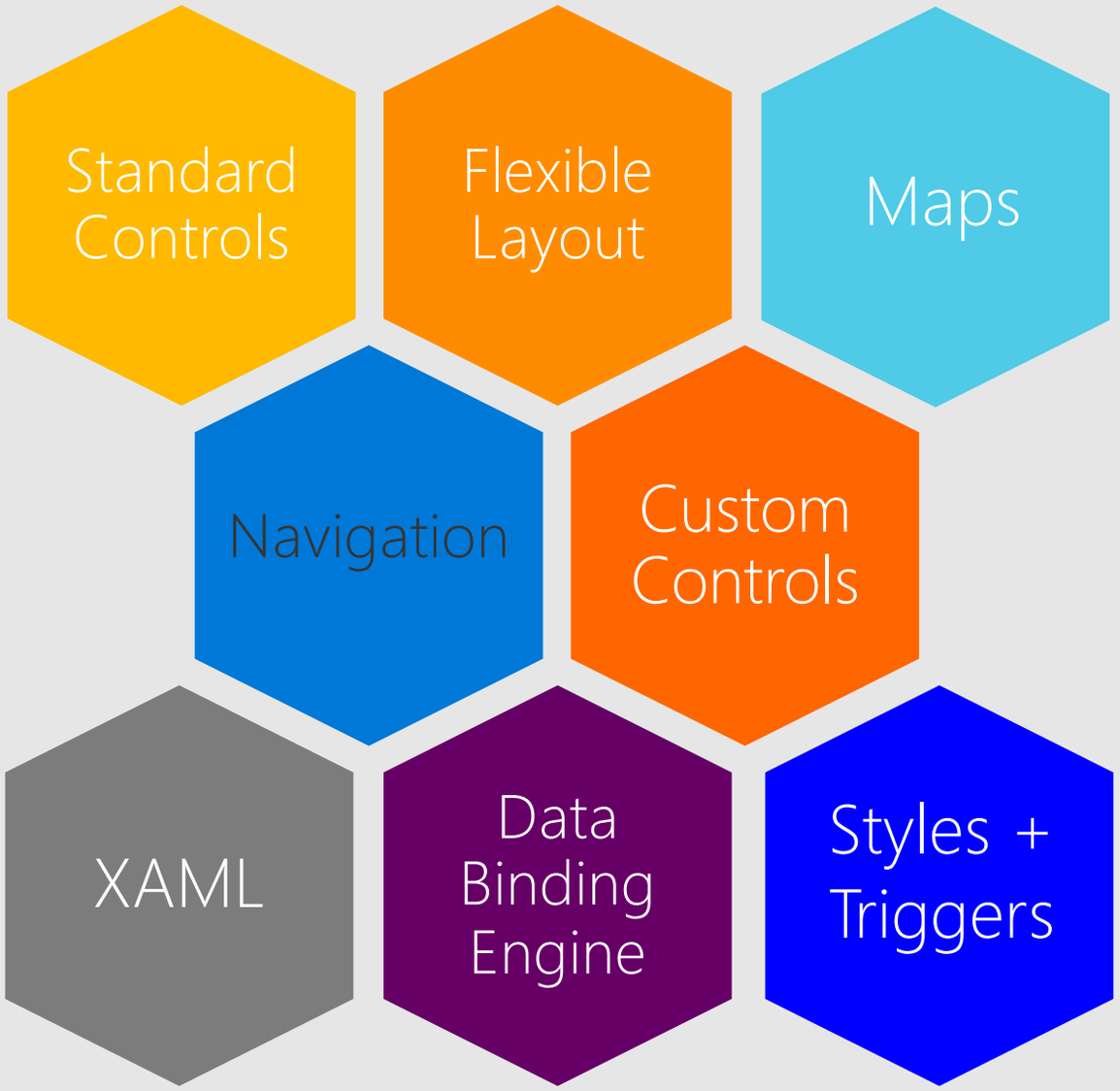
Introduction to Xamarin.Forms

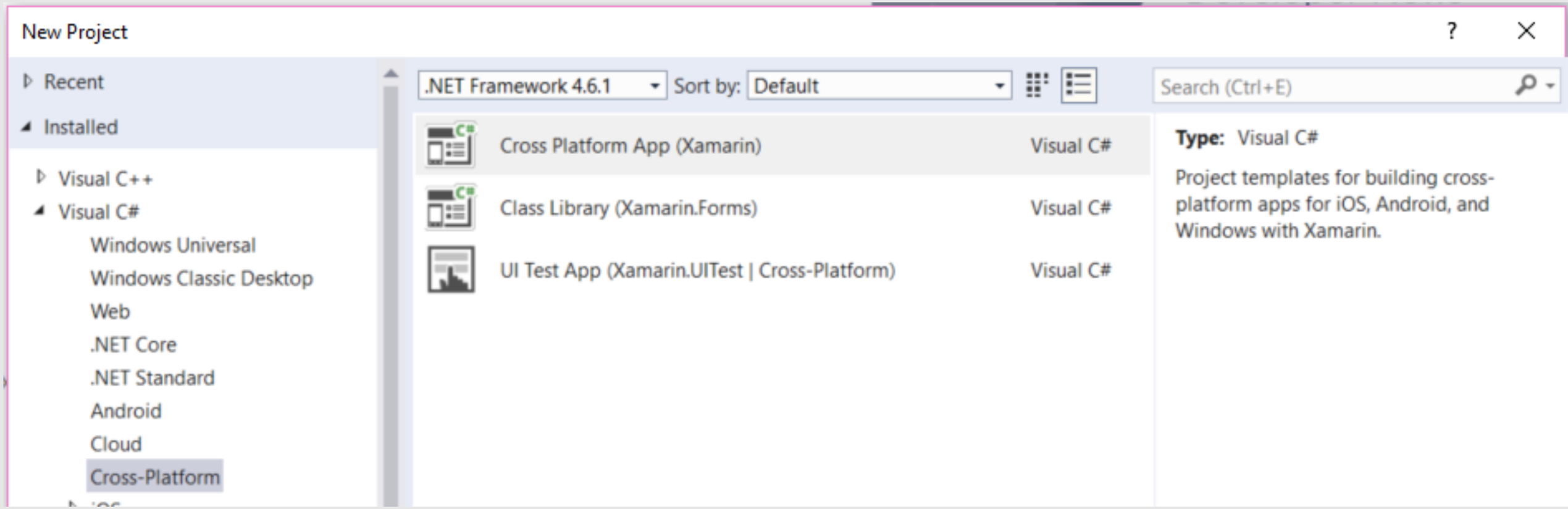
Microsoft

What is Xamarin.Forms?

- Xamarin.Forms is a cross-platform UI framework to create mobile apps for:
  - Android
  - iOS
  - Windows 10 (UWP)



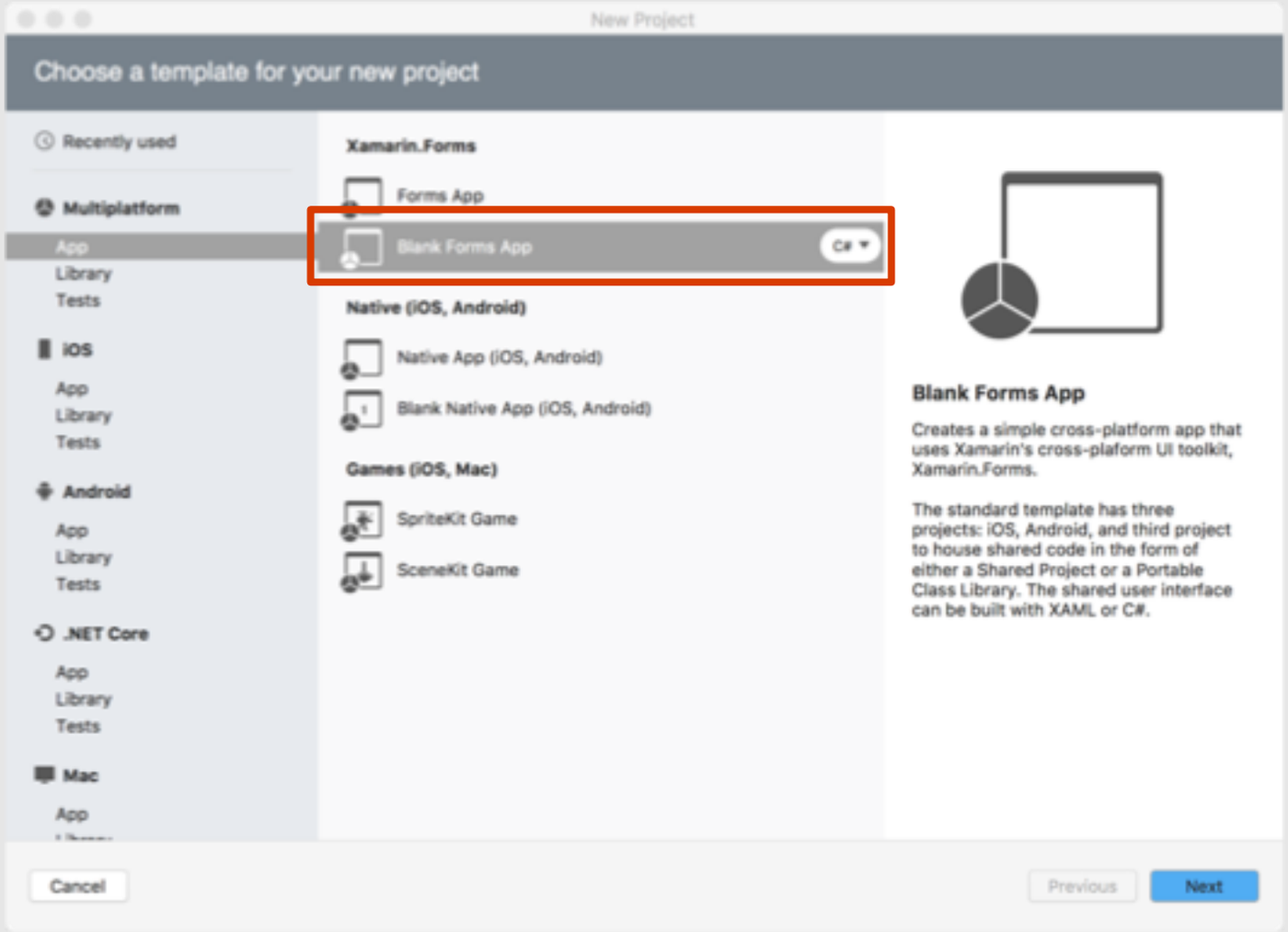
Creating a Xamarin.Forms App [VS]



- Built-in project templates for Xamarin.Forms applications available under **Cross-Platform**

Creating a Xamarin.Forms App [VS Mac]

- Project wizard walks through available options



Available project types

There are **two project styles** available for sharing code – which one you select has an impact on *how* and *what kind* of code is shared

Shared Project (SAP)

Portable Class Library (PCL)

Which one should I use?

Shared Projects

PROS	CONS
All APIs available	Can lead to spaghetti code
Platform-specific logic can be added directly	Harder to unit test when conditional code is used
All file types can be shared	Must be shipped in source form

Portable Class Libraries

PROS	CONS
Enforces architectural design	Limited APIs available
Can be unit tested separately as a binary	Difficult to share non-code files
Can be shipped in binary form (NuGet)	Requires more work to integrate platform-specific code

Demonstration

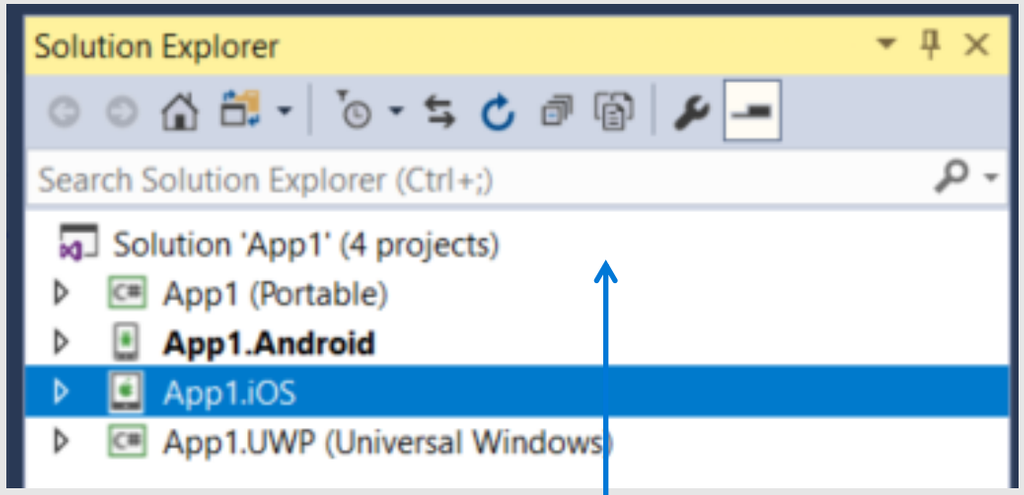
Creating a new Xamarin.Forms application

Xamarin University

Project Structure

Blank App project template creates several related projects

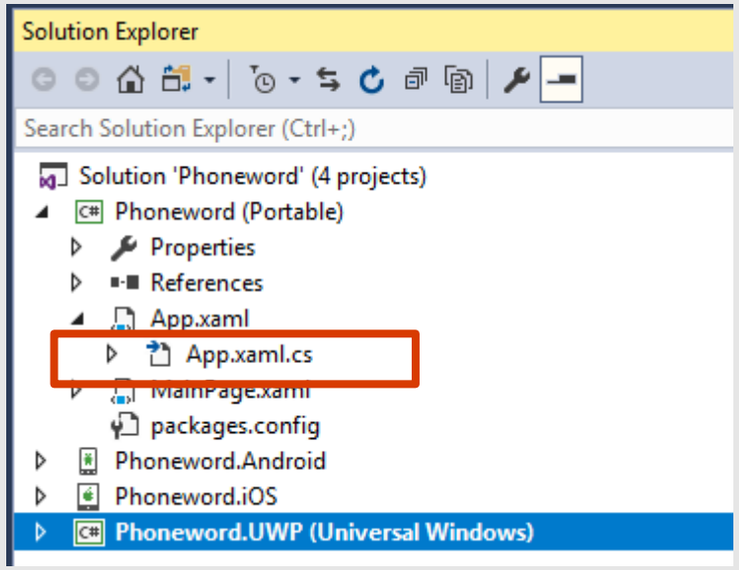
Platform-specific projects act as "host" to create native application



PCL or SAP used to hold shared code that defines UI and logic

Project Structure - PCL

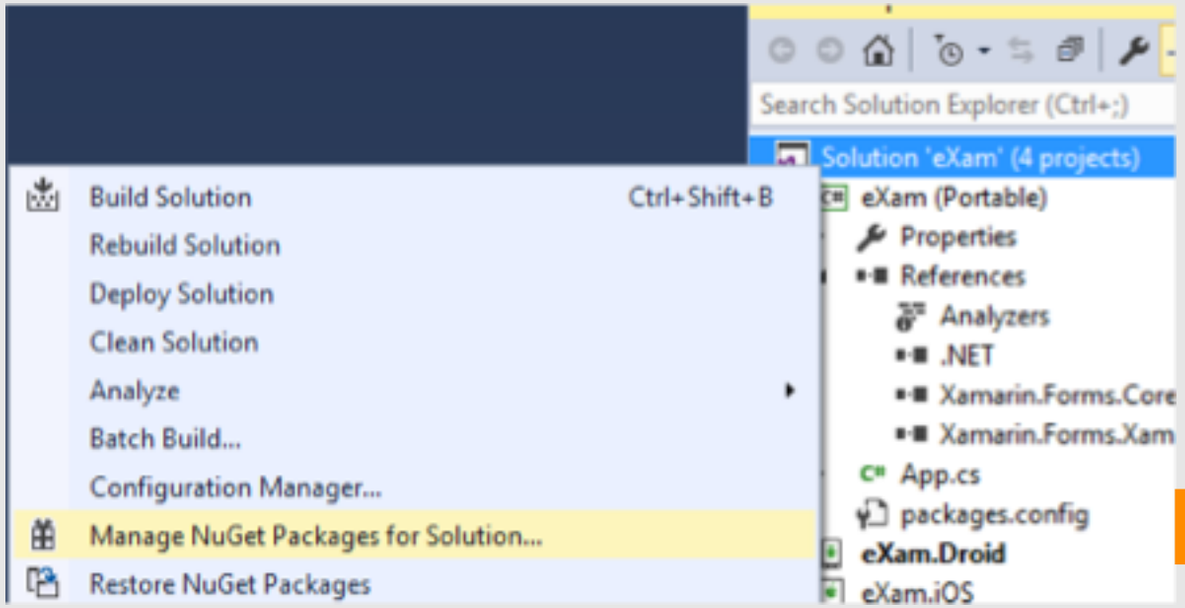
Most of your code will go into the **PCL** used for shared logic + UI

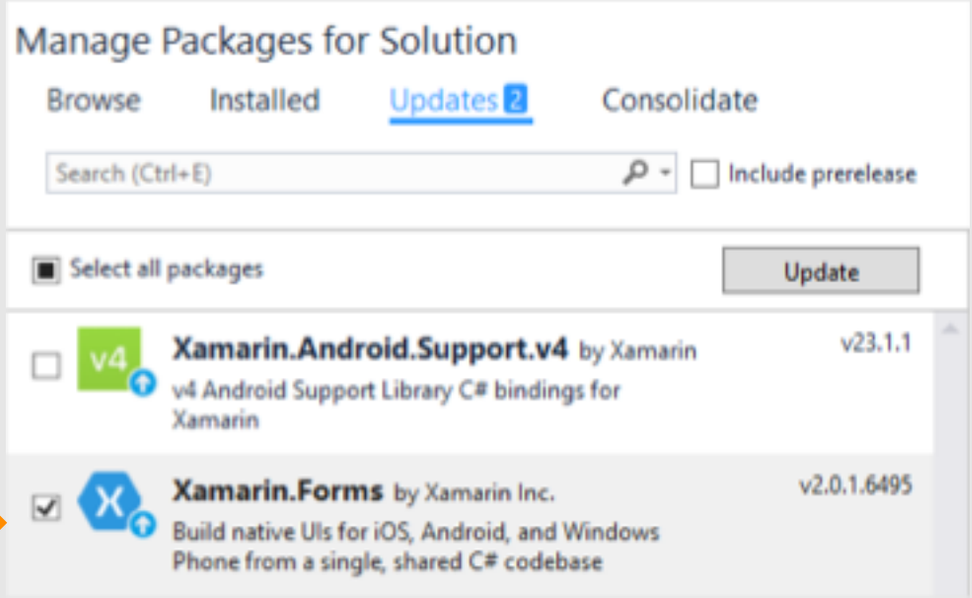


Default template creates a single `App.xaml.cs` file which decides the initial screen for the application

Xamarin.Forms updates

Should update Xamarin.Forms **Nuget package** when starting a new project





Exercise #1

- Create the Xamarin.Forms project

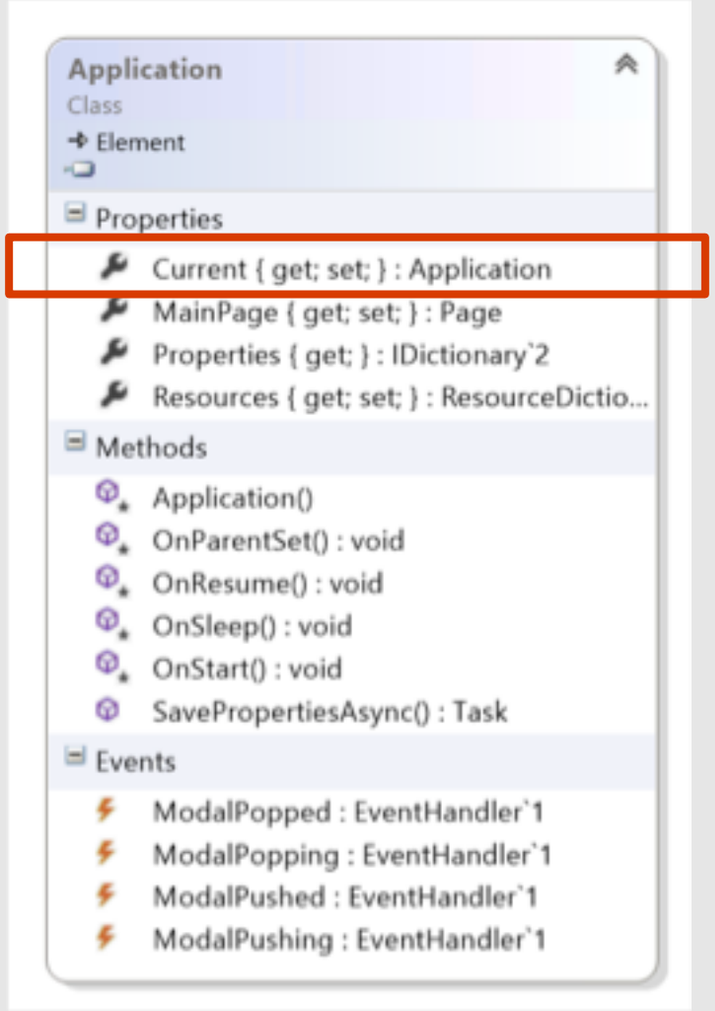
Xamarin.Forms app anatomy

Xamarin.Forms applications have two required components which are provided by the template

Application class

- **`Application`** class provides a *singleton* which manages:
  - Lifecycle methods
  - Modal navigation notifications
  - Currently displayed page
  - Application state persistence

- New projects will have a derived implementation named **`App`**

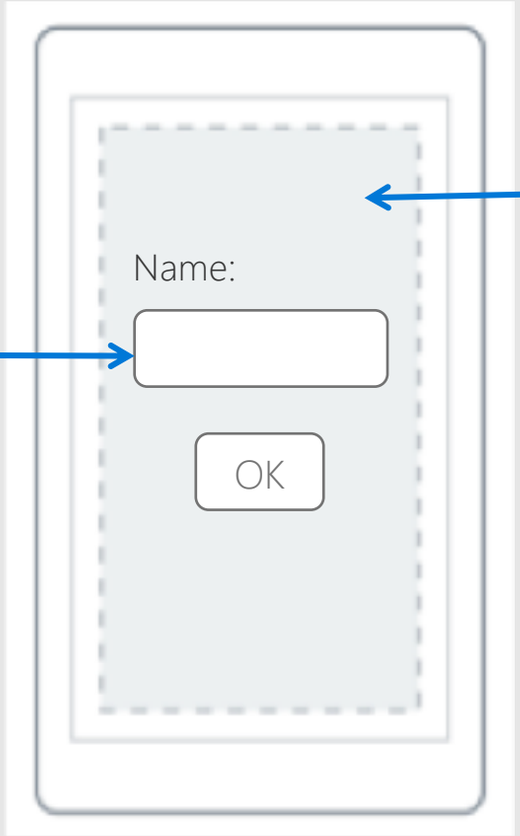


Note: Windows apps *also* have an **`Application`** class, make sure not to confuse them!

Creating the application UI

Application UI is defined in terms of *pages* and *views*

Views are the UI controls the user interacts with

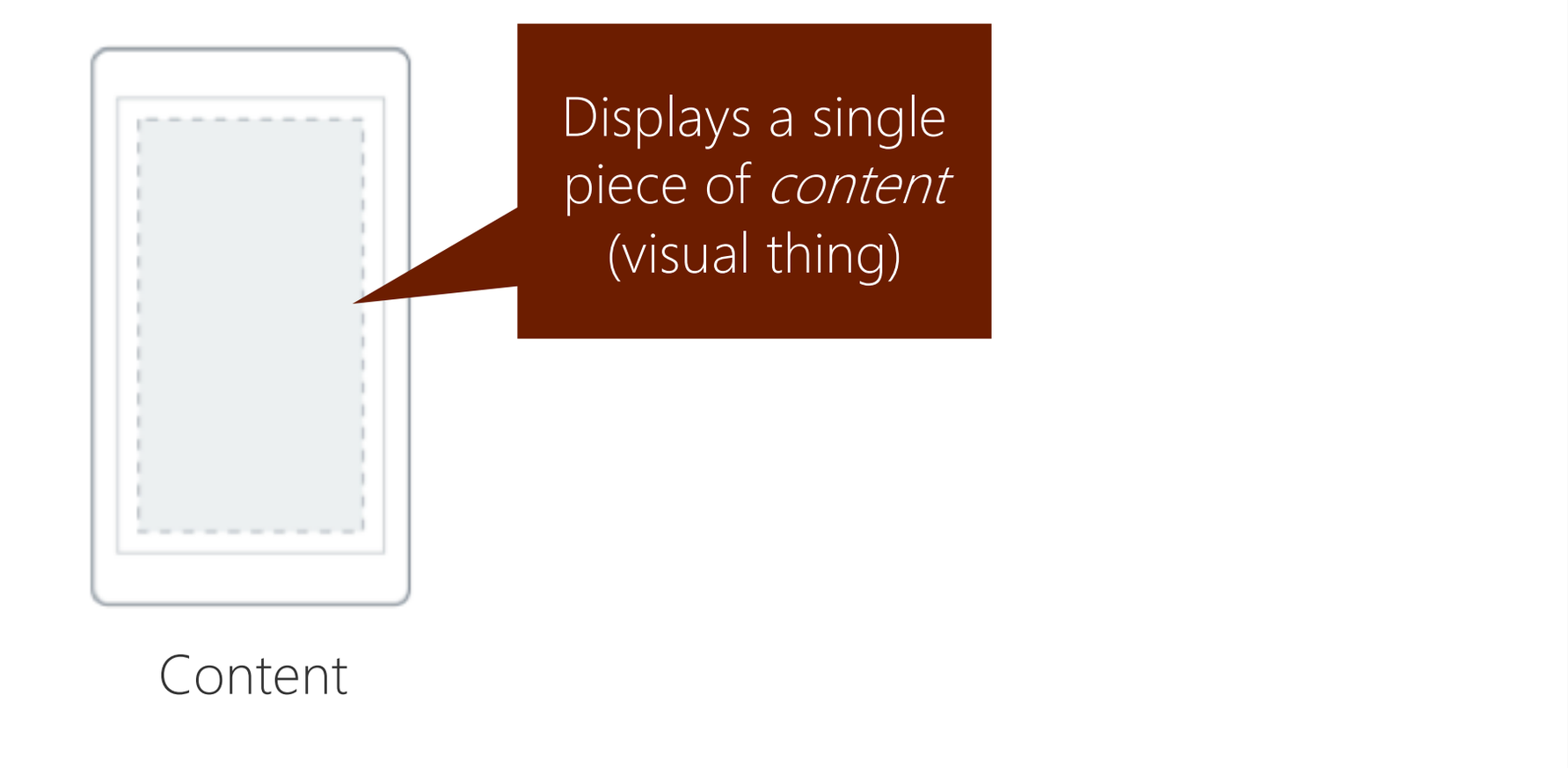


Page represents a single screen displayed in the app

Pages

Page is an abstract class used to define a single screen of content

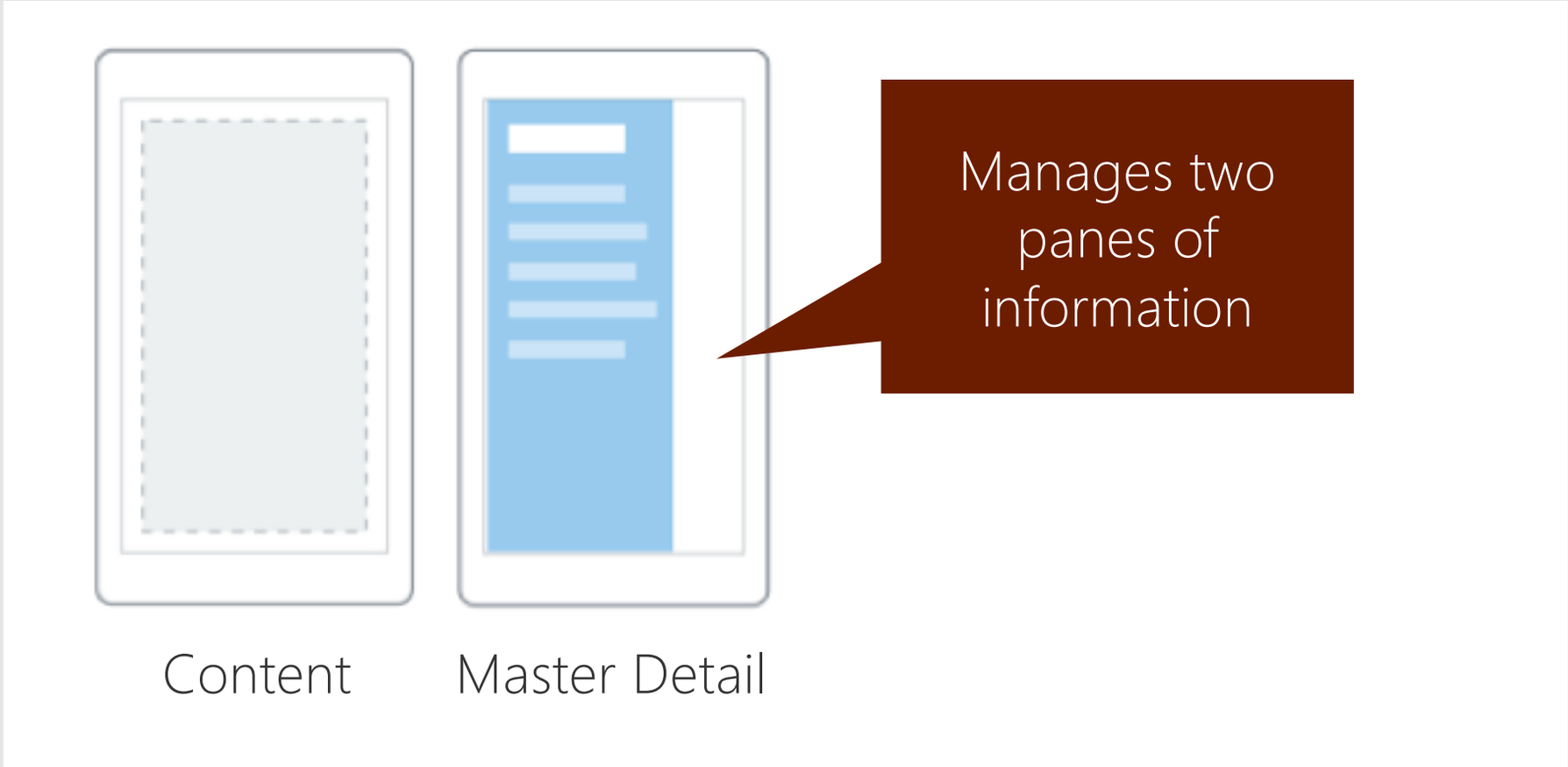
- derived types provide specific visualization / behavior



Pages

Page is an abstract class used to define a single screen of content

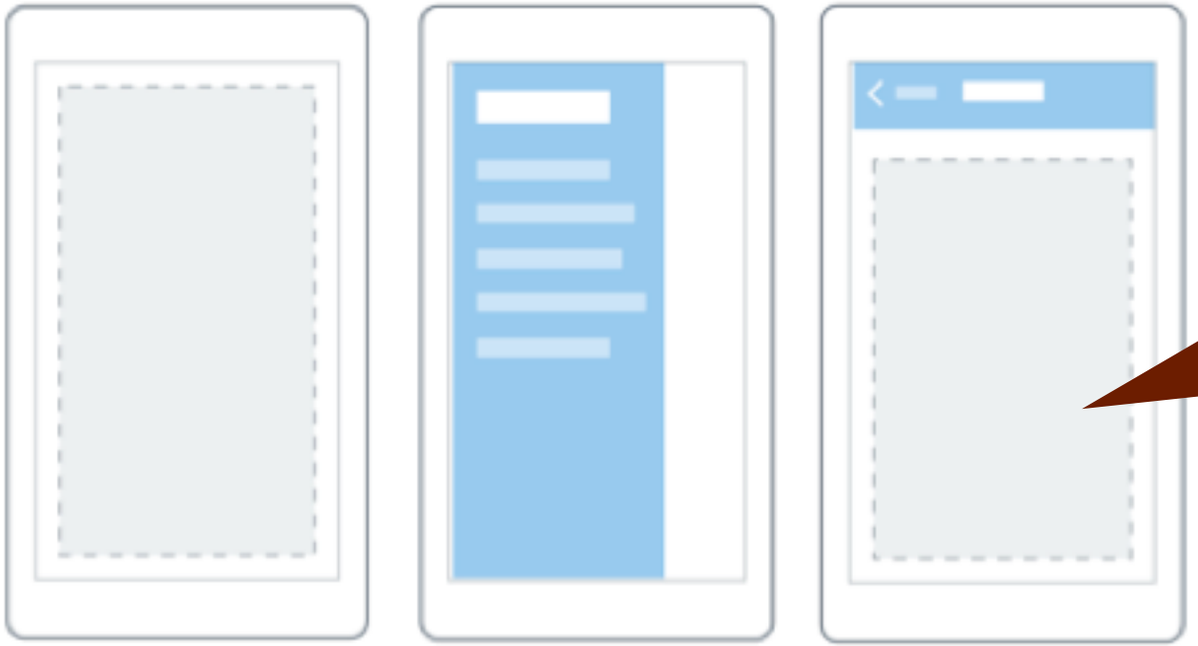
- derived types provide specific visualization / behavior



Pages

Page is an abstract class used to define a single screen of content

- derived types provide specific visualization / behavior



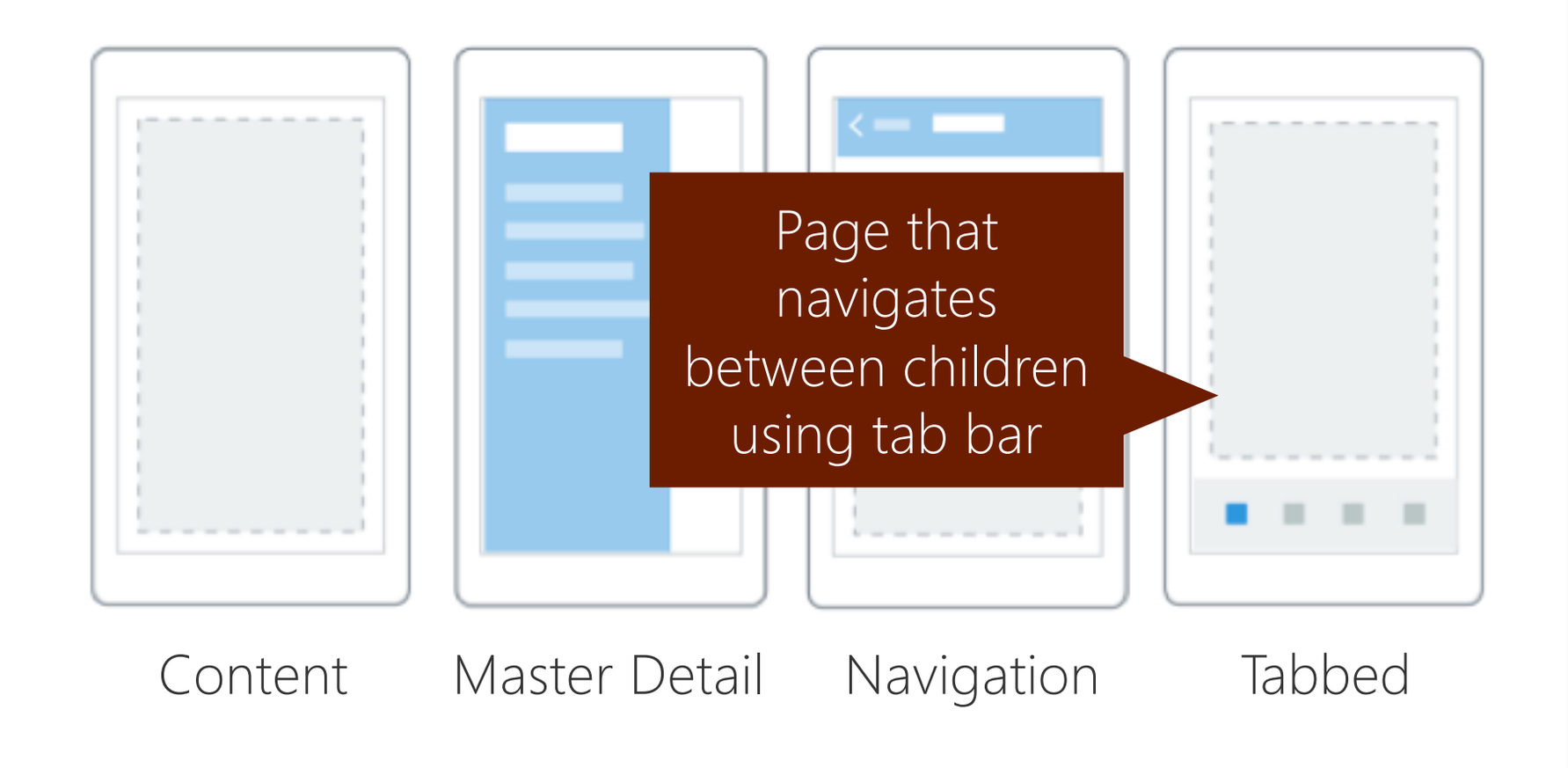
Content Master Detail Navigation

Manages a *stack* of pages with navigation bar

Pages

Page is an abstract class used to define a single screen of content

- derived types provide specific visualization / behavior



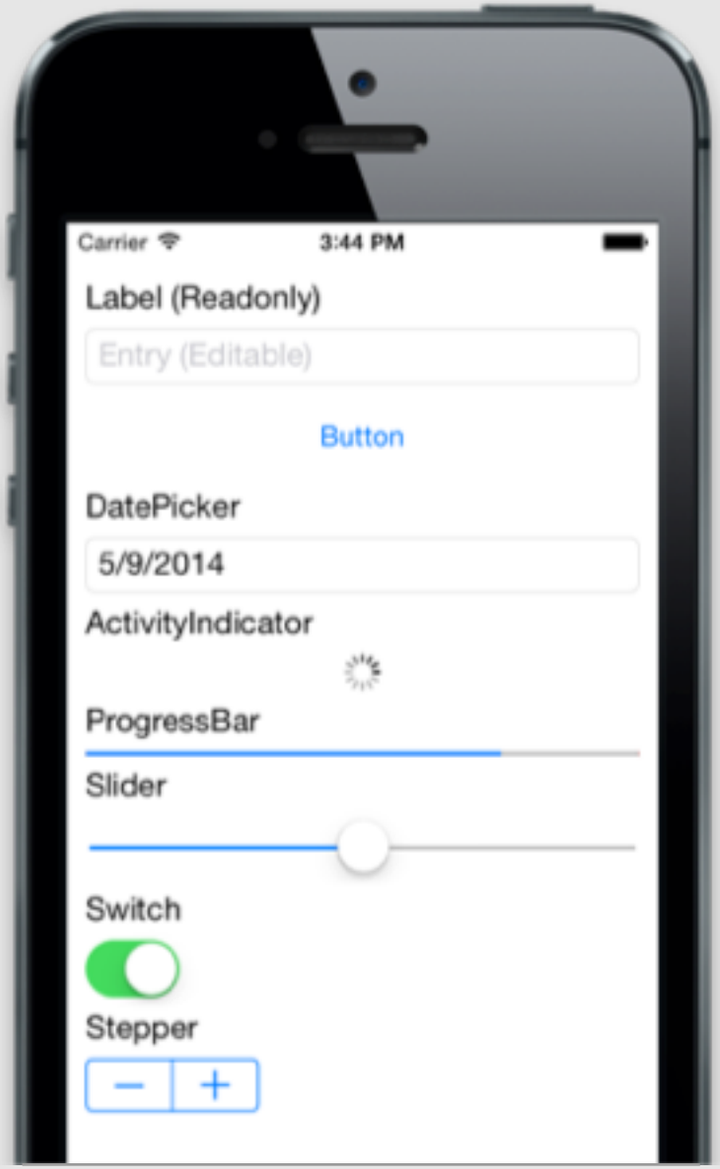
Demonstration

Adding a new ContentPage to a Xamarin.Forms application

Views

View is the base class for all visual controls, most standard controls are present

Label	Image	SearchBar
Entry	ProgressBar	ActivityIndicator
Button	Slider	OpenGLView
Editor	Stepper	WebView
DatePicker	Switch	ListView
BoxView	TimePicker	CarouselView
Frame	Picker	



Views - Button

Button provides a clickable surface with text


```
Button okButton = new Button() {
    Text = "Button”
};
okButton.Clicked += OnClick;
```

```
void OnClick(object sender, EventArgs e) {
  ...
}
```

Views - Label

Use a **`Label`** to display read-only text blocks

Hello, Forms!

```
Label hello = new Label() {
    Text = "Hello, Forms!",
    HorizontalTextAlignment = TextAlignment.Center,
    TextColor = Color.Blue,
    FontFamily = "Arial"
};
```

Views - Entry

Use an **`Entry`** control if you want the user to provide input with an on-screen or hardware keyboard


```
Entry edit = new Entry() {
    Text = "Hello",
    Keyboard = Keyboard.Text,
    PlaceholderText = "Enter Text"
};
```

Visual adjustments

Views utilize **properties** to adjust visual appearance and behavior

```
Entry numEntry = new Entry {
    Placeholder = "Enter Number",
    Keyboard = Keyboard.Numeric
};

Button callButton = new Button {
    Text = "Call",
    BackgroundColor = Color.Blue,
    TextColor =  Color.White
};
```

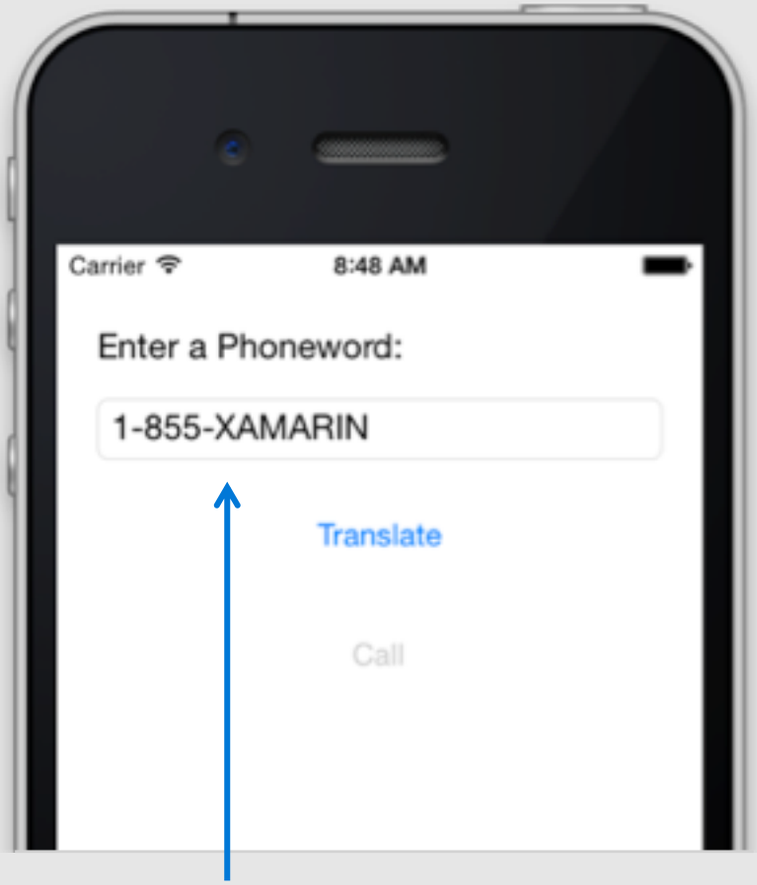
Demonstration

Interacting with a Button

Organizing content

Rather than specifying positions with coordinates (pixels, dips, etc.), you use layout containers to control how views are positioned relative to each other

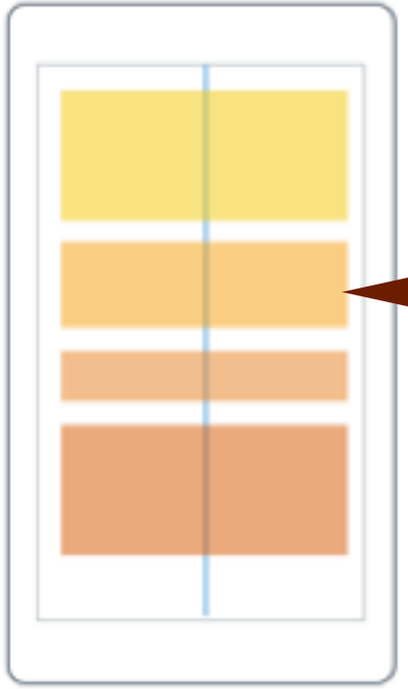
This provides for a more a*daptive* layout which is not as sensitive to dimensions and resolutions



For example, "stacking" views on top of each other with some spacing between them

Layout containers

Layout Containers organize child elements based on specific rules



StackLayout

`StackLayout` places children top-to-bottom (default) or left-to-right based on `Orientation` property setting

Layout containers

Layout Containers organize child elements based on specific rules

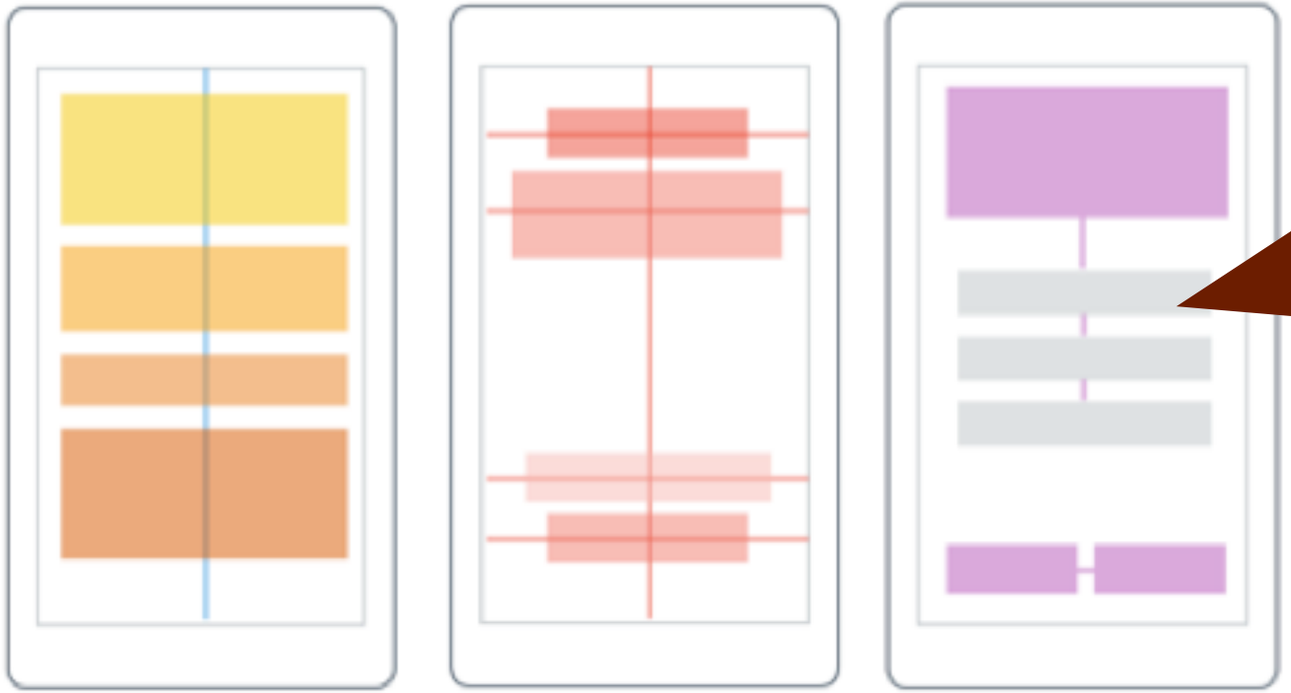
StackLayout

Absolute Layout

`AbsoluteLayout` places children in absolute requested positions based on anchors and bounds

Layout containers

Layout Containers organize child elements based on specific rules



StackLayout

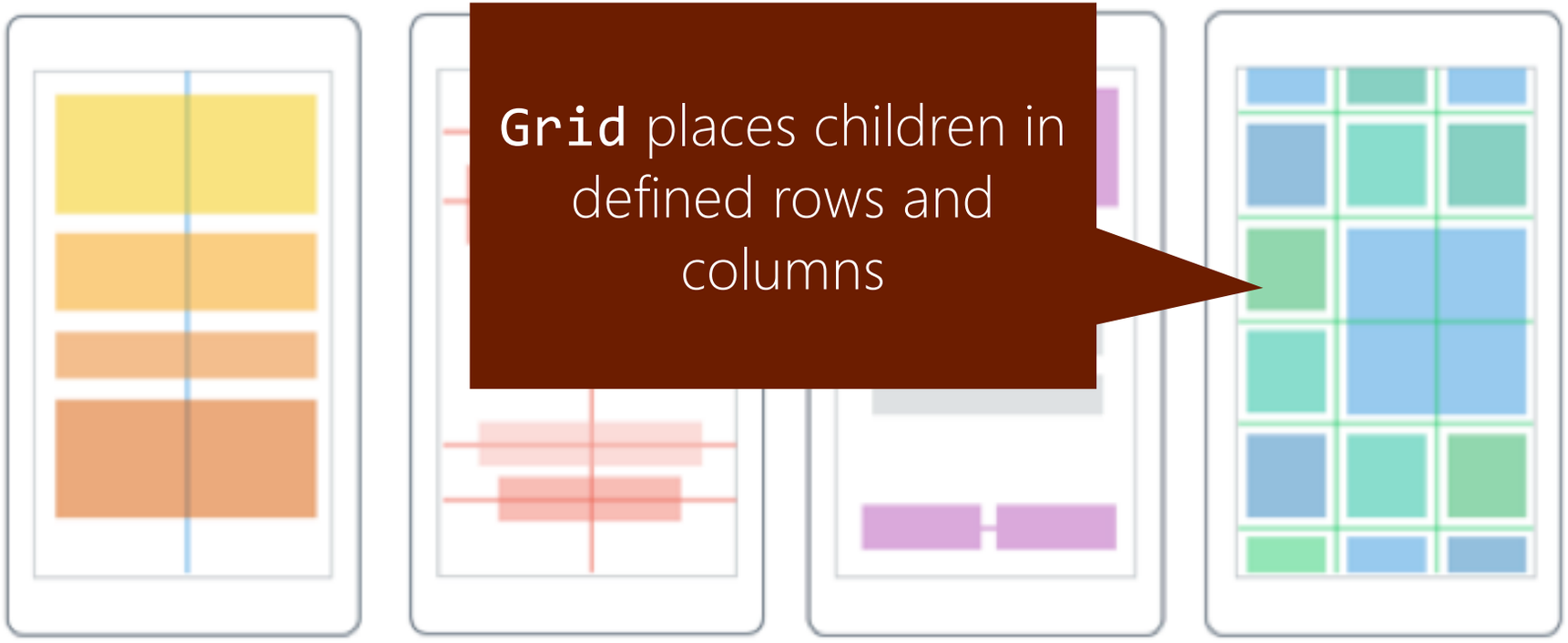
Absolute Layout

Relative Layout

`RelativeLayout` uses constraints to position the children

Layout containers

Layout Containers organize child elements based on specific rules



StackLayout

Absolute Layout

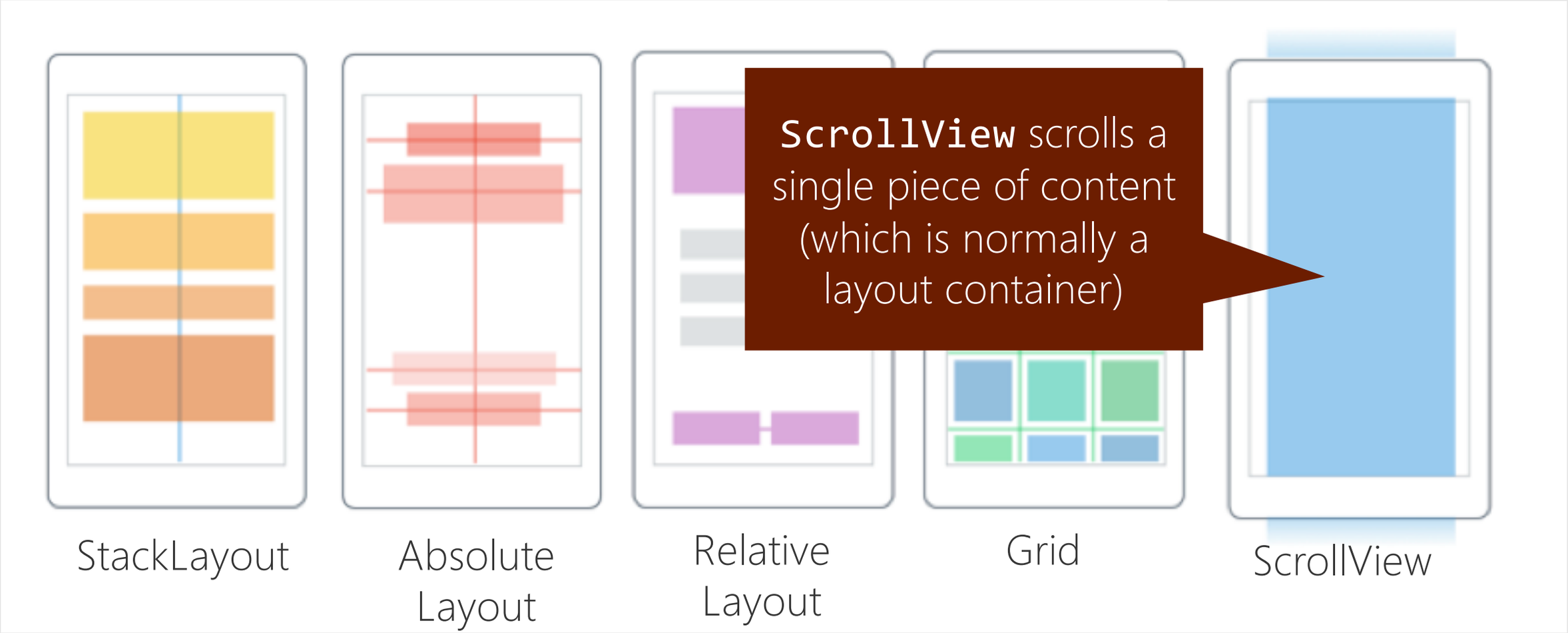
Relative Layout

Grid

`Grid` places children in defined rows and columns

Layout containers

Layout Containers organize child elements based on specific rules



Adding views to layout containers

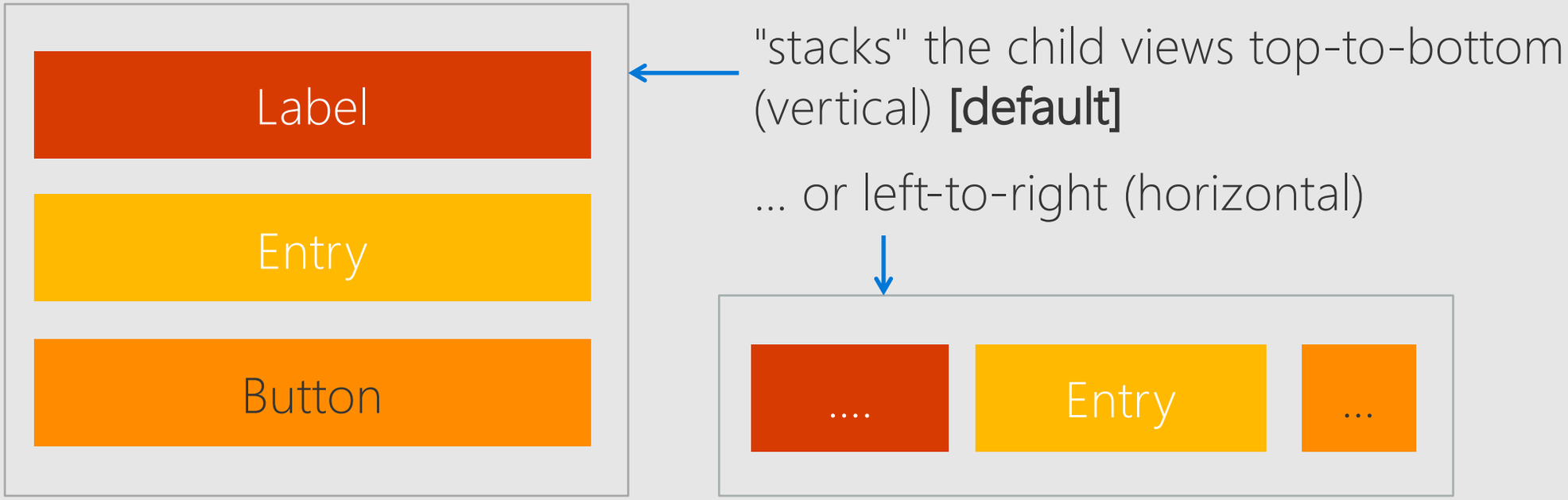
Layout containers have a **`Children`** collection property which is used to hold the views that will be organized by the container

```
Label label = new Label { Text = "Enter Your Name" };
Entry nameEntry = new Entry();

StackLayout layout = new StackLayout();
layout.Children.Add(label);
layout.Children.Add(nameEntry);

this.Content = layout;  // Assign as the page content
```

Views are laid out and rendered in the order they appear in the collection

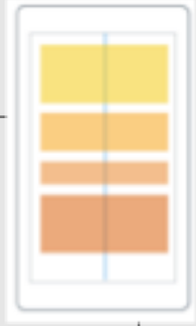
Working with StackLayout

`StackLayout` is used to create typical form style layout, **`Orientation`** property decides the direction that children are stacked



Working with StackLayout

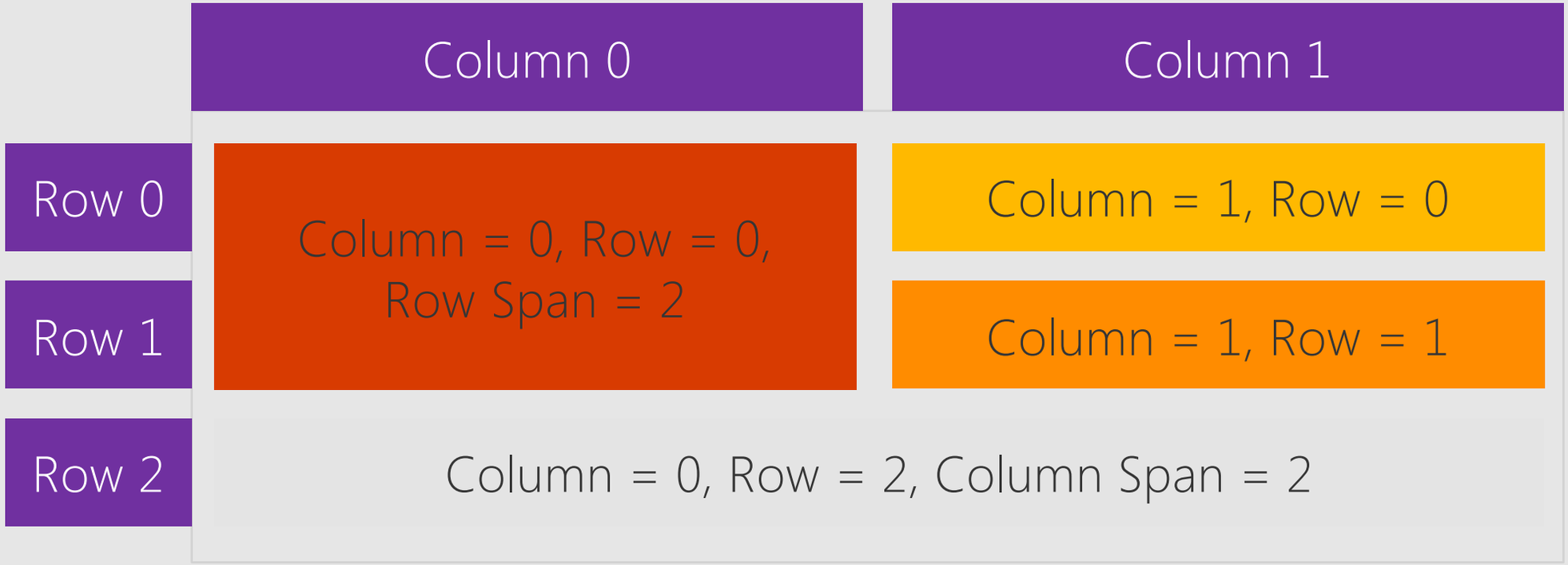
`StackLayout` is used to create typical form style layout, **`Orientation`** property decides the direction that children are stacked



```
var layout = new StackLayout {
   Orientation = StackOrientation.Vertical
};

layout.Children.Add(new Label { Text = "Enter your name:" });
layout.Children.Add(new Entry());
layout.Children.Add(new Button { Text = "OK" });
```

Working with Grid

`Grid` is used to create rows and columns of information, children identify specific column, row and span



Adding items to a Grid

Children in **Grid** must specify the layout properties, or they will default to the first column/row

```
Label label = new Label { Text = "Enter Your Name" };

Grid layout = new Grid();
layout.Children.Add(label);

Grid.SetColumn(label, 1);
Grid.SetRow(label, 1);
Grid.SetColumnSpan(label, 2);
Grid.SetRowSpan(label, 1);
```

Use static methods defined on **Grid** to set layout properties

Adding items to a Grid

Children in **`Grid`** must specify the layout properties, or they will default to the first column/row

```
Grid layout = new Grid();

...

layout.Children.Add(label, 0, 1);          // Left=0 and Top=1
layout.Children.Add(button, 0, 2, 2, 2);   // L=0, R=2, T=2, B=2
```

Can also specify row/column as Left/Right/Top/Bottom values to **`Add`** method

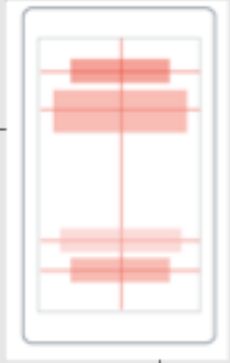
Controlling the shape of the grid

Can influence the determined shape and size of the columns and rows

```
Grid layout = new Grid();
layout.RowDefinitions.Add(new RowDefinition {
    Height = new GridLength(100, GridUnitType.Absolute) // 100px
});
layout.RowDefinitions.Add(new RowDefinition {
    Height = new GridLength(1, GridUnitType.Auto) // "Auto" size
});
layout.ColumnDefinitions.Add(new ColumnDefinition {
    Width = new GridLength(1, GridUnitType.Star)  // "Star" size
});
```

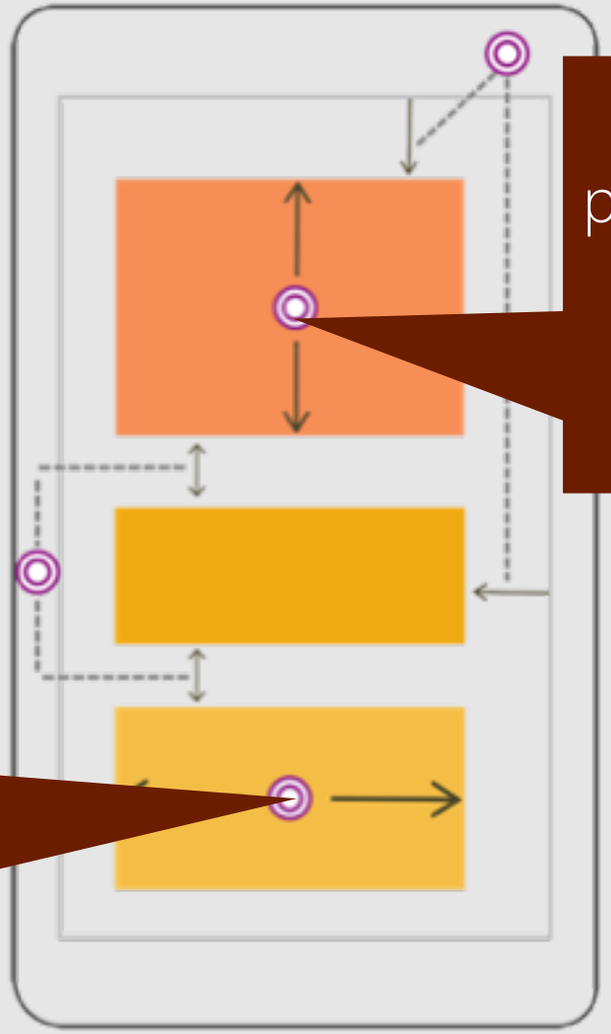
Working with AbsoluteLayout

- **`AbsoluteLayout`** positions and sizes children by **absolute values** through either a coordinate (where the view determines it's own size), or a bounding box



```
var layout = new AbsoluteLayout();
...
// Can do absolute positions by coordinate point
layout.Children.Add(label1, new Point(100, 100));

// Or use a specific bounding box
layout.Children.Add(label2, new Rectangle(20, 20, 100, 25));
```

Adding spacing and padding

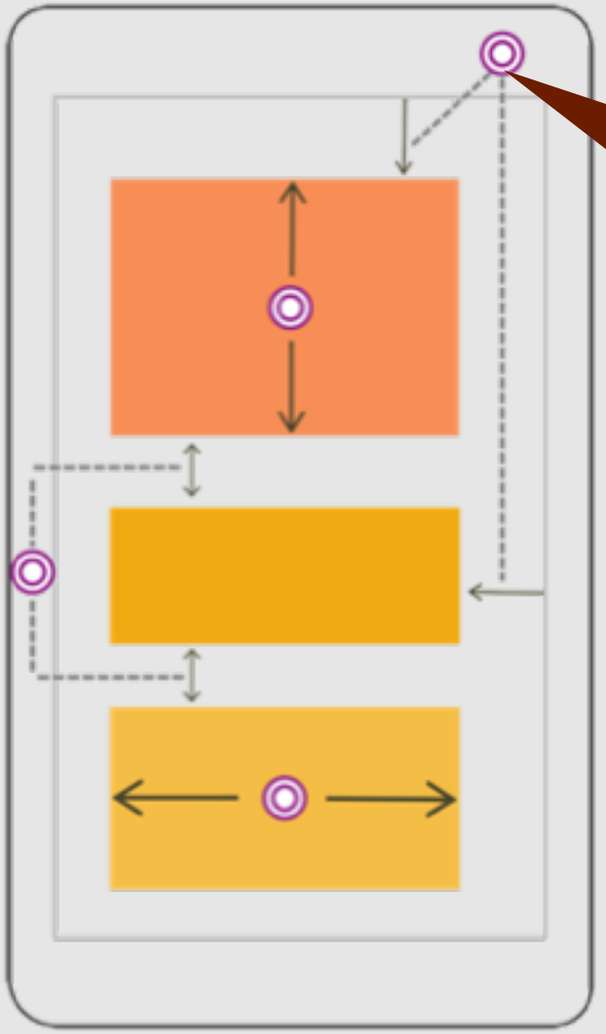


`HeightRequest`
property used to request a specific height on a view in the parent container

`WidthRequest`
property used to request a specific width on a view in the parent container

```
Button okButton = ...;
okButton.WidthRequest = 100;
okButton.HeightRequest = 75;
```

Adding spacing and padding

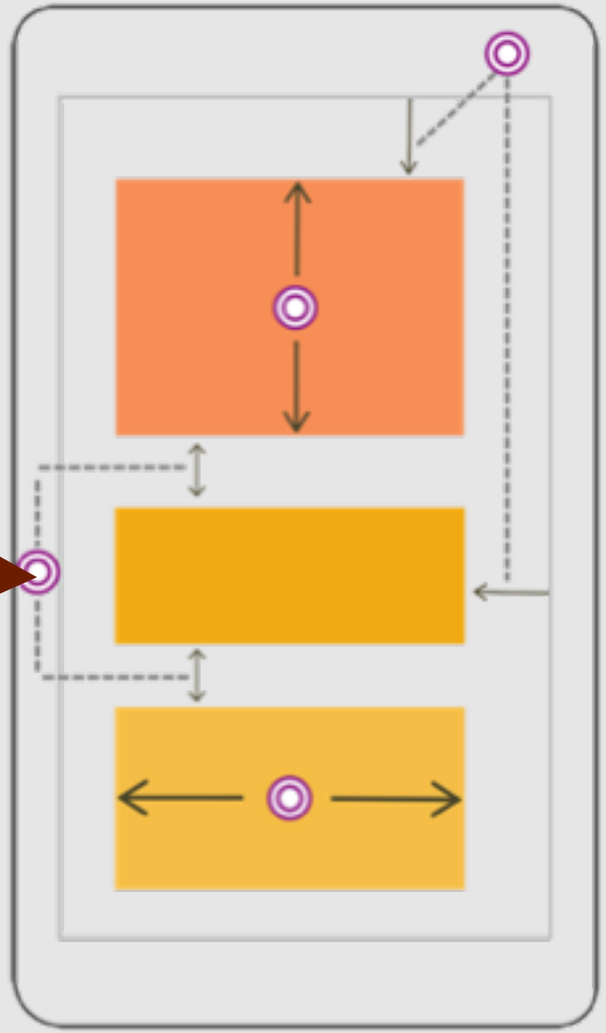


`Padding` property on parent containers is used to add padding *around* the children

```
ContentPage mainPage = ...;
mainPage.Padding =
   new Thickness(5,20,5,5);
```

Adding spacing and padding

`Spacing` property on **`StackLayout`** and **`Grid`** allows you to control spacing *in-between* children



```
StackLayout layout = ...;
layout.Spacing = 20;
```

```
Grid layout = ...;
layout.RowSpacing = 10;
layout.ColumnSpacing = 20;
```

Exercise #2

Add a landing page programmatically

Working with Images

Common to use images in the UI to make the design look professional

```
Image fastCar = new Image() {
    Aspect = Aspect.AspectFill,
    WidthRequest = 200,
    HeightRequest = 200,
    Opacity = .75,
    Source = ImageSource
                 .FromFile("car.jpg"),
};
```

Local Resources

Resources, like images, can be bundled with the application in two ways

Native Resources

Native resources are placed into each native project – this allows the resource to be different per-platform and also to take advantage of platform features such as different resolutions or densities

Local Resources

Resources, like images, can be bundled with the application in two ways

Native Resources

They are referenced directly by filename in your code, use platform-specific build types and must be placed into specific locations in each native project

Local Resources

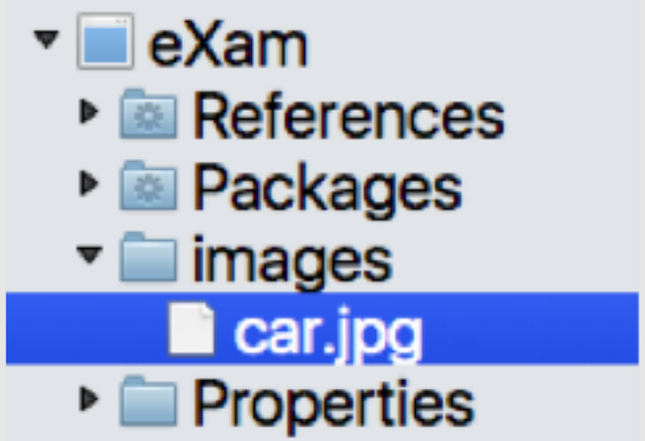
Resources, like images, can be bundled with the application in two ways

Embedded resources are placed into the PCL project and have the build type `EmbeddedResource`, these are shared between all projects

Embedded Resources

Using embedded resources

Use **`ImageSource.FromResource`** to load embedded resource images




```
Image fastCar = ...;
fastCar.Source = ImageSource.FromResource(
            "eXam.images.car.jpg");
```

Build	
Build action	EmbeddedResource
Copy to output directory	Do not copy
Custom Tool	
Custom Tool Namespace	
Resource ID	eXam.images.car.jpg

Path includes dot-separated namespace, folder and filename and can be found in properties for image

Exercise #3

Display an image on the landing page

Summary

- Xamarin.Forms is a cross-platform UI framework for iOS, Android and Windows Phone
- UI definition and primary business logic are defined in shared code; native project used as host
- UI is defined using cross-platform control definitions (Pages, Views and Layout Containers) and then turned into native platform UX at runtime