
IOS101

# Introduction to Xamarin.iOS

Download class materials from
university.xamarin.com

Microsoft

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, trademarked, 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.



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. Introduce the development tools
2. (De)constructing the application
3. Add views and behavior

# Introduce the development tools

Xamarin University

# Tasks

1. Explore the IDE choices
2. Create an app using the project templates

# Reminder: development setup

- You must have the following to build iOS apps:

Mac running macOS

with the latest version of Xcode

Xamarin tools on all your development machines (both macOS and Windows)

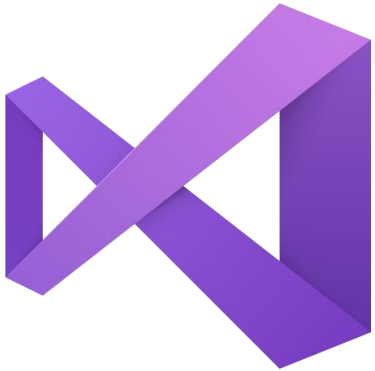


Setup help is provided in the **XAM101** orientation class, if you have not setup your environment yet we highly recommend you attend that class first

# Choose your IDE

- Xamarin allows you to build iOS applications using C# / .NET with either

Microsoft Visual Studio
on Windows



Visual Studio for Mac
on macOS

# Create a new application

- Visual Studio has project templates to create a new applications for Apple platforms

iPhone or iPad

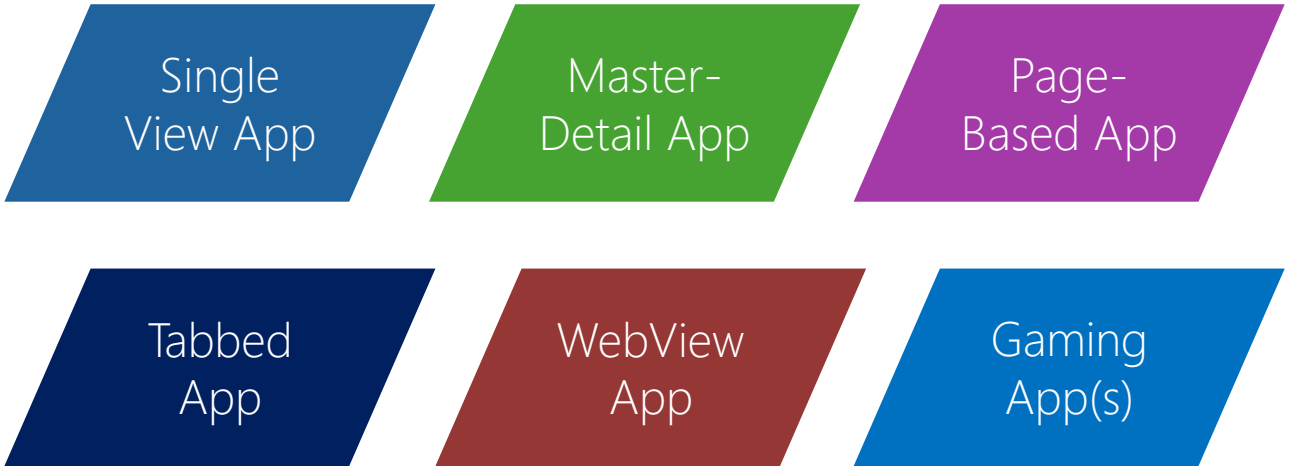
macOS

Apple TV

Apple Watch

# Choosing an iOS project template

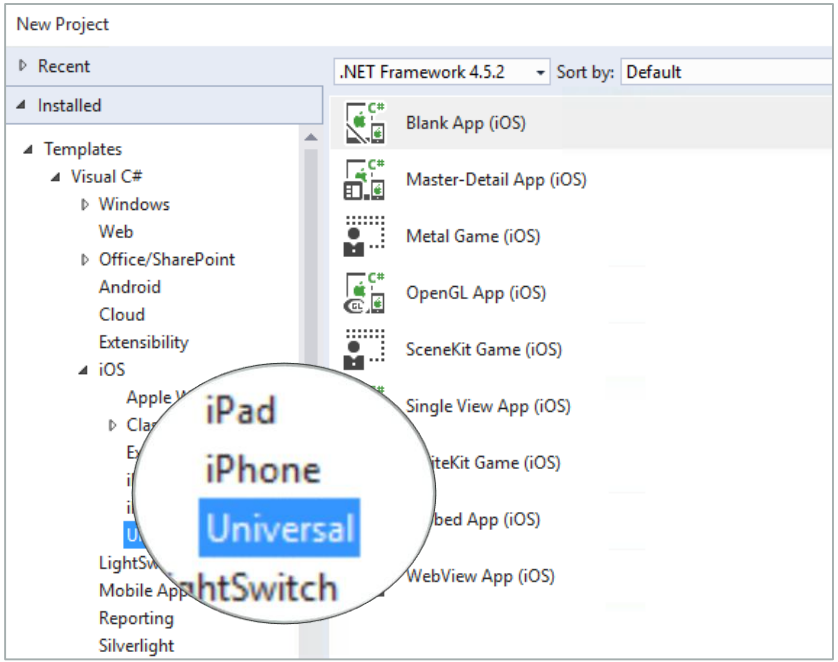
- Project templates provide starting point for different application styles



# Universal application templates

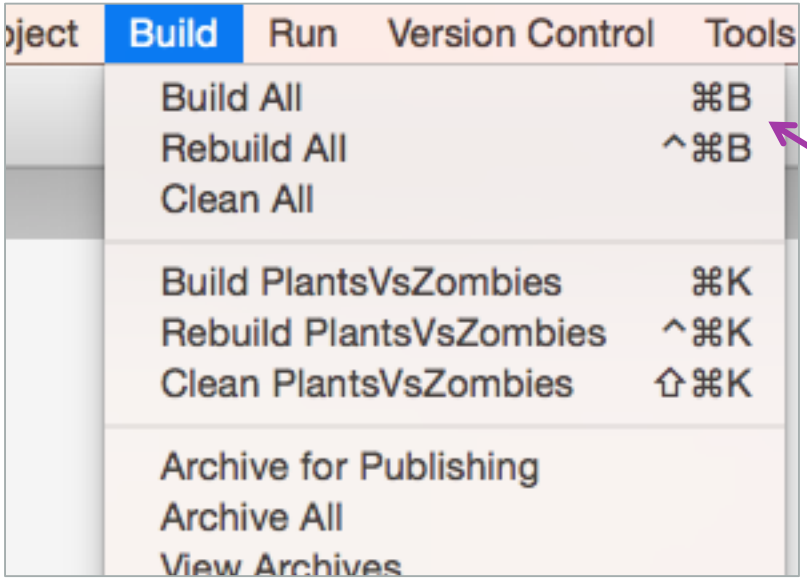
- Visual Studio includes "Universal" templates which support iPhone + iPad in a single app using two separate views

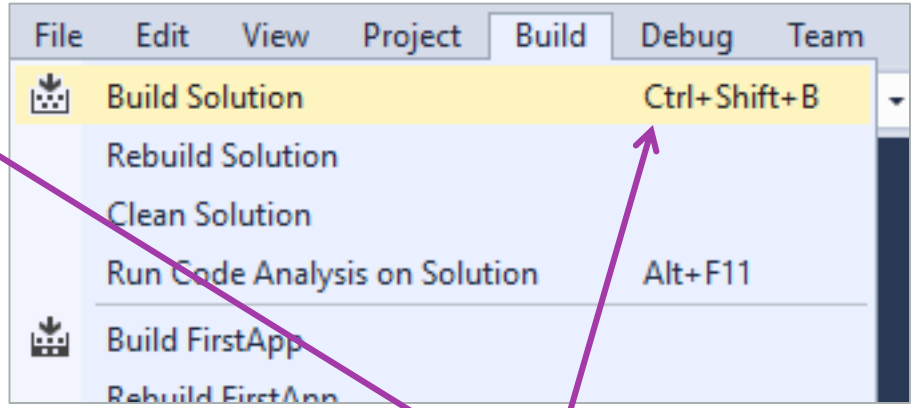
- This is an older set of templates which have been deprecated by new support in iOS8 for adaptive design



# Building your application

- Use the **Build** menu or toolbar to compile/run the application

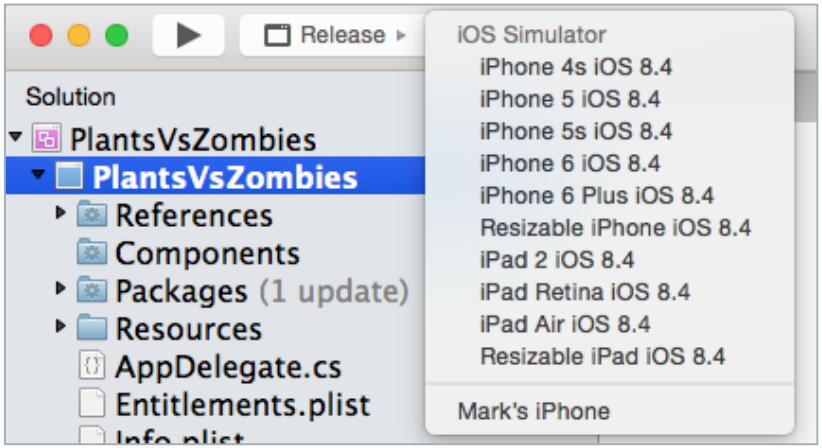


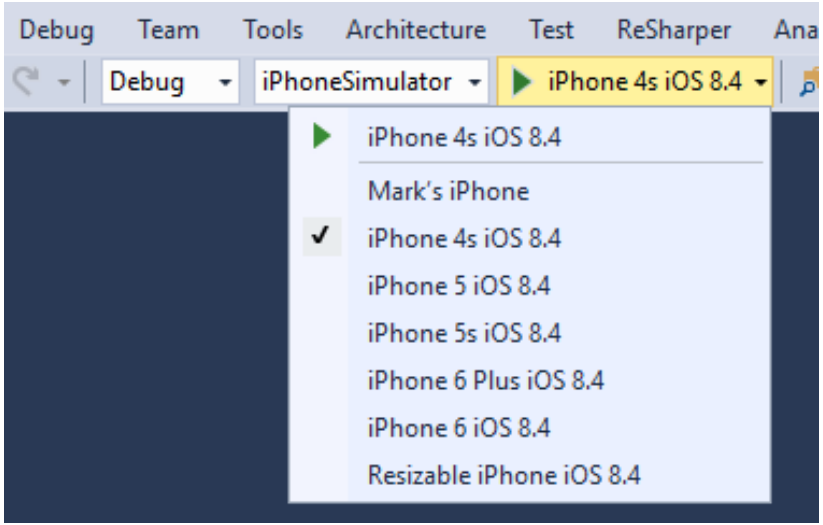

Both IDEs have shortcut keys to build

# Testing your application

- Xcode includes a simulator that can run your app on the Mac, this is the easiest way to test your applications initially

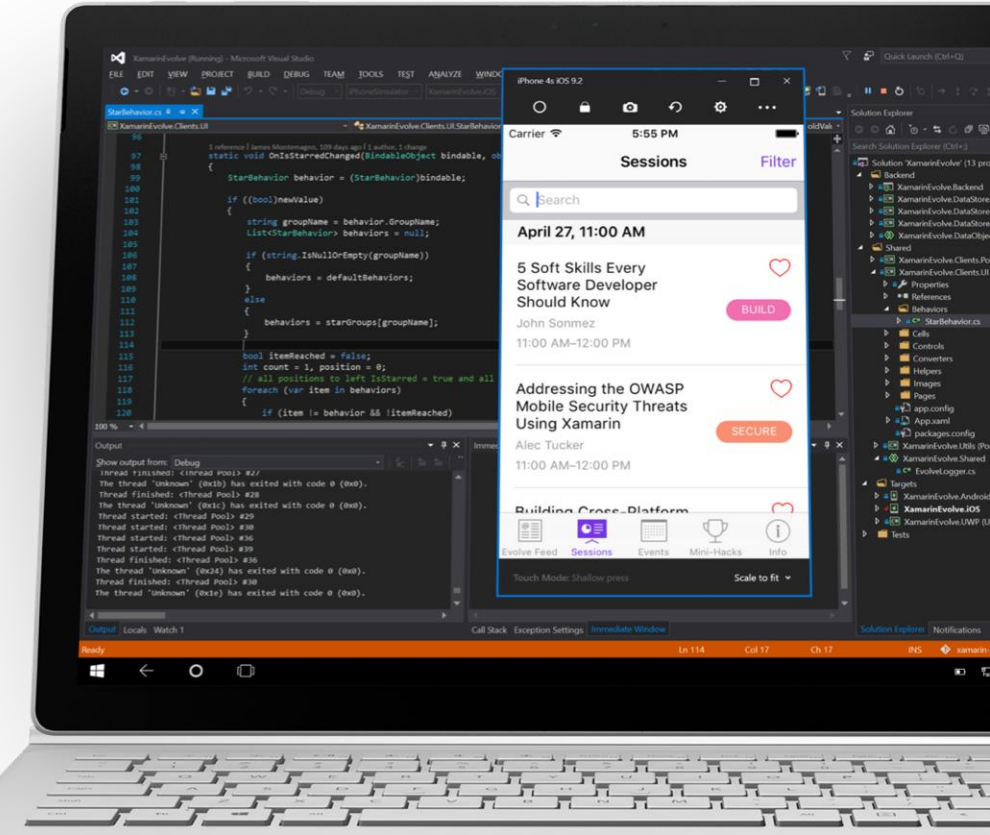



IDEs provide access to the simulator selection directly on the toolbar



# Remoted iOS Simulator for Windows

- Visual Studio Enterprise includes the Remoted iOS Simulator which displays an iOS simulator on Windows
- Multi-touch and pressure-sensitive interaction
- Supports rotation, screenshots, and location changes



# What about deploying to a device?

- To test on a device, you will need to register each device and get a set of signing certificates from Apple
- Must have a registered developer Apple account to deploy to a device (can be paid or free)
- Watch the lightning lecture on provisioning an iOS device for testing

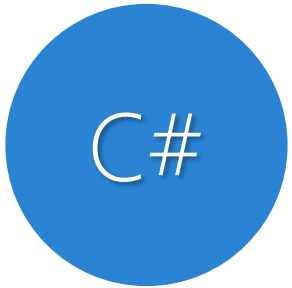
# Group Exercise

Creating and running your first iOS application

Xamarin University

# What is included in Xamarin.iOS?

- Xamarin.iOS includes both compile-time and runtime components




C# compiler for Mac

Native compiler and linker

Runtime services (GC, type checking, etc.)

Core .NET Libraries

# Summary

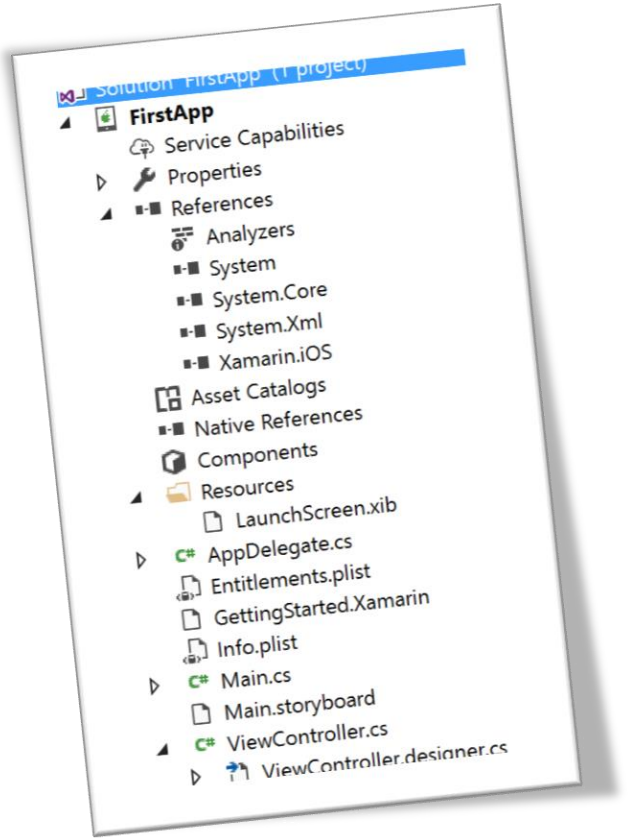
1. Explore the IDE choices
2. Create an app using the project templates

# (De)constructing the application

Xamarin University

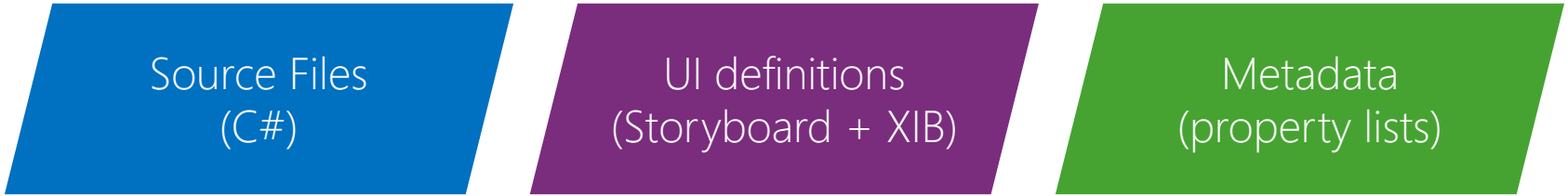
# Tasks

1. Explore a new project
2. Model-View-Controller
3. Delegates and Protocols



# Let's explore the created project

- The created project is contained in a standard .NET solution and has several related files that work together to create the application

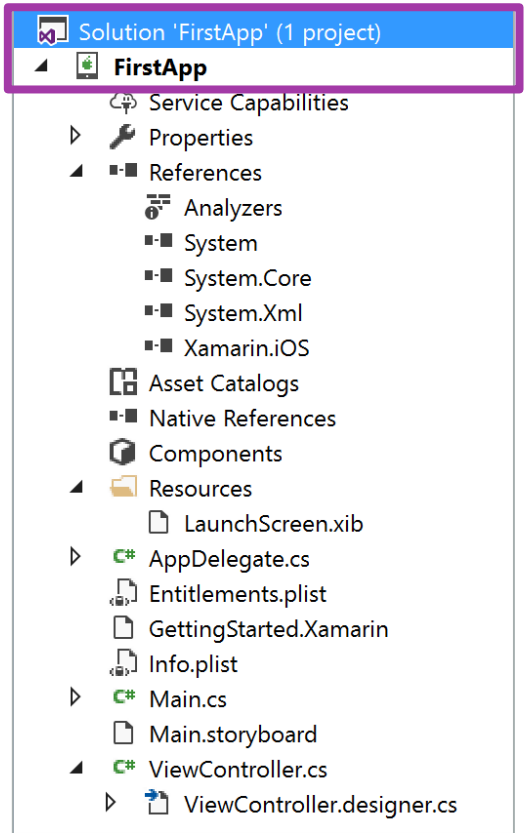


# Demonstration

Explore the created project

Xamarin University

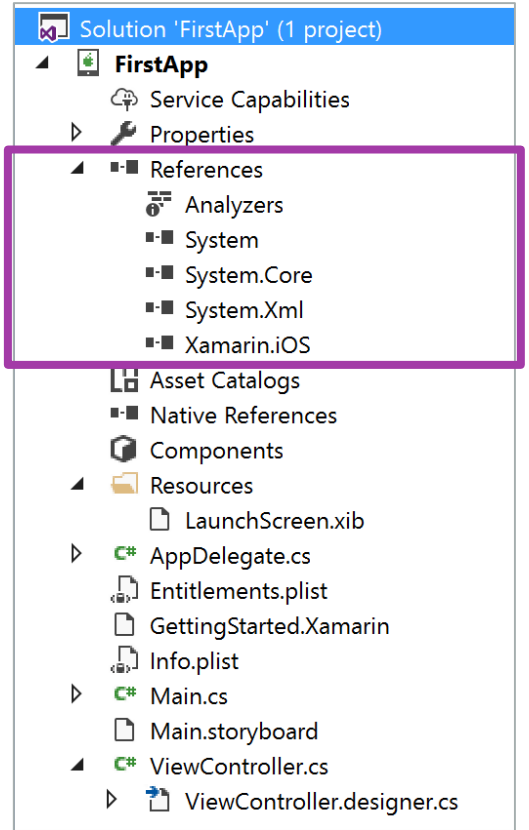
# Let's explore the created project




- IDE loads a *solution file* (**.sln**) which contains one or more *project files* (**.csproj**), each project generates some sort of output – typically an executable or library

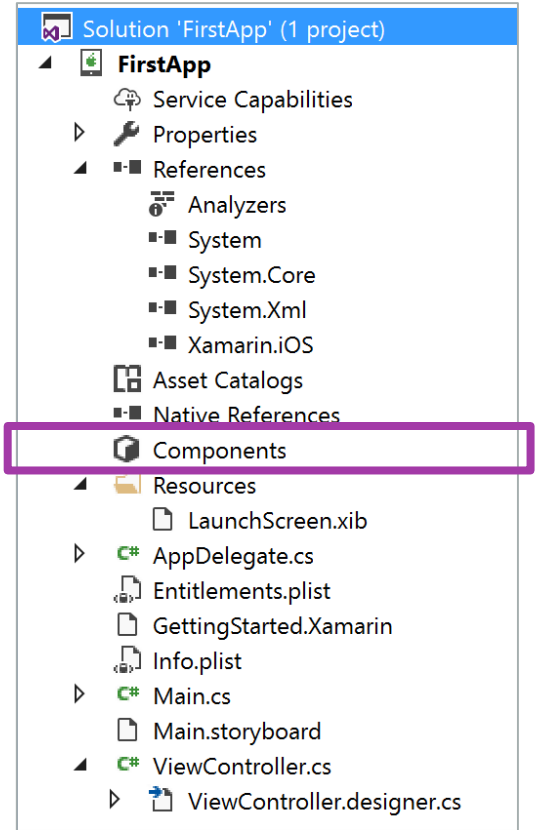
- Uses MSBuild-based projects which can be loaded into either Visual Studio or Visual Studio for Mac – can switch back and forth between macOS and Windows if desired

# Let's explore the created project



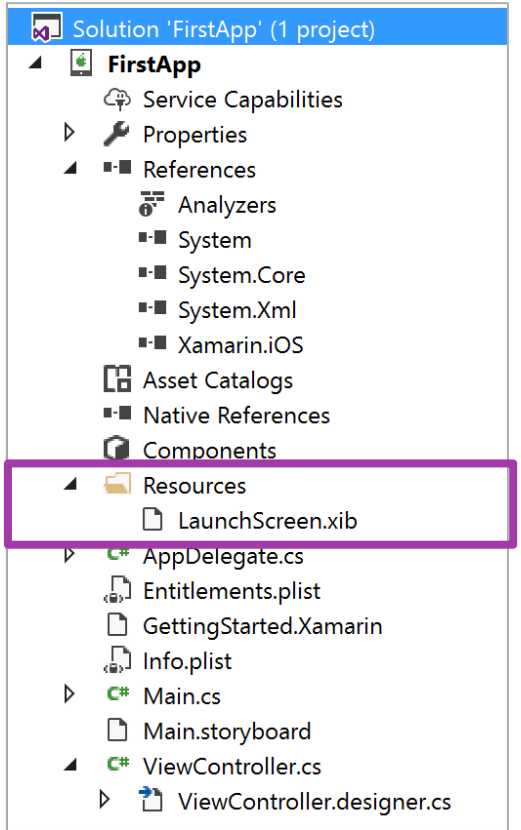

- References folder contains required compile and runtime assemblies
- Can add new assemblies through context menu by right-clicking on the references folder
- Referenced assemblies must either be compatible portable class libraries (PCLs), or compiled against Xamarin.iOS – cannot use desktop .NET assemblies directly

# Let's explore the created project



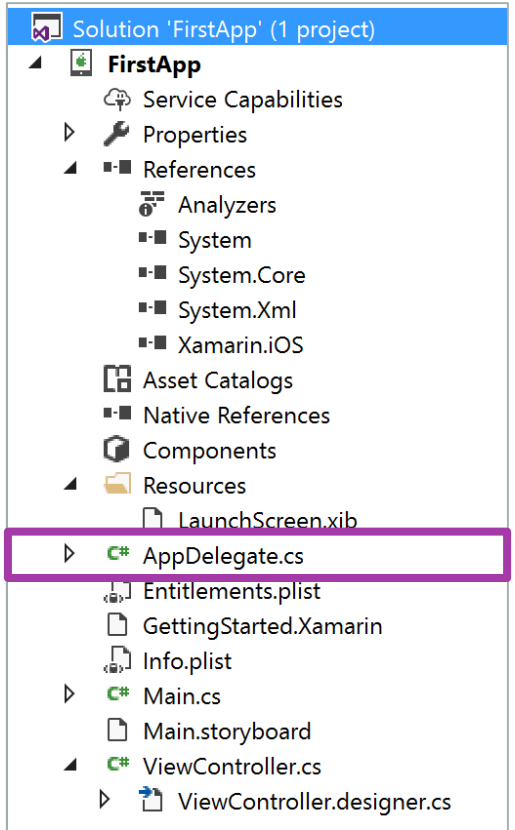

- Components folder contains components downloaded from the Xamarin Component Store (components.xamarin.com)
- Packages folder (only in Visual Studio for Mac) contains any referenced Nuget packages (www.nuget.org)
- Components/Packages must either be compiled as a portable library, or against Xamarin.iOS

# Let's explore the created project



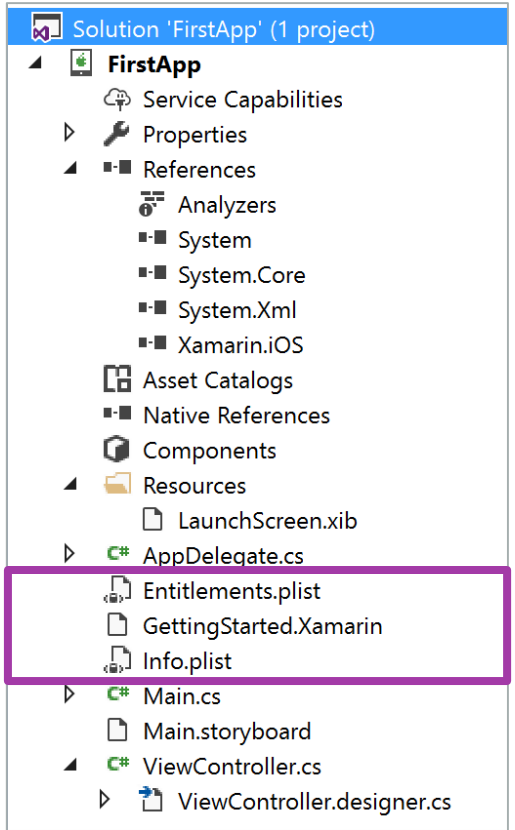

- Resources folder contains additional assets needed at runtime such as images
- Files in this folder typically have a build action of **`BundleResource`** and are included with the generated application package to be installed on a device
- Template creates a launch screen displayed while the app starts

# Let's explore the created project



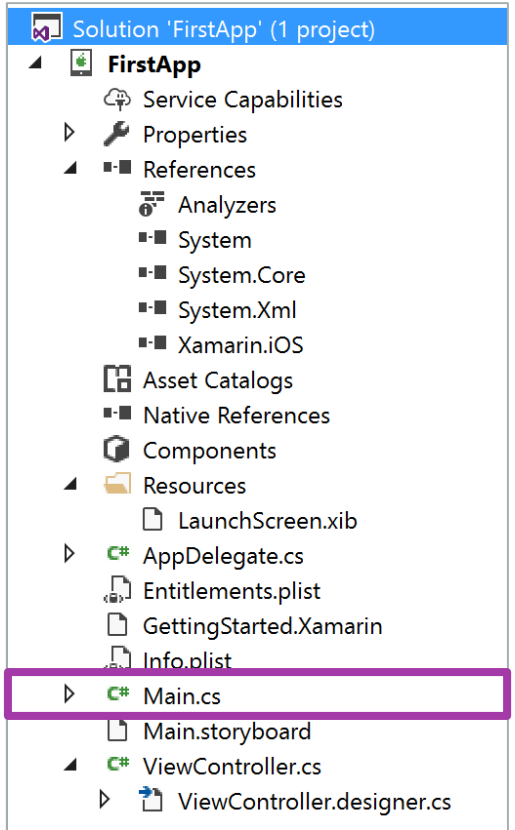

- **`AppDelegate.cs`** is responsible for creating the main window and listening to operating system events
- Contains a class implements that derives from iOS **`UIApplicationDelegate`**
- Must override virtual methods in class to process received operating system events

# Let's explore the created project



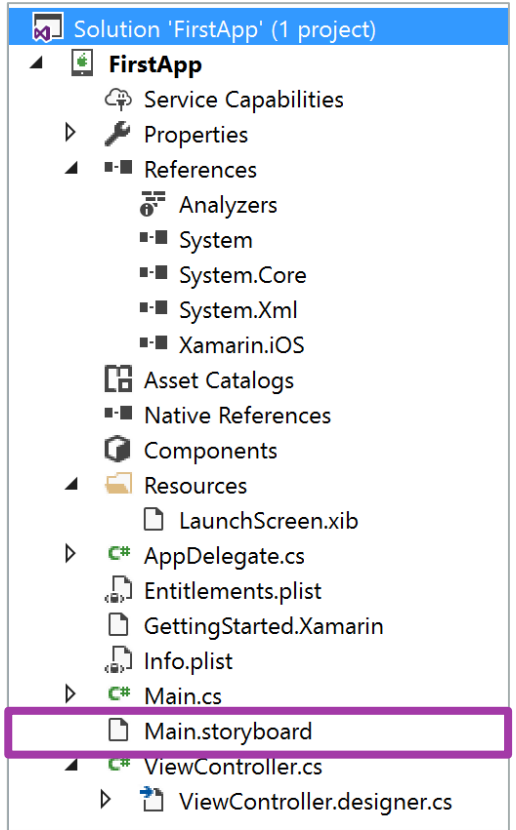

- iOS uses *property list* files to store application metadata as key/value pairs
  - **`Entitlements.plist`** lists external Apple services your app wants to interact with such as in-app purchases, HealthKit or push notifications
  - **`Info.plist`** identifies app icons, version number, app name and other app details
- Both IDEs include a GUI editor for these files to edit the most common settings

# Let's explore the created project



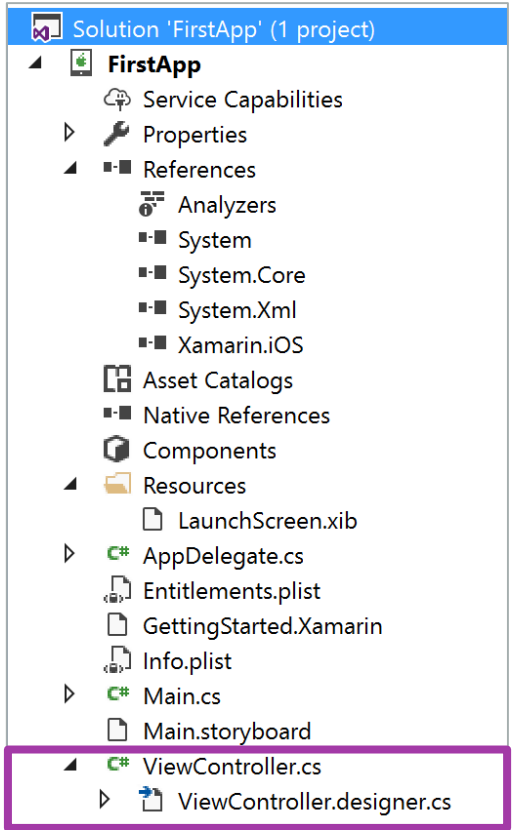

- **`Main.cs`** contains the main entry point for the application in the form of a standard .NET **`static void Main()`**
- It starts up the iOS UI framework (UIKit) and identifies the App Delegate, which will in turn bring up the initial screen for the application
- Be cautious about adding code into the **`Main`** method – iOS has time limits on app launches!

# Let's explore the created project




- **`MainStoryboard.storyboard`** contains the declarative (XML) definition of all the screens in the application (this file is not present for game-based templates)
- Xamarin.iOS includes a built-in designer integrated into both IDEs, or you can use Interface Builder in Xcode
- Primary storyboard is identified in the **`info.plist`**

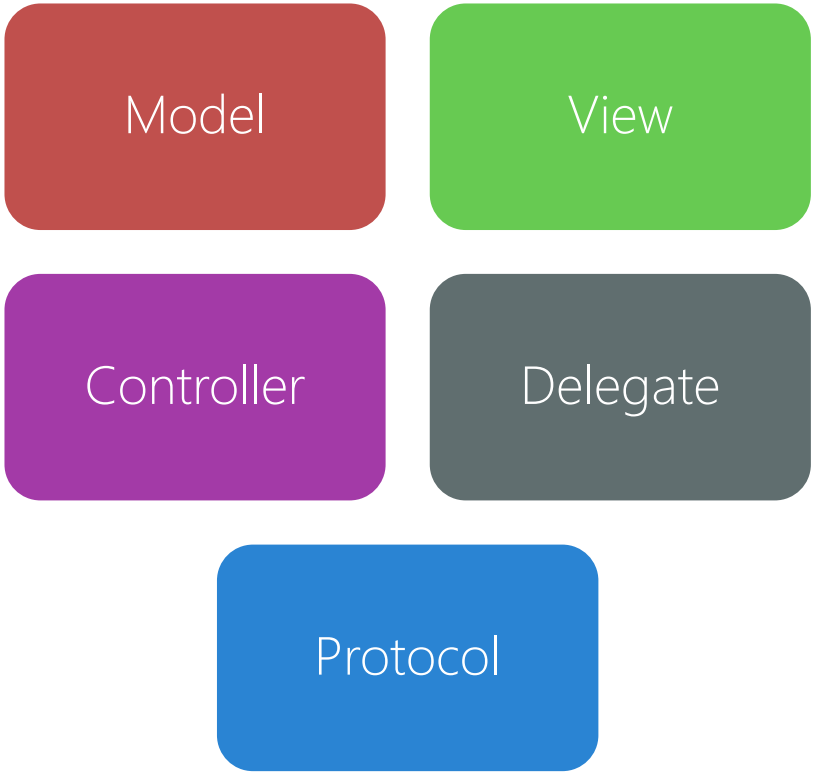
# Let's explore the created project




- `(Root)ViewController.cs` contains the behavior for the initial screen, each screen in your app will have a view controller source file associated with it
- `(Root)ViewController.designer.cs` is a partial-class definition used by the designer to connect elements in the storyboard with the code defined in the view controller
- This follows the **MVC design pattern**

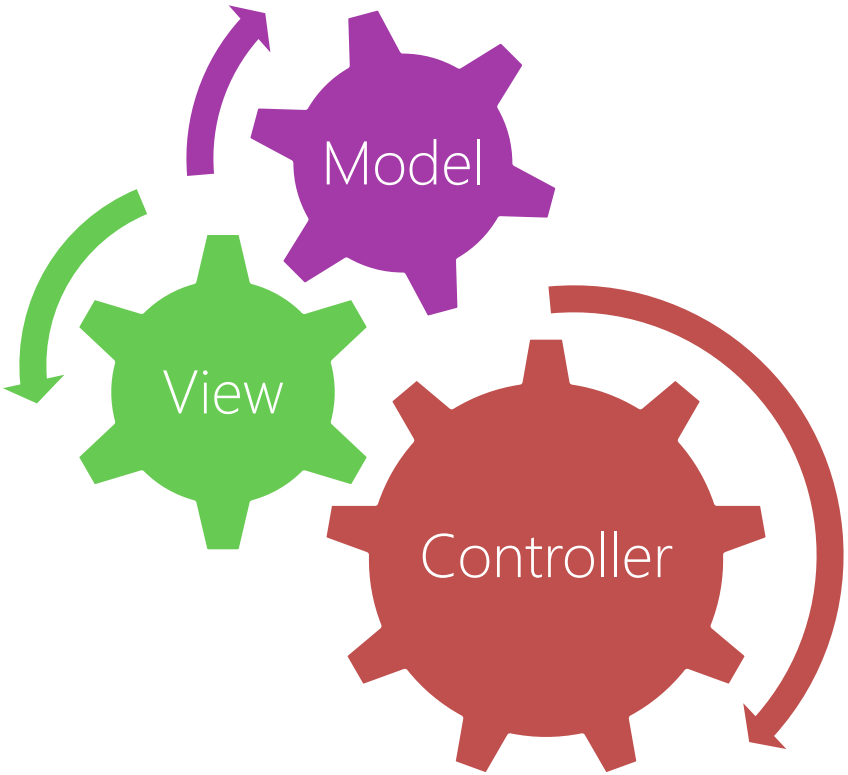
# iOS Terminology

- iOS uses several terms which might be unfamiliar or have different meanings than what you are used to



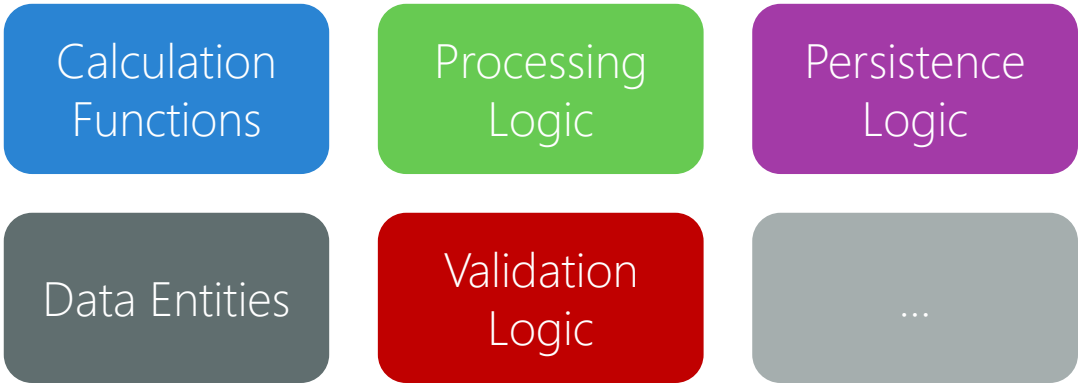
# What is MVC?

- Model-View-Controller (MVC) is an established architectural design pattern to logically separate the UI, data and behavior of an app
- This is the cornerstone design pattern for all iOS applications and it's usage is enforced by the iOS API design



# Model

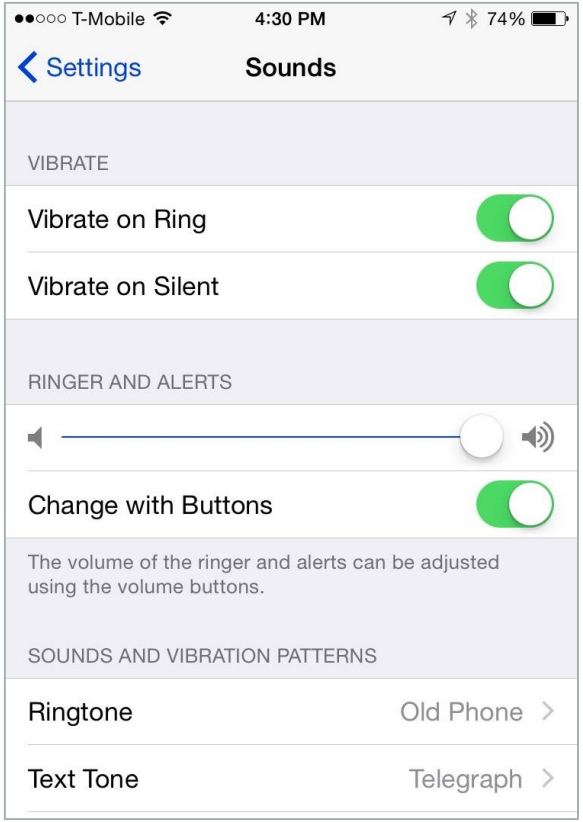
- The **Model** contains data, information and logic that is considered part of the business layer of your application; this is almost all developer-created



# View

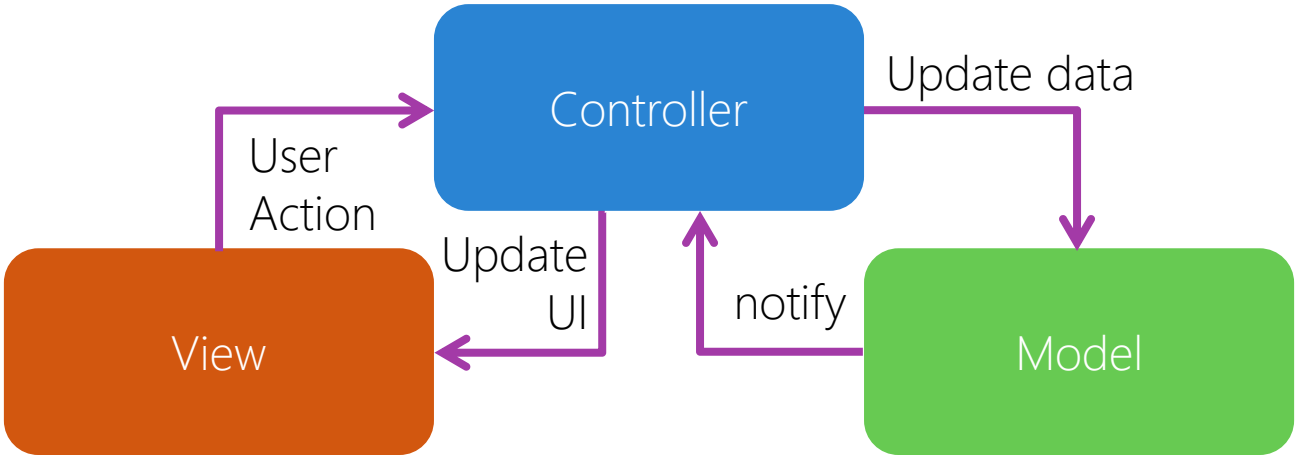
- The **View** contains all the visual components the user sees and interacts with such as buttons, sliders and text, all of which derive from a standard class **`UIView`**

- Views are composed and can be defined in **code** or declaratively using a **Storyboard** or **XIB** file



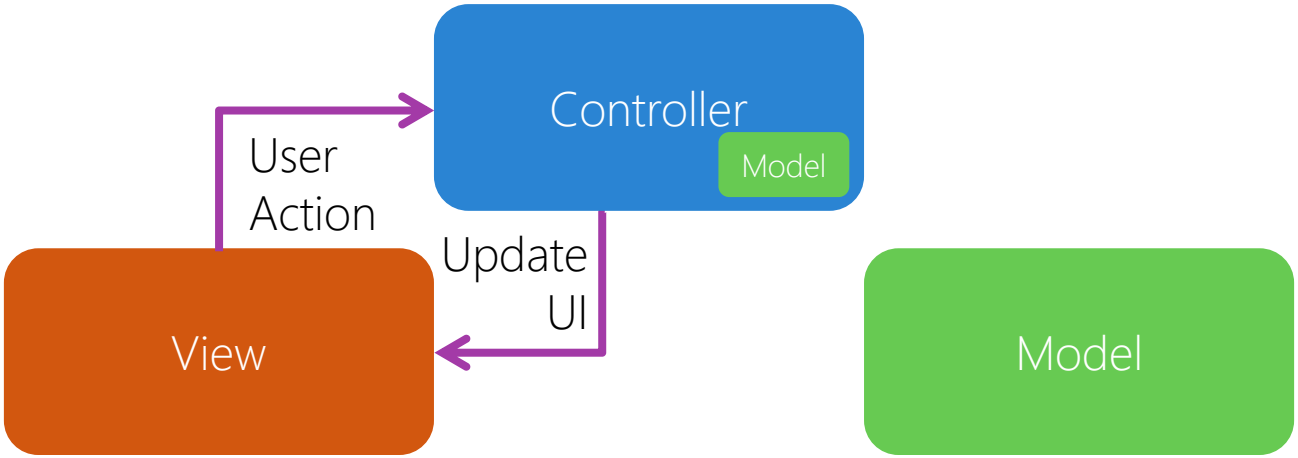
# Controller

- The Controller is the moderator between the model and the view, in iOS these are classes that derive from **`UIViewController`**



# Controller

- The Controller is the moderator between the model and the view, in iOS these are classes that derive from **`UIViewController`**




iOS provides several implementations of **`UIViewController`** to manage different UI styles and behaviors such as navigation, alerts and tables

# Defining a Root Controller

- App must identify a single view controller to be the *starting controller*

```
public class AppDelegate : UIApplicationDelegate
{
    public override UIWindow Window { get; set; }

    public override bool FinishedLaunching(...) {

        ...

        return true;
    }
...
```

# Defining a Root Controller

- App must identify a single view controller to be the *starting controller*

```
public class AppDelegate : UIApplicationDelegate
{
    public override UIWindow Window { get; set; }

    public override bool FinishedLaunching(...) {
        Window = new UIWindow(UIScreen.MainScreen.Bounds);
        ...

        return true;
    }
...
```

# Defining a Root Controller

- App must identify a single view controller to be the *starting controller*

```
public class AppDelegate : UIApplicationDelegate
{
    public override UIWindow Window { get; set; }

    public override bool FinishedLaunching(...) {
        Window = new UIWindow(UIScreen.MainScreen.Bounds);
        Window.RootViewController = new MyViewController();
        ...
        return true;
    }
...
```

```
public class MyViewController
: UIViewController  { … }
```

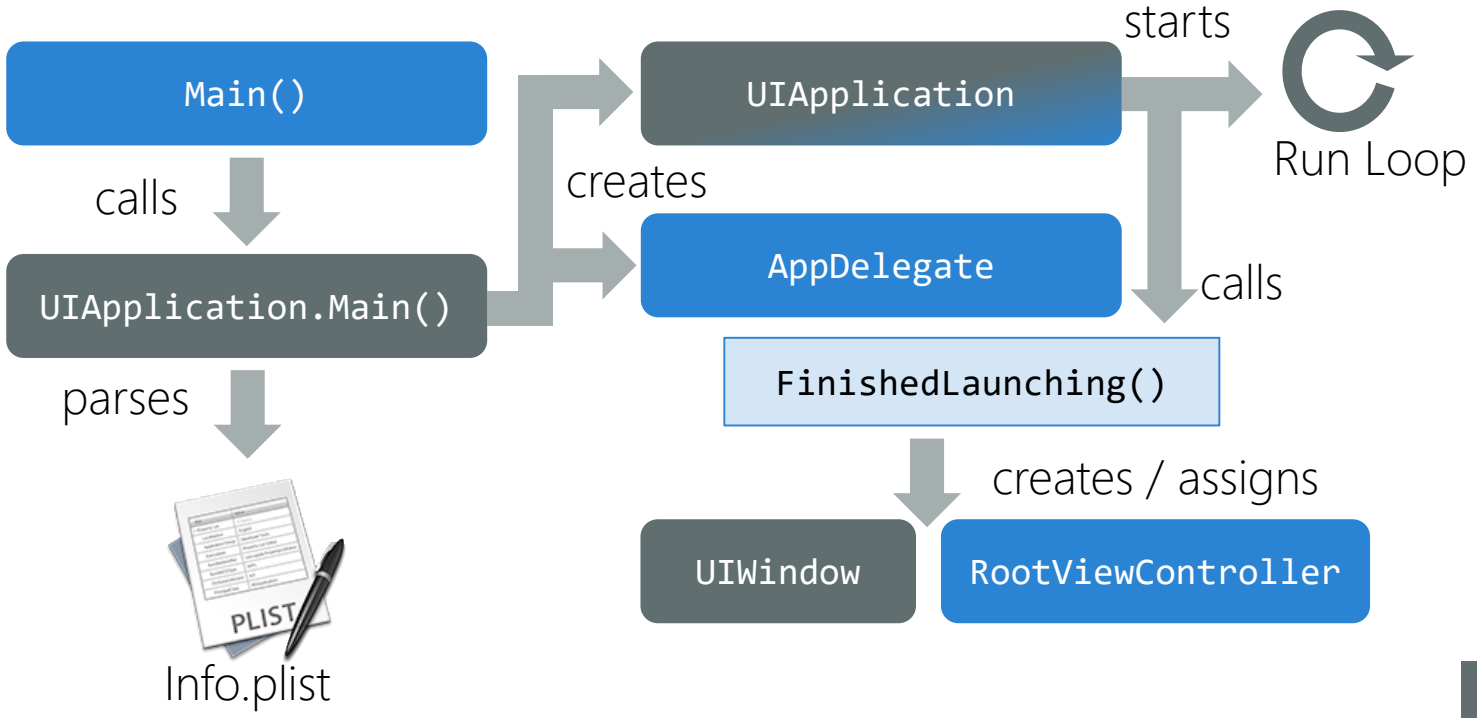
# Defining a Root Controller

- App must identify a single view controller to be the *starting controller*

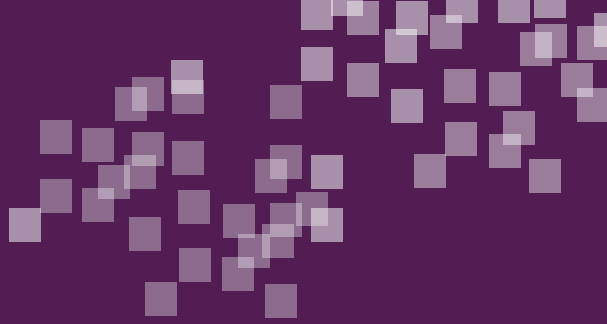
```
public class AppDelegate : UIApplicationDelegate
{
    public override UIWindow Window { get; set; }

    public override bool FinishedLaunching(...) {
        Window = new UIWindow(UIScreen.MainScreen.Bounds);
        Window.RootViewController = new MyViewController();
        Window.MakeKeyAndVisible();
        return true;
    }
...
```

# Putting it all together




iOS
custom

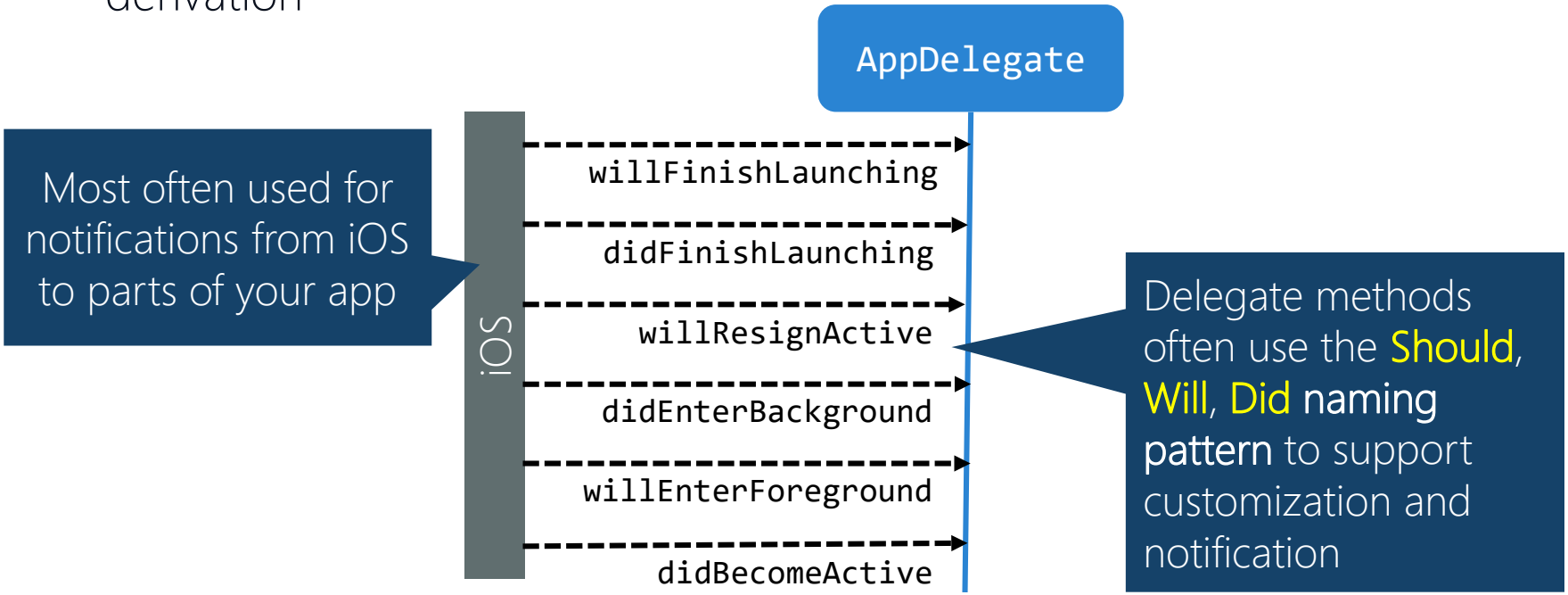


# Individual Exercise

Add a root view controller to the Tip Calculator app

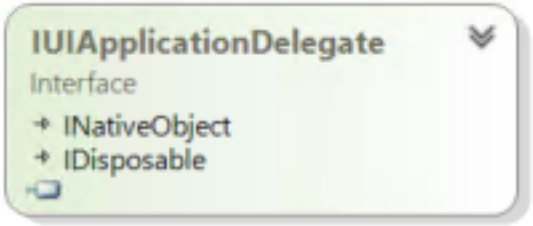
# What is a delegate in iOS?

- iOS uses a *delegation pattern* to provide behavior for classes without derivation



# What is a protocol?

- The operations (messages) a delegate can support are defined by a **protocol**; this defines the contract for the delegate and is similar to an interface in C#

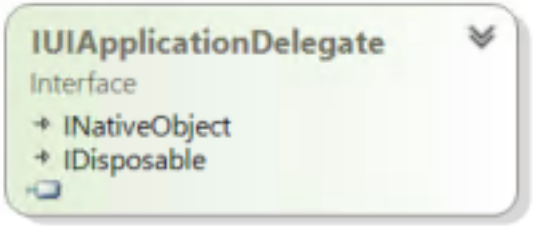



For example, the native **Application Delegate** protocol has an interface definition for C# usage

- ✓ Like interfaces in .NET – iOS objects can implement (**conform**) to multiple protocols and interact with different system services

# What is a protocol?

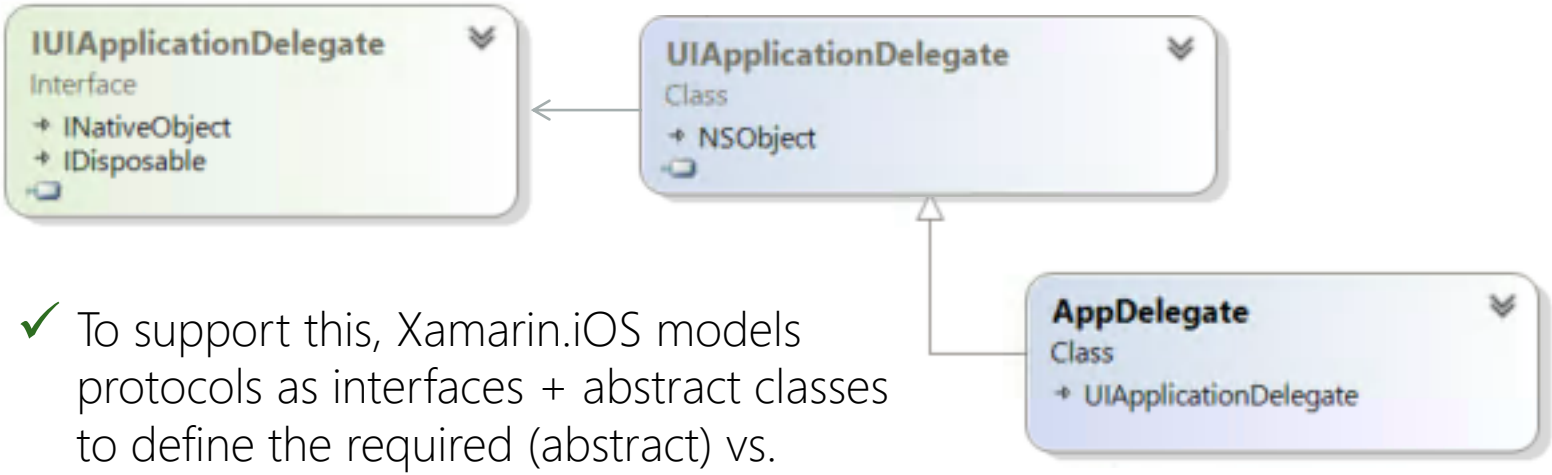
- The operations (messages) a delegate can support are defined by a **protocol**; this defines the contract for the delegate and is similar to an interface in C#




- … But, unlike interfaces, protocols support **optional and static methods** which *cannot be defined* on an interface

# What is a protocol?

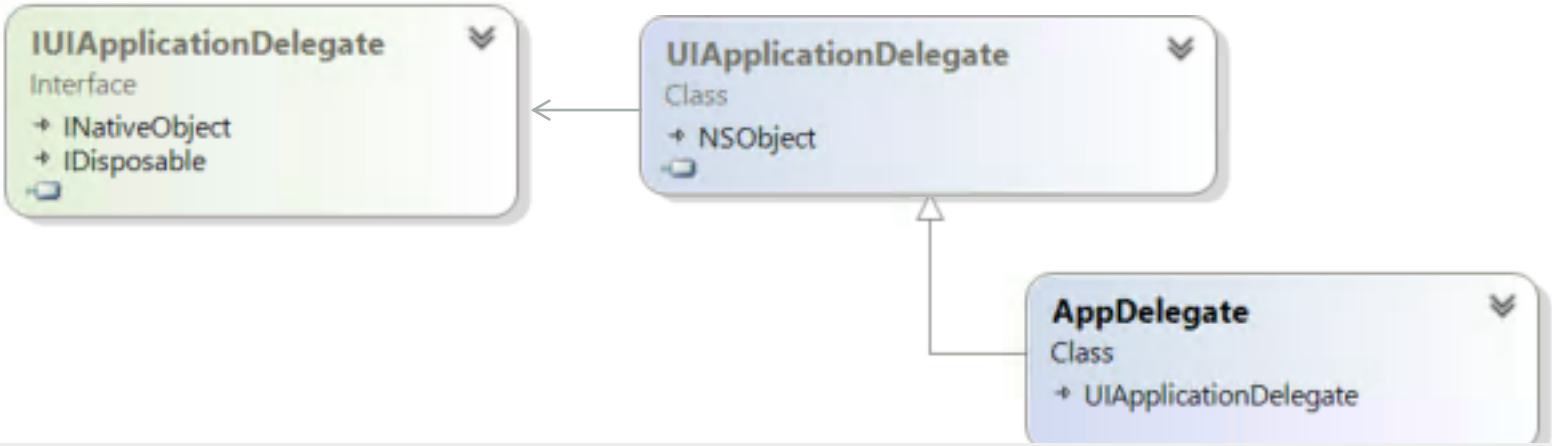
- The operations (messages) a delegate can support are defined by a **protocol**; this defines the contract for the delegate and is similar to an interface in C#




- ✓ To support this, Xamarin.iOS models protocols as interfaces + abstract classes to define the required (abstract) vs. optional (virtual) methods

# What is a protocol?

- The operations (messages) a delegate can support are defined by a **protocol**; this defines the contract for the delegate and is similar to an interface in C#




Always treat protocols like interfaces: method and property implementations should not call the base class as it often will not have one and will throw an exception

# Flash Quiz

Xamarin University

# Flash Quiz

① What file is responsible for creating the window, and listening to operating system events?

a) ViewController.cs

b) Main.storyboard.cs

c) Main.cs

d) AppDelegate.cs

# Flash Quiz

① What file is responsible for creating the window, and listening to operating system events?

a) ViewController.cs

b) Main.storyboard.cs

c) Main.cs

d) AppDelegate.cs

# Flash Quiz

② Visual screens can be created through _____.

a) Storyboard
b) XIB file
c) Code
d) All of the above

# Flash Quiz

② Visual screens can be created through _____.

a) Storyboard
b) XIB file
c) Code
d) All of the above

# Flash Quiz

③ Where can you set application properties such as application name, icons, and launch images?

a) AssemblyInfo.cs
b) Main.storyboard
c) Info.plist
d) Entitlements.plist

# Flash Quiz

③ Where can you set application properties such as application name, icons, and launch images?

a) AssemblyInfo.cs

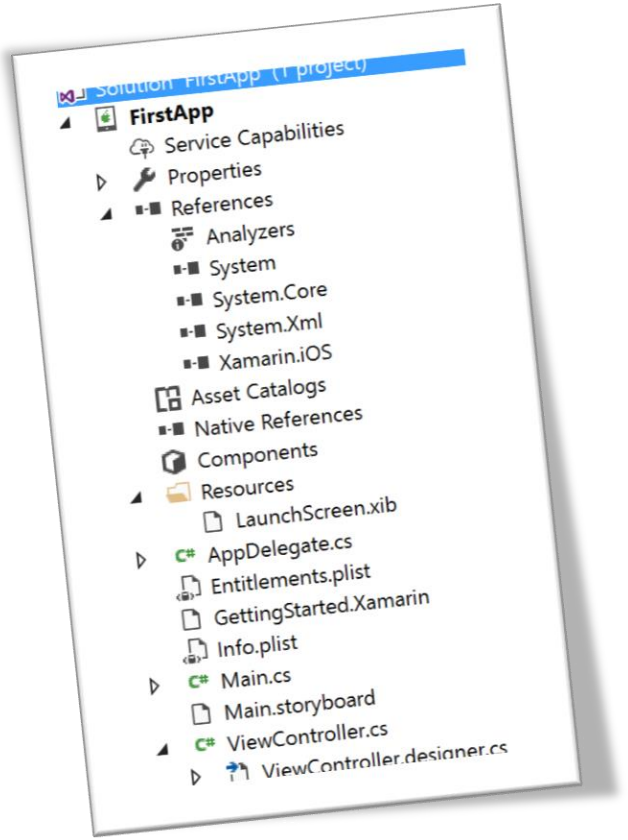
b) Main.storyboard

c) Info.plist

d) Entitlements.plist

# Summary

1. Explore a new project
2. Model-View-Controller
3. Delegates and Protocols

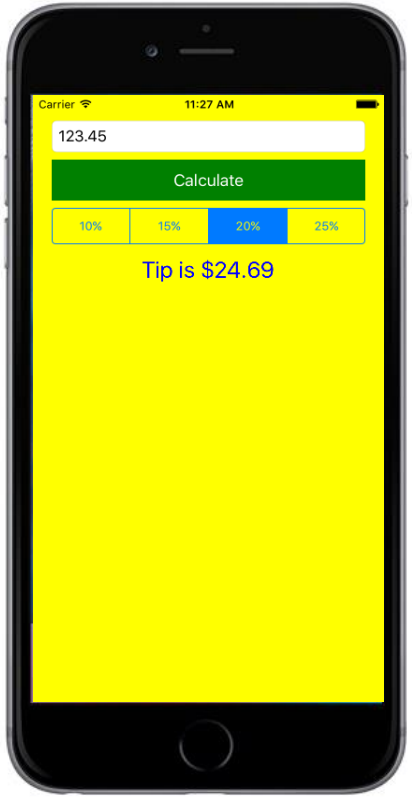


# Add views and behavior

Xamarin University

# Tasks

1. Create screens
2. Manage parent-child relationships
3. Position views
4. Add behavior



# How do you create screens?

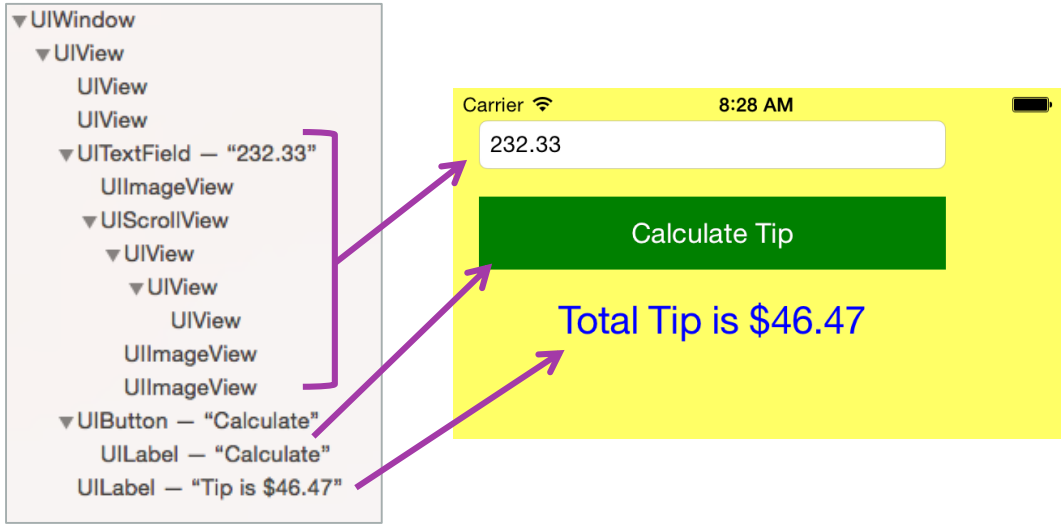
- Screens can be created in code or through the GUI designer


Going to focus the code approach in this class


cover the iOS designer in **iOS102 and iOS300**

# Creating screens 101

- Screens are created through *composition* – each screen is defined by a root parent **`UIView`** (superview) with children (subviews) placed at specific coordinates



# Accessing the view

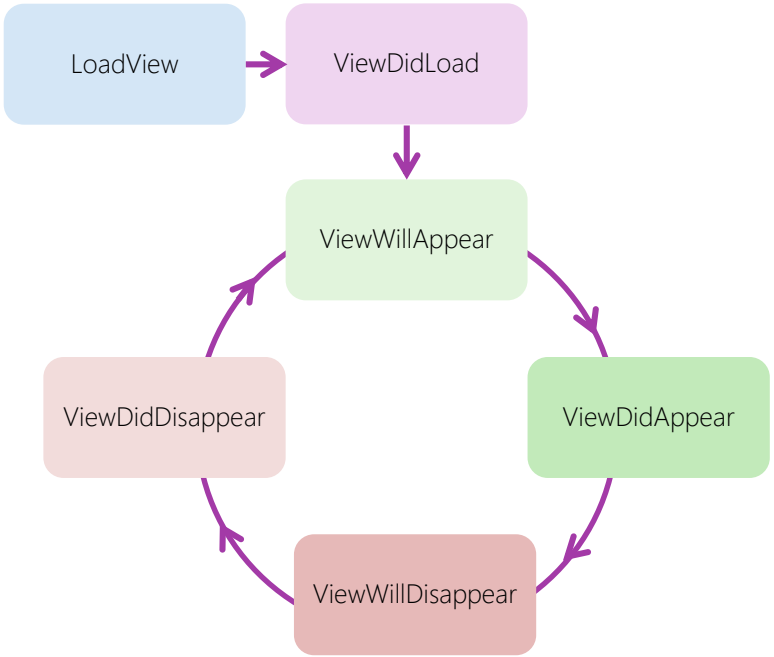
- The View Controller has a **`View`** property which provides access to the root view for the screen and virtual methods which are called at various points in the root view's lifetime

```
public partial class ViewController : UIViewController
{
      ...
      public override void ViewDidLoad()
      {
         base.ViewDidLoad();
         this.View.BackgroundColor = UIColor.Yellow;
      }
}
```

**`ViewDidLoad`** is called after the view is created – this is the place to add new controls and set any visual properties

# Big Picture: view lifecycle

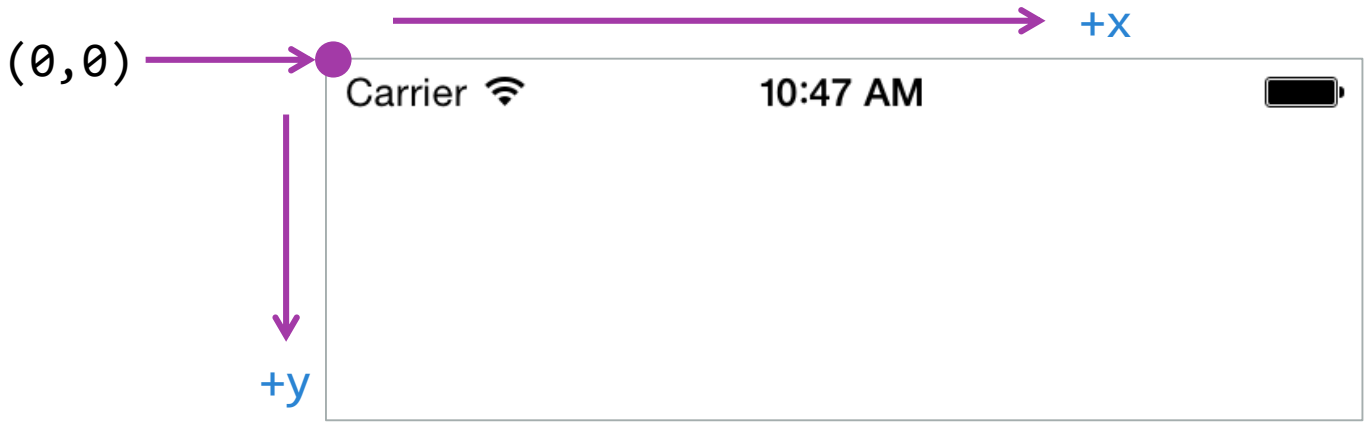
- View controller is notified as the root view is loaded, shown and hidden




| Method | Called when |
|---|---|
| `LoadView` | Creates the root view |
| `ViewDidLoad` | View created / loaded |
| `ViewWillAppear` | View about to be shown |
| `ViewDidAppear` | View has been rendered |
| `ViewWillDisappear` | View about to be hidden |
| `ViewDidDisappear` | View has been hidden |

# Coordinates

- Coordinates are specified as resolution-independent floating point values with **(0,0)** being the top-left corner of the parent **`UIView`**'s location and positive values moving right and down



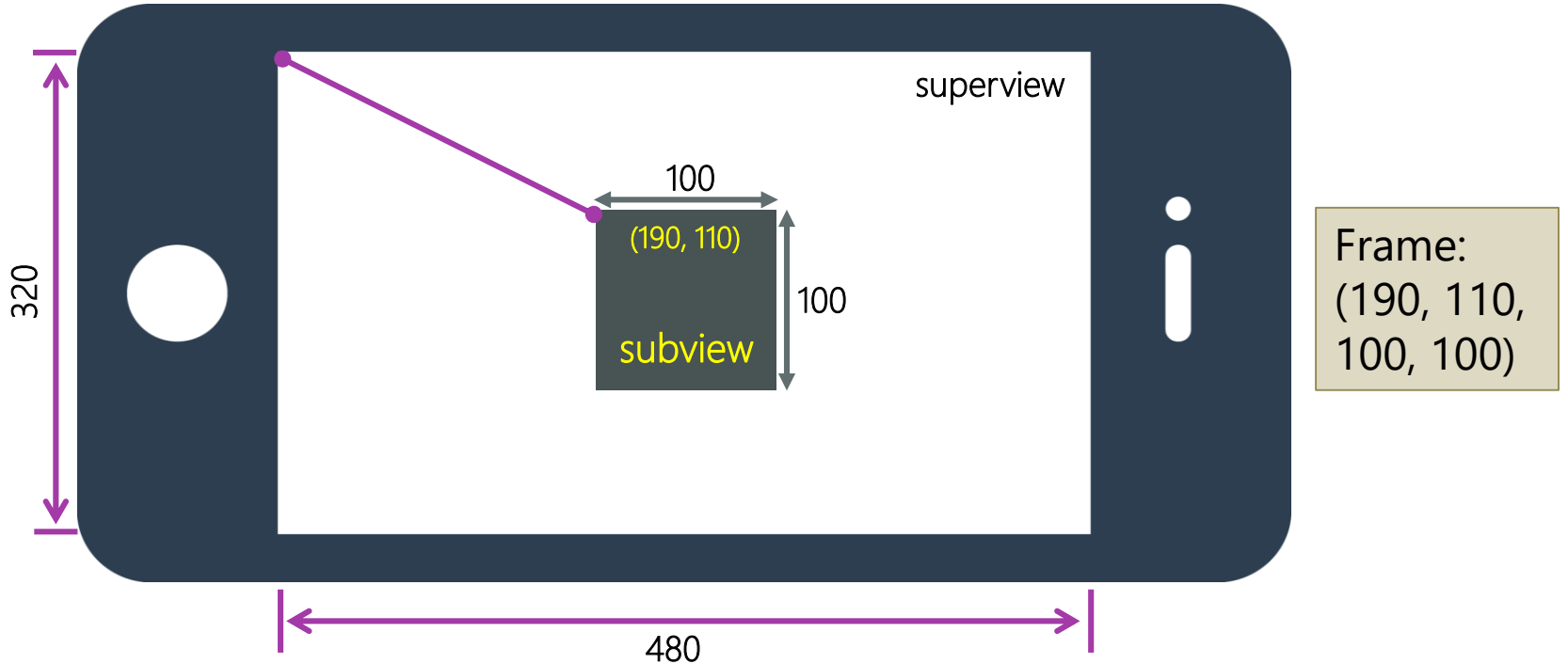
# Positioning subviews

- The position of a child view is decided by several properties

| Property | Definition |
|---|---|
| `Frame` | This is the rectangle (X, Y, width, height) for the view defined in the coordinate system of the superview (parent) and decides the overall drawing area that the view is allowed to render within |
| `Center` | This is the center point (X,Y) for the view in the superview coordinate system. |
| `Bounds` | This is the rectangle of the view in it's own coordinate system. Often this is similar to the frame, however it might not include the space used for margins or shadow effects applied. |

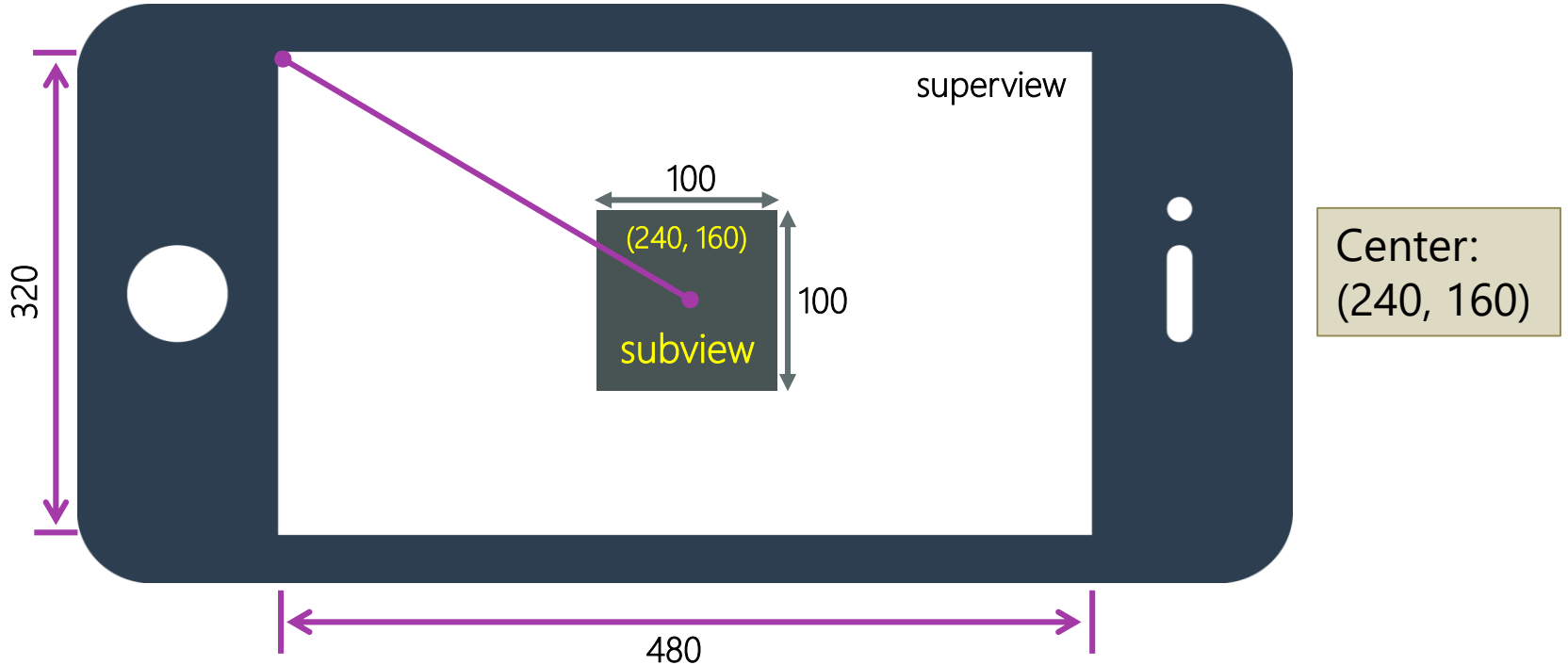
# Positioning subviews

- `Frame` defines the view position and size in superview coordinates



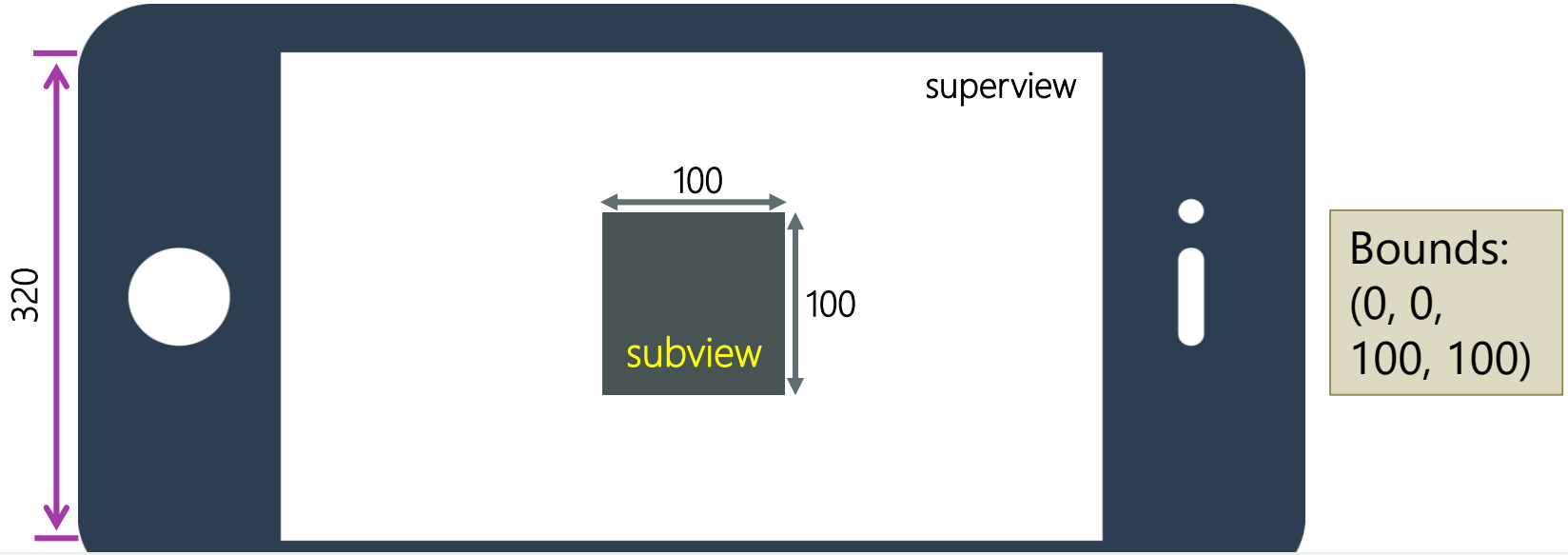
# Positioning subviews

- **`Center`** defines the center point of the view in superview coordinates



# Positioning subviews

- **Bounds** defines the position and size of the view in it's own coordinates



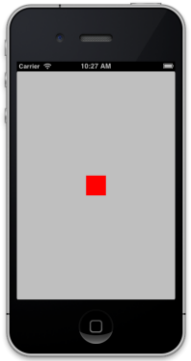

These positioning properties all normally change the same internal values, so setting the **Bounds** + **Center** is the same as setting the **Frame** and vice-versa

# Setting a view's position in code

- You will use both **Frame** and **Bounds** depending on the context – when you are positioning the view within the parent, use **Frame** and when you are working in the current view coordinates, use **Bounds**

```
public override void ViewDidLoad()
{
    nfloat height = View.Bounds.Height;  // Current view coordinates
    nfloat width = View.Bounds.Width;

    var subview = new UIView() {
        Frame = new CGRect(width/2-20, height/2-20, 40,40)
    };
    ...
}
```



# Flash Quiz

Xamarin University

# Flash Quiz

① When creating a subview, you should set the _____ property to set it's position and size

a) Center
b) Bounds
c) Frame
d) Rectangle

# Flash Quiz

① When creating a subview, you should set the _____ property to set it's position and size

a) Center
b) Bounds
c) Frame
d) Rectangle

# Flash Quiz

② If you change the Center property, the Frame property will also change

a) True

b) False

# Flash Quiz

② If you change the Center property, the Frame property will also change

a) True

b) False

# Flash Quiz

③ A Frame's (0,0) is always the top-left corner of the screen

a) True

b) False

# Flash Quiz

③ A Frame's (0,0) is always the top-left corner of the screen

a) True

b) False

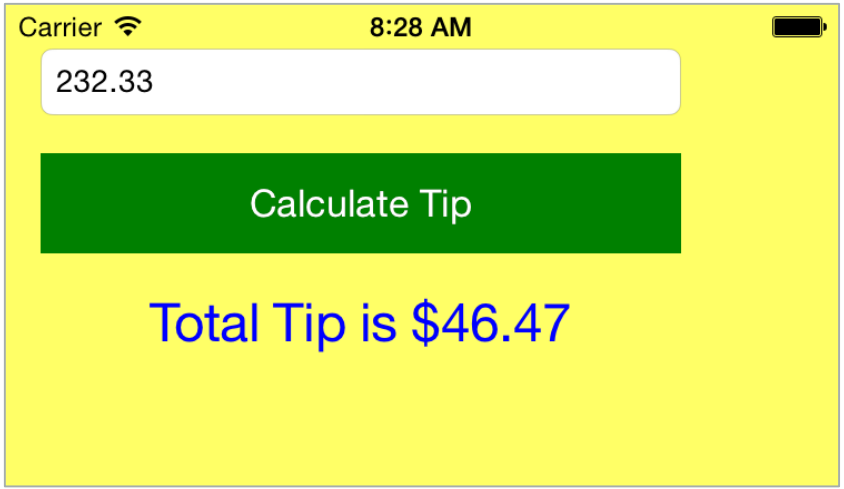
# Controls in iOS

- iOS defines standard controls (views) in the **UIKit** framework that you will use when creating your application screens
- These ultimately derive from **`UIView`**; you create them, set the **`Frame`** and add to a superview to display it

| Windows | Android | iOS |
|---|---|---|
| Button | Button | UIButton |
| CheckBox | CheckBox | UISwitch |
| ComboBox | Spinner | UIPickerView |
| Image | ImageView | UIImageView |
| Label | TextView | UILabel |
| ListBox | ListView | UITableView |
| ProgressBar | ProgressBar | UIProgressView |
| Slider | Slider | UISlider |
| TextBox | EditText | UITextField |

# Let's build a Tip Calculator UI

- `UITextField` at the top to enter the total amount
- `UIButton` to execute the tip calculation logic
- `UILabel` to display the results



# Adding entry fields

- Use **`UITextField`** to add edit controls to a screen, automatically displays an on-screen keyboard when control is tapped

```
UITextField emailEntry = new UITextField() {
      Frame = new CGRect(10, 20, View.Bounds.Width - 20, 35),
      KeyboardType = UIKeyboardType.EmailAddress,
      BorderStyle = UITextBorderStyle.RoundedRect,
      Placeholder = "Email Address"
};
```

# Adding buttons

- Use **`UIButton`** to add buttons to a screen – standard button type only displays title with no border or background color

supply the button type to the constructor

```
UIButton button = new UIButton(UIButtonType.Custom) {
    Frame = new CGRect(....),
    BackgroundColor = UIColor.FromRGB(0.5f, 0, 0),
};
button.SetTitle("Login", UIControlState.Normal);
```

Must call method to set the title – can set different text values for different button states (Normal, Highlighted, Disabled, etc.)

# Adding text

- Use the **UILabel** control to add read-only text to a screen

```
UILabel label = new UILabel(new CGRect(190, 110, 100, 35)) {
    Text = "This is a label",
    TextAlignment = UITextAlignment.Center,
    TextColor = UIColor.Blue
};
```

Properties control appearance

**Frame** can be set through constructor parameter when creating most **UIView** types

# Adding subviews to the screen

- View manages a collection of subviews to display which are rendered in the order you add them to the collection (bottom-up)

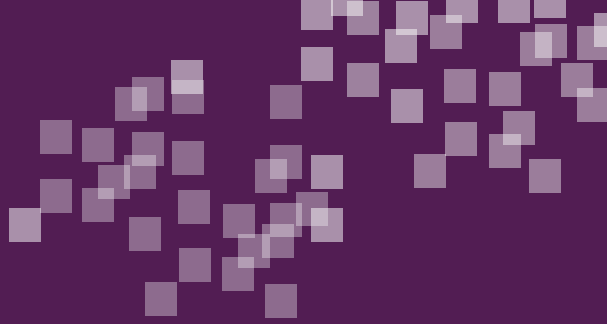
```
public override void ViewDidLoad()
{
    ...
    var label = new UILabel() { ... }
    var entry = new UITextField() { ... }
    var button = new UIButton() { ... }

    View.AddSubview(label);            // add one view
     // or View.Add(label)
    View.AddSubviews(entry, button);  // add multiple views
}
```

# Examining subviews

- **`UIView`** is enumerable and supports iterating through the children

```
void RemoveAllContent()
{
    foreach (UIView subview in View)
    {
        // Remove from the parent view
        subview.RemoveFromSuperview();
    }
}
```



# Individual Exercise

Create the UI for a Tip Calculator

Xamarin University

# Keyboard Dismissal

- Views do not automatically dismiss the keyboard – must *resign first responder* status on the active **`UITextField`** to hide the keyboard

```
UITextField emailEntry = ...;
...
void HideKeyboard()
{
    emailEntry.ResignFirstResponder();
}
```

# Adding behavior to a screen

- Many controls in Xamarin.iOS expose .NET events to provide interactivity notification, these are mapped on top of the event actions of the native control

```
button.TouchUpInside += delegate(object sender, EventArgs e)
```

```
entry.EditingDidBegin += delegate(object sender, EventArgs e)
entry.ValueChanged += delegate(object sender, EventArgs e)
entry.EditingDidEnd += delegate(object sender, EventArgs e)
```



Since these are regular .NET events, you can use any .NET technique to handle them, e.g. a delegate method, anonymous method or lambda expression

# Wiring up to a button

- **`UIButton`** exposes standard control **`TouchUpInside`** event to represent a tap or click – wire up to event in the **`ViewDidLoad`** override to add behavior logic

```
public override void ViewDidLoad() {
    ...
    button.TouchUpInside += OnLoginButtonClicked;
}

void OnLoginButtonClicked(object sender, EventArgs e) {
    ... // Do login logic here
}
```

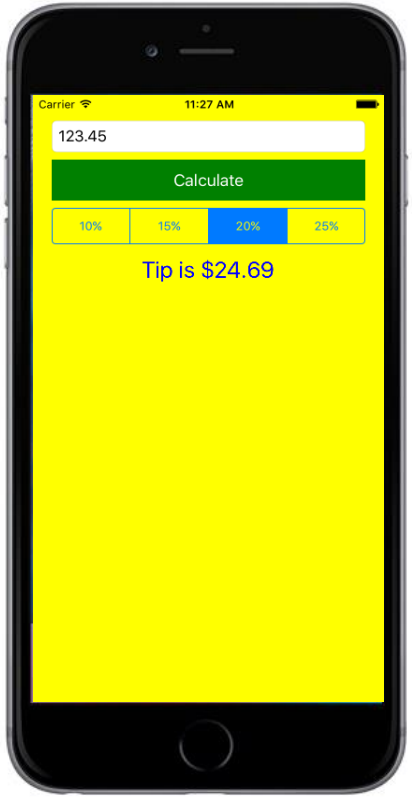
# Individual Exercise

Add logic to your Tip Calculator

Xamarin University

# Summary

1. Create screens
2. Manage parent-child relationships
3. Position views
4. Add behavior



# Next Steps

- This class has covered the basics of iOS development using the Xamarin tools
- The next class, **IOS102** covers building your Views with the iOS designer
- There is a homework assignment to continue practicing your app development skills!

# Thank You!

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