UWP101

Introduction to the Universal Windows Platform

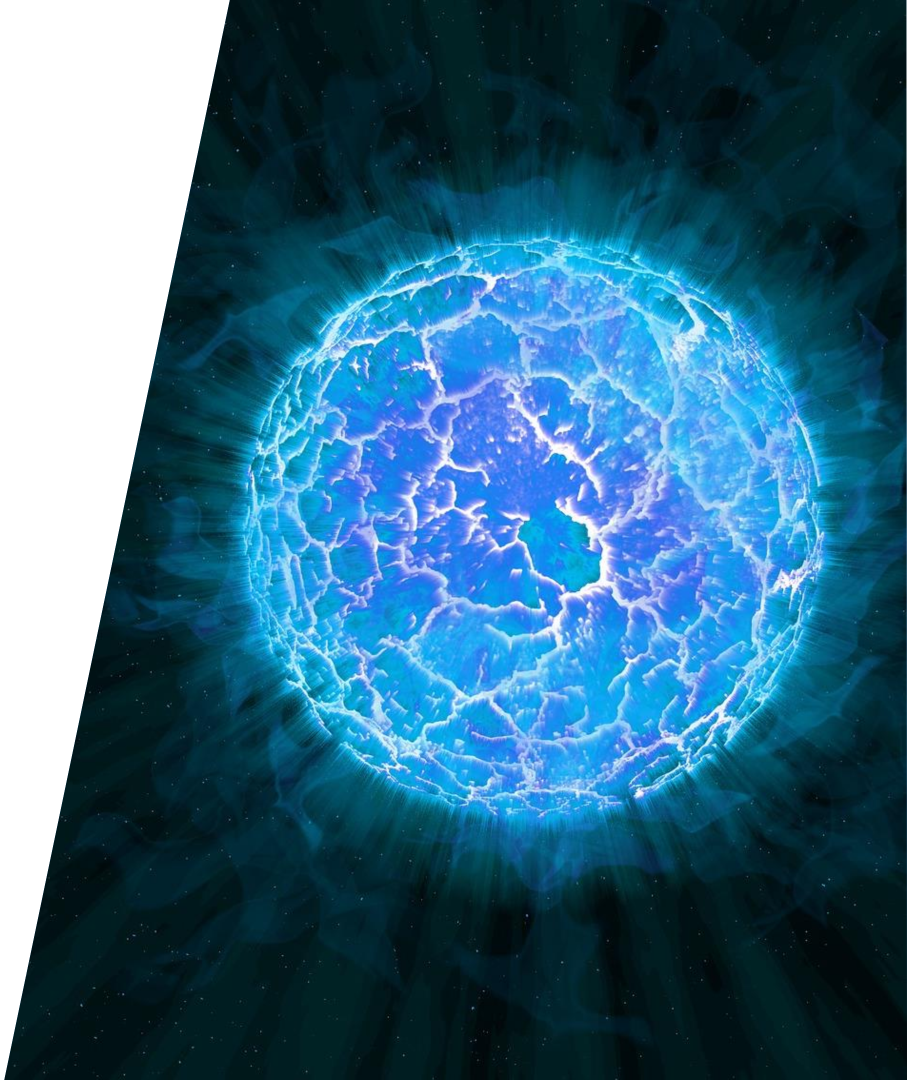
Download class materials from
university.xamarin.com

Microsoft

Xamarin University

Objectives

1. Create and run a UWP application
2. Respond to lifecycle events
3. Write platform-adaptive code
4. Write version-adaptive code



Create and run a UWP application

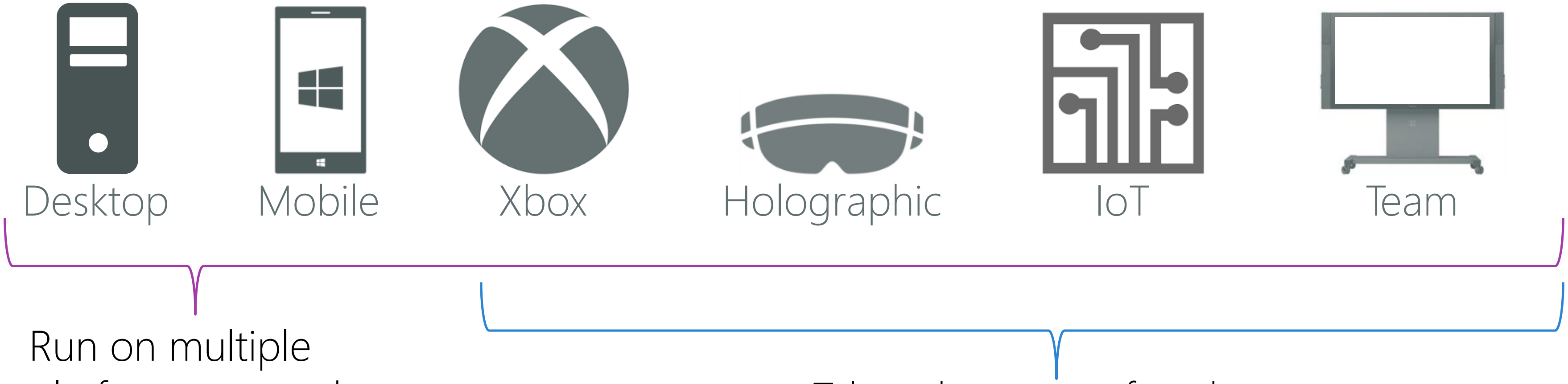
Xamarin University

Tasks

1. Explore app structure
2. Create and run a UWP application

Motivation

❖ Windows 10 runs on a wide variety of hardware platforms with diverse form factors and device capabilities



Run on multiple platforms to reach more customers

Take advantage of each form factor's characteristics

What is UWP?

- The *Universal Windows Platform* (UWP) is an application architecture that lets developers target the full range of Windows 10 devices with one app

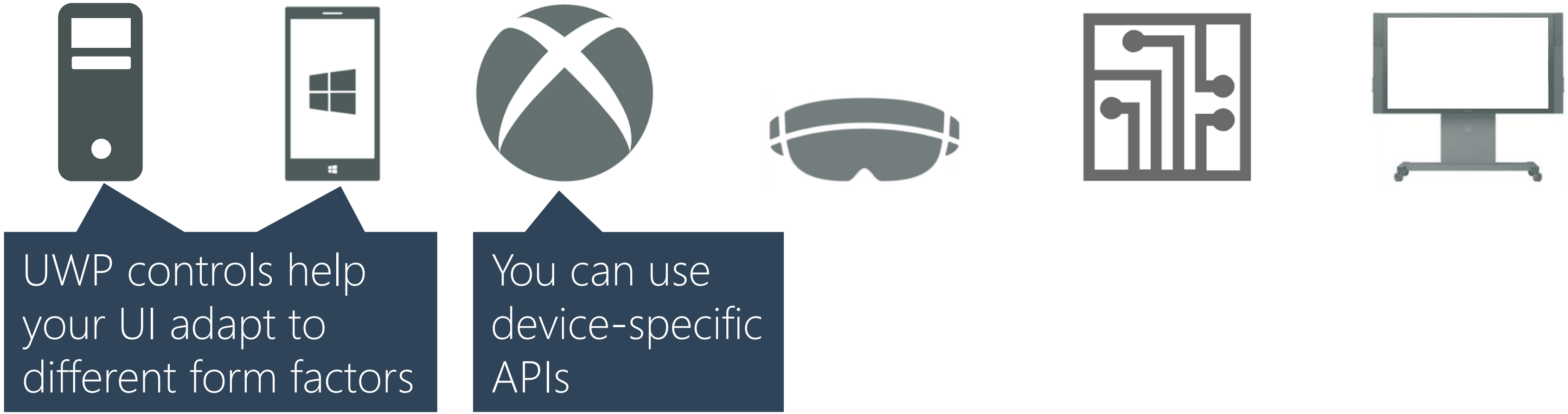
All devices

Common API

Common store

All devices

- UWP apps can run on all Windows 10 devices (although you can limit your app to specific device types if needed)

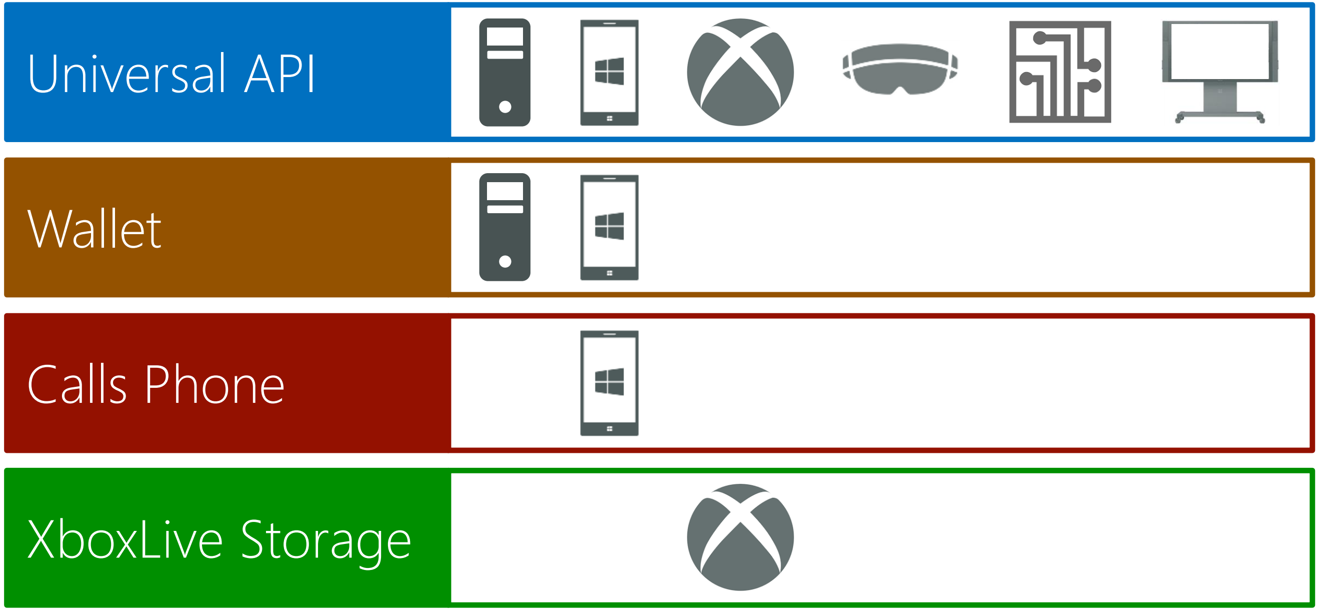


UWP controls help your UI adapt to different form factors

You can use device-specific APIs

Common API

- The UWP API consists of a universal API supported everywhere and specific APIs supported where appropriate



Universal API

Wallet

Calls Phone

XboxLive Storage

Common store

- You deploy your app to the Windows Store which then distributes it to your customers

You build a single .appx package that specifies the devices your app targets

The store offers your app only to customers with the targeted devices

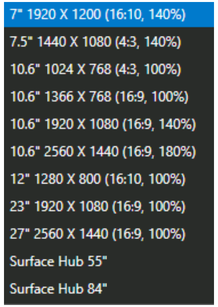
Development environment

- Microsoft tools cover the full range of app development and testing

Develop using Visual Studio 2015/2017

Use C#, C++, VB, or JavaScript

Code on Win 7/8/10 (local deployment on Win10 only)



7" 1920 X 1200 (16:10, 140%)
7.5" 1440 X 1080 (4:3, 140%)
10.6" 1024 X 768 (4:3, 100%)
10.6" 1366 X 768 (16:9, 100%)
10.6" 1920 X 1080 (16:9, 140%)
10.6" 2560 X 1440 (16:9, 180%)
12" 1280 X 800 (16:10, 100%)
23" 1920 X 1080 (16:9, 100%)
27" 2560 X 1440 (16:9, 100%)
Surface Hub 55"
Surface Hub 84"

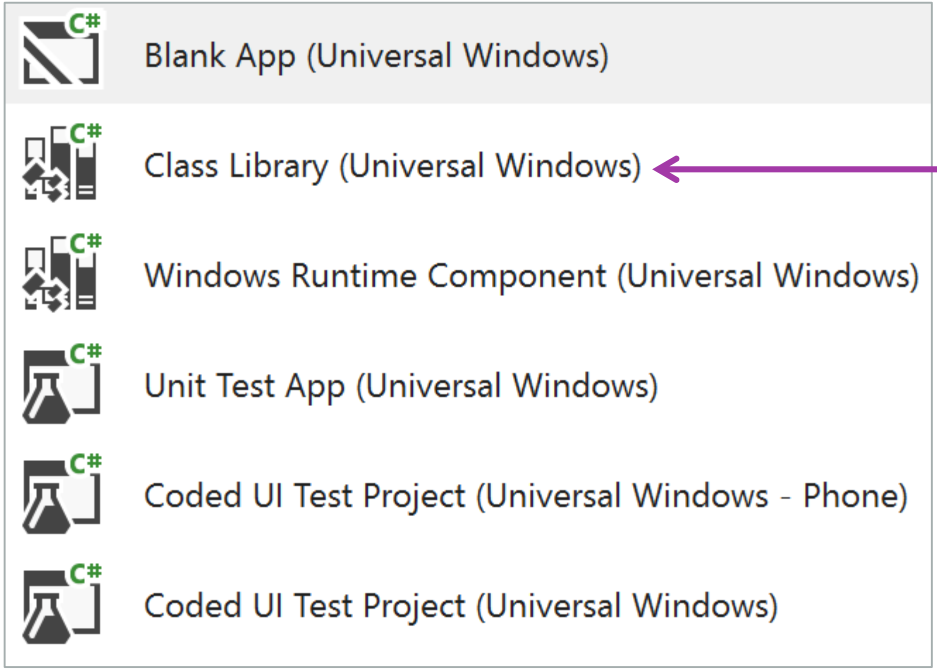
Test on emulators for various form factors

Project templates

- Visual Studio offers several UWP project templates

App with minimal starter code

Several testing projects



Blank App (Universal Windows)

Class Library (Universal Windows)

Windows Runtime Component (Universal Windows)

Unit Test App (Universal Windows)

Coded UI Test Project (Universal Windows - Phone)

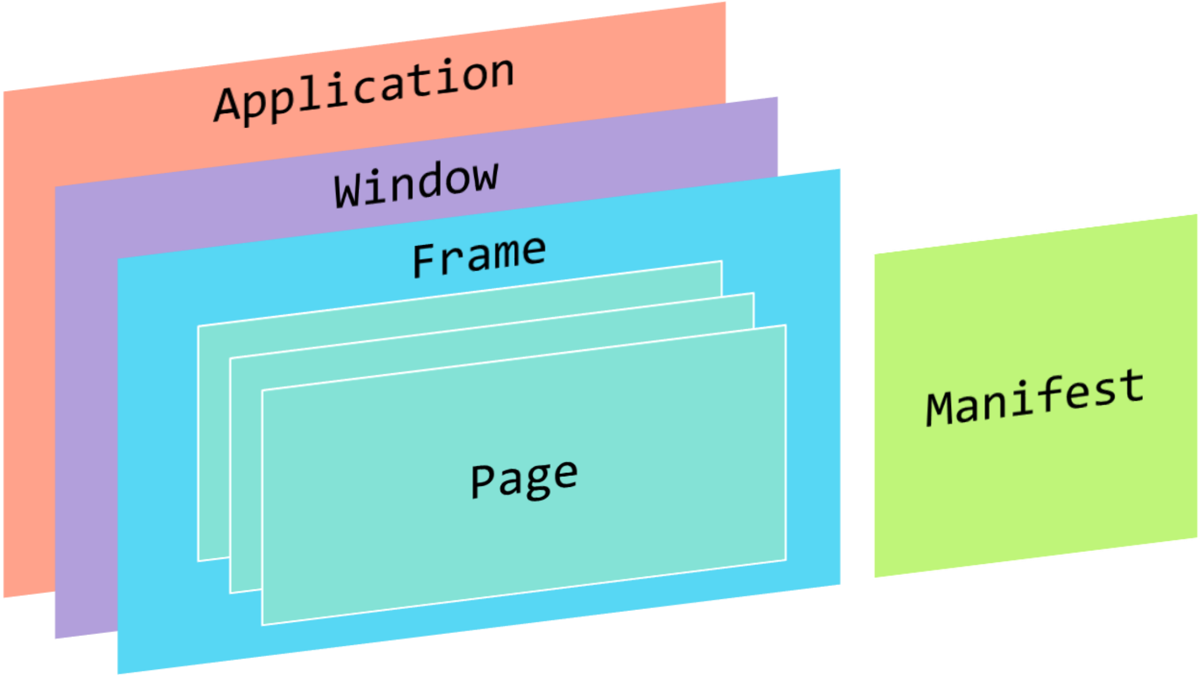
Coded UI Test Project (Universal Windows)

Library to share with C#/VB UWP apps

Library to share with UWP apps coded in all languages

Application structure

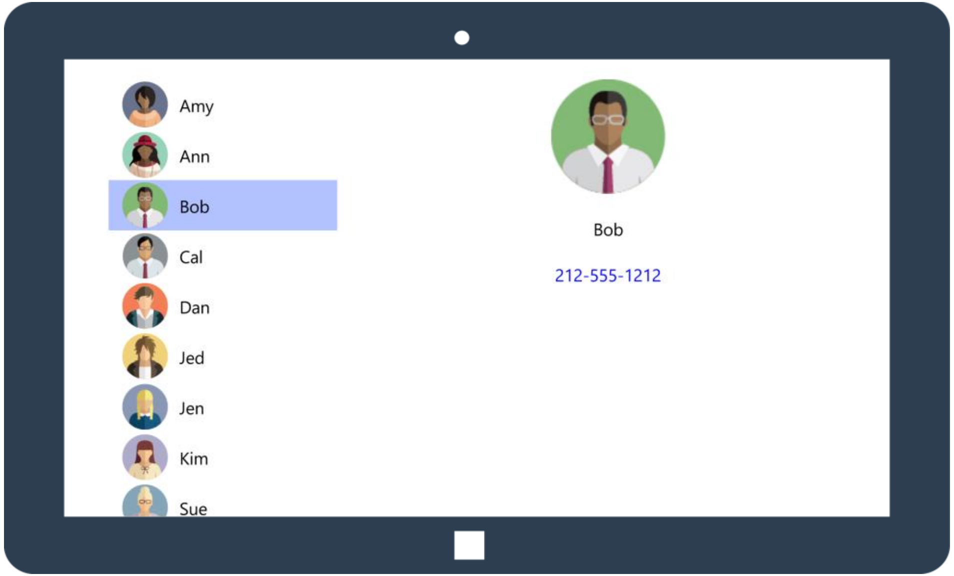
- UWP applications are made up of several parts



The Page class

- A **Page** implements the UI and behavior for a specific app feature

The UI contains layout panel(s) and controls and generally occupies the entire screen →



Your app's pages

- Typically, you will create one **Page**-derived class for each screen of content in your app

```
public partial class ContactsPage : Page { ... }
public partial class DetailsPage  : Page { ... }
public partial class EditPage     : Page { ... }
```

A *Contacts* app might have these pages

Pages must derive from the **Page** class

Page implementation

- The definition of each page is typically split across two files

ContactsPage.xaml

```
<Page x:Class="MyContacts.ContactsPage">
  <Grid>
    <ListView...>...</ListView>
    ...
  </Grid>
<Page>
```

UI declared in XAML (can use code but it is not common)

ContactsPage.xaml.cs

```
public partial class ContactsPage : Page
{
   ...
}
```

Behavior implemented in code-behind file

The Frame class

❖ The **`Frame`** class represents an area of your UI that shows a single **`Page`** and implements forward/back navigation

You navigate the Frame to new Pages to change your UI

```
public class Frame : ContentControl, INavigate
{  ...
   public bool Navigate(Type sourcePageType);
   public bool Navigate(Type sourcePageType, object parameter)

   public bool CanGoBack    { get; }
   public bool CanGoForward { get; }
   public bool GoBack();
   public bool GoForward();
}
```

Frame maintains the navigation history using a classic navigation paradigm: it behaves like your web browser

Frame navigation

- You navigate the Frame in response to user action

```
public class DetailsPage : Page
{  ...
   void OnBackClick(object sender, RoutedEventArgs e)
   {
      if (base.Frame.CanGoBack)
         base.Frame.GoBack();
   }

   void OnEditClick(object sender, RoutedEventArgs e)
   {
     base.Frame.Navigate(typeof(EditPage), currentContactId);
   }
}
```

Page has a **Frame** property to let you access the frame that is hosting it

Destination

Parameter

The Window class

- The **`Window`** class represent the user-viewable window of a UWP app

```
public sealed class Window
{  ...
   public UIElement Content { get; set; }

   public void Activate();

   public static Window Current { get; }
}
```

You load this with the UI you want to display (usually a Frame)

You call this when your UI is loaded and ready

Used to access your app's Window

Application class

- The `Application` class encapsulates your application – the Blank App template generates a subclass named **App** for you

Entry point

Lifecycle management

App-wide resources

Unhandled exceptions

App startup code

- At startup, your **App** class loads the Window's UI and navigates to your home page

Called at startup

Create the UI

Load and show the Window

```
partial class App : Application
{  ...
   protected override void OnLaunched(LaunchActivatedEventArgs e)
   {
      var frame = new Frame();
      frame.Navigate(typeof(ContactsPage));

      Window.Current.Content = frame;
      Window.Current.Activate();
   }
}
```

App manifest

- Each UWP app has an XML manifest named **`Package.appxmanifest`** that contains app metadata for use by the Store and Windows

Identity

App name, version, publisher, etc.

Visual Assets

Logo, splash screen, etc.

Capabilities

Needed resources (e.g. app must access Webcam, Bluetooth, Pictures library, etc.)

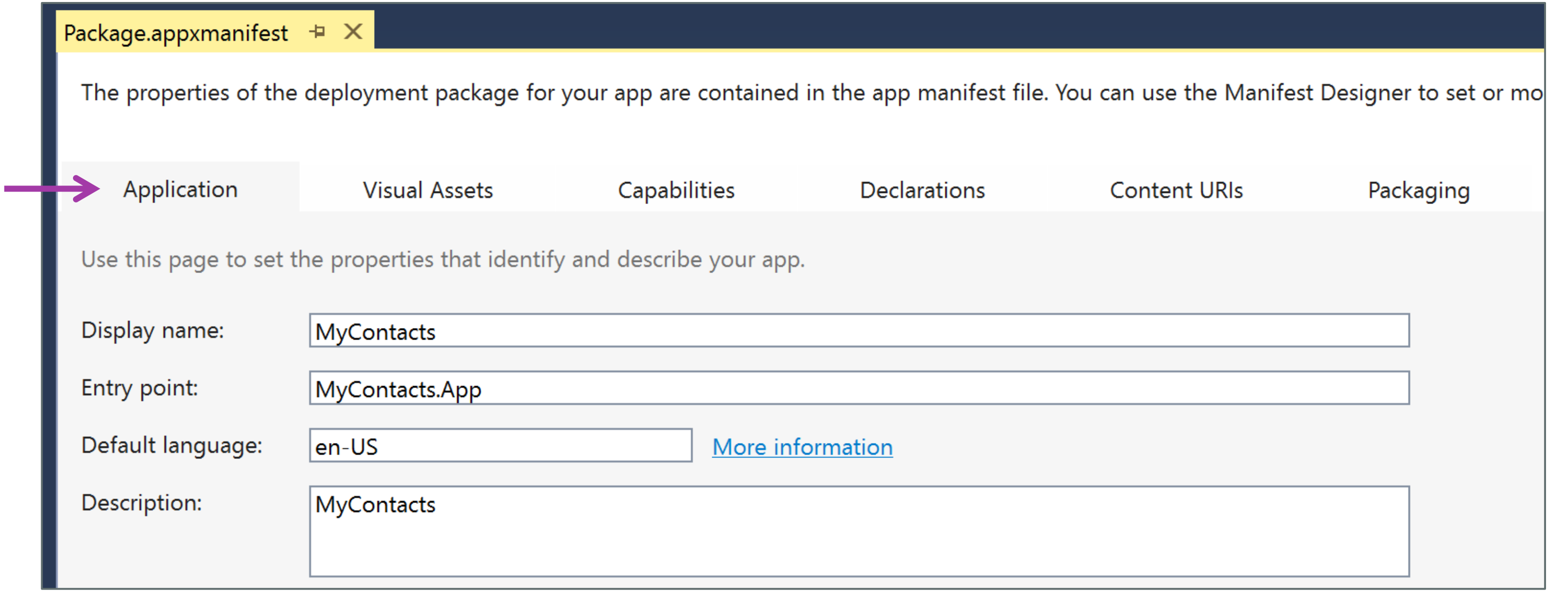
Declarations

Provide services to other apps (e.g. Share Target, Account Picture Provider, AutoPlay support, etc.)

Manifest editor

- Visual Studio contains a GUI editor for `Package.appxmanifest`

Configure your app's properties



The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or mo

Use this page to set the properties that identify and describe your app.

Display name: MyContacts

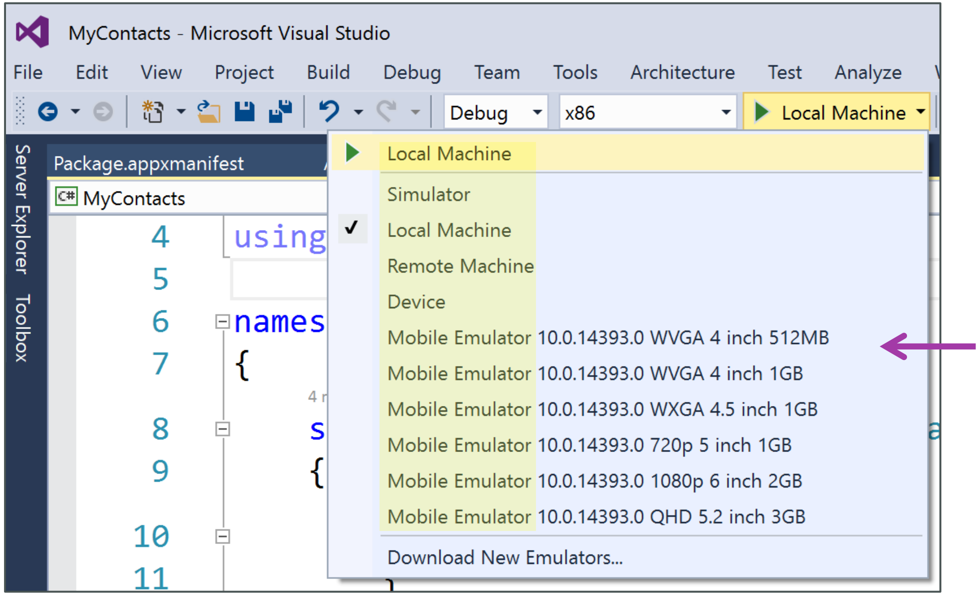
Entry point: MyContacts.App

Default language: en-US More information

Description: MyContacts

Deployment target

- Use the **Standard** toolbar to select your deployment target



MyContacts - Microsoft Visual Studio
File
Edit
View
Project
Build
Debug
Team
Tools
Architecture
Test
Analyze
Debug
x86
Local Machine
Package.appxmanifest
MyContacts
Server Explorer
Toolbox
Local Machine
Simulator
✓ Local Machine
Remote Machine
Device
Mobile Emulator 10.0.14393.0 WVGA 4 inch 512MB
Mobile Emulator 10.0.14393.0 WVGA 4 inch 1GB
Mobile Emulator 10.0.14393.0 WXGA 4.5 inch 1GB
Mobile Emulator 10.0.14393.0 720p 5 inch 1GB
Mobile Emulator 10.0.14393.0 1080p 6 inch 2GB
Mobile Emulator 10.0.14393.0 QHD 5.2 inch 3GB
Download New Emulators...

Can run on the local machine, a connected device, or one of many different emulators

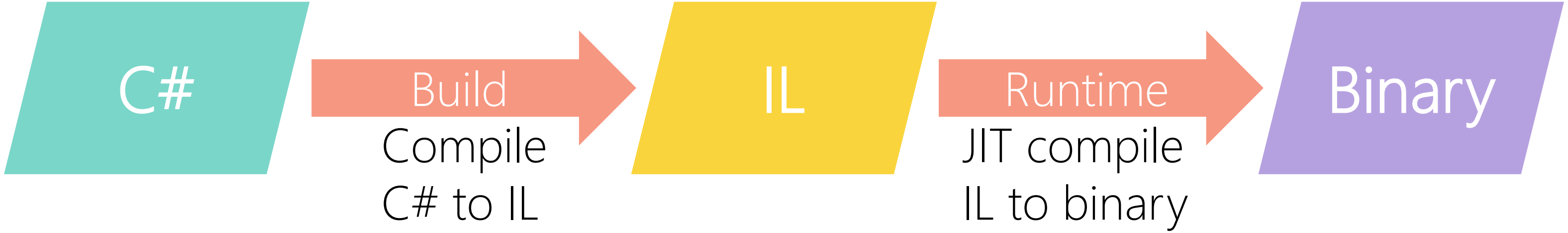
Save the project to the local disk – you will get an error if you run from a network drive

Individual Exercise

Create and run a UWP application

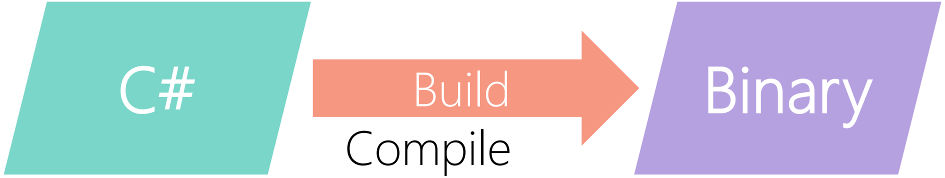
Debug builds

- Debug build produces an Intermediate Language (IL) executable – the CLR then translates the IL to binary at runtime



Release builds

- Release build produces a binary executable using the **.NET Native** tool chain

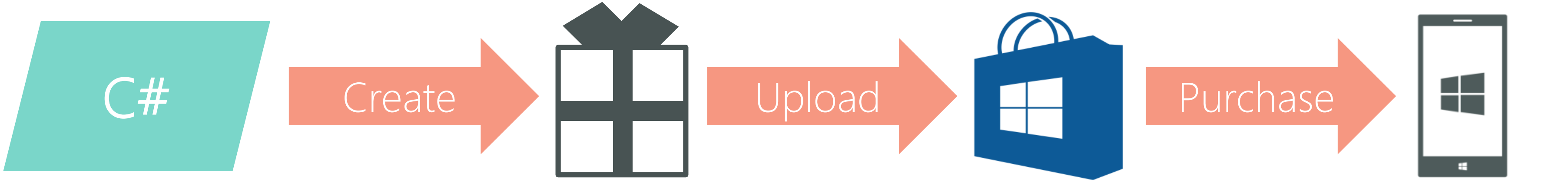


Compile C# directly to binary

Test your apps periodically in "release mode" to identify bugs that may occur during the .NET Native compilation process – this is the environment your users will experience

Store build

- To deploy to the Store, you use Visual Studio to create an `.appxupload` package (Project → Store → Create App Packages…)



Package contains IL

IL is compiled to native code in the Store

Native code installed on device

Summary

1. Explore app structure
2. Create and run a UWP application

Respond to lifecycle events

Xamarin University

Tasks

1. Determine previous execution state
2. Save state when entering the background
3. Restore state at startup if appropriate

App launch and shutdown

- Windows imposes some rules on the lifecycle of UWP apps

Single instance

Only one copy running at a time

Suspension

Inactive apps may be denied CPU time

Termination

Apps may be closed to reclaim resources

Single instance

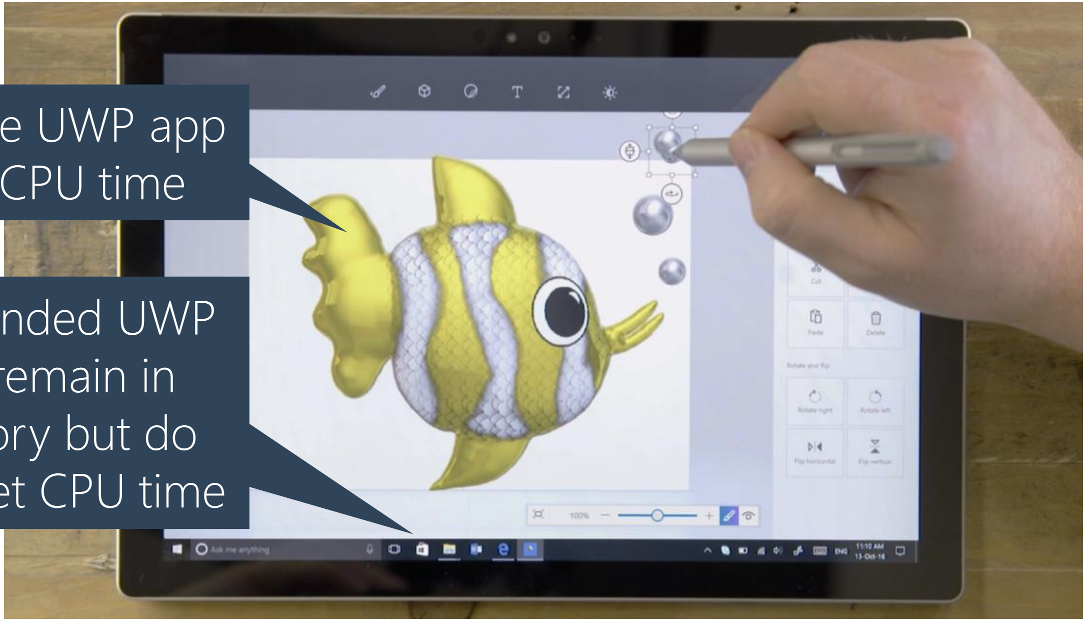
- UWP apps are single instance – launching an app that's already running does not create a new copy

```
partial class App : Application
{  ...
   protected override void OnLaunched(LaunchActivatedEventArgs e)
   {
      ...
   }
}
```

This method will be called again inside the already-running app

Suspension

- Windows devotes resources to active apps and suspends other apps to conserve power and improve responsiveness



Active UWP app gets CPU time

Suspended UWP apps remain in memory but do not get CPU time

When do apps get suspended?

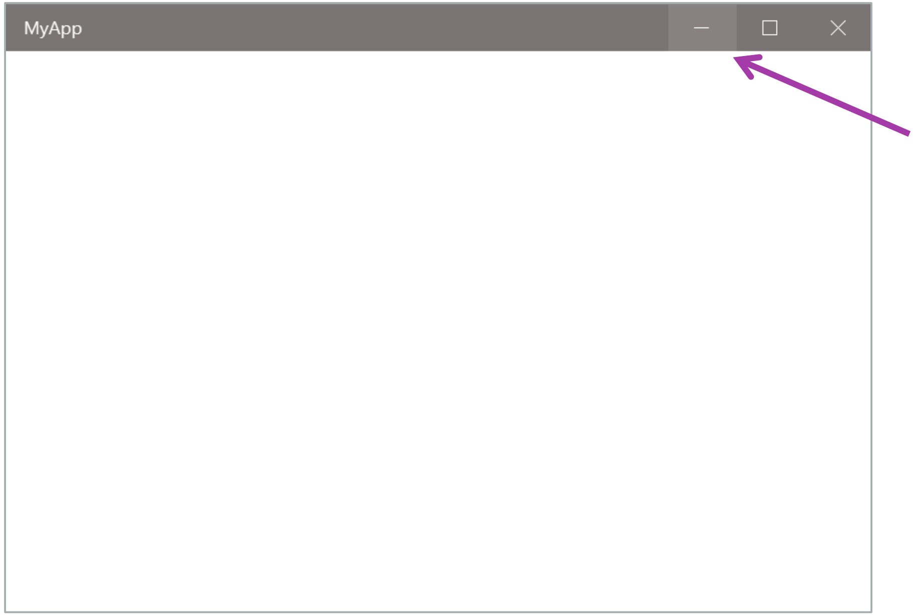
- Intuitively, an app becomes eligible for suspension when the user is no longer working with it – the exact conditions vary by device

On the desktop, an app becomes inactive when minimized or the screen is locked

On the phone, an app becomes inactive when the user switches to the home screen or another app

Termination

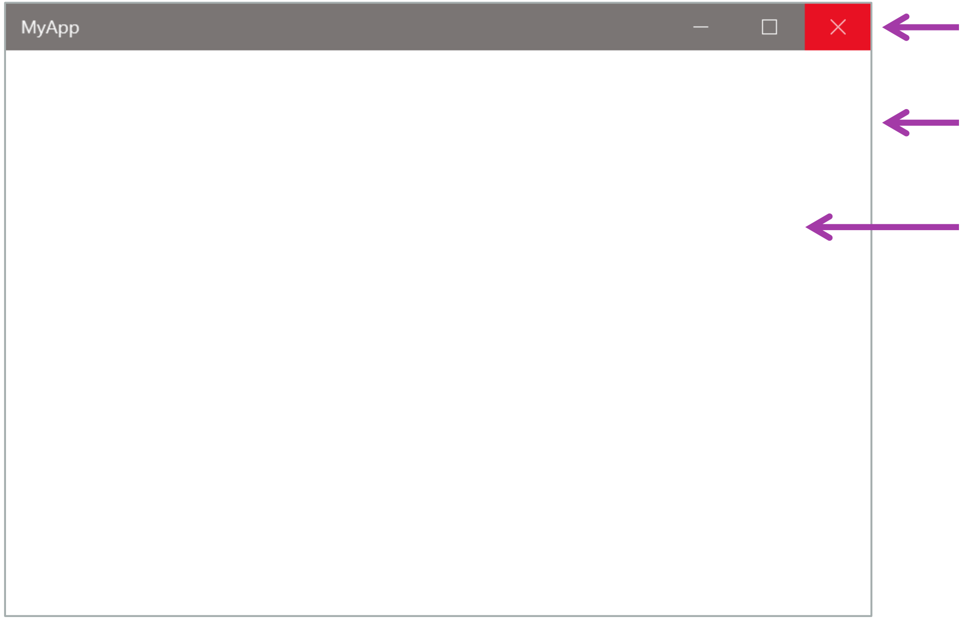
- Windows may close a suspended app if Windows needs to reallocate resources to other apps



User minimizes the app and later Windows terminates it

Voluntary shutdown

- The user may explicitly close an application



Button

ALT+F4 keys

Gesture (drag to bottom on touch devices)

Previous execution state

- Windows records how an app was shut down the last time it ran and gives this information to the app the next time it runs

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
   ApplicationExecutionState last = e.PreviousExecutionState;

   switch (last)
   {
      case ApplicationExecutionState.NotRunning:   ... break;
      case ApplicationExecutionState.Running:      ... break;
      case ApplicationExecutionState.Suspended:    ... break;
      case ApplicationExecutionState.Terminated:   ... break;
      case ApplicationExecutionState.ClosedByUser: ... break;
   }
}
```

Passed to the activation method in the arguments

How to generate each state

- Generally, user action during the app's last run determines that app's `PreviousExecutionState`

State	Explanation
`NotRunning`	App just installed, app crashed, killed with Task Manager
`Suspended`	User relaunched a suspended app
`Terminated`	App was suspended and then closed by Windows
`ClosedByUser`	User ended the app with the Close button, ALT+F4, etc.
`Running`	User relaunched an already-running app

This shows the most common conditions for each state, see the docs for a complete list: https://msdn.microsoft.com/en-us/library/windows/apps/windows.applicationmodel.activation.applicationexecutionstate.aspx

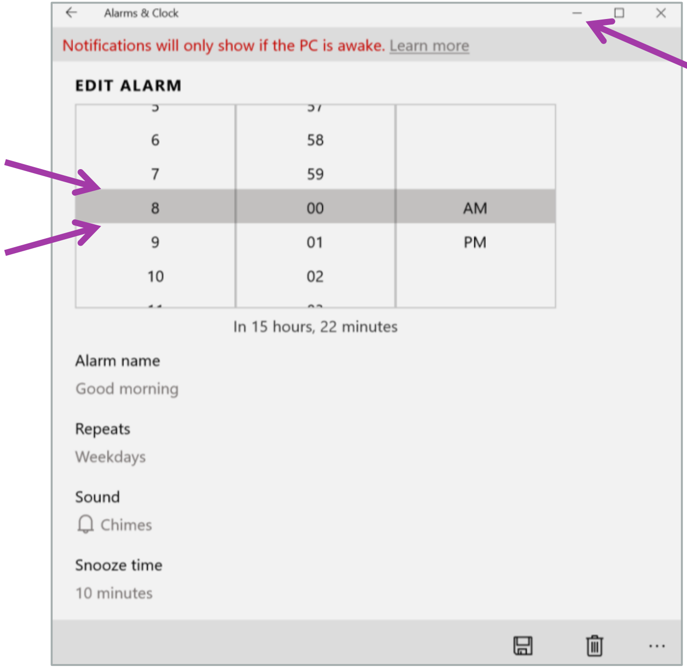
Demonstration

PreviousExecutionState

Xamarin University

User experience

- The user should not lose their data if they close/relaunch the app or if Windows terminates the app



Alarms & Clock

Notifications will only show if the PC is awake. Learn more

EDIT ALARM

6	58	
7	59	
8	00	AM
9	01	PM
10	02	

In 15 hours, 22 minutes

Alarm name
Good morning

Repeats
Weekdays

Sound
Chimes

Snooze time
10 minutes

1. User enters data into the UI

2. User minimizes the app; Windows may suspend and then terminate it

3. You need to store their data and reload it at the next launch

Application states

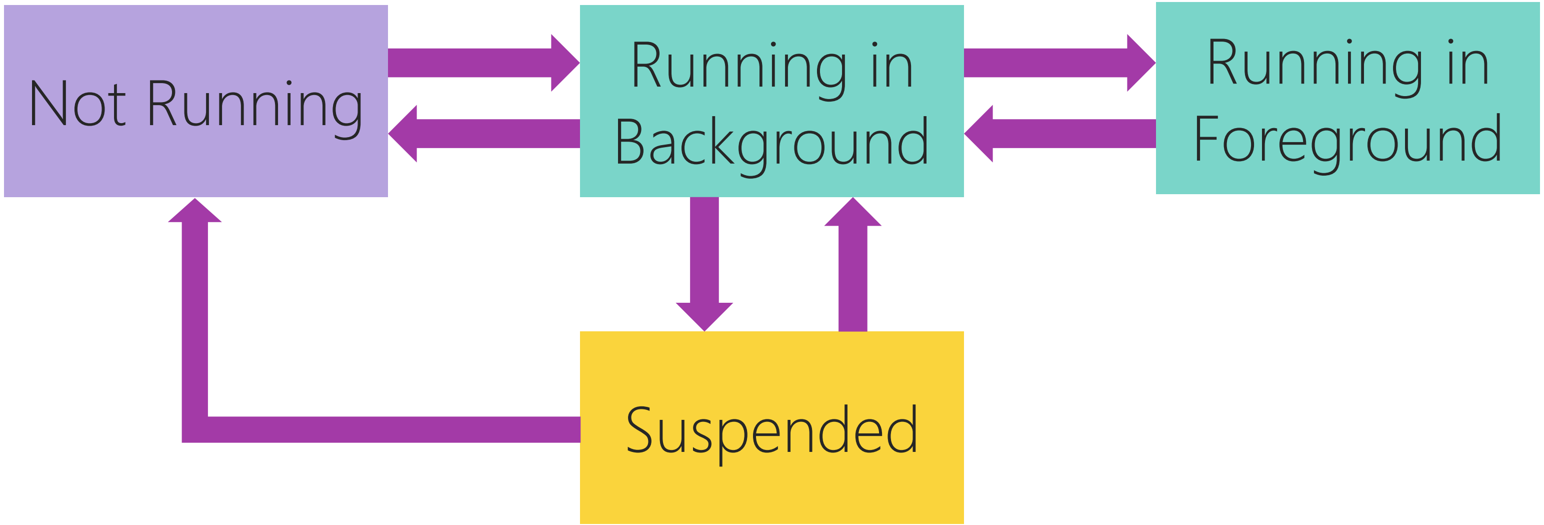
- An application *state* is the app's current set of resources provided by Windows (CPU time and memory)

Not Running	Running in Background	Running in Foreground	Suspended
Inactive and not in memory	Active and receiving CPU time, UI **not** visible	Active and receiving CPU time, UI is visible	Inactive but in memory

Foreground and background states are new in Windows 10 Anniversary edition and above – previous Windows versions combined these into a single **`Running`** state

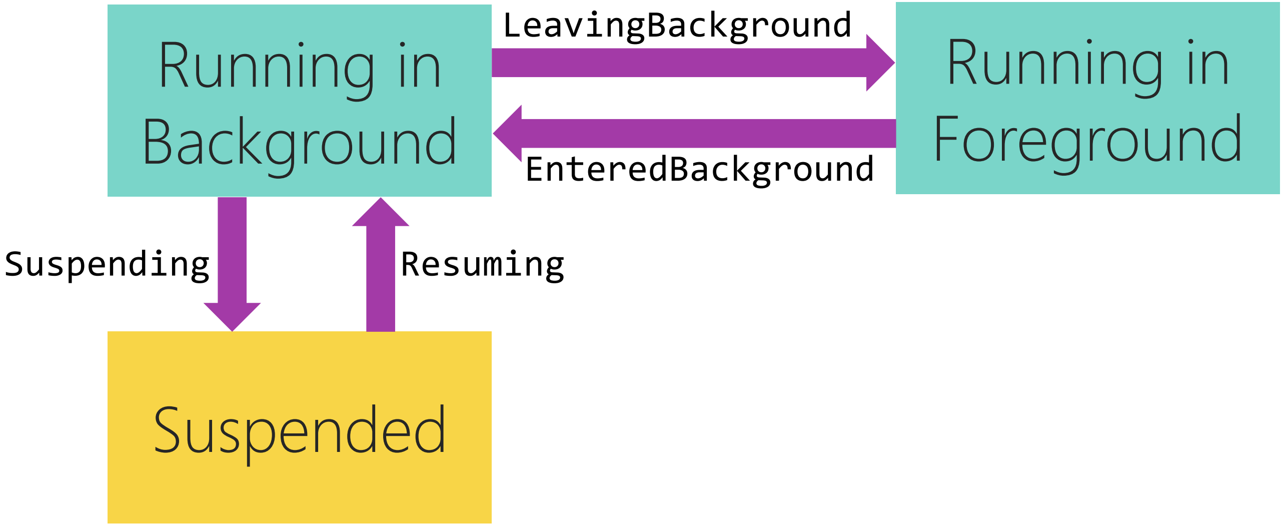
Application Lifecycle

- An application's *lifecycle* is the sequence of states the application moves through in response to user actions or Windows directives



Lifecycle notifications

- The application receives lifecycle notification events through the **`Application`** class for some state transitions



Running in Background

Running in Foreground

Suspended

`LeavingBackground`

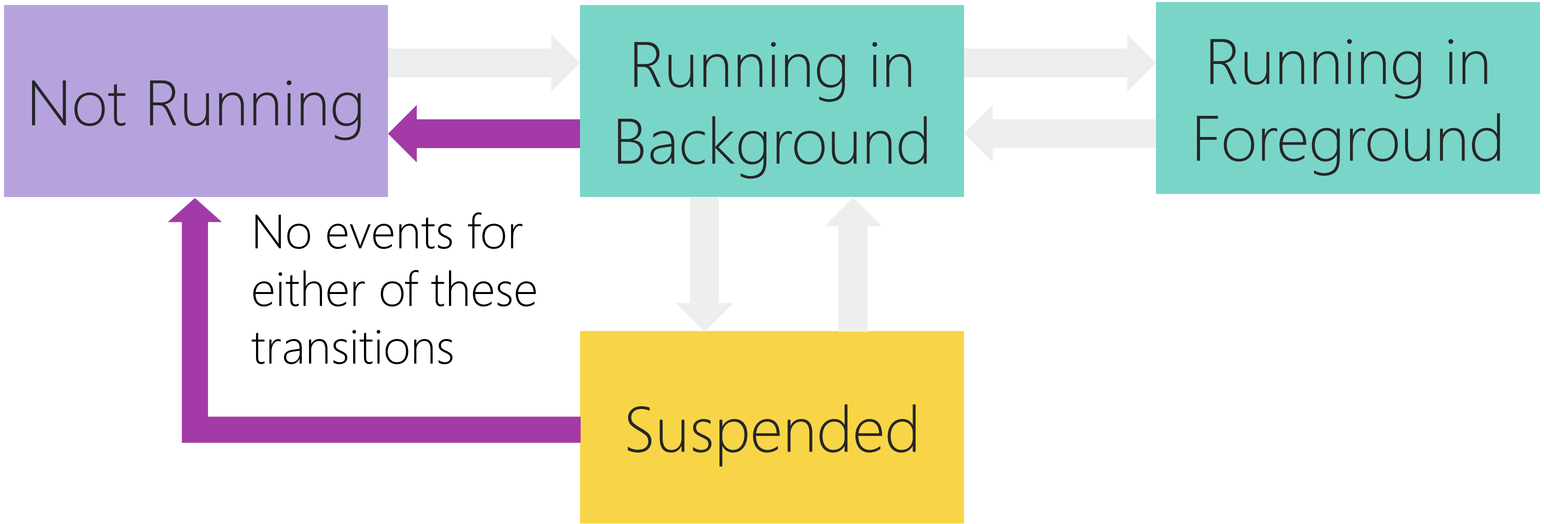
`EnteredBackground`

`Suspending`

`Resuming`

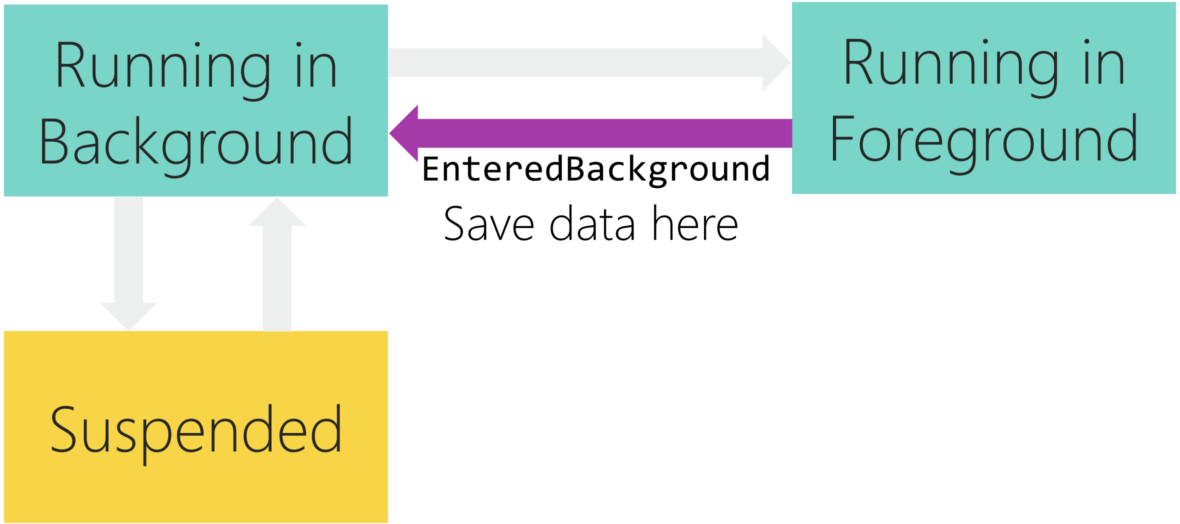
No shutdown notifications

- The app is given no notice when entering the **Not Running** state



When to save user data

- You should save the user's data and application state when your app enters the background



How to save user data

❖ In your handler for the app's **`EnteredBackground`** event, use the Application Data APIs to persist user data

```
void OnEnteredBackground(object sender, EnteredBackgroundEventArgs e)
{
   string data = ...;

   ApplicationData.Current.LocalSettings.Values["MyKey"] = data;
}
```

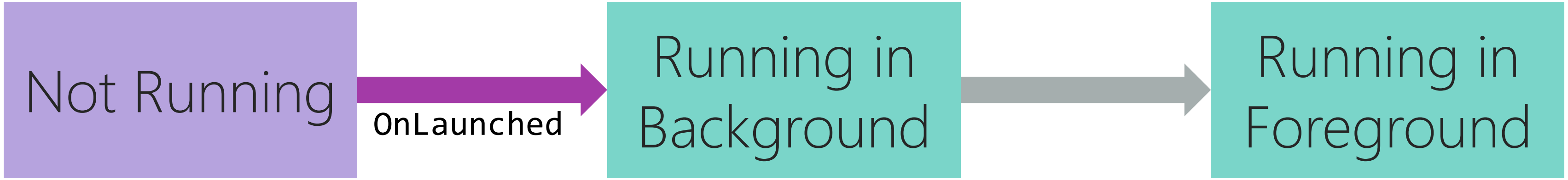
Application data provides a simple persistent-storage API

Built-in dictionary with synchronous methods

The details of the Application Data API are not covered in this course, please see:
https://msdn.microsoft.com/en-us/library/windows/apps/windows.storage.applicationdata.aspx

When to restore data

- Restore saved data when the app launches if the previous execution state is **`ClosedByUser`** or **`Terminated`**



Check the previous execution state, then restore data if needed

How to restore user data

- Use the Application Data APIs to retrieve the stored user data

Restore data at startup

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{  ...
   ApplicationExecutionState last = e.PreviousExecutionState;

   if (last == ApplicationExecutionState.ClosedByUser
    || last == ApplicationExecutionState.Terminated)
   {
      if (ApplicationData.Current.LocalSettings.Values.ContainsKey("MyKey"))
      {
         string data = (string)ApplicationData.Current.LocalSettings.Values["MyKey"];
         ...
      }
   }
}
```

Only in these two cases

Individual Exercise

Respond to lifecycle events

Summary

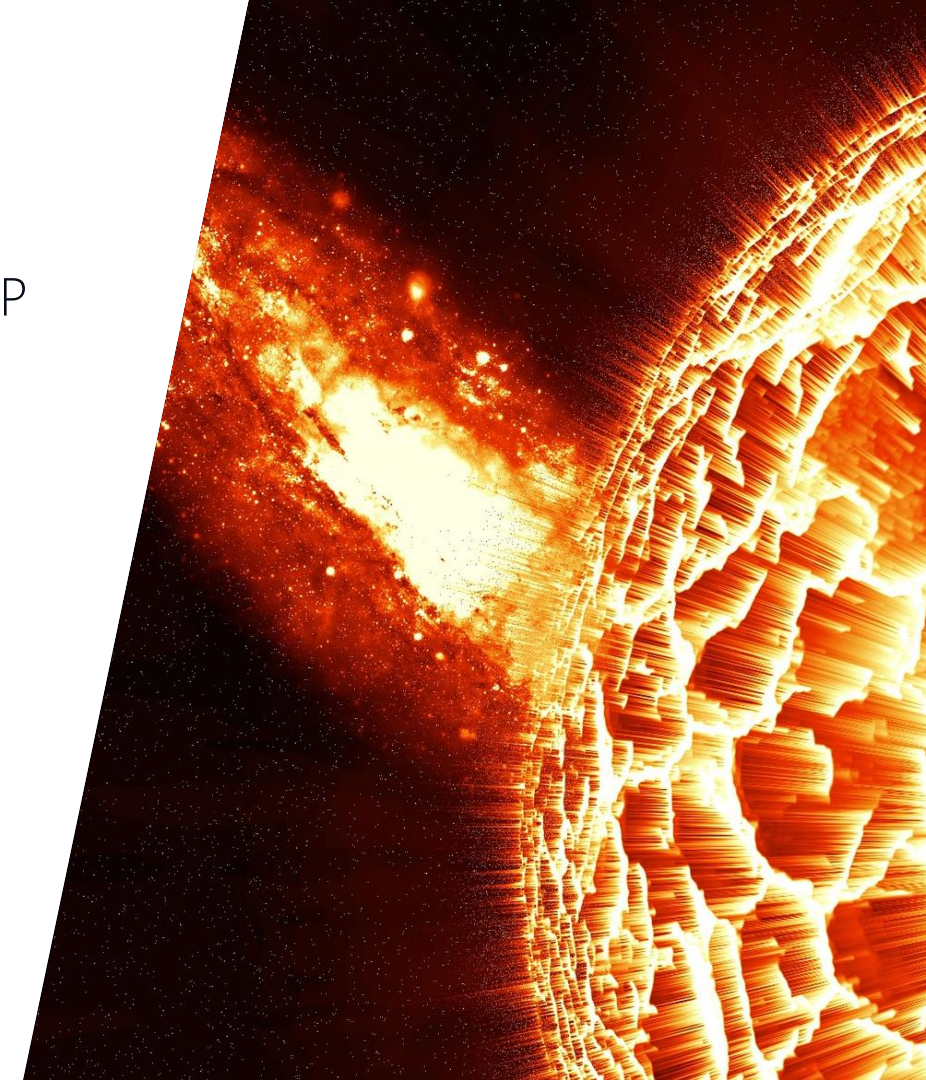
1. Determine previous execution state
2. Save state when entering the background
3. Restore state at startup if appropriate

Write platform-adaptive code

Xamarin University

Tasks

1. Define the adaptive features of UWP
2. Target specific device families
3. Use device-specific functionality



How are UWP apps adaptive?

❖ There are three ways UWP apps adapt to their runtime device

UI Adaptive

You code an adaptive UI and/or one UI for each device type

Platform Adaptive

You take advantage of device-specific APIs and/or you target specific device types

Version Adaptive

You take advantage of version-specific APIs and/or you target specify API versions

What is platform adaptive?

- An app is *platform adaptive* if it enables features on devices where they are available

E.g. an Expenses app uses the Scanner API available to Desktop apps...

...but uses the camera on non-desktop devices where the Scanner API is not supported

API partitioning

- The UWP APIs are partitioned into functional areas called *contracts*

Universal

XboxLive Storage

Dual SIM Tile

CallsPhone

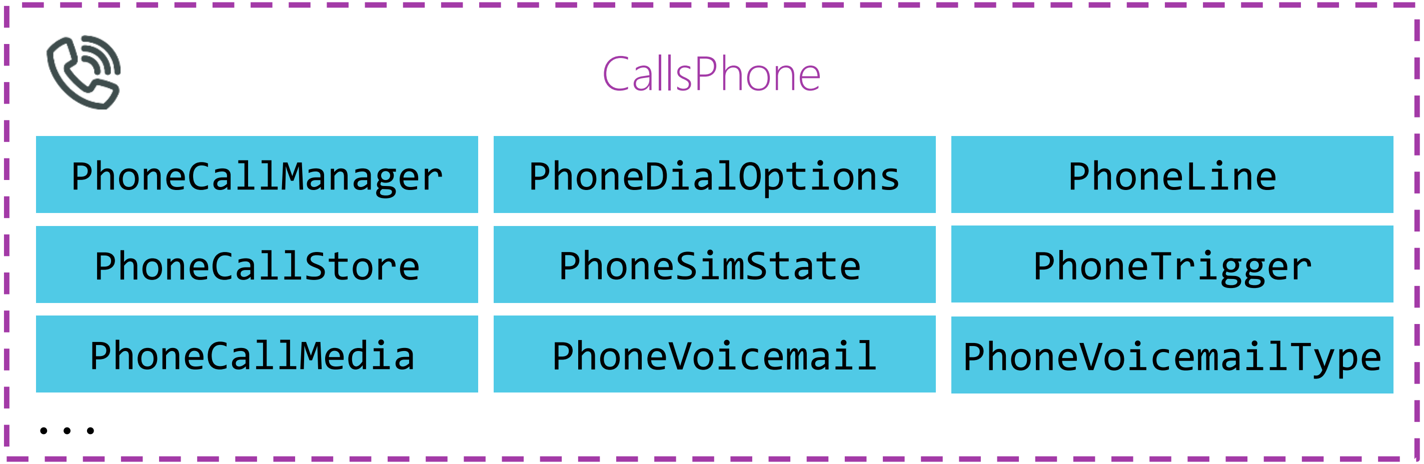
Wallet

Scanner Device

This shows only a few of the available contracts, explore all of them on MSDN: http://bit.ly/2dkrhK0

What is an API contract?

- An *API Contract* is set of related APIs that deliver a particular feature or functionality



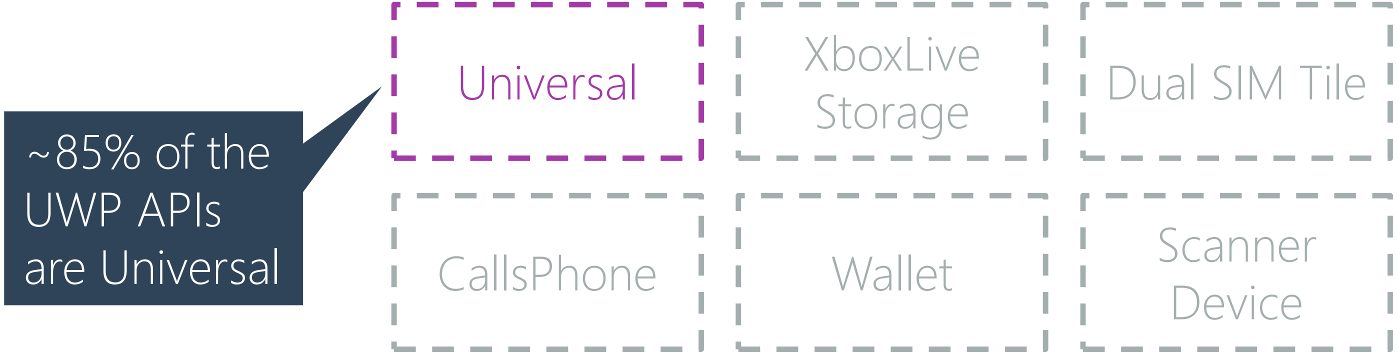
CallsPhone

PhoneCallManager	PhoneDialOptions	PhoneLine
PhoneCallStore	PhoneSimState	PhoneTrigger
PhoneCallMedia	PhoneVoicemail	PhoneVoicemailType

...

Universal API Contract

- The *Universal API Contract* is the set of APIs that are available on all device types



~85% of the UWP APIs are Universal

Universal

XboxLive Storage

Dual SIM Tile

CallsPhone

Wallet

Scanner Device

UWP apps can also use many .NET libs – they are available everywhere but not technically part of the Universal API Contract https://msdn.microsoft.com/en-us/library/windows/apps/mt185501.aspx

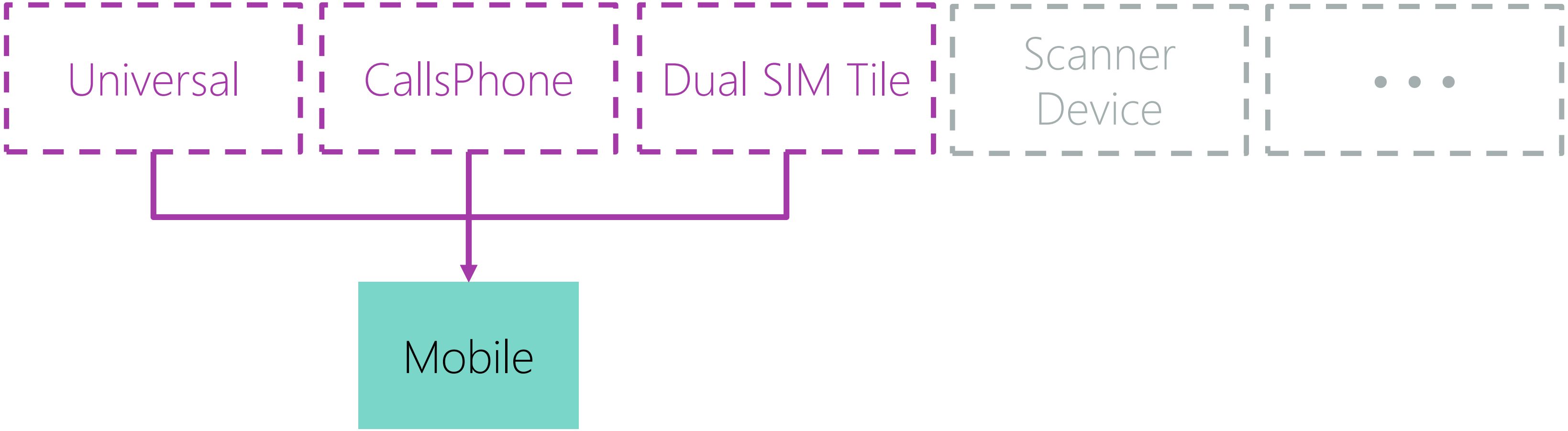
What is a device family?

- A *device family* is a collection of API Contracts

Device family	Example	Identifier string
Universal	N/A	`Windows.Universal`
Desktop	Surface Studio	`Windows.Desktop`
Mobile	Lumia 950	`Windows.Mobile`
Xbox	Xbox One S	`Windows.Xbox`
Holographic	HoloLens	`Windows.Holographic`
IoT	Raspberry Pi	`Windows.IoT`
IoT Headless	Minnowboard Max	`Windows.IoTHeadless`
Team	Surface Hub	`Windows.Team`

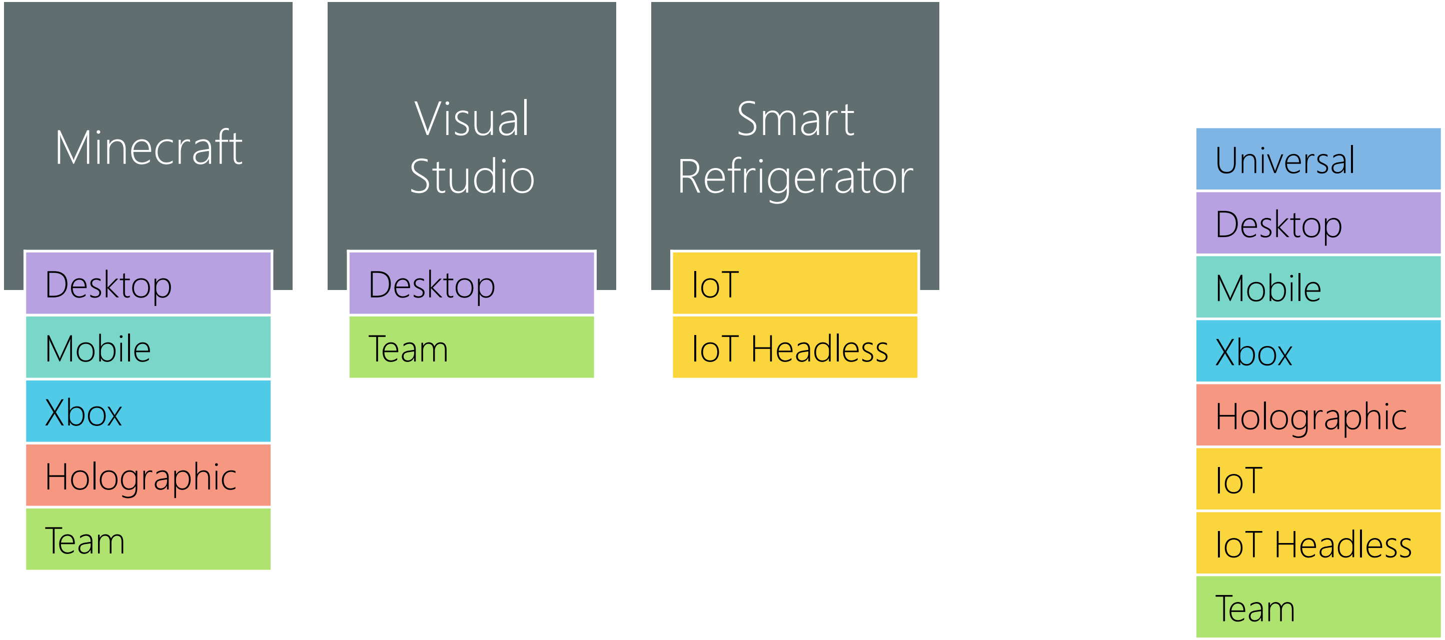
Device families and API Contracts

- Each device family supports the Universal API Contract and a selection of other API Contracts as appropriate for their hardware



Discussion

- Which device families would be appropriate for each of these apps?



Minecraft
- Desktop
- Mobile
- Xbox
- Holographic
- Team

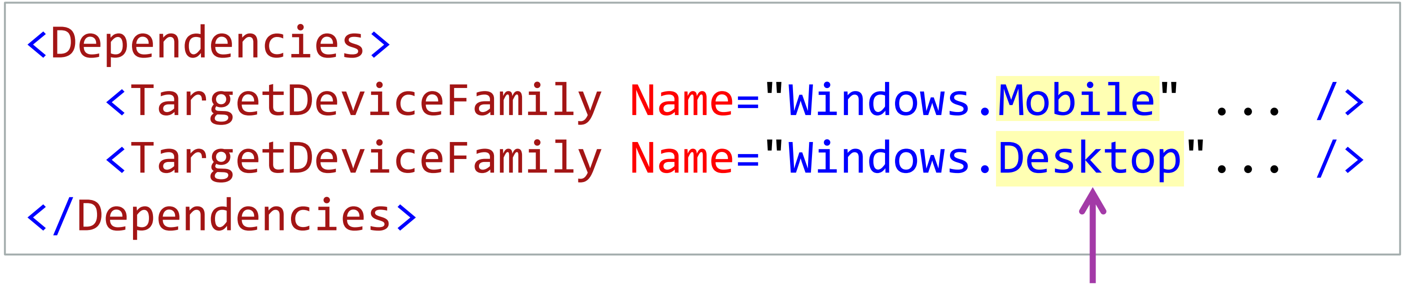
Visual Studio
- Desktop
- Team

Smart Refrigerator
- IoT
- IoT Headless

- Universal
- Desktop
- Mobile
- Xbox
- Holographic
- IoT
- IoT Headless
- Team

How to target device families

- To target specific device families, add **`TargetDeviceFamily`** entries to your app's manifest



`Package.appxmanifest`

```
<Dependencies>
   <TargetDeviceFamily Name="Windows.Mobile" ... />
   <TargetDeviceFamily Name="Windows.Desktop"... />
</Dependencies>
```

Use the family identifier string

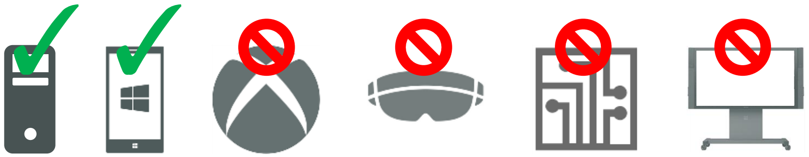
Open the manifest with the **XML (Text) Editor** to edit your app's **`Dependencies`**.

Effect of targeting on installation

- Your app can only be installed on devices in the families that you target

Package.appxmanifest

```
<Dependencies>
   <TargetDeviceFamily Name="Windows.Mobile" ... />
   <TargetDeviceFamily Name="Windows.Desktop"... />
</Dependencies>
```



Universal target

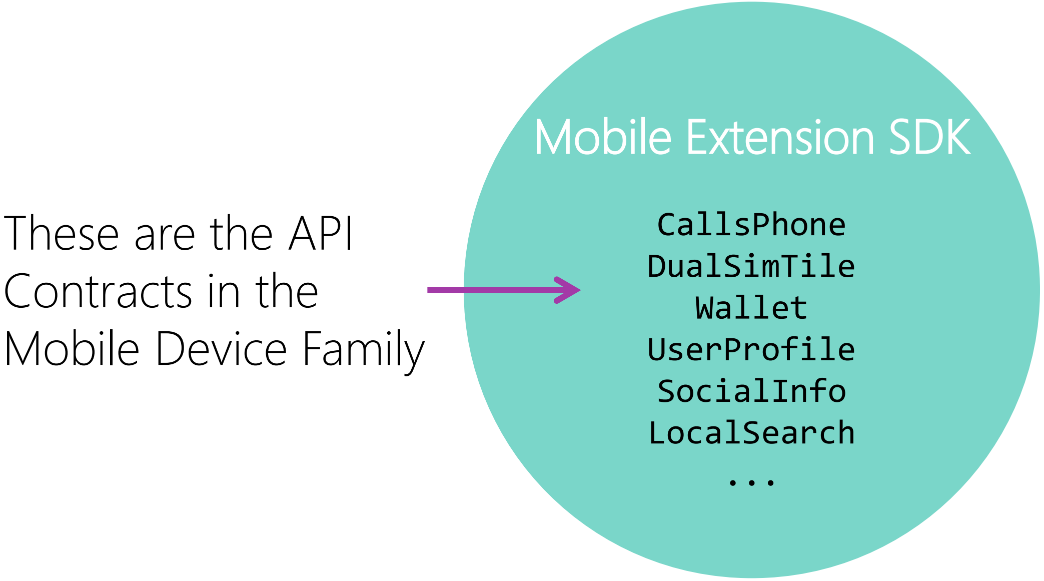
- Target the Universal device family to make your app installable on all UWP devices (this is the default for new projects)

Package.appxmanifest

```
<Dependencies>
   <TargetDeviceFamily Name="Windows.Universal" ... />
</Dependencies>
```

What is an Extension SDK?

❖ An *Extension SDK* is the component that defines the available APIs for a specific device family



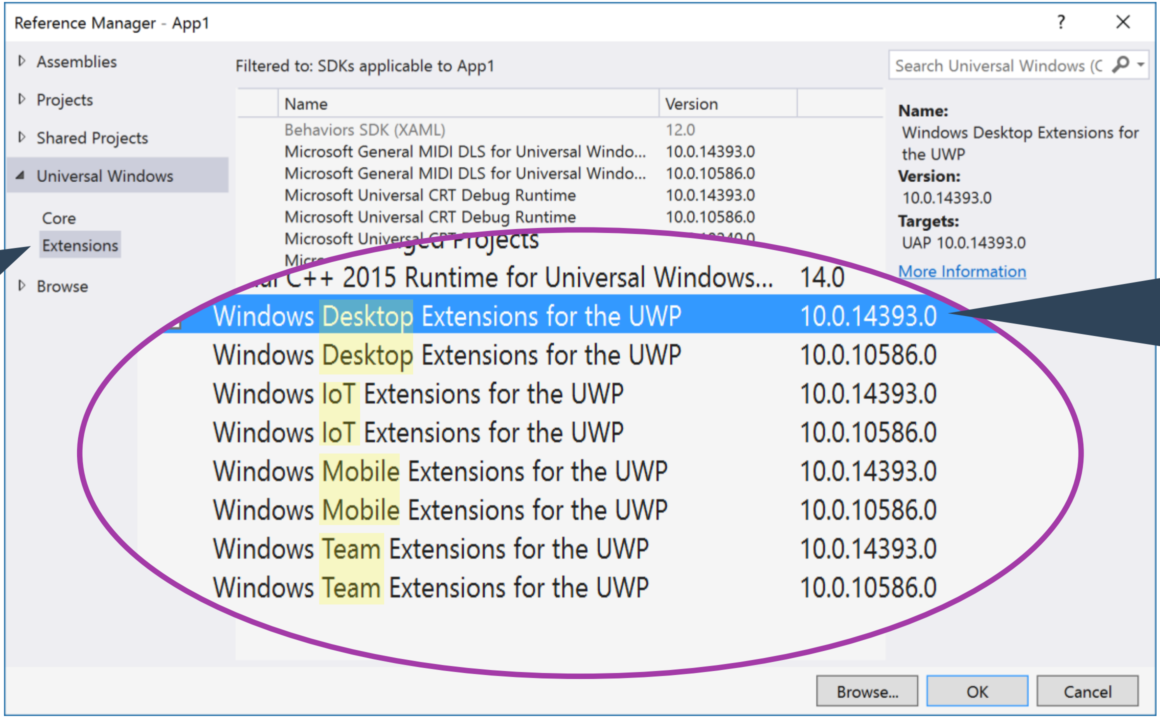
These are the API Contracts in the Mobile Device Family

Mobile Extension SDK

```
CallsPhone
DualSimTile
Wallet
UserProfile
SocialInfo
LocalSearch
...
```

How to use an Extension SDK

- You must reference an Extension SDK to use those APIs



Reference Manager - App1

Filtered to: SDKs applicable to App1

Name	Version
Behaviors SDK (XAML)	12.0
Microsoft General MIDI DLS for Universal Windo...	10.0.14393.0
Microsoft General MIDI DLS for Universal Windo...	10.0.10586.0
Microsoft Universal CRT Debug Runtime	10.0.14393.0
Microsoft Universal CRT Debug Runtime	10.0.10586.0
Microsoft Universal CRT	10.0.10240.0
Micro	
al C++ 2015 Runtime for Universal Windows...	14.0
Windows Desktop Extensions for the UWP	10.0.14393.0
Windows Desktop Extensions for the UWP	10.0.10586.0
Windows IoT Extensions for the UWP	10.0.14393.0
Windows IoT Extensions for the UWP	10.0.10586.0
Windows Mobile Extensions for the UWP	10.0.14393.0
Windows Mobile Extensions for the UWP	10.0.10586.0
Windows Team Extensions for the UWP	10.0.14393.0
Windows Team Extensions for the UWP	10.0.10586.0

ged Projects

Name: Windows Desktop Extensions for the UWP

Version: 10.0.14393.0

Targets: UAP 10.0.14393.0

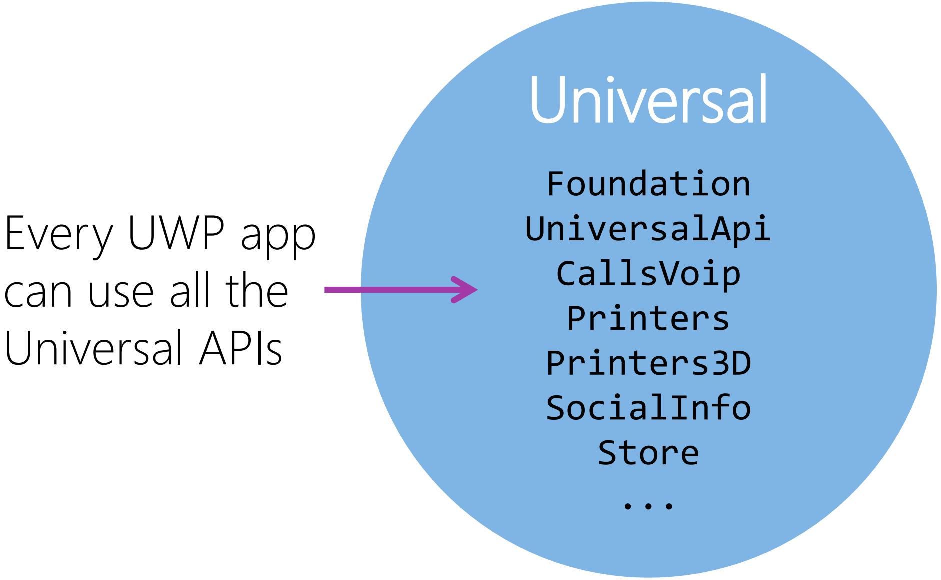
More Information

Listed in the Extensions category

Typically you will have multiple versions available

Using Universal APIs

- Your app can freely utilize the APIs in the Universal API Contract regardless of which device families your app target



Every UWP app can use all the Universal APIs

Universal

```
Foundation
UniversalApi
CallsVoip
Printers
Printers3D
SocialInfo
Store
...
```

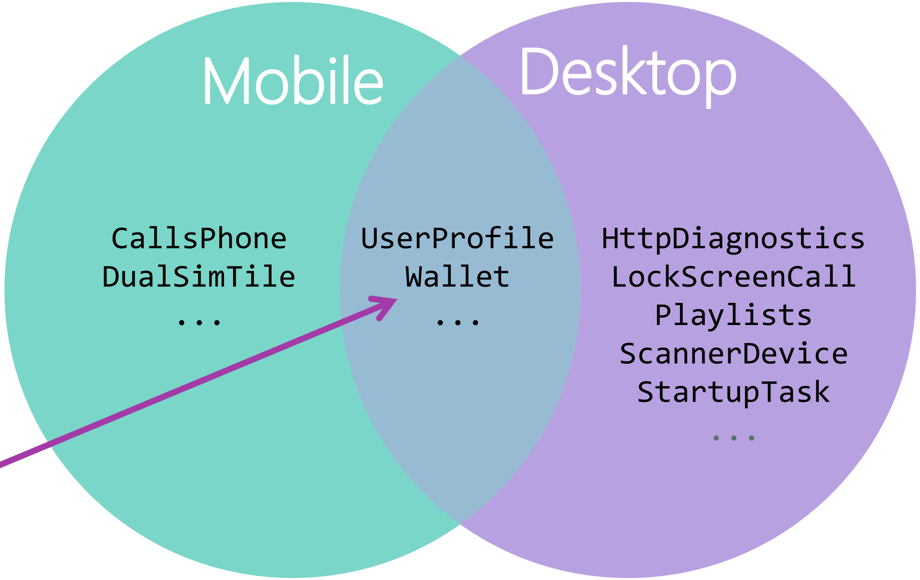
Using APIs that exist in all your targets

- Your app can freely utilize the APIs from the intersection of all device families you target

You target Mobile and Desktop

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Mobile" .../>
  <TargetDeviceFamily Name="Windows.Desktop".../>
</Dependencies>
```

The intersection of the APIs
will be available at runtime



Mobile

CallsPhone
DualSimTile
...

UserProfile
Wallet
...

Desktop

HttpDiagnostics
LockScreenCall
Playlists
ScannerDevice
StartupTask
...

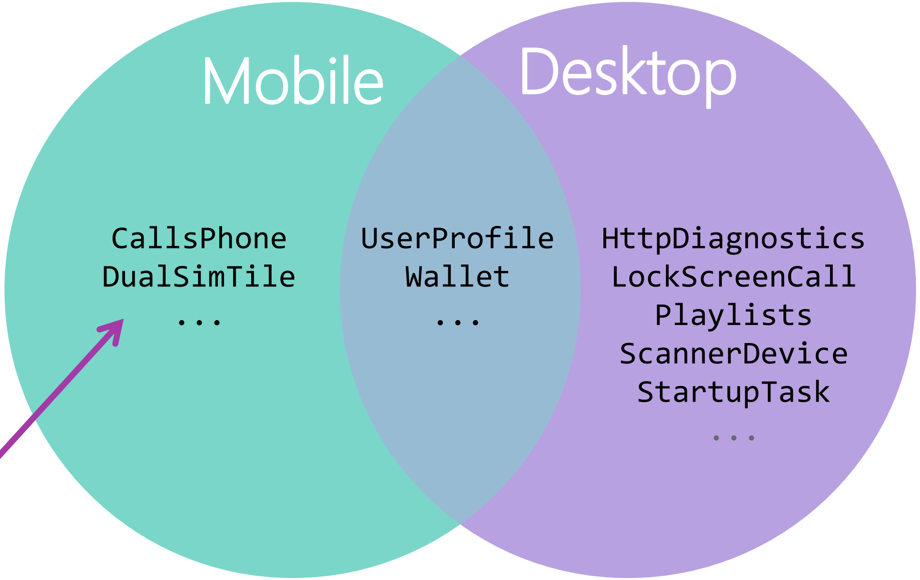
Using APIs that exist in one target

- APIs that are not part of all the device families you target can be used, but you must do a runtime test to see if they are available

App can be installed on either a Mobile or Desktop device

```
<Dependencies>
  <TargetDeviceFamily Name="Windows.Mobile" .../>
  <TargetDeviceFamily Name="Windows.Desktop".../>
</Dependencies>
```

These APIs will only be available when running on a Mobile device



Mobile

Desktop

```
CallsPhone
DualSimTile
...
```

```
UserProfile
Wallet
...
```

```
HttpDiagnostics
LockScreenCall
Playlists
ScannerDevice
StartupTask
...
```

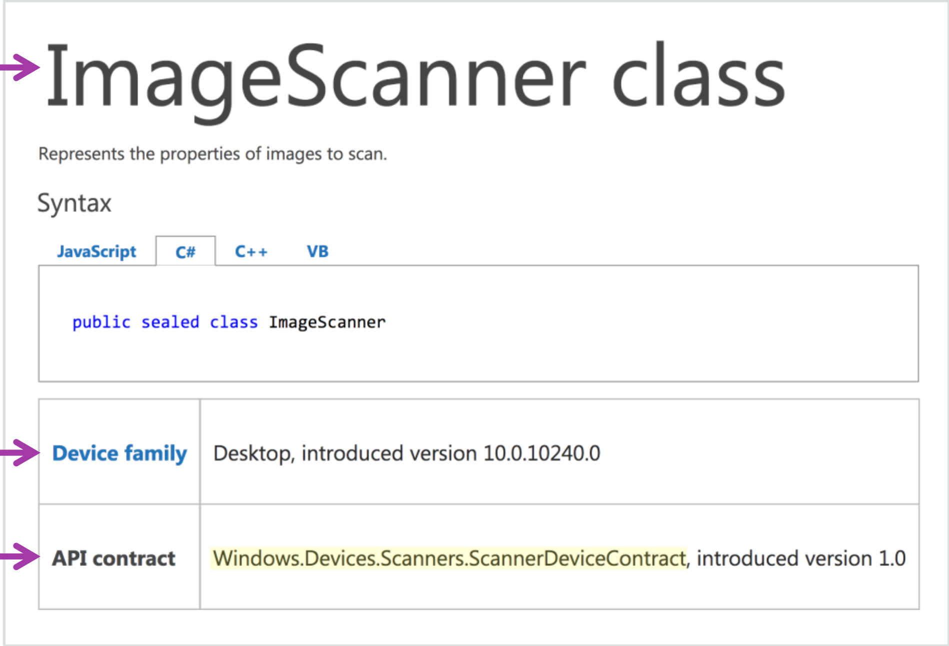
How is an API Contract identified?

- Each API Contract is identified by its name

Documentation for the type you want to use

Supported Device families

Contract name



ImageScanner class

Represents the properties of images to scan.

Syntax

```
public sealed class ImageScanner
```

Device family	Desktop, introduced version 10.0.10240.0
API contract	Windows.Devices.Scanners.ScannerDeviceContract, introduced version 1.0

What is ApiInformation?

- The **`ApiInformation`** class lets you programmatically test for the presence of an API Contract on the runtime device

Can test for an API Contract

```
public static class ApiInformation
{ ...
  public static bool IsApiContractPresent(string contractName, ushort majorVersion, ushort minorVersion)

  public static bool IsTypePresent    (string typeName)
  public static bool IsMethodPresent  (string typeName, string methodName, uint inputParameterCount)
  public static bool IsPropertyPresent(string typeName, string propertyName)
  public static bool IsEventPresent   (string typeName, string eventName)
}
```

Can test for types/members

When one API in a contract is found, all APIs in that contract will be available

How to write platform-adaptive code

- Use **`ApiInformation`** to test if an API contract is available on your runtime device, then enable those features in your app

```
string contract = "Windows.Devices.Scanners.ScannerDeviceContract";

if (ApiInformation.IsApiContractPresent(contract, 1, 0))
{
   EnableMyScannerButton();

   ImageScanner scanner = await ImageScanner.FromIdAsync("...");
}
```

Contract name

Test for availability

Use the API Contract

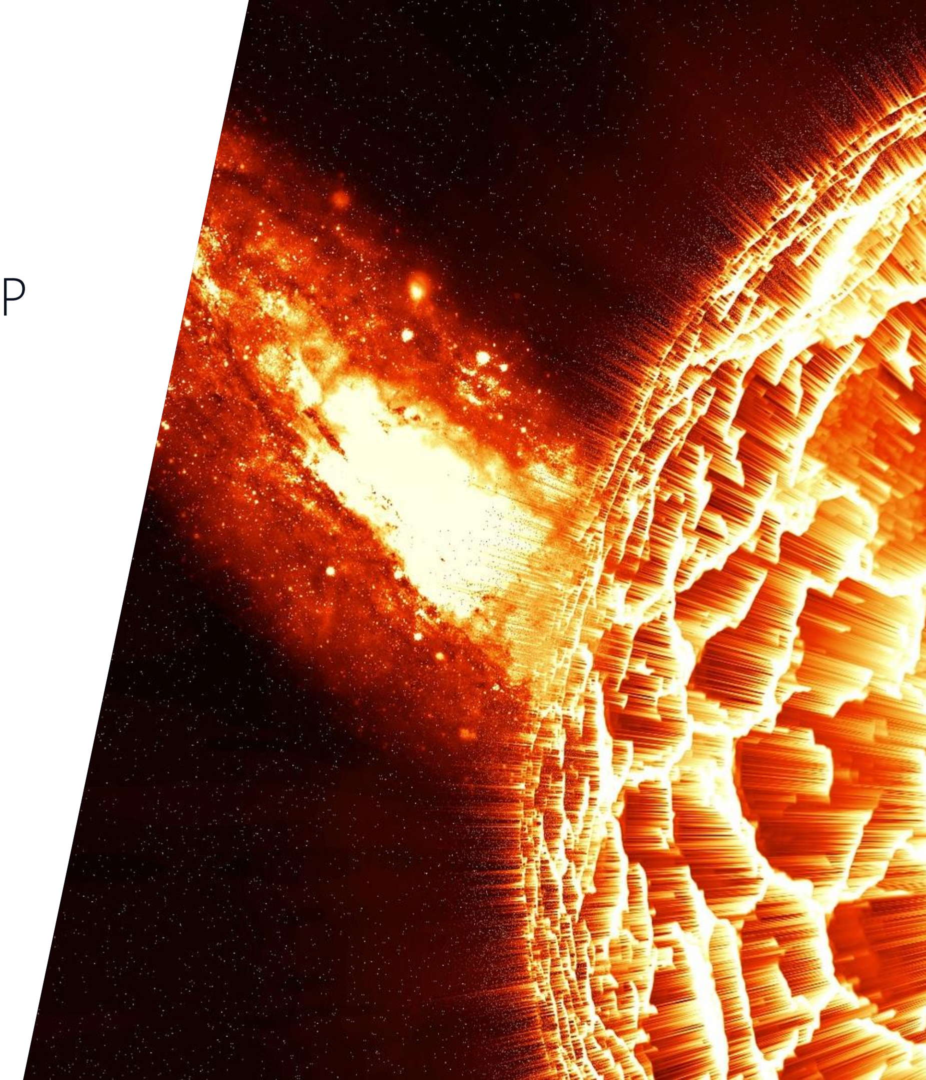
Individual Exercise

Write platform-adaptive code

Xamarin
University

Summary

1. Define the adaptive features of UWP
2. Target specific device families
3. Use device-specific functionality



Write version-adaptive code

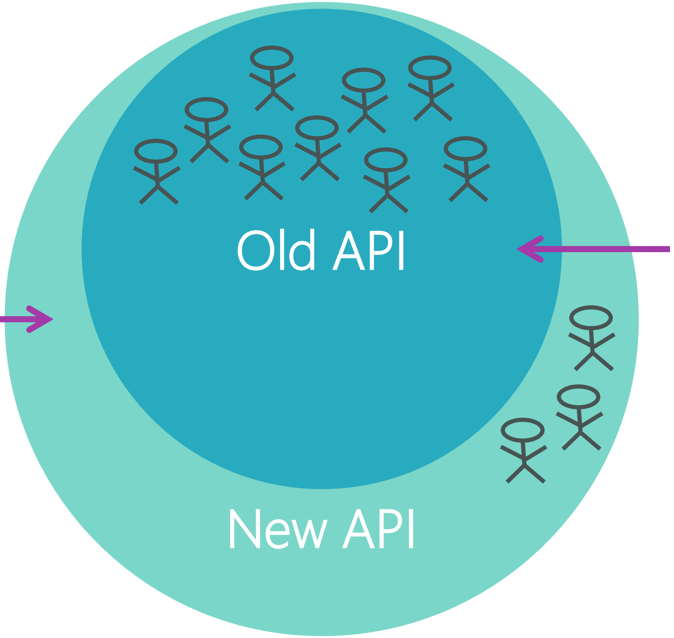
Xamarin University

Tasks

1. Specify your app's target OS versions
2. Use version-specific functionality

Motivation

- Before UWP, it could be difficult for Windows developers to decide when to update their app to use new APIs

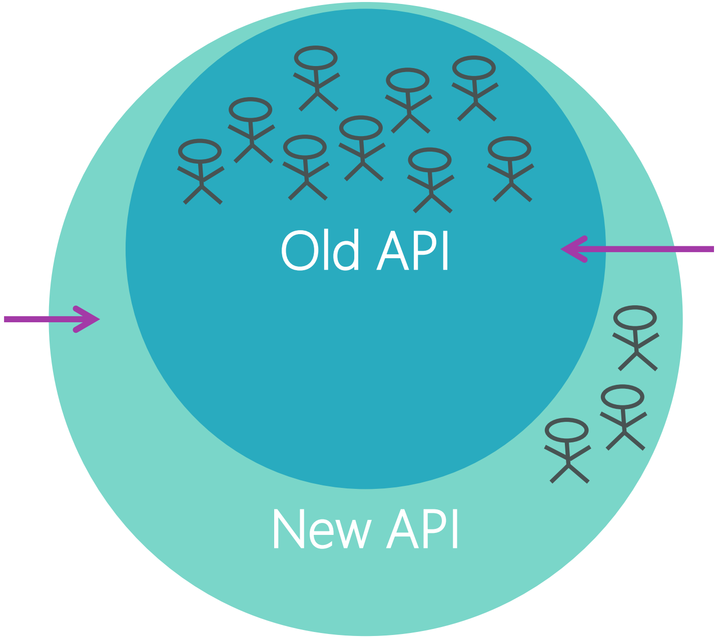


Update to the new API and use the new features?

Stick with the old API and access the largest user market?

What is version-adaptive code?

- *Version-adaptive code* is code that uses one codebase to support multiple API versions; using new APIs when running on updated devices

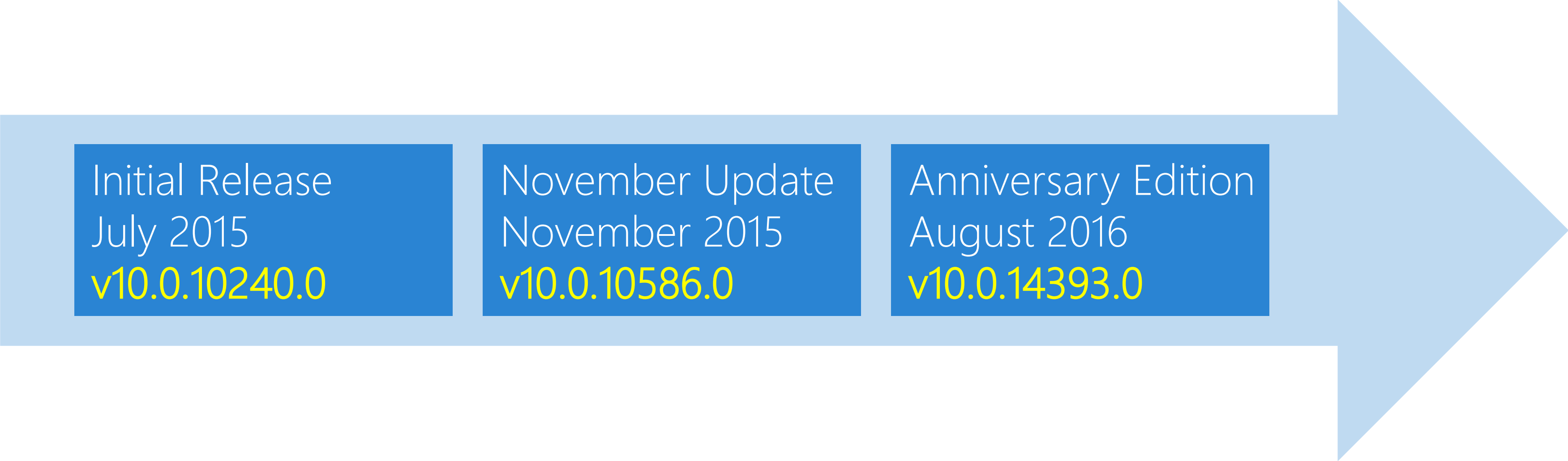


App uses new APIs when available and gracefully degrades when running on older installations

App can be installed on devices running old or new versions to maximize the size of the target market

Windows 10 versions

- Windows 10 has several released versions



Initial Release
July 2015
v10.0.10240.0

November Update
November 2015
v10.0.10586.0

Anniversary Edition
August 2016
v10.0.14393.0

Target versions

- You must declare which Windows version(s) your app supports

`MinVersion`

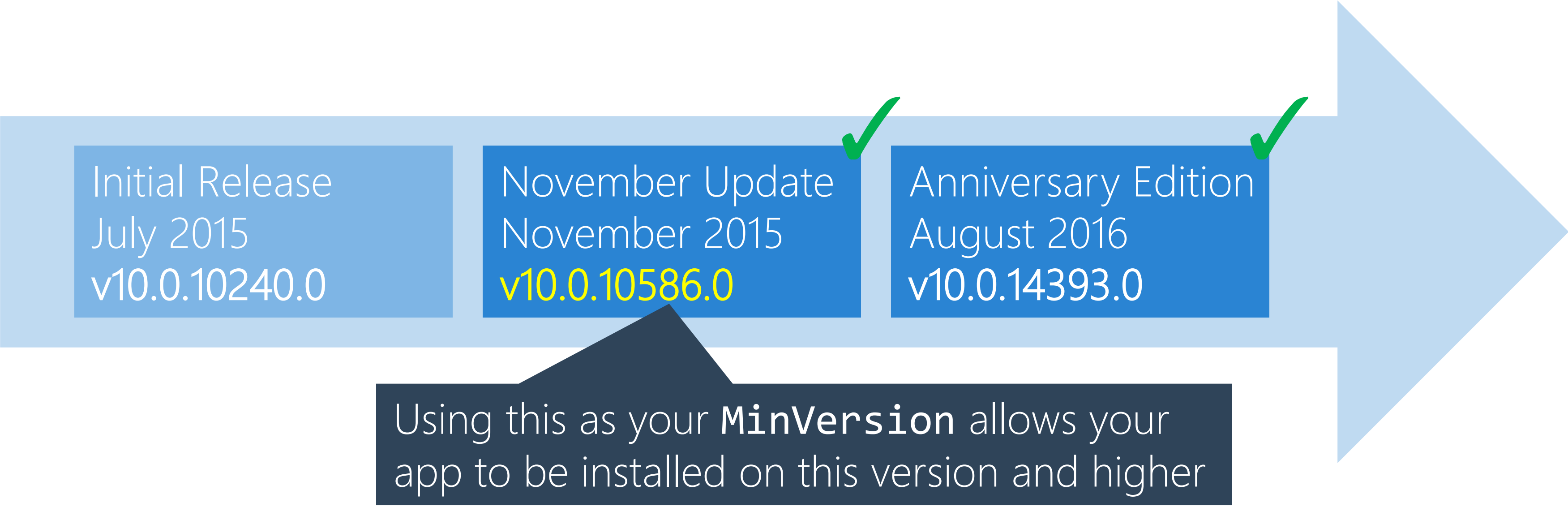
Used during installation from the Store

`MaxVersionTested`

Used to preserve target API behavior

What is MinVersion?

- Your app's **`MinVersion`** determines which Windows version(s) your app can be installed on through the Store (you can side-load onto lower versions)



Initial Release
July 2015
v10.0.10240.0

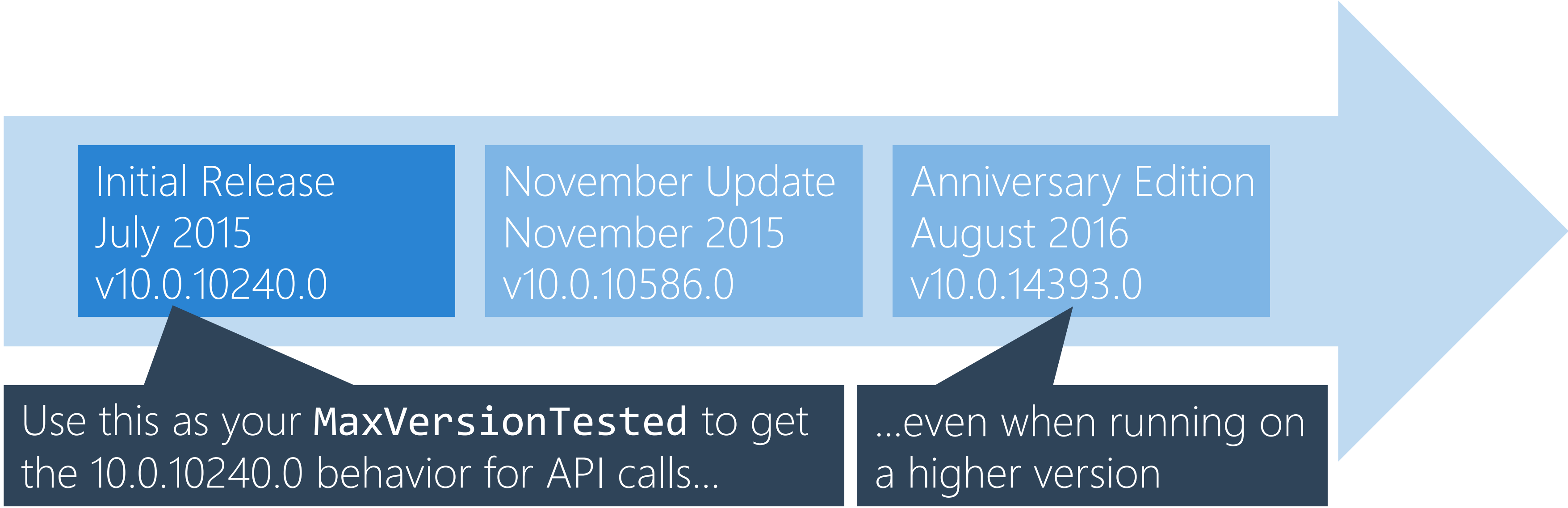
✓ November Update
November 2015
v10.0.10586.0

✓ Anniversary Edition
August 2016
v10.0.14393.0

Using this as your **`MinVersion`** allows your app to be installed on this version and higher

What is MaxVersionTested?

- Your app's **`MaxVersionTested`** determines which implementation of an API you get at runtime, it preserves your app's original behavior when a newer API implementation has changed the behavior (called a *quirk*)



Initial Release
July 2015
v10.0.10240.0

November Update
November 2015
v10.0.10586.0

Anniversary Edition
August 2016
v10.0.14393.0

Use this as your **`MaxVersionTested`** to get the 10.0.10240.0 behavior for API calls…

…even when running on a higher version

How to specify Target versions

- There are two places you can specify your app's Target versions

`Package.appxmanifest`

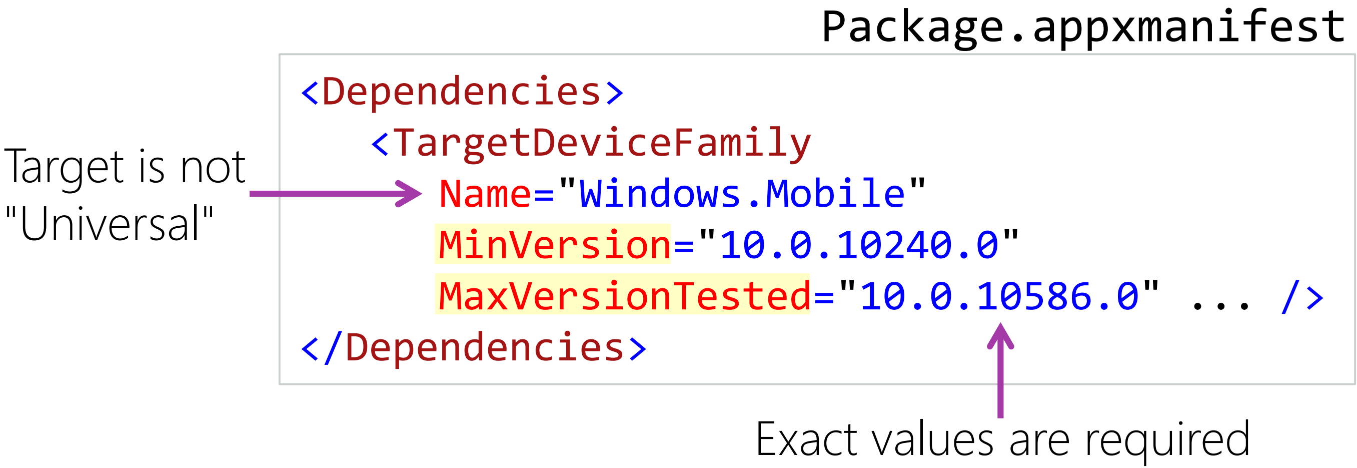
This is the only way when targeting specific Device Families

`MyApp.csproj`

This is an option when targeting the Universal Device Family

Target versions [manifest]

- When targeting specific Device Families, apps must specify precise target versions in their manifest



Package.appxmanifest

```
<Dependencies>
   <TargetDeviceFamily
      Name="Windows.Mobile"
      MinVersion="10.0.10240.0"
      MaxVersionTested="10.0.10586.0" ... />
</Dependencies>
```

Target is not "Universal"

Exact values are required

Target versions [.csproj]

- When targeting the Universal Device Family, apps can use the special value "**`10.0.0.0`**" in their Manifest to force the Target version values to be taken from elements in the **`.csproj`**

`Package.appxmanifest`

```
<TargetDeviceFamily
   Name="Windows.Universal"
   MinVersion="10.0.0.0"
   MaxVersionTested="10.0.0.0" ... />
```

Target is "Universal" and versions are 10.0.0.0

`MyApp.csproj`

```
<TargetPlatformMinVersion>10.0.10240.0<TargetPlatformMinVersion/>
<TargetPlatformVersion>10.0.10586.0<TargetPlatformVersion/>
```

Values are set from the .csproj file (the Project Properties GUI exposes these settings)

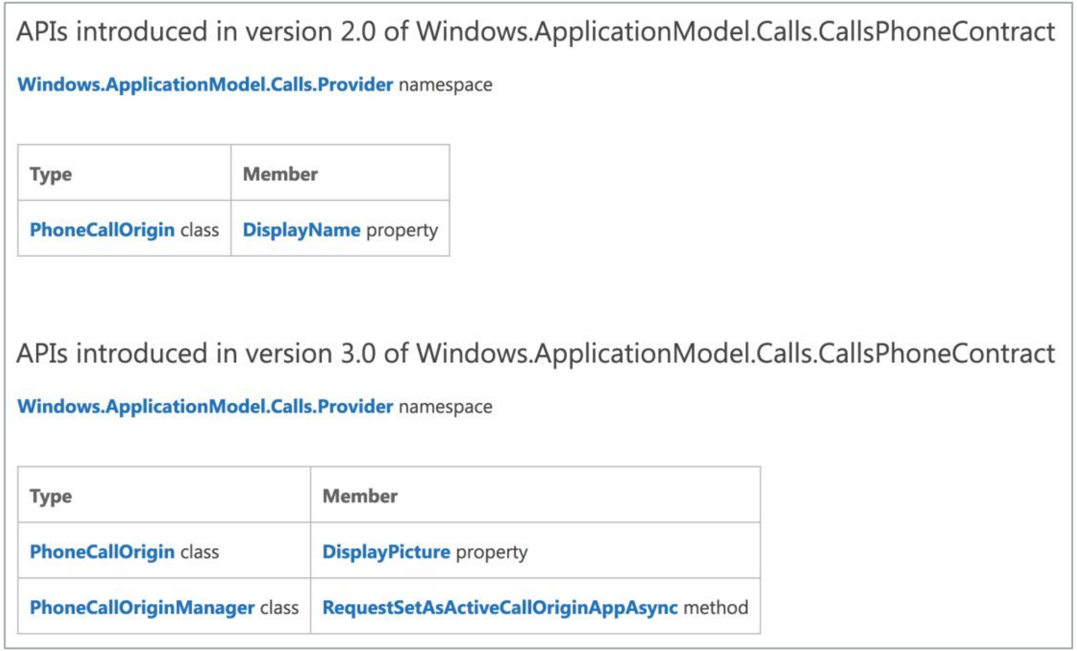
API Contract versions

- Each release of Windows 10 changes the set of available API Contracts

	Windows v10.0.10240.0	Windows v10.0.10586.0	Windows v10.0.14393.0
CallsPhone	v1.0	v2.0	v3.0
Wallet	v1.0	→	→
Maps.Guidance	v1.0	v2.0	→
ControlChannelTrigger	v1.0	→	v2.0
CallsBackground	N/A	v1.0	→
CallsVoip	v1.0	Moved to Universal	N/A

API Contract version features

- New releases of API Contracts typically add new features



APIs introduced in version 2.0 of Windows.ApplicationModel.Calls.CallsPhoneContract

Windows.ApplicationModel.Calls.Provider namespace

Type	Member
PhoneCallOrigin class	**DisplayName** property

APIs introduced in version 3.0 of Windows.ApplicationModel.Calls.CallsPhoneContract

Windows.ApplicationModel.Calls.Provider namespace

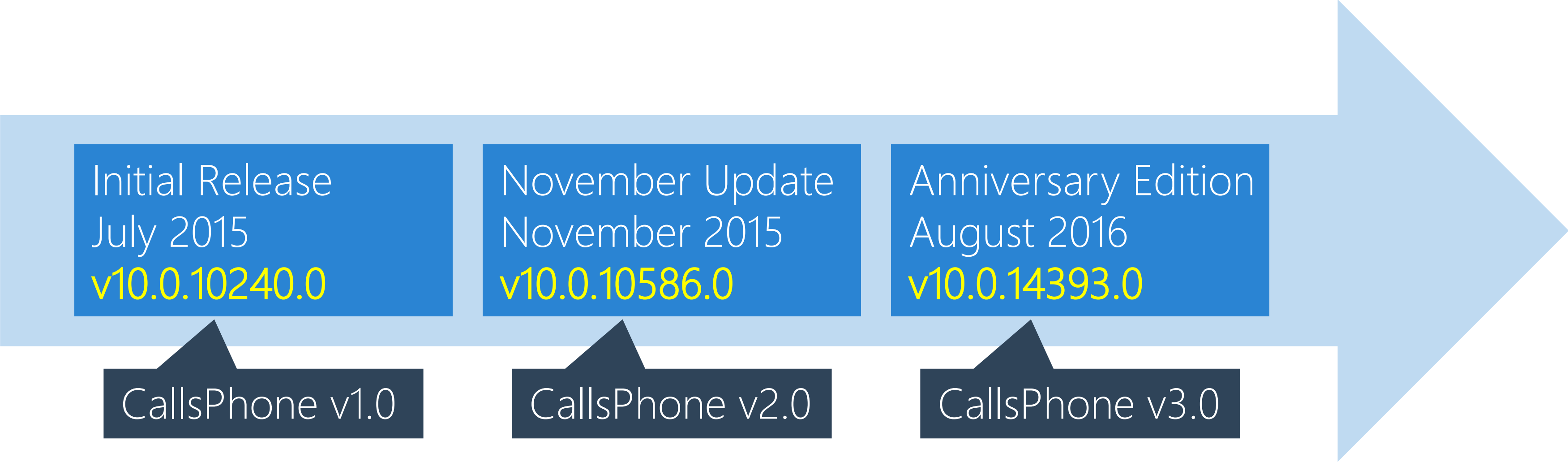
Type	Member
PhoneCallOrigin class	**DisplayPicture** property
PhoneCallOriginManager class	**RequestSetAsActiveCallOriginAppAsync** method

Added in v2.0 →

Added in v3.0 →

API Contract version availability

- The version of Windows 10 your app is running on determines which version of an API Contract is available



Initial Release
July 2015
v10.0.10240.0

CallsPhone v1.0

November Update
November 2015
v10.0.10586.0

CallsPhone v2.0

Anniversary Edition
August 2016
v10.0.14393.0

CallsPhone v3.0

Identify API Contract version

- You use **`IsApiContractPresent`** to determine whether a specific API Contract version is available to your app at runtime

```
public static class ApiInformation
{
  ...
  public static bool IsApiContractPresent(string contractName, ushort majorVersion, ushort minorVersion)
  ...
}
```

Version parameters

Note: UWP guidance is to test for API Contract version, not Windows version

Version adaptive code

- Use **`ApiInformation`** to determine available API version(s)

```
string contract = "Windows.ApplicationModel.Calls.CallsPhoneContract";

if (ApiInformation.IsApiContractPresent(contract, 3, 0))
{
   var origin = new PhoneCallOrigin();
   var picture = origin.DisplayPicture; // only in version 3
   ...
}
if (ApiInformation.IsApiContractPresent(contract, 2, 0))
{
   var origin = new PhoneCallOrigin();
   string name = origin.DisplayName; // in versions 2 and 3
   ...
}
if (ApiInformation.IsApiContractPresent(contract, 1, 0))
{
   bool active = PhoneCallManager.IsCallActive; // in versions 1, 2, and 3
   ...
}
```

Include picture → `var picture = origin.DisplayPicture;`

Include name → `string name = origin.DisplayName;`

Include active indicator → `bool active = PhoneCallManager.IsCallActive;`

Individual Exercise

Write version-adaptive code

Summary

1. Specify your app's target OS versions
2. Use version-specific functionality

Additional Resources

- Microsoft Virtual Academy
  https://mva.microsoft.com/
- Channel 9
  https://channel9.msdn.com/
- Microsoft Docs
  https://docs.microsoft.com

Thank You!

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

Microsoft