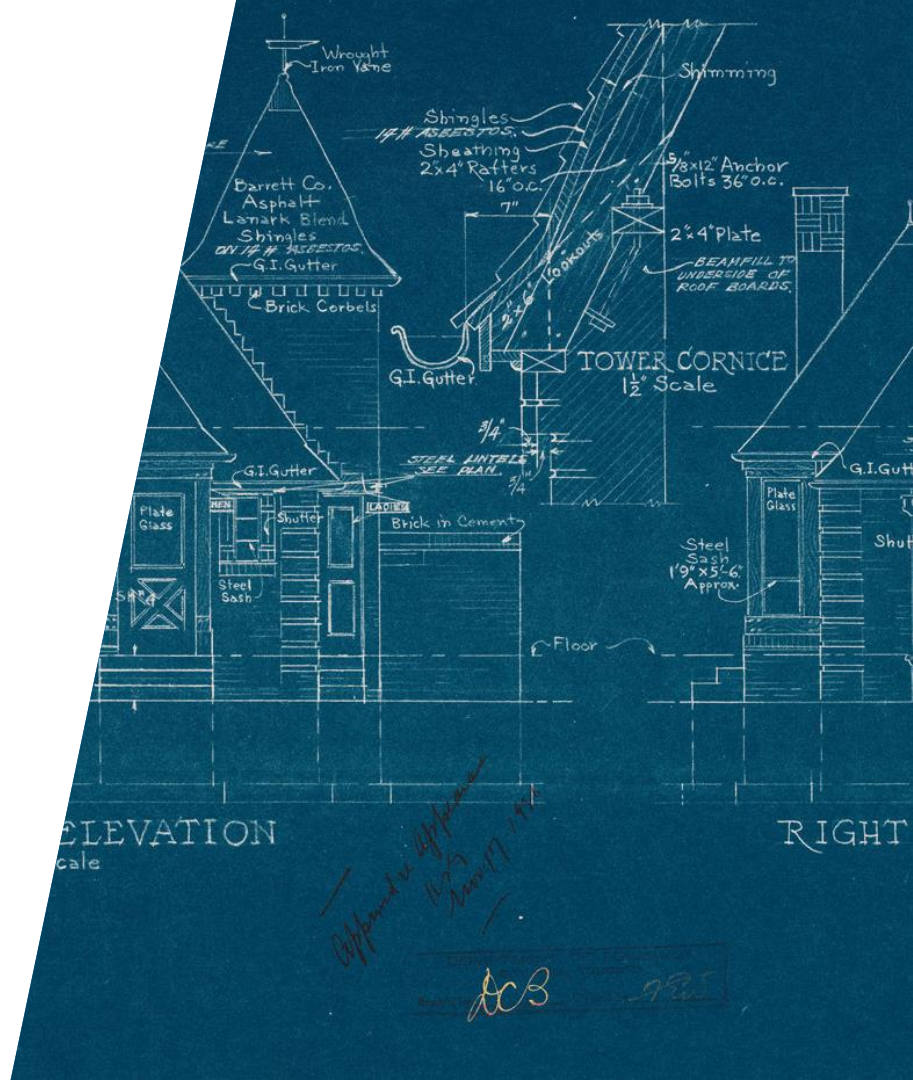
Make your Resources and Styles  
available across your entire app



**Xamarin**  
University

# Tasks

1. Create App.xaml
2. Use application-wide resources



# Motivation

- ❖ You will often need to share resources across multiple pages of your app; however, page-level resources are only available on one page

```
<ContentPage ...>

  <ContentPage.Resources>
    <ResourceDictionary>
      <x:Double x:Key="size">32</x:Double>
    </ResourceDictionary>
  </ContentPage.Resources>
  ...
  <Label FontSize="{StaticResource size}" />
  ...
</ContentPage>
```



OK, definition and use are in the same page

```
<ContentPage ...>
  ...
  ...
  ...
  ...
  ...
  ...
  <Button FontSize="{StaticResource size}" />
  ...
</ContentPage>
```



Resources defined in one page are not available in a different page

# Available dictionaries

- ❖ **VisualElement** and **Application** have built-in resource dictionaries
  - these are initialized to **null** by default

```
public class VisualElement : ...  
{ ...  
    public ResourceDictionary Resources  
    {  
        get;  
        set;  
    }  
}
```

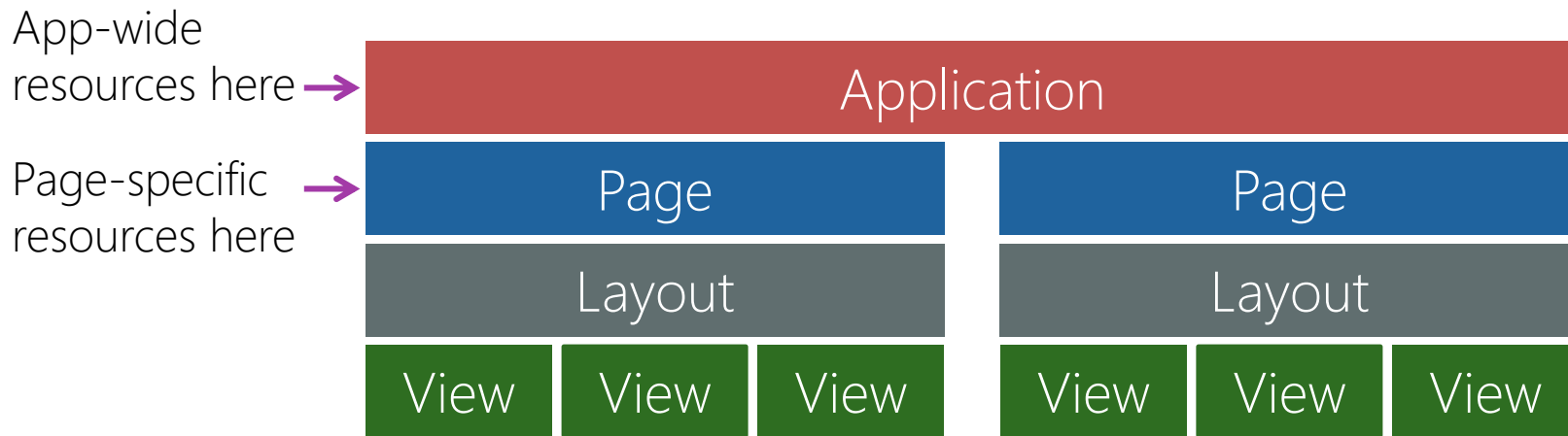
Pages, layouts, and views  
inherit from **VisualElement**

```
public class Application : ...  
{ ...  
    public ResourceDictionary Resources  
    {  
        get;  
        set;  
    }  
}
```

Your app class inherits  
from **Application**

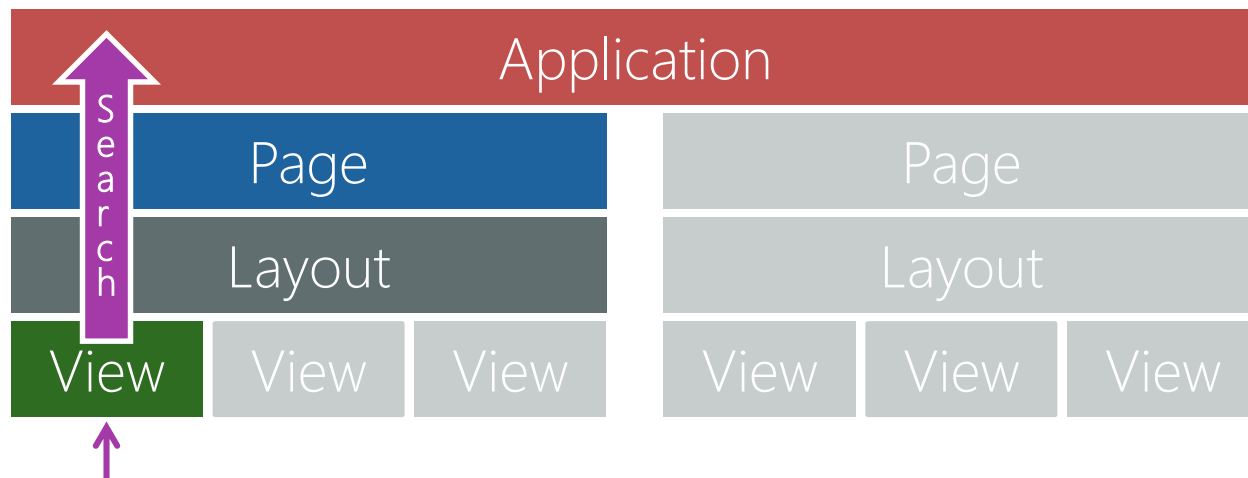
# Resource scope

- ❖ Resources can be defined at different levels so they are scoped to a specific usage area in the application



# Lookup rules

- ❖ Dictionaries are searched starting at the point a resource is applied, then up the visual tree to the Page, and finally to the App



Apply a resource to a view, lookup will proceed up the hierarchy

Place resources close to where they are used to minimize lookup cost

# Defining application-level resources

- ❖ **App.xaml** and **App.xaml.cs** files are needed in order to use an application-wide resource dictionary in xaml

App.xaml

```
<Application
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="MyApp.App">

  <Application.Resources>
    <ResourceDictionary>
      <x:Double x:Key="size">32</x:Double>
    </ResourceDictionary>
  </Application.Resources>

</Application>
```

App.xaml.cs

```
namespace MyApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new MainPage();
        }
    }
}
```



# Using application-level resources

- ❖ Can use either **StaticResource** or **DynamicResource** to apply an application-level resource

```
<ContentPage ...>  
...  
  <Label FontSize="{StaticResource size}" />  
...  
</ContentPage>
```

```
<ContentPage ...>  
...  
  <Button FontSize="{StaticResource size}" />  
...  
</ContentPage>
```

The resource will be available in all pages of the app

# Duplicate keys

- ❖ Keys can be repeated in different dictionaries, the first matching key on the search path is used

```
<Application.Resources>
  <ResourceDictionary>
    <x:String x:Key="msg">Two</x:String>
  </ResourceDictionary>
</Application.Resources>
```

App.xaml

```
<ContentPage.Resources>
  <ResourceDictionary>
    <x:String x:Key="msg">One</x:String>
  </ResourceDictionary>
</ContentPage.Resources>
```

MainPage.xaml

Text set  
to One



```
<Label Text="{StaticResource msg}">
```

# Group Exercise

Use application-wide Resources



**Xamarin**  
University

# Merged Dictionaries

- ❖ Xamarin.Forms allows you to import a dictionary into another dictionary by assigning the **MergedWith** property of a **ResourceDictionary**

AboutPage

AboutPage owns a  
ResourceDictionary

```
<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary>
      <x:Double x:Key="size">32</x:Double>
    </ResourceDictionary>
  </ContentPage.Resources>
  ...
</ContentPage>
```

SettingsPage

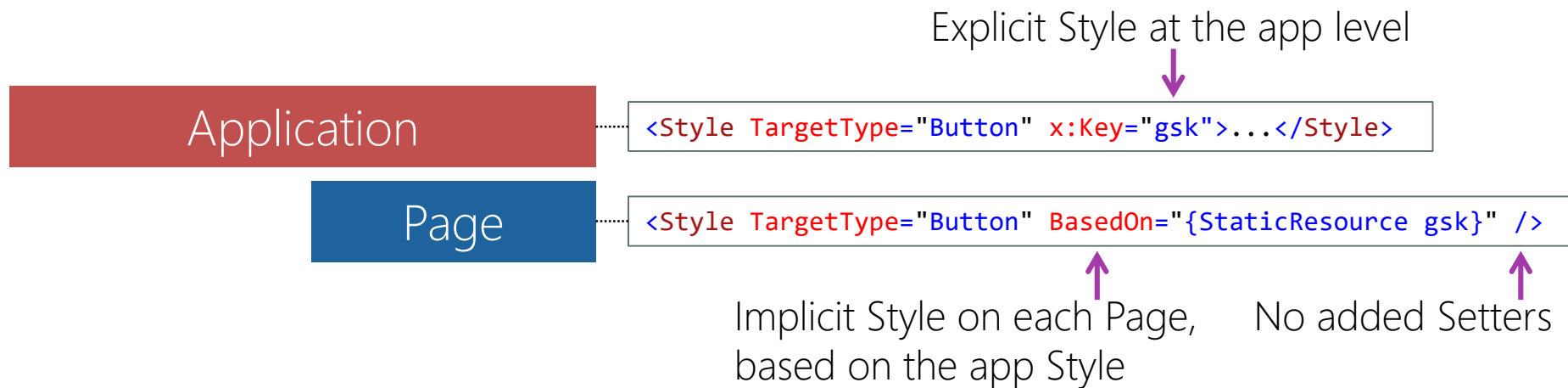
dictionary is referenced  
by owning class name

```
<ContentPage ...>
  <ContentPage.Resources>
    <ResourceDictionary
      MergedWith="theme:AboutPage" />
    </ContentPage.Resources>
    ...
  </ContentPage>
```

SettingsPage now  
has access to resources  
defined in AboutPage

# Guideline for global styles

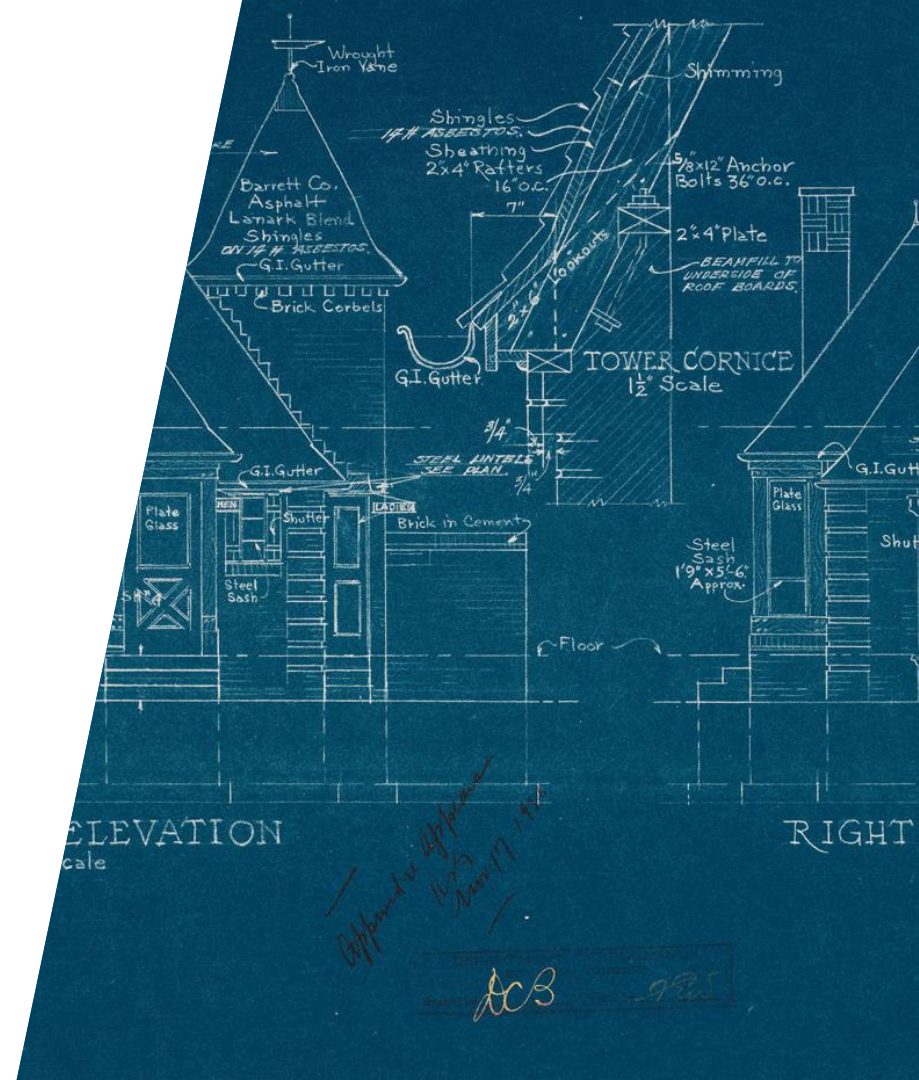
- ❖ Use explicit styles at the application level and then put an implicit style in each page that uses **BasedOn**



 This approach makes it clear which implicit Styles will be applied on each page

# Summary

1. Create App.xaml
2. Use application-wide resources





Apply the user's Accessibility choices  
with built-in Styles



**Xamarin**  
University

# Tasks

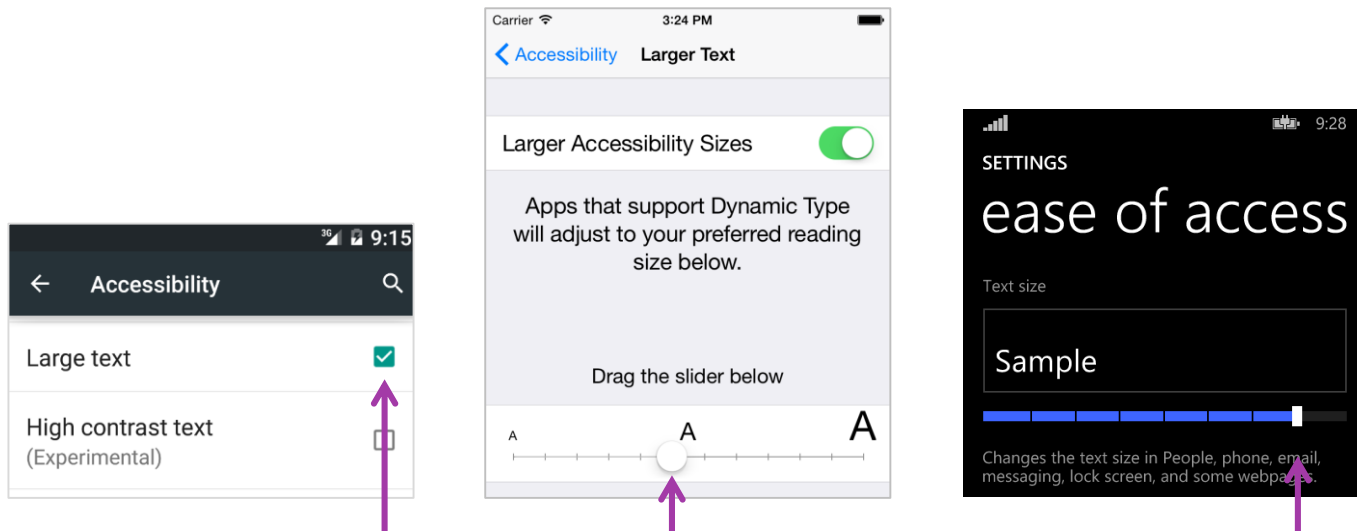
1. Apply a built-in Style
2. Customize a built-in Style





# Motivation


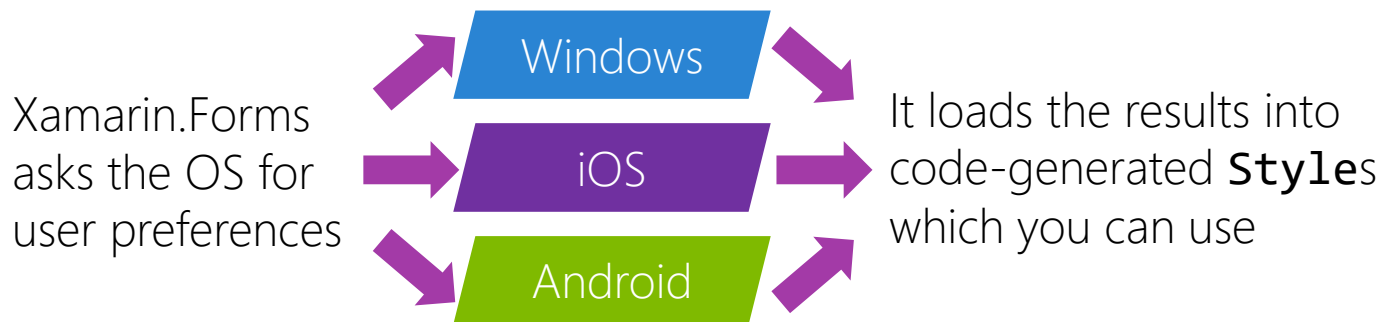
- ❖ Apps should respect the user's device-wide preferences for appearance and accessibility; ideally, apps update their UI when settings change



Apps should try to use the text size the user requested

# What is a built-in Style?

- ❖ Xamarin.Forms maps the user's device-wide preferences to Styles, it keeps those Styles updated as the user changes their settings




Built-in Styles are under development, please expect changes and additions.

# Implementation

- ❖ The built-in styles are provided as **Style** objects in **Device.Styles**

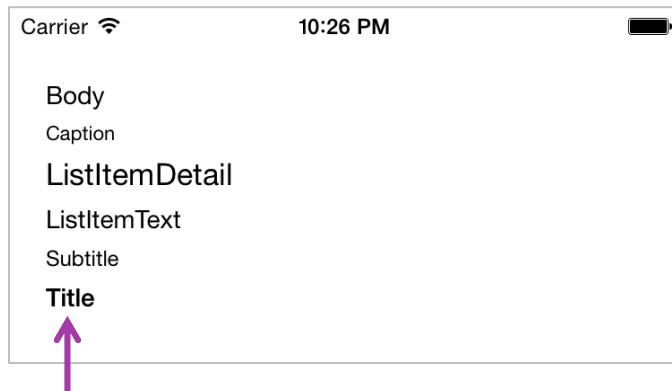
```
public static class Styles
{
    ...
    public static readonly Style BodyStyle;
    public static readonly Style CaptionStyle;
    public static readonly Style ListItemDetailTextStyle;
    public static readonly Style ListItemTextStyle;
    public static readonly Style SubtitleStyle;
    public static readonly Style TitleStyle;
}
```



Styles are for common UI  
like titles, body text, and lists

# Targets

- ❖ The built-in Styles use a **TargetType** of **Label**



The Styles have setters for common properties such as fonts and colors

# Resource keys

- ❖ Symbolic constants from **Device.Styles** identify the built-in Styles in XAML

```
public static class Styles
{ ...
    public static readonly string BodyStyleKey           = "BodyStyle";
    public static readonly string CaptionStyleKey        = "CaptionStyle";
    public static readonly string ListItemDetailTextStyleKey = "ListItemDetailTextStyle";
    public static readonly string ListItemTextStyleKey    = "ListItemTextStyle";
    public static readonly string SubtitleStyleKey        = "SubtitleStyle";
    public static readonly string TitleStyleKey          = "TitleStyle";
}
```

You use these in your XAML

# Using a built-in Style

- ❖ Must use **DynamicResource** to access a built-in Style

```
public static class Styles
{
    ...
    public static readonly string TitleStyleKey = "TitleStyle";
}
```

Use the predefined string resource key

```
<Label Text="Welcome" Style="{DynamicResource TitleStyle}" />
```



**DynamicResource** is required because these styles are generated via code and can change at runtime if the user changes their preferences

# Customizing built-in Styles

- ❖ **BaseResourceKey** lets you use a built-in Style as a base, it performs a dynamic lookup which keeps the property values synchronized to the user preferences

```
<Style BaseResourceKey="TitleStyle" TargetType="Label" x:Key="MyTitleStyle">  
    ...  
</Style>
```



Property identifies the Resource to use as the **BasedOn** style  
(i.e. you are supplying a key that will be used for Resource lookup)

# Flash Quiz



# Flash Quiz

- ① The built-in Styles work with \_\_\_\_\_
- a) **Label**
  - b) **Entry**
  - c) **ListView**
  - d) All of the above

# Flash Quiz

- ① The built-in Styles work with \_\_\_\_\_
- a) Label
  - b) Entry
  - c) ListView
  - d) All of the above

# Flash Quiz

- ② The built-in Styles generally set \_\_\_\_\_
- a) Text properties
  - b) Layout properties

# Flash Quiz

- ② The built-in Styles generally set \_\_\_\_\_
- a) Text properties
  - b) Layout properties

# Summary

1. Apply a built-in style
2. Customize a built-in style



# Thank You!

Please complete the class survey in your profile:  
[university.xamarin.com/profile](https://university.xamarin.com/profile)