# C# Fundamentals and Xamarin Forms

OzCode
Your Road to Magical Debugging

CODEVALUE

**Alex Pshul (@AlexPshul)**
Software Architect & Consultant
alexp@codevalue.net
https://www.pshul.com

# About Me

- Alex Pshul
  - @AlexPshul
  - www.pshul.com
  - alexp@codevalue.net
- Software Architect & Consultant @CodeValue Ltd.
- More than 8 years of hands on experience
- OzCode Evangelist (www.oz-code.com)
- Talk to me about:
  - Software Development
  - Hardware and Gadgets
  - Gaming
  - Animals

# C# Language Fundamentals

# Agenda

- Statements and Expressions
- Comments
- Identifiers
- Variables
- Literals
- Operators
- Control Statements
- Summary

# Statements, Expressions and Comments

- Statement
  - Comment, declaration, expression, control statement, block
- Expression
  - Anything that evaluates to a value
- Comments

```
// single line comment
```

```
/* this is a multiline comment
   can span as many lines as you like
*/
```

```
/// <summary>
/// this is XML comments
/// typically used for documentation
/// </summary>
```

# Identifiers

- Names used for fields, types, methods, etc.
- Must start with a letter or an underscore
  - Can continue with letters, underscore or digits
- C# is case sensitive
  - However, non private code elements must not be different by case alone
- C# has a number of reserved words
  - Cannot be used as identifiers
    - unless prefixed by @
- Naming convention is important from a design and maintenance perspective

# Variables

- Symbolic name for an address in memory
- All variables must be declared before used
- Must be initialized before read from
- Variable declaration is type and variable name

```
void DoSomething() {
    int counter = 0;
    double pi;
    string name = "Bart Simpson";
    int x = 4, y = 10;

// does not compile: Console.WriteLine(pi);

    pi = 3.1415653589;
}
```

# .NET / C# Integral Predefined Types

- int and the like are never “platform dependent” as they are in C/C++

| .NET type name | C# keyword | Size (bytes) | Description |
|---|---|---|---|
| System.Sbyte | sbyte | 1 | Signed byte (-128 to +127) |
| Systen.Byte | byte | 1 | Unsigned byte (0 to 255) |
| System.Int16 | short | 2 | Signed 16-bit integer (-32768 to +32767) |
| System.UInt16 | ushort | 2 | Unsigned 16-bit integer (0 to 65535) |
| System.Int32 | int | 4 | Signed 32-bit integer |
| System.UInt32 | uint | 4 | Unsigned 32-bit integer |
| System.Int64 | long | 8 | Signed 64 bit integer |
| System.UInt64 | ulong | 8 | Unsigned 64-bit integer |
| System.Char | char | 2 | A Unicode (UTF-16) character |
| System.Boolean | bool | 1 | A boolean value (false or true) |

# .NET / C# Other Predefined Types

| .NET type name | C# keyword | Size (bytes) | Description |
|---|---|---|---|
| System.Single | float | 4 | IEEE 754 single precision floating point value |
| Systen.Double | double | 8 | IEEE 754 double precision floating point value |
| System.Decimal | decimal | 16 | 28 digit precision (used for monetary values) |
| System.String | string | Depends | A sequence of Unicode characters |

# Numeric Literals

- Integer literals
  - Decimal or hexadecimal (**0x** prefix for hex)
  - Defaults to **int**, **uint**, **long**, **ulong**
  - **L** or **l** specifies long
  - **U** or **u** specifies unsigned

```
56      -65     7623u   0x56a
0x2000L         77L     123UL
```

- Floating point literals
  - Standard or scientific notation
  - **double** by default
  - **F** or **f** specifies float
  - **D** or **d** specifies double
  - **M** or **m** specifies decimal

```
4.0     3.2f    .67     3.88M
2.4e-9          1e8
6.66e-10
```

# Other Literals

- Boolean
  - Can be `true` or `false`
- Single character
  - Single quotes
  - Character, escape (backslash), hex, Unicode

```
'a'          '\n'
'\x041'      '\u006f'
```

- String of characters
  - Double quotes
  - Can prefix with @ to disable escaping

```
"Hello, world!\n"
@"c:\temp\myfile.txt"
```

# Operators

- Mathematical
- Bitwise
- Logical / Relational
- Assignment
- Increment / decrement
- Conversions and Casts

# Mathematical Operators

- Precedence
  - Multiplication, division, modulo
  - Addition, subtraction
- Can use parenthesis to change precedence

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulo (remainder) |

# Bitwise Operators

| Operator | Description |
|---|---|
| & | AND |
| \| | OR |
| ^ | XOR |
| ~ | NOT |
| >> | Shift right |
| << | Shift left |

OR truth table

| A | B | A \| B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

XOR truth table

| A | B | A ^ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

AND truth table

| A | B | A & B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Relational / Logical Operators

| Operator | Description |
|---|---|
| == | Equal |
| != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

- Relational operators return **`true`** or **`false`**

| Operator | Description |
|---|---|
| && | And |
| \|\| | Or |
| ! | not |

- Precedence
  - Not
  - And
  - Or
  - Relational operators
- Logical operators use “short circuit” evaluation

```
bool r1 = x > 7;
bool r2 = y == 6 || x * 3 < 200;
bool r3 = z >= y && !y < 10 || x % 2 == 0;
```

# Assignment Operators

| Operator | Description |
|---|---|
| = | Simple assignment |
| op= | Compound assignment<br>“op” is one of:<br>+ - * / % >> << & \| ^ |
| ++ | Increment |
| -- | decrement |

- Increment and decrement
  - Support postfix and prefix
  - Prefix means lvalue is updated first
  - Postfix means current value is used first

```
x = y + 3;      // simple assignment
z *= 2;         // compound
x = y++ * 3;    // value of y used before increment
z = 2 + ++y;    // y is incremented and its new value used
```

# The var Keyword

- Introduced in C# 3.0
- Instructs the compiler to infer the type implicitly based on the right side of an assignment
- Mostly useful in LINQ scenarios (see Advanced .NET course)

```
// C# 2.0
int x = 5;
string name = "Bart Simpson";
Dictionary<string, object> data = new Dictionary<string, object>();
int size = name.Length;
```

```
// C# 3.0
var x = 5;
var name = "Bart Simpson";
var data = new Dictionary<string, object>();
var size = name.Length;
var y = x * 2.5;
var keys = data.Keys;   // Dictionary<string, object>.KeyCollection
```

# Arithmetic Operations

- Arithmetic operations use: **`int`**, **`uint`**, **`long`**, **`ulong`**, **`float`**, **`double`**, **`decimal`**
  - Result type is the widest of operand types
  - **`int`** used if both operands are integral and narrower
  - Cannot mix **`decimal`** and **`float`**, or **`ulong`** with any signed type
- Otherwise, a cast may be needed (next slide)

```
byte + short              // widened to int
short + long              // widened to long
uint + short              // widened to long
ulong + int               // does not compile
                          // widened to decimal
```

# Conversions and Casts

- Implicit cast silently convert to larger type
- Explicit cast is necessary when loss of information possible
- Casting is done by specifying the type to convert to in parenthesis

```
int x = 4;
long z = x;             // widening conversion
int k = z;              // does not compile
short s = (short)z;     // explicit cast - might lose data
```

# Control Statements

```
if (boolean_expression)
   Statement;
else          // optional
   Statement;
```

```
switch(value) {
   case constant1:
      Statements;
   default:      // optional
      Statements;
}
```

```
while (boolean_expression)
   Statement;
```

```
do
   Statement;
while (boolean_expression);
```

```
for(init_expr; boolean_expression; update_expr)  // all optional
   Statement;
```

```
foreach(SomeType item in collection)
   Statement;
```

- "Statement" may be a single statement or a block (statements enclosed between {})

# Loop Constructs

- `while`
  - The loop body may not execute at all
- `do-while`
  - The loop body executes at least once
- `for`
  - Similar to while with an initialization expression
  - Can use a declaration inside
  - Any part can be dropped

# Loop Examples: Sum of an Array

```
int index = 0, sum = 0;
while(index < 10) {
    sum += values[index];
    index++;
}
```

```
int index = 0, sum = 0;
do {
    sum += values[index++];
} while(index < 10);
```

```
int sum = 0;
for(int i = 0; i < 10; i++)
    sum += values[i];
```

```
int sum = 0;
for(int i = 0; i < 10; sum += values[i++])
    ;
```

```
int sum = 0;
foreach(int n in values)
    sum += n;
```

# Loop Control

- The **`break`** keyword
  - Causes exit from the innermost loop
- The **`continue`** keyword
  - Abandons the current iteration and jumps back to the start of the update clause or the body
- The **`goto`** keyword
  - Allows unconditional jump to a label

# Loop Control Examples

```
// find the first fibonacci number
// greater than 10000
int a = 1, b = 1, c;
for(; ;) {
   c = a + b;
   if(c > 10000)
      break;
   a = b;
   b = c;
}
Console.WriteLine(c);
```

```
// display all 2 digit numbers
// whose digit sum is less than 10
for(int d1 = 1; d1 <= 9; d1++)
   for(int d2 = 0; d2 <= 9; d2++) {
      if(d1 + d2 >= 10)
         continue;
      Console.WriteLine($"{d1}{d2}");
   }
```

# The `switch` Statement

- Can be used as a simple replacement of if/else if/else construct
- Checked value must be integer, **`enum`**, **`char`** or **`string`**
- No fall-through (as in C/C++)
  - Unless the **`case`** is empty
  - Must use **`break`**
  - Can use **`goto case`** to explicitly jump to another case
  - **`goto default`** is also supported
- **`case`** values may be in any order

```
string desc;
switch(level) {
   case 0:
      desc = "very low";
      break;
   case 1:
   case 2:
      desc = "medium";
      break;
   default:
      desc = "high";
      break;
}
```

# The Ternary Operator

- Contains 3 parts
- Syntax

```
A ? B : C
```

- Equivalent to

```
if(A) B else C
```

- But it’s an expression, not a statement

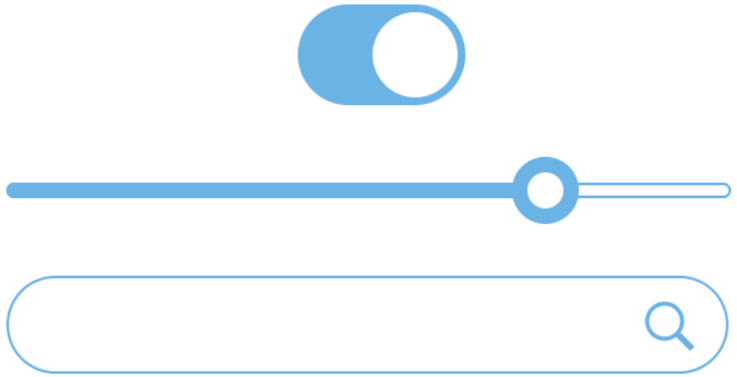
# Summary

- C# borrows its syntax from C++ and Java
- Supports the conventional constructs and control flow statements

# Xamarin (Forms)

# What is native?

# The Anatomy of a Native App



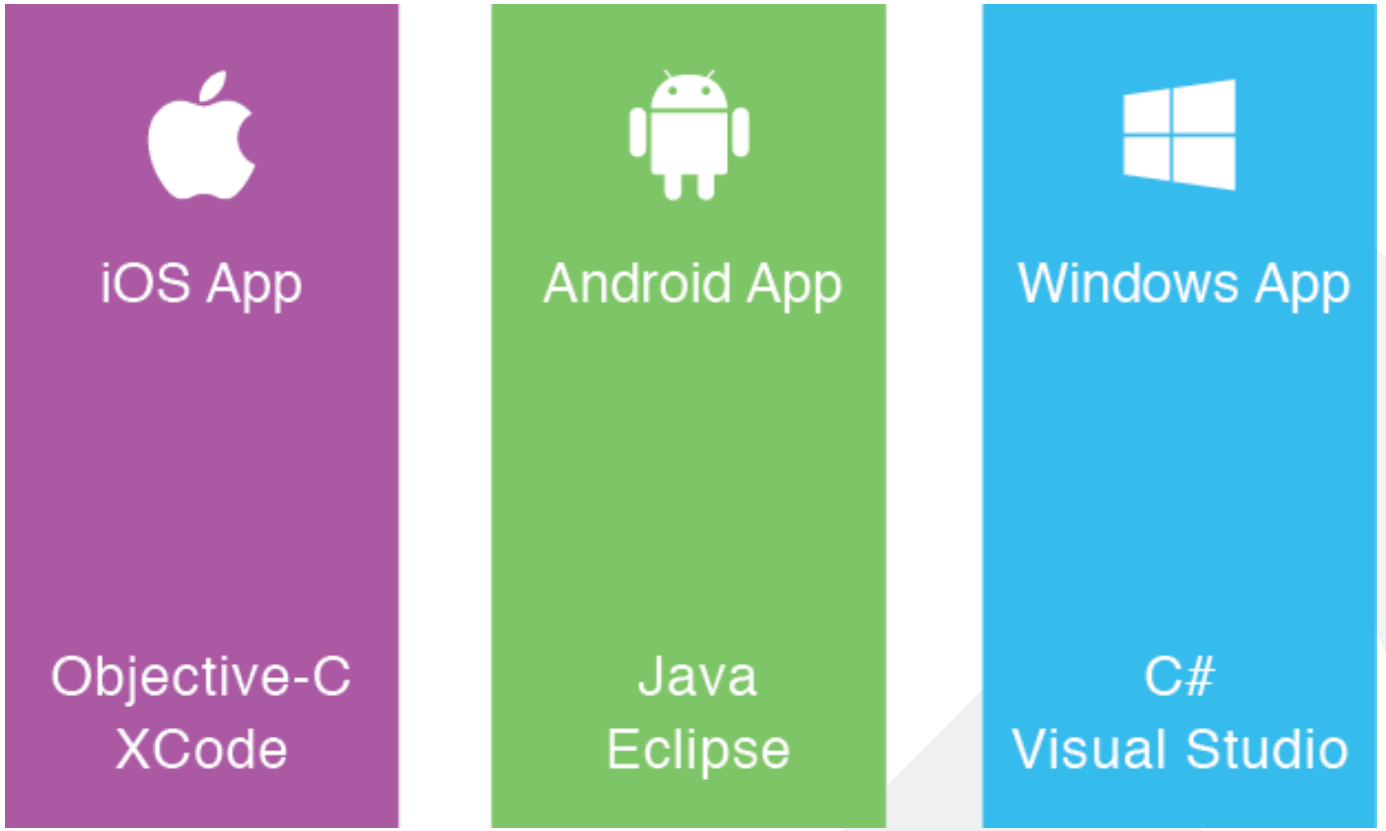
Native User Interfaces

# Architecting

# Mobile Apps

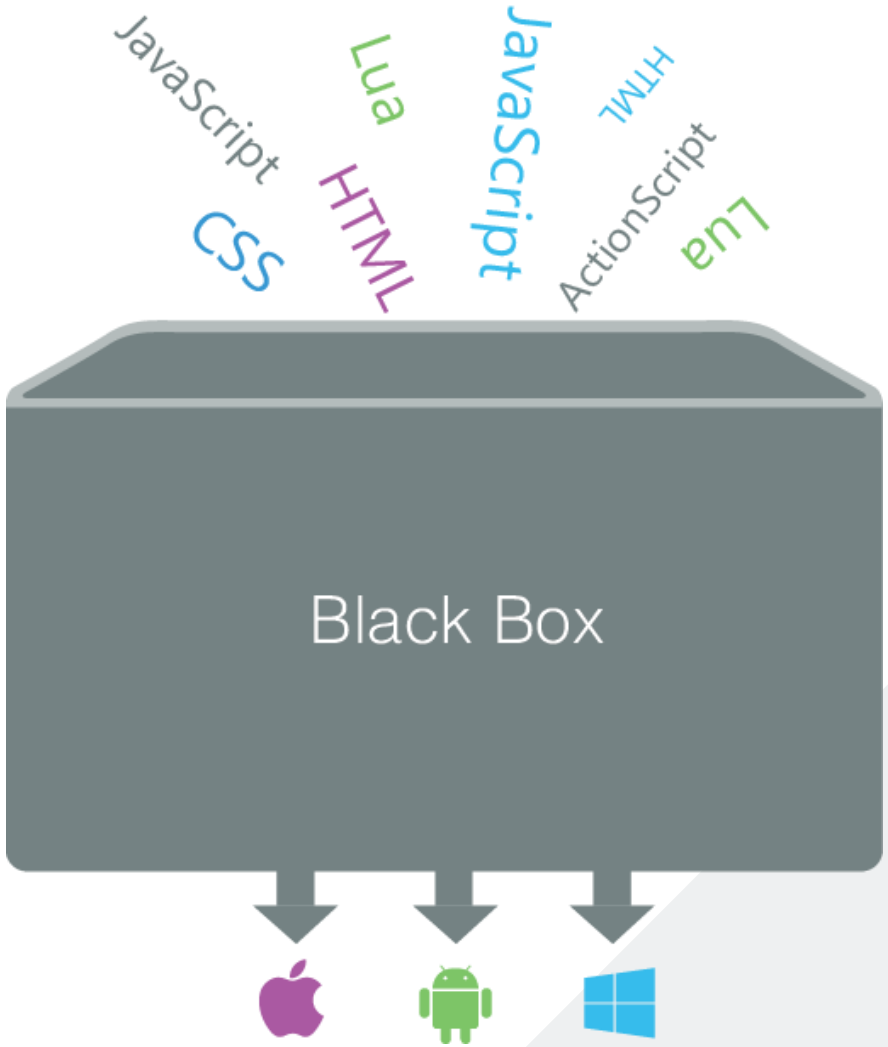
# The Silo Approach

Build App
Multiple Times



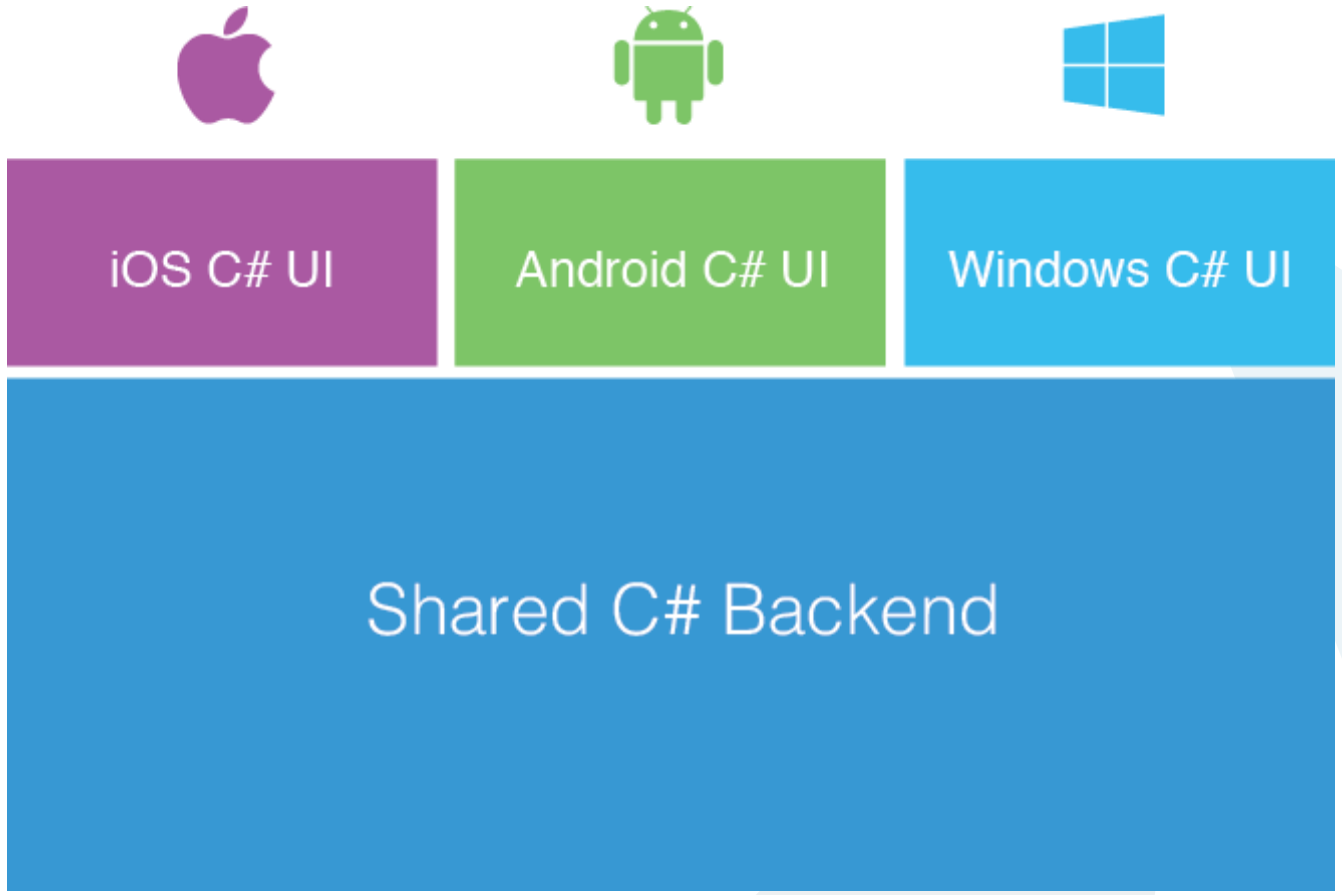
# The Write-Once-Run-Anywhere Approach

Lowest Common Denominator

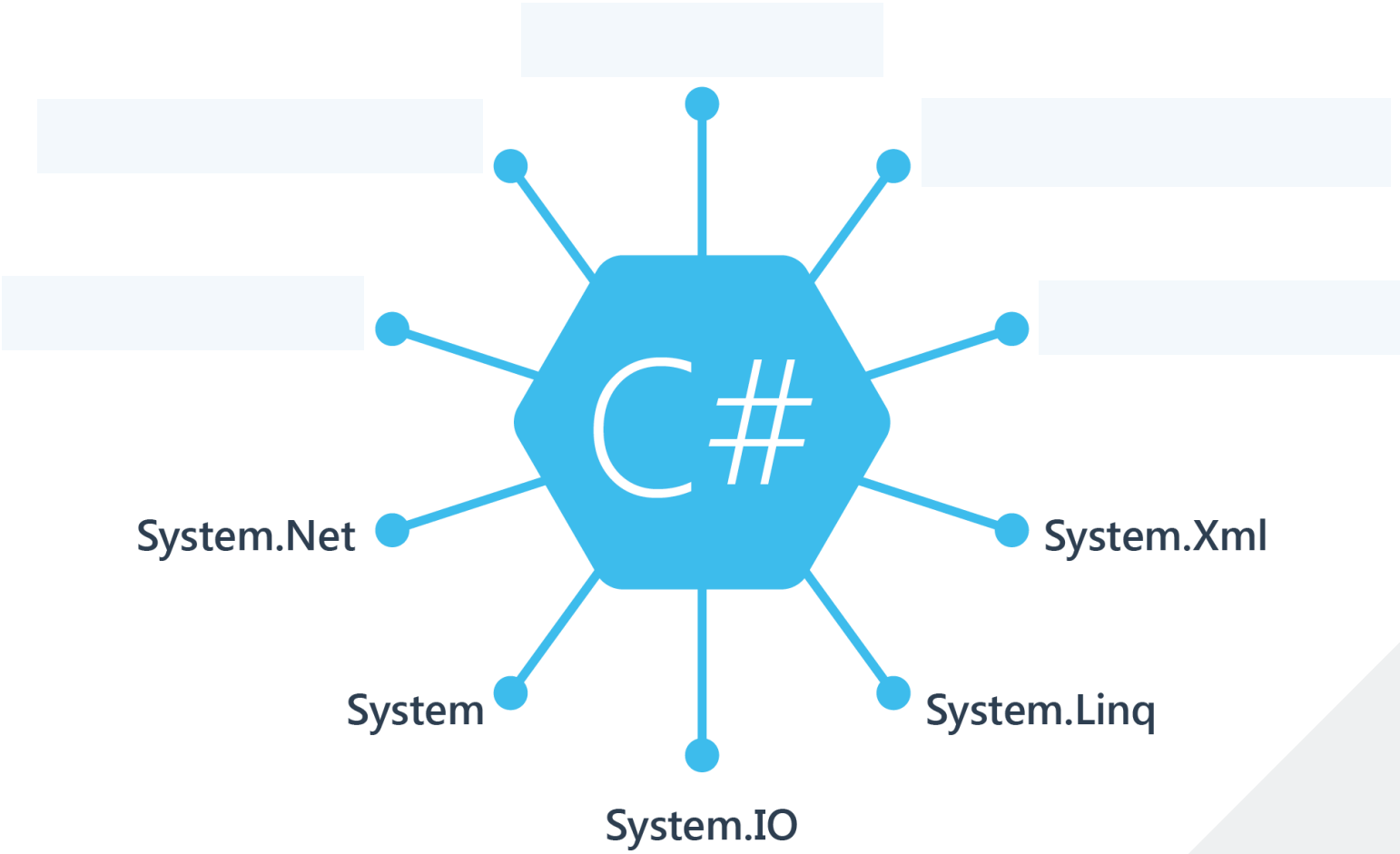


# Xamarin's Unique Approach

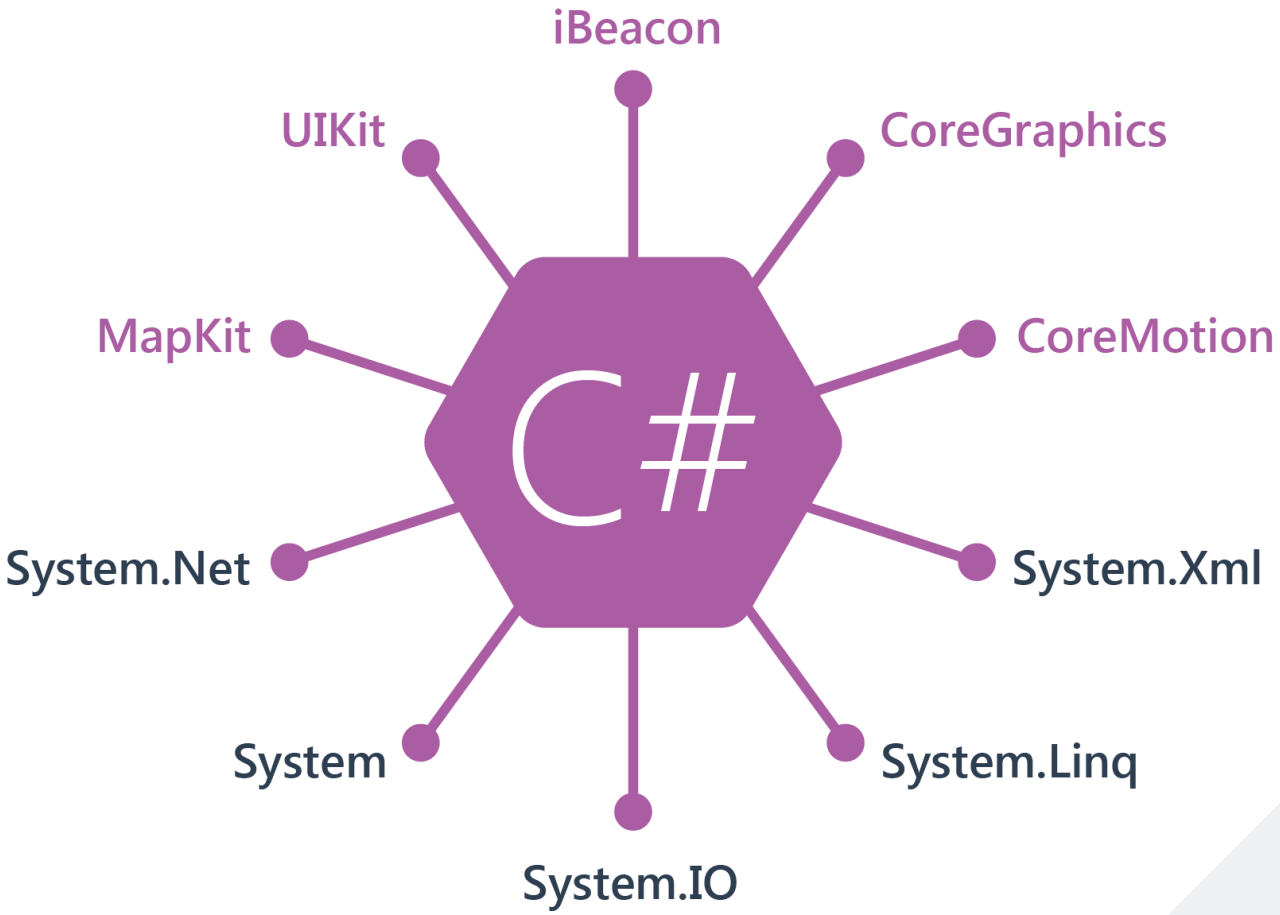
Native With
Code Sharing



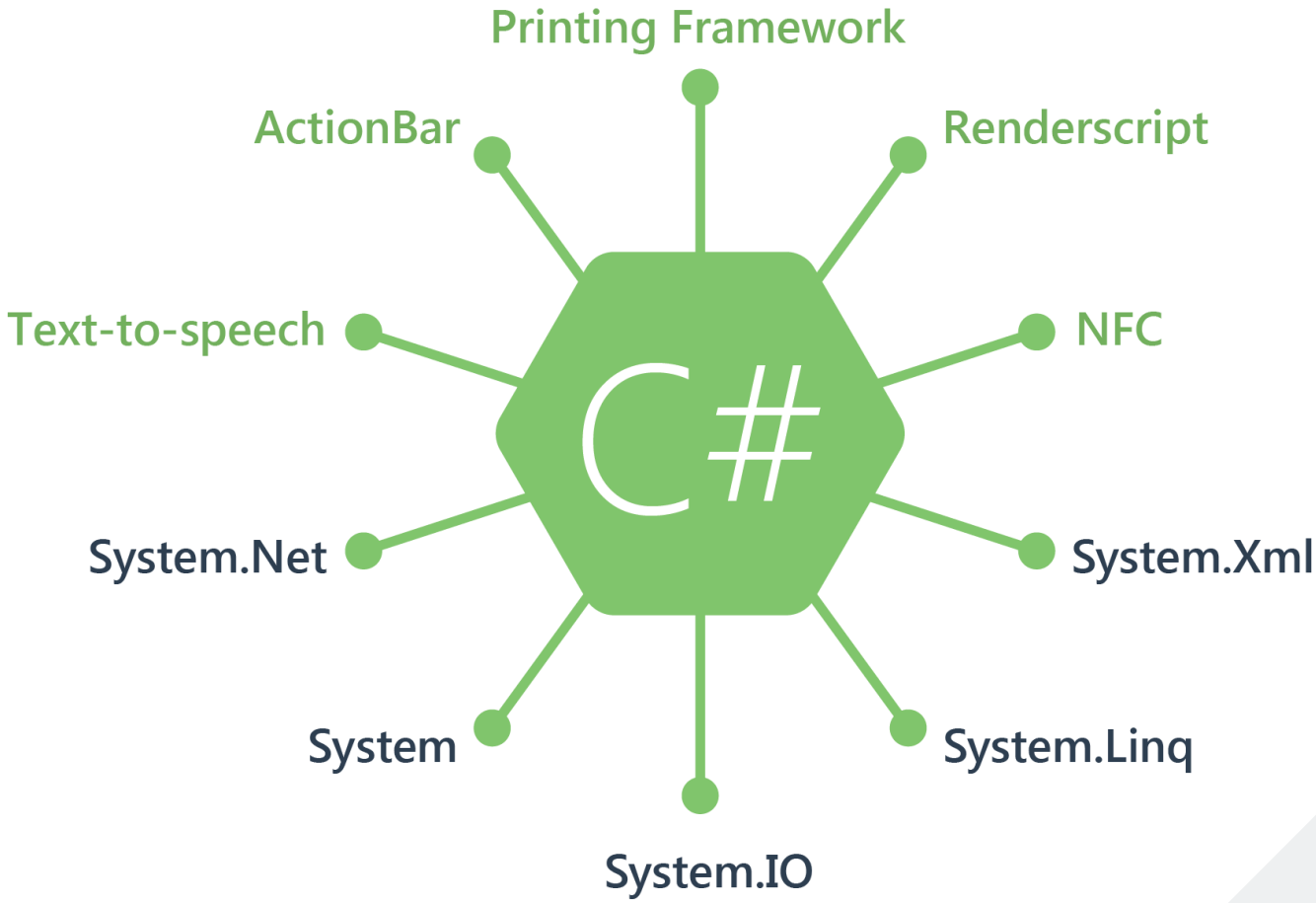
# Windows APIs



# iOS APIs | 100% Coverage



# Android APIs | 100% Coverage



Anything you can do in Objective-C, Swift, or Java can be done in C# with Xamarin using Visual Studio

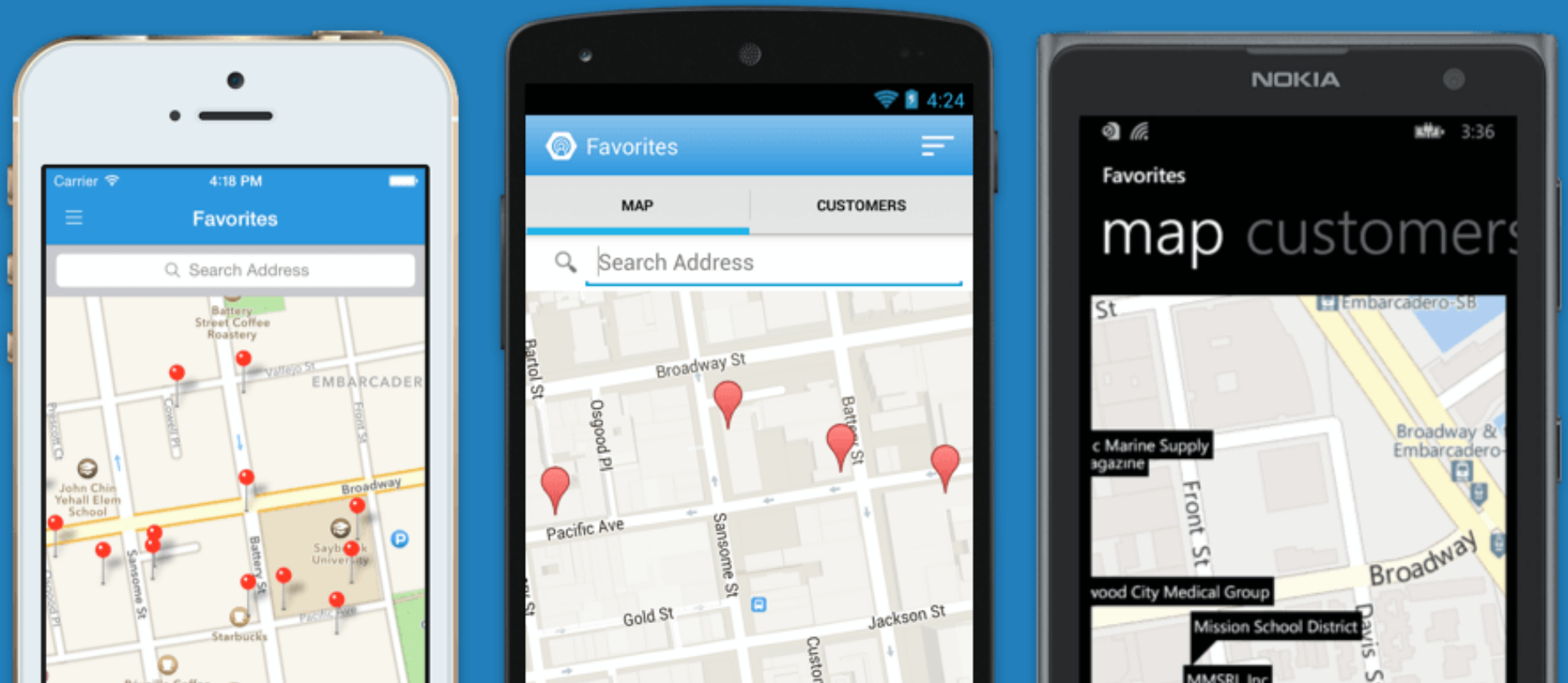
# Native Performance


Xamarin.iOS does full Ahead Of Time (AOT) compilation to produce an ARM binary for Apple's App Store.


Xamarin.Android takes advantage of Just In Time (JIT) compilation on the Android device.

# Meet Xamarin.Forms

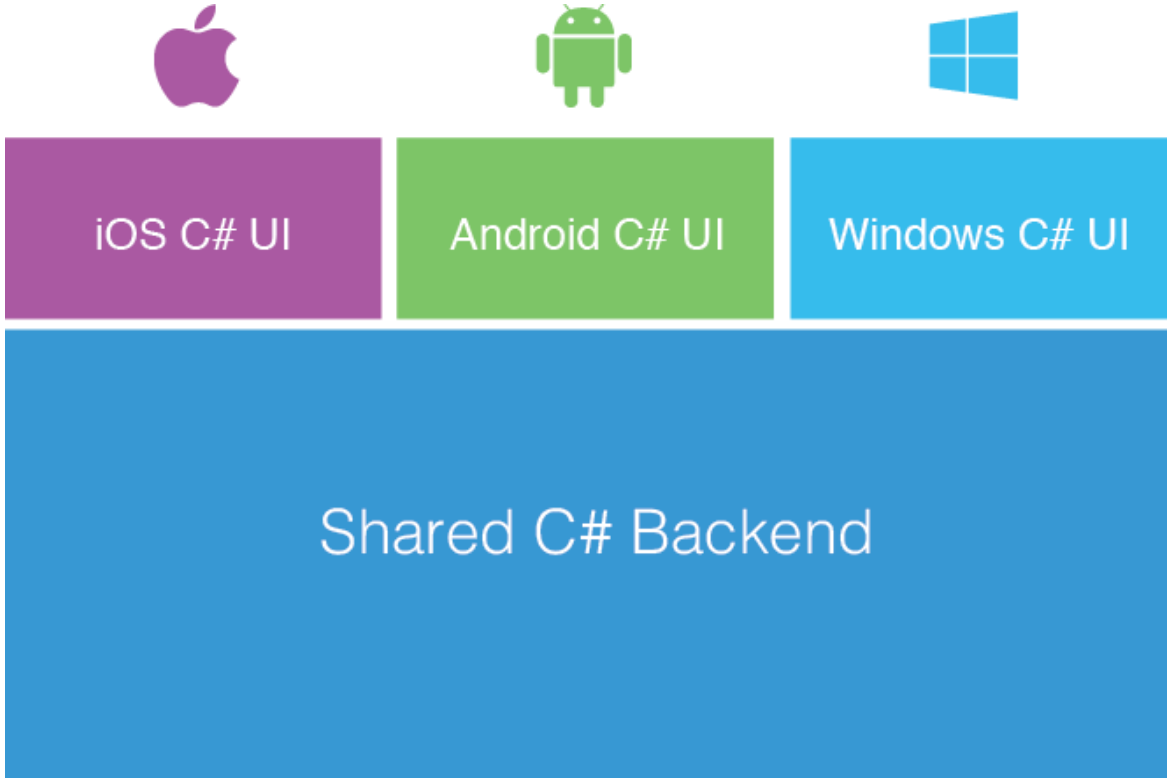
## Build native UIs for iOS, Android and Windows Phone from a single, shared C# codebase.



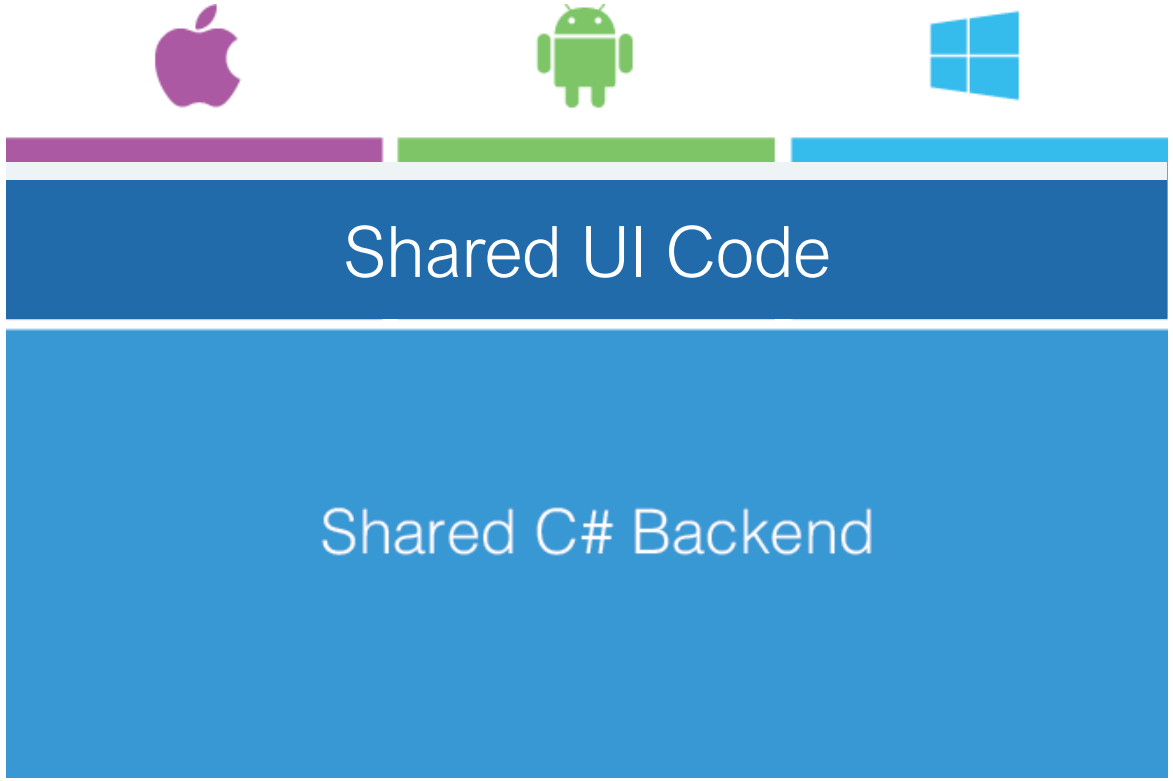
# Xamarin + Xamarin.Forms

Xamarin

Traditional Xamarin approach




With Xamarin.Forms:
more code-sharing, native controls



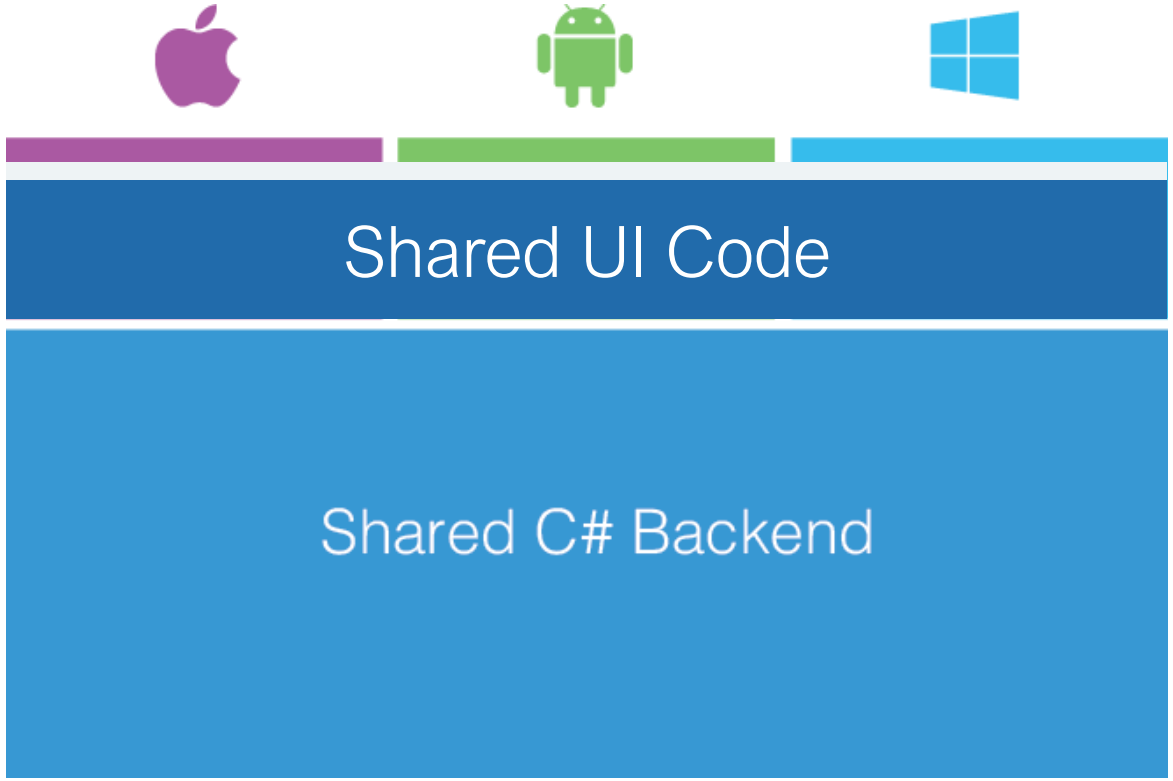
# Xamarin + Xamarin.Forms

Xamarin

Quickly and easily build native user interfaces using shared code

Xamarin.Forms elements map to native controls and behaviors

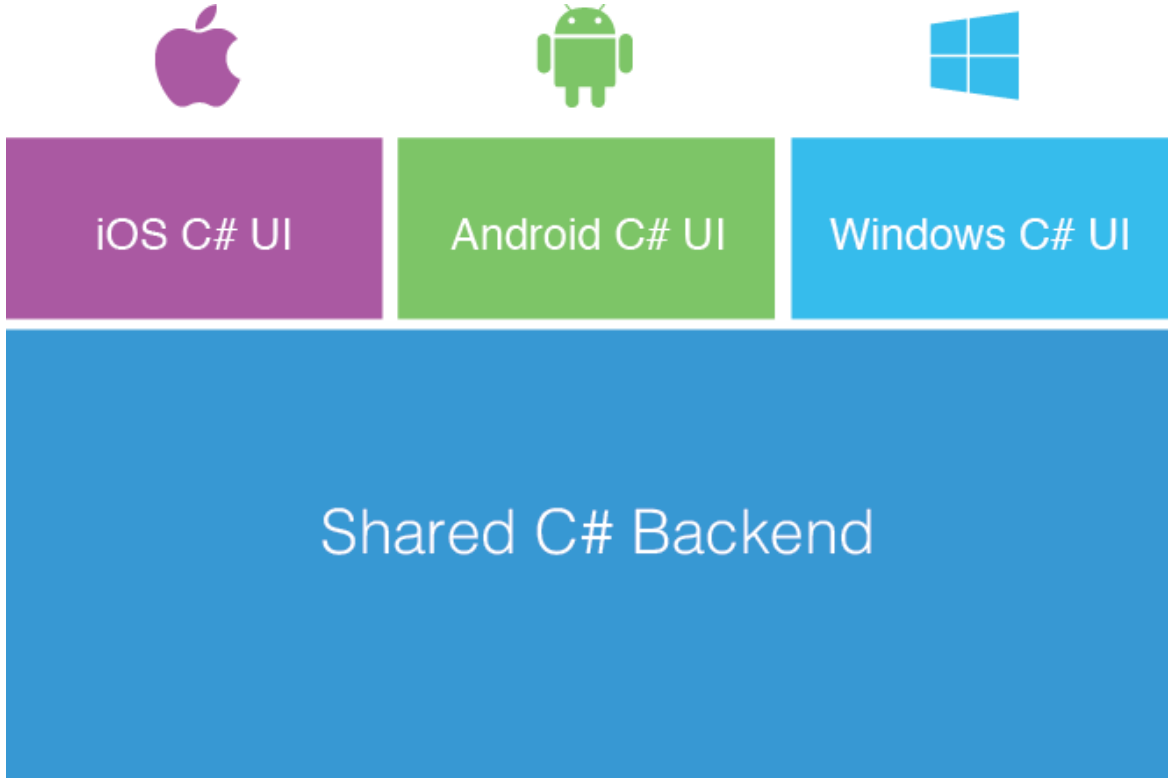
Mix-and-match Xamarin.Forms with native APIs



# What's Included

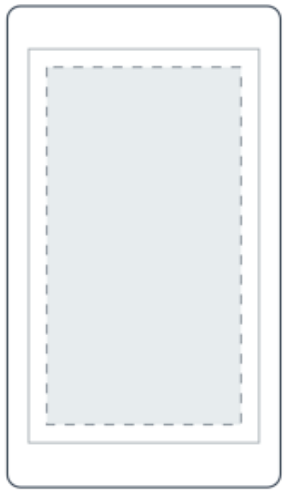
Xamarin

- 40+ Pages, Layouts, and Controls
  - Build from code behind or XAML
- Two-way Data Binding
- Navigation
- Animation API
- Dependency Service
- Messaging Center



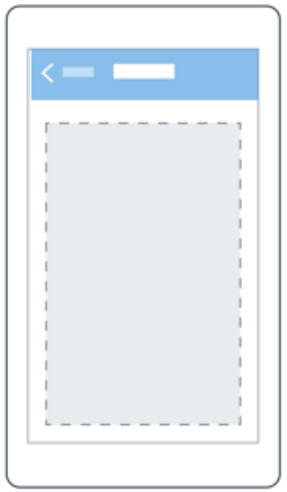
# Pages

Xamarin

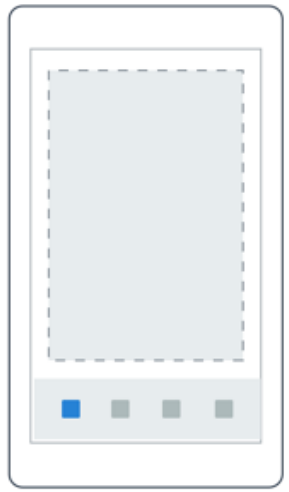


Content

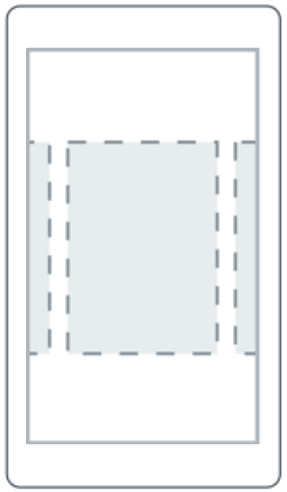
MasterDetail



Navigation



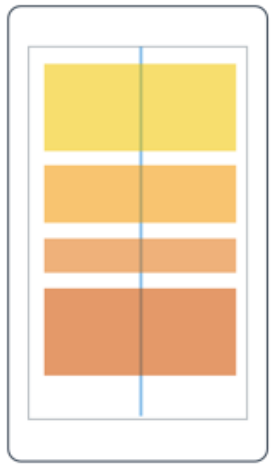
Tabbed



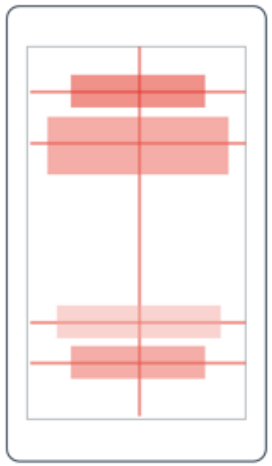
Carousel

# Layouts

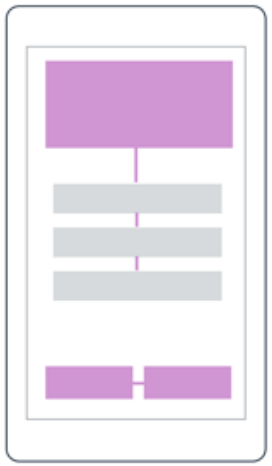
Xamarin



Stack

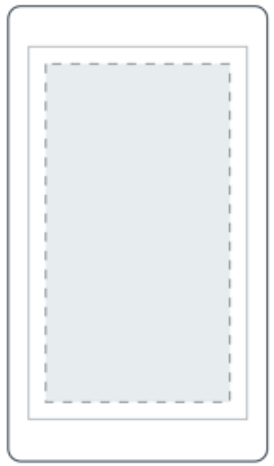


Absolute

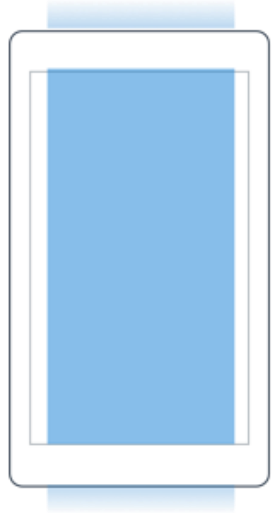


Relative

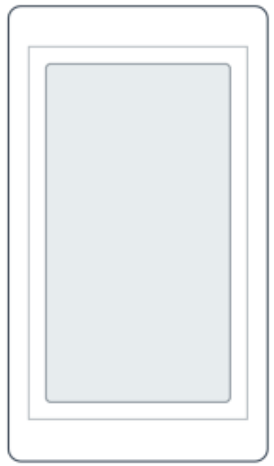
Grid



ContentView



ScrollView



Frame

# Controls

Xamarin

| | | | | |
|---|---|---|---|---|
| ActivityIndicator | BoxView | Button | DatePicker | Editor |
| Entry | Image | Label | ListView | Map |
| OpenGLView | Picker | ProgressBar | SearchBar | Slider |
| Stepper | TableView | TimePicker | WebView | EntryCell |
| ImageCell | SwitchCell | TextCell | ViewCell | |

# Model-View-ViewModel

Xamarin

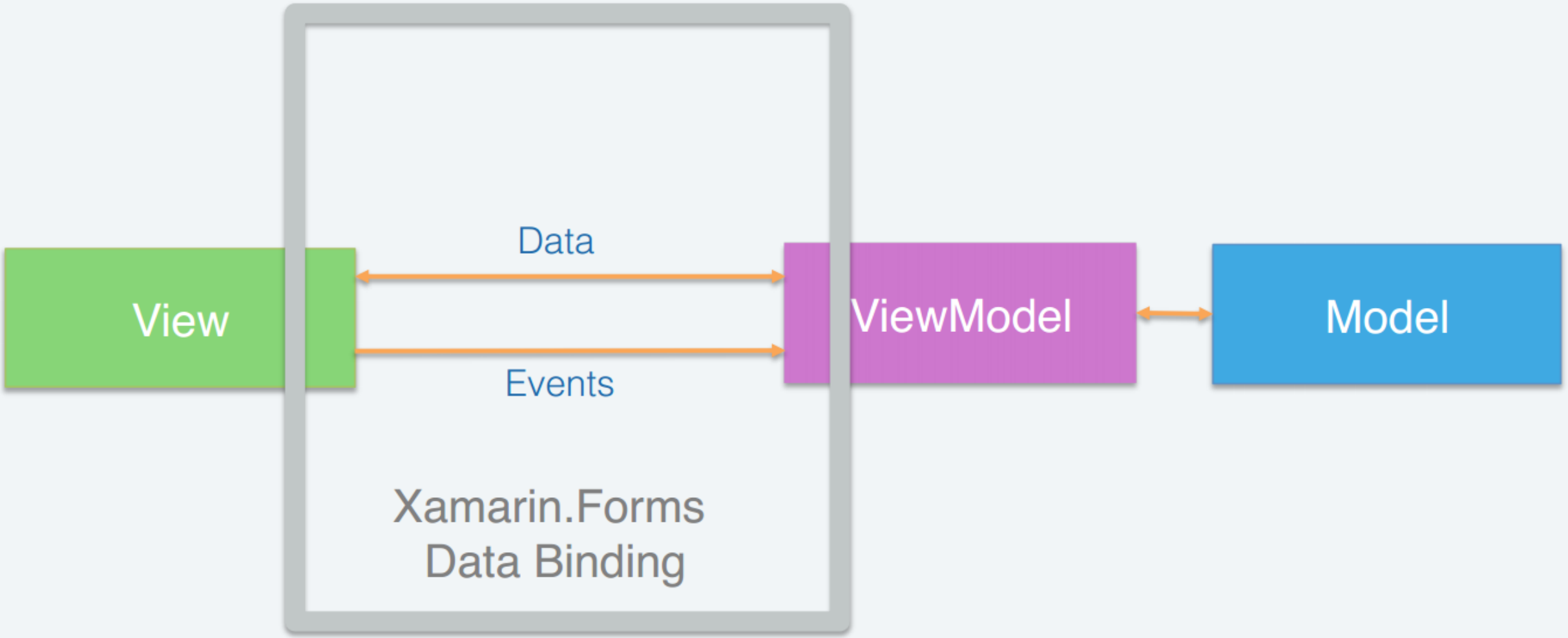

How to display information

What to display
Flow of interaction

Business Logic
Data objects

# Model-View-ViewModel

Xamarin



# DataBinding

- Xamarin support rich DataBindings mechanism.
- Support for INotifyPropertyChanged notifications.
- Declare Bindings in Code/XAML

```
var label = new Label() {VerticalOptions = LayoutOptions.Center};
          label.SetBinding(Label.TextProperty, new Binding("MyName"));
```

```
<Entry Placeholder="Please input your User Name"
       Text="{Binding UserName, Mode=TwoWay}"/>
```

# Commands

- Used to Execute a method when an action is performs, such as button click.
- Ability to pass parameter
- Ability to have CanExecute

```
public interface ICommand
{
    //
    // Methods
    //
    bool CanExecute (object parameter);

    void Execute (object parameter);

    //
    // Events
    //
    event EventHandler CanExecuteChanged
}
```

# Commands

- Command type is part of the Xamarin Forms framework (no need for the 3rd Party).

```
  private Command _remindMeCommand;

public Command RemindMeCommand
{
    get
    {
        return _remindMeCommand ?? (_remindMeCommand = new Command(
            () =>
            {
                UserName = "Alon Fliess";
            }));
    }
}
```

```
<Button Text="Remind Me…" Command="{Binding RemindMeCommand}"/>
```

# Animations

- Cross-platform animations
- Platform-specific animation APIs
- Async/Await API

# Login ViewModel

Xamarin

```
public class LoginViewModel : INotifyPropertyChanged
{
  private string username = string.Empty;
  public string Username
  {
    get { return username; }
    set { username = value; OnPropertyChanged ("Username"); }
  }

  private string password = string.Empty;
  public string Password
  {
    get { return password; }
    set { password = value; OnPropertyChanged ("Password"); }
  }

  public Command LoginCommand
  {
    get {
      return new Command (() => {
        //Log into Server here
      });
    }
  }
```

# Login Page – Code Behind

Xamarin

```
public class LoginPage : ContentPage
{
  public LoginPage()
  {
    //set binding context
    this.BindingContext = new LoginViewModel ();

    //create UI & bind to properties
    var username = new Entry { Placeholder = "Username" };
    username.SetBinding (Entry.TextProperty, "Username");

    var password = new Entry { Placeholder = "Password", IsPassword = true };
    password.SetBinding (Entry.TextProperty, "Password");

    var loginButton = new Button {
      Text = "Login",
      TextColor = Color.White,
      BackgroundColor = Color.FromHex("77D065")
    };

    loginButton.SetBinding (Button.CommandProperty, "LoginCommand");

    //set main content of page
    Content = new StackLayout{
      VerticalOptions = LayoutOptions.Center,
      Padding = 50, Spacing = 10,
      Children = { username, password, loginButton }
    };

  }
}
```

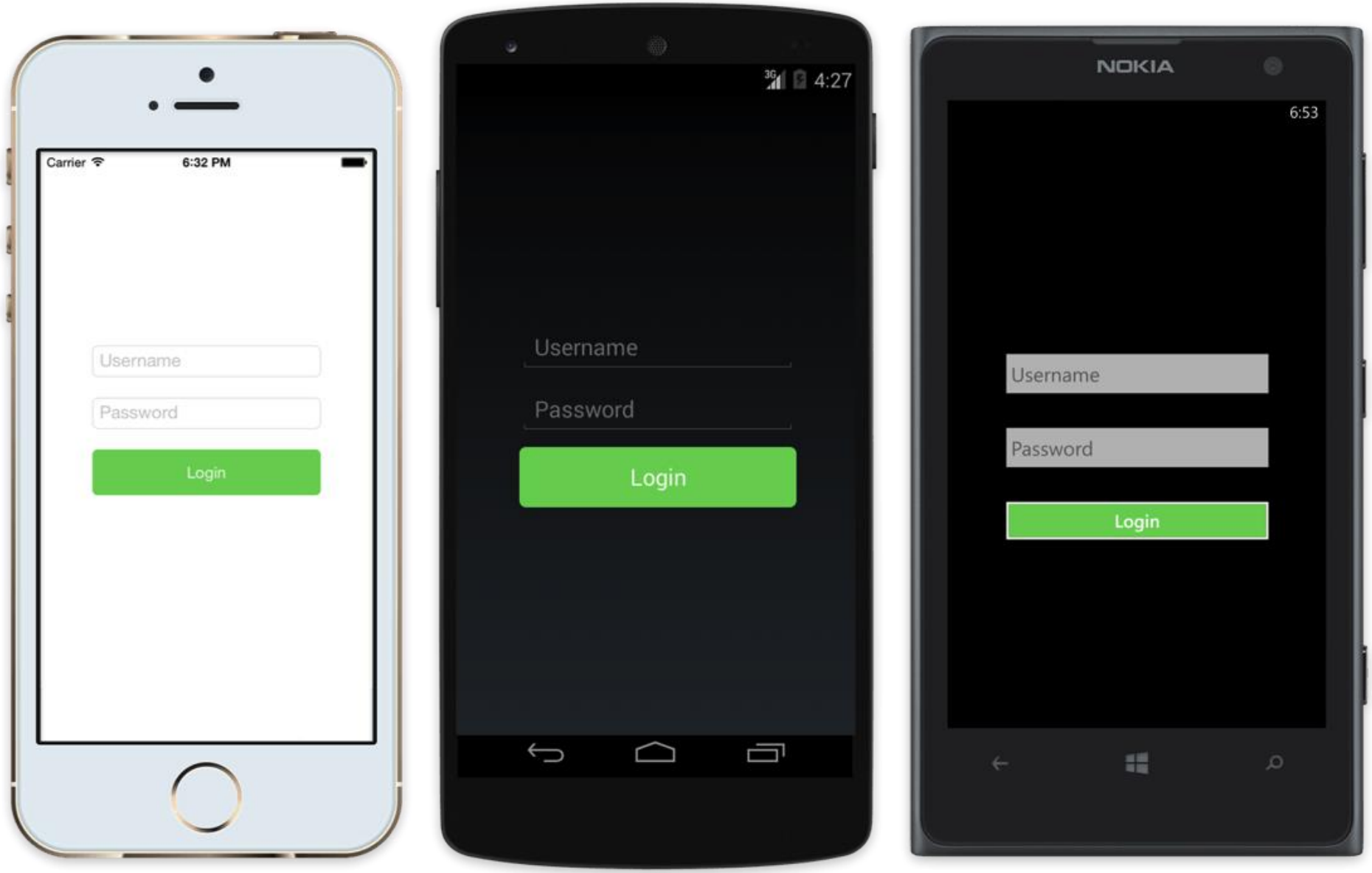
# Login Page – XAML

Xamarin

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="LoginExampleForms.LoginPageXAML">
    <ContentPage.Content>

    <StackLayout VerticalOptions="Center" Padding="50" Spacing="10">

        <Entry Placeholder="Username" Text="{Binding Username}"/>
        <Entry Placeholder="Password" Text="{Binding Password}"/>
        <Button Text="Login"
                TextColor="#FFFFFF"
                BackgroundColor="#77D065"
                Command="{Binding LoginCommand}"/>

    </StackLayout>

    </ContentPage.Content>
</ContentPage>
```

# Login Page

Xamarin



# Summary

- Quickly and easily build native user interfaces using shared code
- Xamarin.Forms elements map to native controls and behaviors
- Mix-and-match Xamarin.Forms with native APIs

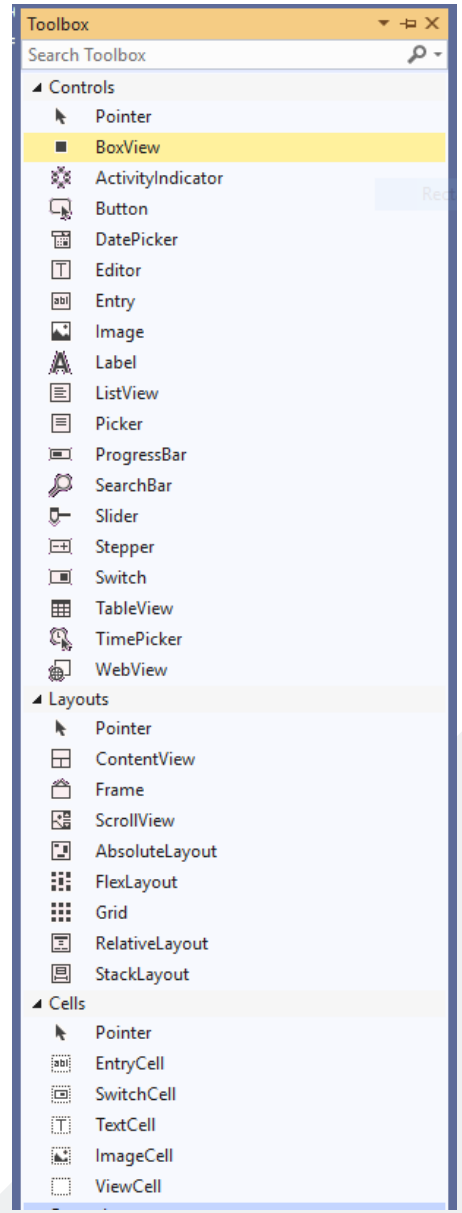
# XAML Fundamentals

# XAML Fundamentals

- Visual Studio
  - Toolbox
  - Properties View
  - Device Emulator
- What is XAML?
- Basic XAML
- Markup Extensions
- Naming Elements
- XAML Rules
- Summary

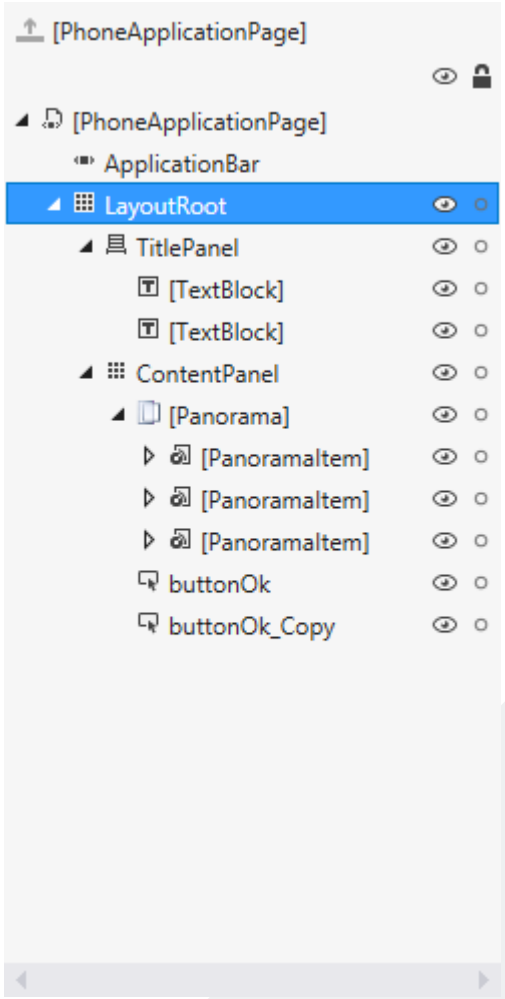
# Toolbox

- The toolbox groups UI controls available at design time
- 3rd party and custom controls are also available from the toolbox
- Simple drag and drop a control from the toolbox
- Controls can be searched within the search box and can be sorted
- Additional groups can be created



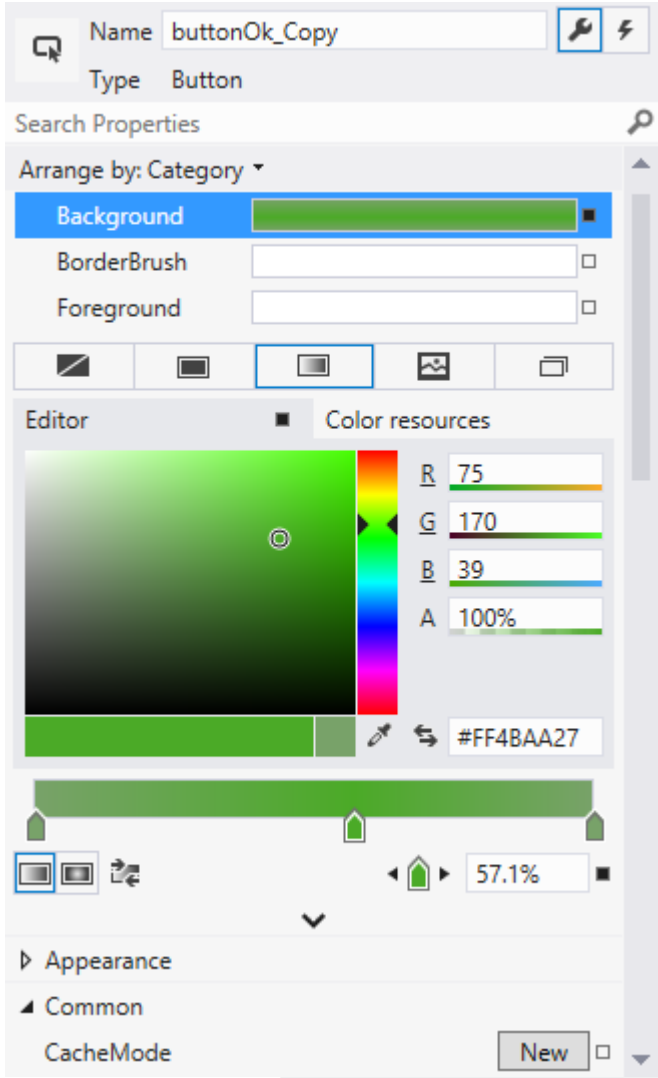
# Document Outline

- The document outline view displays the logical tree of UI elements, each element name or type and an icon of the element's type
- Each element in the hierarchy can be design-time locked and hidden
- The document outline is very useful to navigate between UI elements, especially in complex XAML files



# Properties View

- The properties view provides an easy a rapid way to:
  - Search for an element's property by its name
  - Set simple property value using plain text
  - Set complex property value using designers
  - Easily select color brushes, styles, font size, transformations and more
  - Register element's events
  - Arrange properties in groups
  - Create data bindings
  - Reset to default values



# What is XAML?

- XML based language
- Enable separation of UI and behavior (code)
- XAML allows
  - Creation of objects
  - Setting of properties
  - Connection to events
  - Custom behaviors
- XAML cannot call methods directly

# XAML vs. Code

- Anything that can be done in XAML can be done in code
  - But not vice versa
- XAML is usually shorter and more concise than the equivalent code
  - Thanks to type converters and markup extensions
- XAML should be used for initial UI
- Code will handle events and change items dynamically

# Simple XAML Example

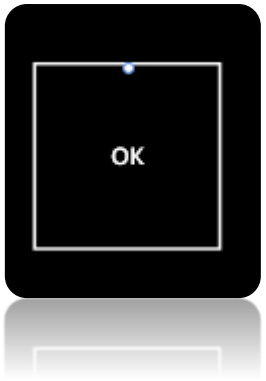
```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="OK" />
```

```
Windows.UI.Xaml.Controls.Button b = new
Windows.UI.Xaml.Controls.Button();
b.Content = "OK";
```

- Visual Studio UI designer generates XAML on each control picked from the toolbox
- XAML Can be visually viewed in the UI designer (Not in all platforms)

# XAML Example

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button x:Name="buttonOk"
            Width="200"
            Height="200"
            Content="OK"
            Click="buttonOk_Click" />
</Grid>
```



# Elements and Attributes

- Elements with type names only designate object creation (via the default constructor)
- Attributes indicate property or event values
  - Event values are event handlers (methods) names

# XAML Example

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Button x:Name="buttonOk"
            Width="200"
            Height="200"
            Content="OK"
            Click="buttonOk_Click" >
        <Button.Background>
            <LinearGradientBrush EndPoint="0.5,1“
                                 StartPoint="0.5,0">
                <GradientStop Color="#FFB2D9FF" Offset="0.004"/>
                <GradientStop Color="#FFB0D8FF" Offset="1"/>
                <GradientStop Color="#FF0A85FF" Offset="0.571"/>
            </LinearGradientBrush>
        </Button.Background>
    </Button>
</Grid>
```

# XAML And Code Behind

- A root element, usually **Page** or **UserControl** classes, can have code behind file
- The name of the code behind file is correlated to the XAML file name
- For example: MainPage.xaml and MainPage.xaml.cs
- The code behind full class name is specified from XAML using the **x:Class** directive

```
<Page
    x:Class="UWPDemo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UWPDemo"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Image x:Name="image" HorizontalAlignment="Left"  VerticalAlignment="Top" Stretch="Fill" Opacity="0.5"/>
        <Button x:Name="button" Content="Button" HorizontalAlignment="Left" " Height="61" Width="127"/>
    </Grid>
</Page>
```

# Child Elements

- Child elements (that are not property elements) can be one of
  - The **Content** property of the object
    - A property adorned with the attribute **Windows.UI.Xaml.Controls.ContentProperty**
  - Collection items
    - The object implements **`IList`** or **`IDictionary`**
  - A value that can be type-converted

# Content Property

- A single property that is designated with the **ContentProperty** attribute on the type
- Allows shortening the markup

```
<Button Content="OK" >
</Button>
```

```
<Button>
        OK
</Button>
```

```
<Button>
      <Button.Content>
         <Rectangle Fill="Blue"/>
      </Button.Content>
</Button>
```

```
<Button>
    <Rectangle Fill="Blue"/>
</Button>
```

# Collection Items

- List (IList)

```
<ListBox>
    <ListBox.Items>
        <ListBoxItem Content="Item 1"/>
        <ListBoxItem Content="Item 2"/>
    </ListBox.Items>
</ListBox>
```

- Dictionary (IDictionary)

```
<ResourceDictionary>
    <SolidColorBrush x:Key="br1" Color="Aqua" />
    <Rectangle x:Key="rc1" Fill="Brown" />
</ResourceDictionary>
```

# Summary of XAML Rules

- XML Element – create a new instance
- XML attribute – set a property or register an event
  - Type converter may execute
- **`ContentProperty`** attribute – no need to specify **`Type.Property`**
- Property of type **`IList`** or **`IDictionary`**
  - Add child elements (XAML calls appropriate **`Add`** method)
  - Need a **`x:Key`** in case of a dictionary

# Naming Elements

- Elements can be named using the **`x:Name`** XAML attribute
- The code-behind file will contain a field with that name
- Allows to access the element in the XAML as well:
  - For binding scenarios
  - For passing the whole element in some scenarios

# XAML Keywords

| Keyword | Valid on | Meaning |
|---|---|---|
| **x:Class** | Root element | The class that derives from the element type |
| **x:Key** | Element that its parent implements IDictionary | Key in a dictionary |
| **x:Name** | Element | The element's name, used for a field name for that element |

# Mapping custom types to XAML namespaces

- You can define your own custom types in C# and then reference your custom types in XAML markup
- To use XAML for custom types  - those that come from libraries other than the core libraries:
  - You must declare and map a XAML namespace with a prefix
  - Use that prefix in element usages to reference the types that were defined in your library
  - You declare prefix mappings as xmlns attributes
- For example:
  - the attribute syntax to map a prefix myTypes to the namespace myCompany.myTypes is
  - xmlns:myTypes="clr-namespace:myCompany.myTypes"
  - The representative element usage is: <myTypes:CustomButton/>

# XAML Markup Extensions

- Represent some kind of "shortcut" that enables a XAML file to access a value or behavior that isn't simply declaring elements based on backing types
- In XAML attribute syntax, curly braces "{" and "}" indicate a XAML markup extension usage
- A XAML parser calls code that provides behavior for that particular markup extension
  - That code provides an alternate object or behavior result that the XAML parser needs
- Examples:
  - {Binding} {StaticResource} {DynamicResource}

# Markup Extension Example

```
<Canvas.Resources>
    <Style TargetType="Border" x:Key="PageBackground">
        <Setter Property="BorderBrush" Value="Blue"/>
        <Setter Property="BorderThickness" Value="5"/>
    </Style>
    </Canvas.Resources>
        ...
    <Border Style="{StaticResource PageBackground}">
        ...
    </Border>
```

# XAML and .NET Events

- XAML has a syntax for attaching event handlers to objects in the markup
- You specify the name of the event as an attribute name on the object where the event is handled
  - For the attribute value, you specify the name of an event-handler function that you define in code
- The XAML processor uses this name to create a delegate representation in the loaded object tree, and adds the specified handler to an internal handler list

```
<Button Clicked="ShowUpdatesButton_Click">Show updates</Button>
```

# Summary

- XAML is mainly used to create a user interface
- It declaratively allows object creation, property and event assignment
- A code-behind file will usually contain the procedural logic
- Sharing with designers is easier
- Tools such as Expression Blend generate XAML that is immediately usable

# Basic Concepts

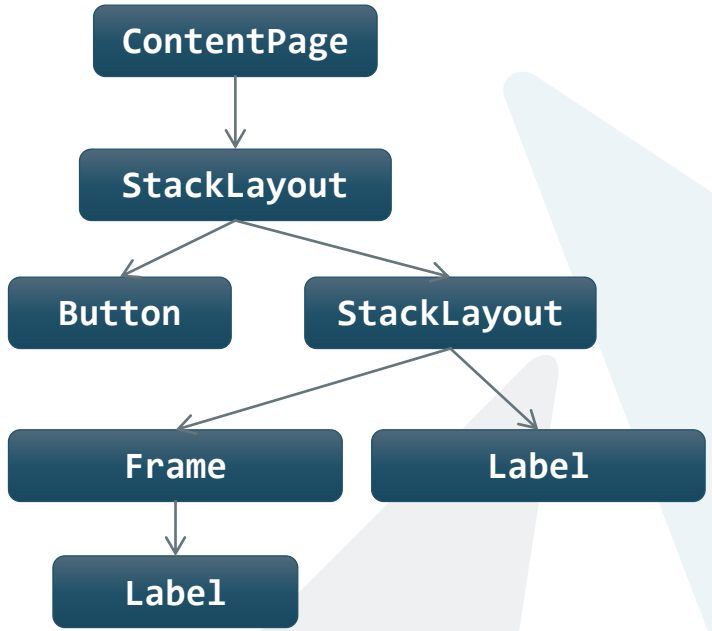
# Agenda

- Logical Tree
- Resources
- Layouts
- Summary

# Logical Tree

- A tree of elements/controls making up the user interface

```
<ContentPage x:Class="Demo.MainPage"
             xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml">
    <StackLayout>
        <Button Content="OK" Background="Red" Margin="6"/>
        <StackLayout Margin="6">
            <Frame BorderColor="Green">
                <Label Text="Hello" FontSize="20" />
            </Frame>
            <Label Text="World" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```



# Logical Resources

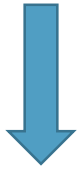
- Arbitrary named .NET objects, stored in the **`Resources`** collection property of an element
  - Typically for sharing the resource among child objects
- The **`VisualElement(Xamarin)`** type define a `Resources` property (of type **`ResourceDictionary`**)

# Creating and Using Resources

- Add a **`Resources`** property to some element
  - Usually a **`ContentPage`** or the **`Application`**
  - Any child element can reference those resources
- Add the objects with a **`x:Key`** attribute (must be unique in this resource dictionary)
- Use the **`StaticResource`** markup extension with the resource key name

# Resources Example

```
<StackLayout Orientation="Horizontal" HorizontalOptions="Center">
    <Button BackgroundColor="Green">BUTTON A</Button>
    <Button BackgroundColor="Green">BUTTON B</Button>
    <Button BackgroundColor="Green">BUTTON C</Button>
</StackLayout>
```



```
<StackLayout Orientation="Horizontal" HorizontalOptions="Center">
    <StackLayout.Resources>
        <ResourceDictionary>
            <Color x:Key="MainBackgroundColor">Green</Color>
        </ResourceDictionary>
    </StackLayout.Resources>
    <Button BackgroundColor="{StaticResource MainBackgroundColor}">BUTTON A</Button>
    <Button BackgroundColor="{StaticResource MainBackgroundColor}">BUTTON B</Button>
    <Button BackgroundColor="{StaticResource MainBackgroundColor}">BUTTON C</Button>
</StackLayout>
```

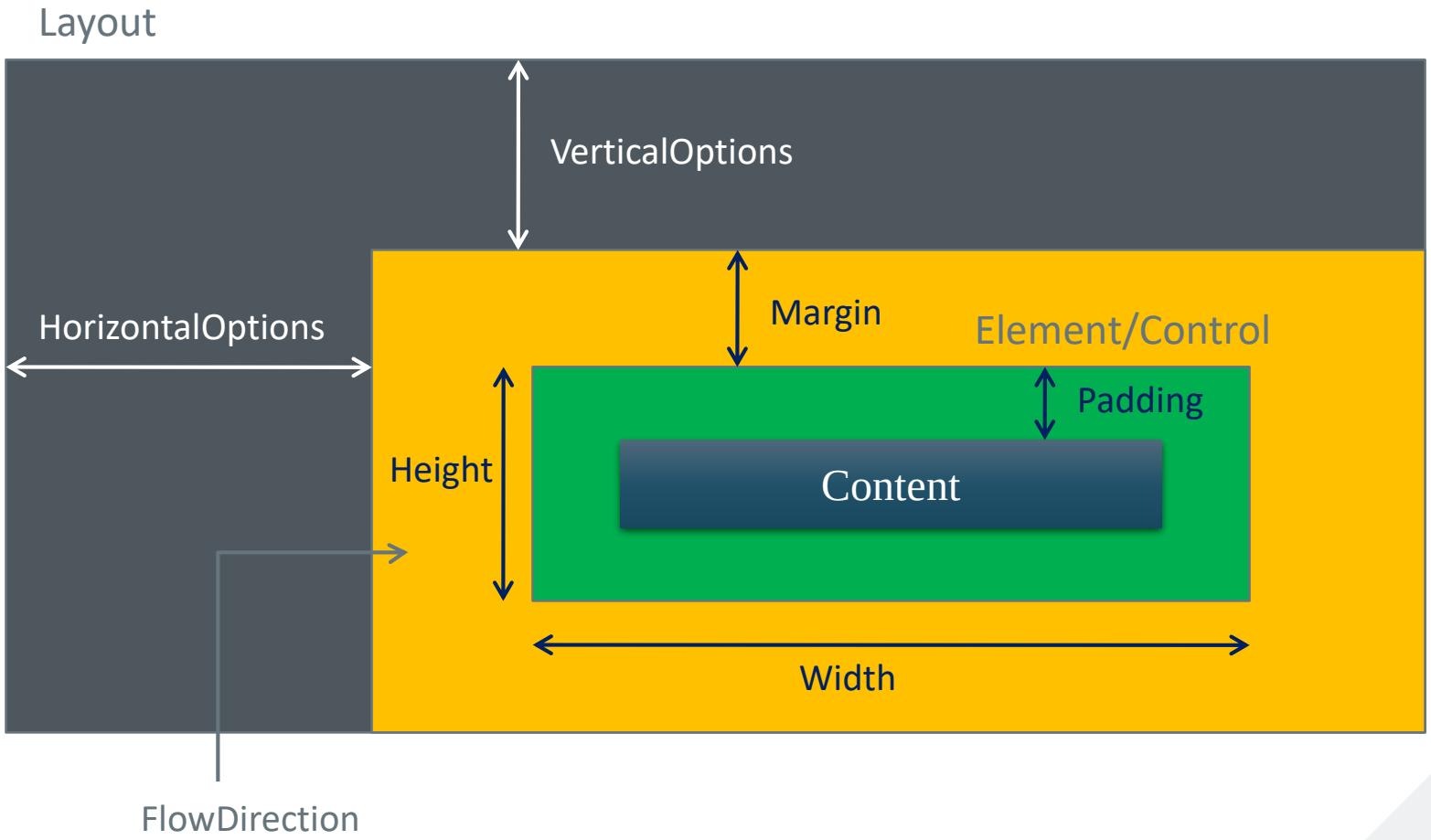
# Layout

- Layout is the arranging of user interface elements within some container
- Older technologies (e.g. Windows Forms) mostly used exact position and sizes
  - Limited in flexibility and adaptability
- XAML based technologies provide several layout panels that can control dynamically size and placement of elements
- Elements may advertise their size and position needs
- Two layout kind:
  - Static layout: Explicit pixel sizes and positions
    - Canvas (UWP)
    - AbsoluteLayout (Xamarin Forms)
  - Fluid layout: shrink, grow and reflow to adapt the visual space available

# Size and Position of Elements

- Element sizing and positioning is determined by the element itself and its logical parent
- A child element may request various settings
- The parent panel does not have to comply

# Element Layout Properties



# Element Size

- `WidthRequest` and `HeightRequest` properties
  - Request the exact size of the element
  - Default value is **`Double.NaN`**
    - Meaning: be as large as it needs to be
  - usually a bad idea to use these properties
    - Prevents smart resizing by panel
- `MinimumWidthRequest` and `MinimumHeightRequest` properties
  - Defaults are -1, allowing to ignore this values

# Margin and Padding

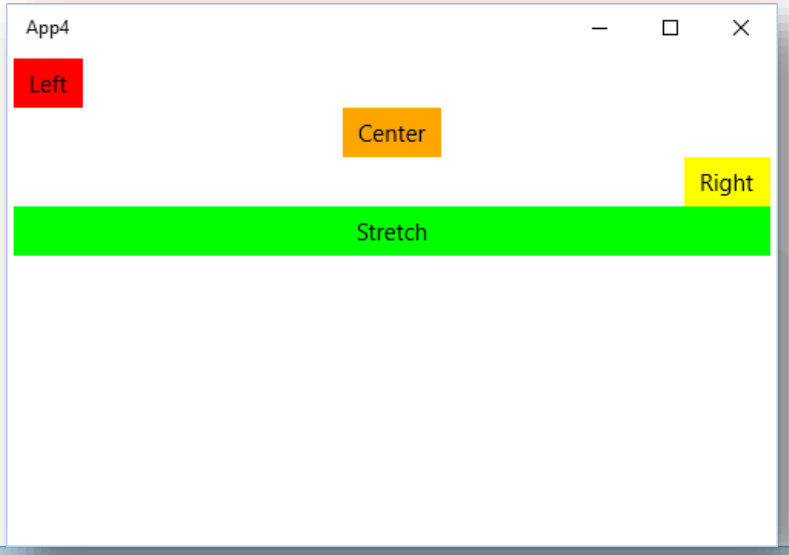
- Both of type **Thickness** (value type)
  - Maintains the properties **Left**, **Top**, **Right**, **Bottom** indicating distance from the corresponding edge
- **Margin**
  - The amount of space to add around the element
- **Padding**
  - The amount of space to add around the content of the control
- In XAML, can supply one, two or four numbers

# Visibility

- Visibility of elements is determined by the boolean **`IsVisibile`** property
- Can be changed during runtime
  - Via the code behind
  - Via binding to a property on the BindingContext

# Alignment

- Alignment indicates what should be done with any extra space given to an element
- **HorizontalOptions/VerticalOptions**
  - **Start, Center, End, Fill, StartAndExpand, CenterAndExpand, EndAndExpand, FillAndExpand**




```
<StackLayout Margin="4">
  <Button HorizontalOptions="Start" BackgroundColor="Red">Left</Button>
  <Button HorizontalOptions="Center" BackgroundColor="Orange">Center</Button>
  <Button HorizontalOptions="End" BackgroundColor="Yellow">Right</Button>
  <Button HorizontalOptions="Fill" BackgroundColor="Lime">Stretch</Button>
</StackLayout>
```

# Flow Direction

- The **`FlowDirection`** property indicates the flow of layout
  - **`LeftToRight`** (the default)
  - **`RightToLeft`**

# Layout Views

- Layout views derive from the abstract ***`Xamarin.Forms.Layout`*** class
- Maintain a **`Children`** property of type **`IReadOnlyList<Element>`**
- Each child element can be a Layout View as well
  - Allows creation of complex and adaptive user interfaces
- Xamarin Forms provides several built in panels
  - Custom layout views can be created as well

# StackLayout

- Stacks its elements in a vertical or horizontal “stack”
- **`Orientation`** property
  - **`Vertical`** (default) or **`Horizontal`**
- Alignment is ignored in the direction of stacking
- In the direction specified by the Orientation property, an element sizes to its content

# What is a View?

- Views are elements capable of receiving focus and handling input
  - You add a control to your app UI.
  - You set properties on the control, such as width, height or color
  - You hook up some code to the control so that it does something
- Many controls are available “out of the box”
- Custom controls can be created
  - User controls that wrap one or more controls and expose higher level properties
  - Custom controls that derive from an existing control and extend its functionality

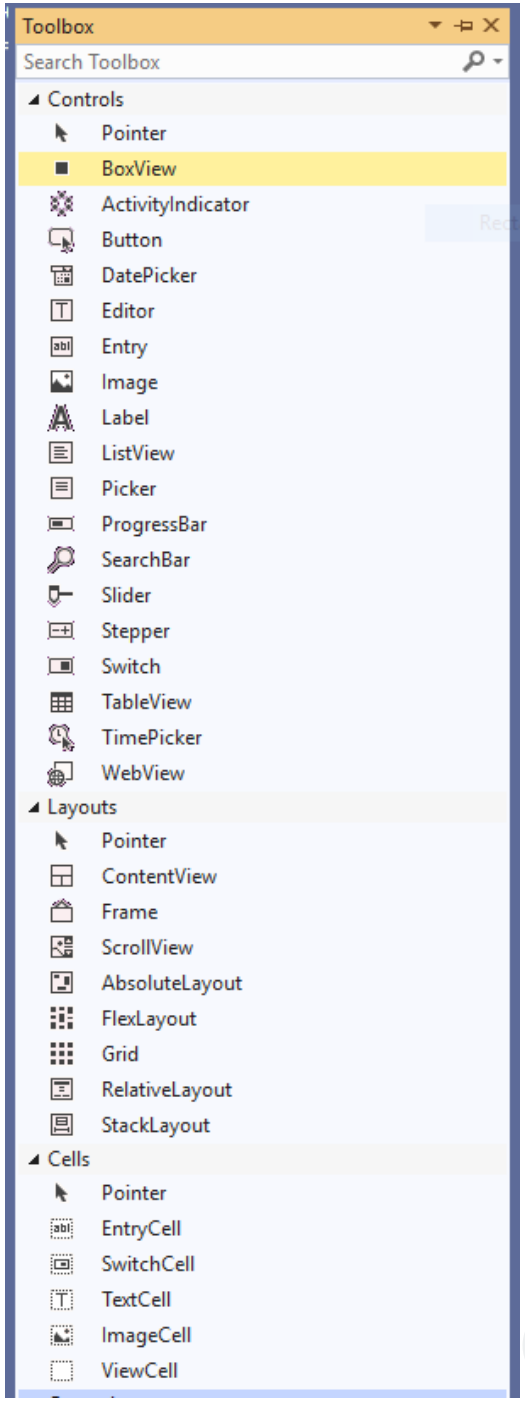
# Example: Static Text

- The **`Label`** View
  - The **`Text`** property
  - Font related properties
    - **`FontSize`**, **`FontFamily`**, etc.
  - **`TextAlignment`**, **`TextDecorations`**

```
<Label FontSize="16" Margin="4" Text="Hello World“/>
```

# Other Controls

- Here you can find the full control list



# Data Binding

# What is Data Binding?

- A way for your app's UI to display data, and optionally to stay in sync with that data
- Data binding allows you to separate the concern of data from the concern of UI, for better:
  - Readability, Testability, and Maintainability
- Data binding means tying two arbitrary objects
- Typical scenario is a non-visual object (or collection) to a visual element
  - Any changes to the non-visual object are reflected in the visual element (and optionally vice versa)

# Data Binding Concepts

- Source
  - The data object to bind to
- Property Path
  - The property on the source object to use
  - May be a nested property, an array element or an indexer
- Binding Mode
  - Typically one way or two way (target update source)

# Using Data Binding

- Typically done in XAML using the `{Binding}` markup extension
  - The Binding class is the workhorse behind the scenes
  - Set on the target property

# Binding Direction

- The **`Binding`** object allows specifying how the target / source properties are updated
- **`Mode`** property (of type enum **`BindingMode`**)

| Binding Mode | Meaning |
|---|---|
| `OneWay` | The target property is updated by source property changes |
| `TwoWay` | OneWay + the source property is updated by changes of the target property |
| `OneTime` | Target is updated by the source the first time they are bound |

# Binding to Objects

- **`Source`**
  - Reference to the source object
- **`BindingContext`**
  - Used by default if **`Source`** is not specified
  - Searches up the element tree if not found on target element

# Change Notifications

- An object must notify when one of its properties changes
  - By defining the property as a bindable property
  - Or by implementing the **`INotifyPropertyChanged`** interface
    - Raise the **`PropertyChanged`** event

# The `BindingContext`

- Sometimes many elements bind to the same object
  - Perhaps with different properties
- The object may be specified as the **`BindingContext`** property on any common parent element
- Whenever the **`Source`** property is not specified in the **`Binding`**, a binding context object is searched up the element hierarchy
  - If found, becomes the binding source object
- Can be used programmatically without the need to create the source object in XAML

# Bind Example

```
<Label Text="{Binding FirstName}"/>
```

```
private string _firstName;
public string FirstName
{
    get => _firstName;
    set
    {
        _firstName = value;
        OnPropertyChanged();
    }
}

private void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

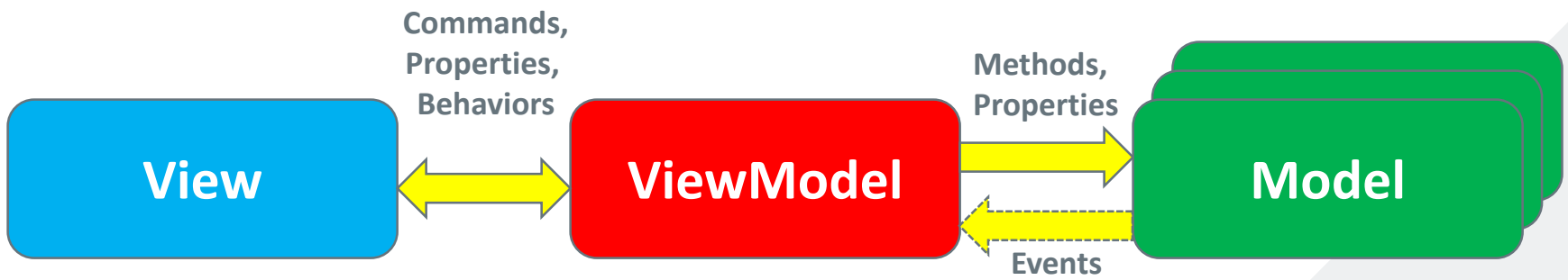
# Commands and MVVM

# Agenda

- Introduction to the MVVM Pattern
- Commands
- Implementing Commands for MVVM
- Summary

# The MVVM Pattern

- Model – View – ViewModel
- Based on similar principles of Model View Controller (MVC) and Model View Presenter (MVP)
- Natural pattern for XAML based applications
  - Data binding is key
- Enables developer-designer workflow
- Increases application testability



# MVVM Participants

- Model
  - Business logic and data
  - May implement change notification for properties and collections
- View
  - Data display and user interactivity
  - Implemented as a **`ContentPage`**, **`ContentView`** or custom control
  - Has little or ideally no code behind at all
- ViewModel
  - UI logic and data for the View
  - Abstracts the Model for View usage
  - Exposes commands (**`ICommand`**) to be used by the View
  - Implements change notifications
  - Maintains state for the View (communicates via data binding)

# The View

- Provides the user interface and interaction
- The `BindingContext` property points to the View Model
- Updated using property changes from the ViewModel
- Binds to commands (on `internal interface IButtonElement` elements) provided by the ViewModel

# The ViewModel

- Exposes properties the View binds to
- Can be an adapter if some functionality missing from Model classes
- Exposes commands to be invoked by the view
- Maintains state for the View
- Implements change notifications (`INotifyPropertyChanged`)
  - Uses `ObservableCollection<T>` that already implements **`INotifyCollectionChanged`**

# The Model

- Responsible for business logic and data, e.g.
  - Data Transfer Objects (DTO)
  - POCOs (Plain Old CLR Objects)
  - Generated entity objects
  - Generated proxy objects
- May provide change notifications
- Provides validation if appropriate

# Introduction to Commands

- Handling events and executing some code is fine for simple applications
- Sometimes the same code needs to execute from unrelated events (e.g. tap,menu item, toolbar)
- Maintaining UI state (e.g. enabled/disabled) becomes difficult
- Higher level functionality, such as an undo / redo system is not possible
- Solution: use commands (the “Command” design pattern) with some support from Xamarin Forms

# The Command

- A command is an object implementing the `System.Windows.Input.ICommand` interface

```
public interface ICommand {
    event EventHandler CanExecuteChanged;

    bool CanExecute(object parameter);
    void Execute(object parameter);
}
```

# Commands for MVVM

- Xamarin Forms provides a basic `ICommand` implementation that uses a delegate called simply **Command**
- Other implementations possible
  - E.g. the **`CompositeCommand`** class from PRISM framework that holds a list of commands
- Using commands in MVVM
  - Some controls expose a **`Command`** and **`CommandParameter`** properties that can be bound to a command exposed by the ViewModel

# Wiring the View and the View Model

- The View's **`BindingContext`** must be set to its supporting ViewModel
- Some options
  - The View can create an instance of the right VM (even in XAML)
  - The ViewModel can be injected using some dependency injection technique (e.g. Unity or MEF)
  - Use some global ViewModel locator object
  - A Main VM can be set explicitly on the main View, and other VMs can be exposed as properties, which will be bound by child views

# Summary

- Commands allow high level segregation of tasks
- The MVVM pattern is common in Xamarin Forms to separate logic from UI and increase testability