Lecture 4 – Xamarin Froms

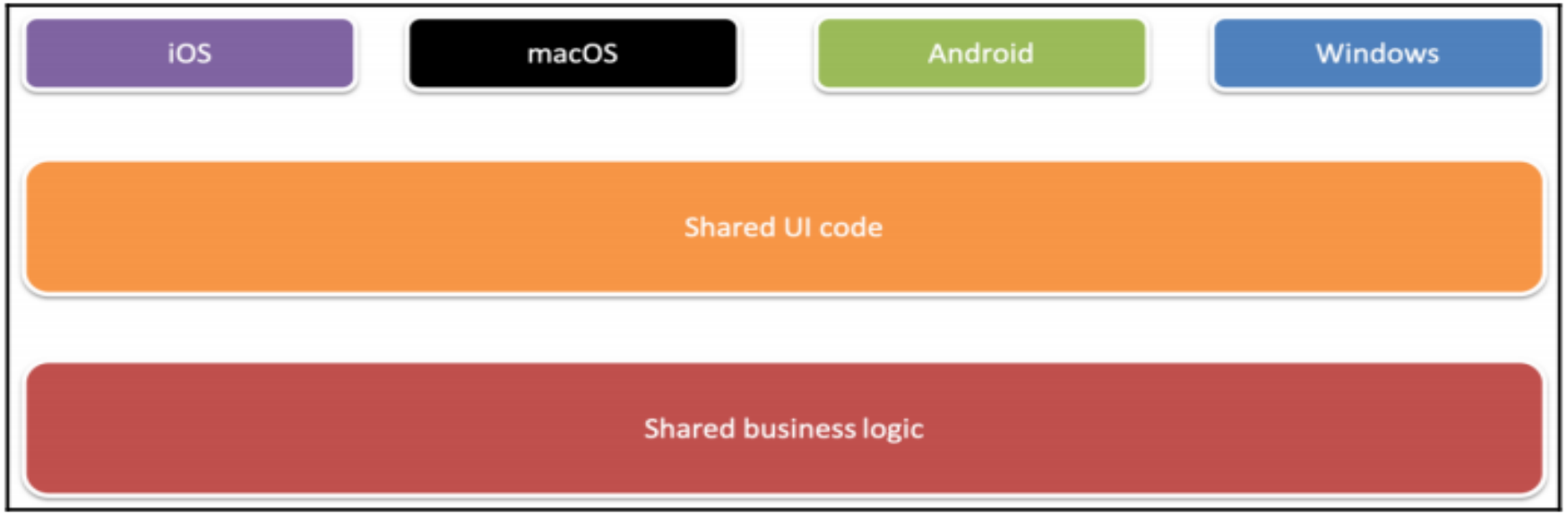
Mr. Yousif Garabet Arshak
Computer Science Department
University of Zakho
yousif.arshak@uoz.edu.krd

Outlines

- What is Xamarin Forms
- The architecture of Xamarin.Forms
- Defining a UI using XAML
- Defining a Label control
- Creating a page in XAML
- Creating a page in C#
- XAML or C#?
- Xamarin.Forms versus traditional Xamarin
- When to use Xamarin.Forms

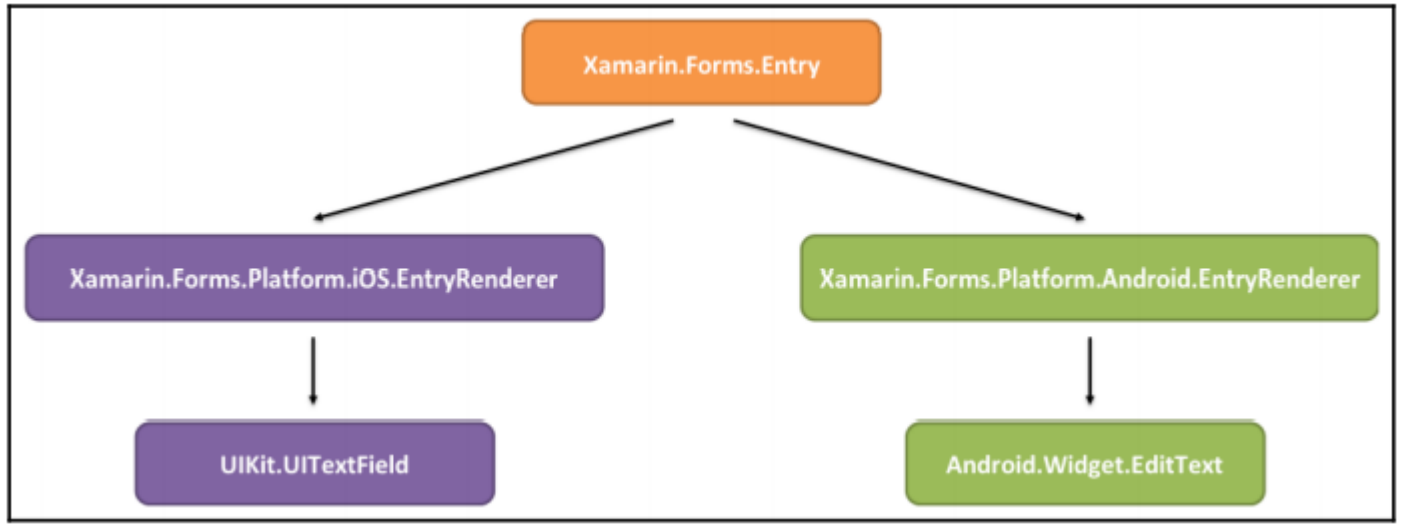
What is Xamarin.Forms?

- Xamarin.Forms is a UI framework that is built on top of Xamarin (for iOS and Android) and the Universal Windows Platform (UWP). Xamarin.Forms allows developers to create a UI for iOS, Android, and UWP with one shared code base, as illustrated in the following diagram. If we build an app with Xamarin.Forms, we can use XAML, C#, or a combination of both to create the UI:



The architecture of Xamarin.Forms

- Xamarin.Forms is more or less just an abstract layer on top of each platform. Xamarin.Forms has a shared layer that is used by all platforms, as well as a platform-specific layer. The platform-specific layer contains renderers. A renderer is a class that maps a Xamarin.Forms control to a platform-specific native control. Each Xamarin.Forms control has a platform-specific renderer.
- The following diagram illustrates how entry control in Xamarin.Forms is rendered to a UITextField control from the UIKit namespace when the shared Xamarin.Forms code is used in an iOS app. The same code in Android renders an EditText control from the Android.Widget namespace:



Defining a UI using XAML

- The most common way to declare our UI in Xamarin.Forms is by defining it in a XAML document. It is also possible to create the GUI in C#, since XAML is really only a markup language for instantiating objects. We could, in theory, use XAML to create any type of object, as long as it has a parameterless constructor. A XAML document is an Extensible Markup Language (XML) document with a specific schema.

Defining a Label control

- As a simple example, let's look at the following snippet of a XAML document:

  <Label Text="Hello World!" />
- When the XAML parser encounters this snippet, it creates an instance of a Label object and
- then sets the properties of the object that correspond to the attributes in the XAML. This
- means that if we set a Text property in XAML, it sets the Text property on the instance of
- the Label object that is created. The XAML in the preceding example has the same effect as
- the following:

```
var obj = new Label()
{
 Text = "Hello World!"
};
```

XAML exists to make it easier to view the object hierarchy that we need to create in order to

make a GUI. An object model for a GUI is also hierarchical by design, so XAML supports

adding child objects. We can simply add them as child nodes, as follows:

```
<StackLayout>
 <Label Text="Hello World" />
 <Entry Text="Ducks are us" />
</StackLayout>
```

StackLayout is a container control that organizes the children vertically or horizontally

within a container. Vertical organization is the default value and is used unless we specify

otherwise. There are also a number of other containers, such as Grid and FlexLayout.

These will be used in many of the projects in the following chapters.

Creating a page in XAML

- A single control is no use unless it has a container that hosts it. Let's see what an entire page
- would look like. A fully valid ContentPage object defined in XAML is an XML document.
- This means that we must start with an XML declaration. After that, we must have one—and
- only one—root node, as shown:

```
<?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="MyApp.MainPage">
    <StackLayout>
        <Label Text="Hello world!" />
    </StackLayout>
</ContentPage>
```

- In the preceding example, we defined a ContentPage object that translates into a single view on each platform. In order to make it a valid XAML, we need to specify a default namespace (xmlns="http://xamarin.com/schemas/2014/forms") and then add the x namespace (xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml").
- The default namespace lets us create objects without prefixing them, such as the StackLayout object. The x namespace lets us access properties such as x:Class, which tells the XAML parser which class to instantiate to control the page when the ContentPage object is created.
- A ContentPage object can have only one child. In this case, it's a StackLayout control. Unless we specify otherwise, the default layout orientation is vertical. A StackLayout object can, therefore, have multiple children. Later on, we will touch on more advanced layout controls, such as the Grid and FlexLayout controls.

Creating a page in C#

- For clarity, the following code shows you how the previous example would look in C#:

  public class MainPage : ContentPage { }

page is a class that inherits from Xamarin.Forms.ContentPage. This class is automatically generated for us if we create an XAML page, but if we just use code, we will need to define it ourself.

Let's create the same control hierarchy as the XAML page we defined earlier using the following code:

```
var page = new MainPage();
var stacklayout = new StackLayout();
stacklayout.Children.Add(
 new Label()
 {
 Text = "Welcome to Xamarin.Forms"
 });
page.Content = stacklayout;
```

The first statement creates a page object. We could, in theory, create a new ContentPage page directly, but this would prohibit us from writing any code behind it. For this reason, it's good practice to subclass each page that we plan to create.

The block following this first statement creates the StackLayout control, which contains the Label control that is added to the Children collection.

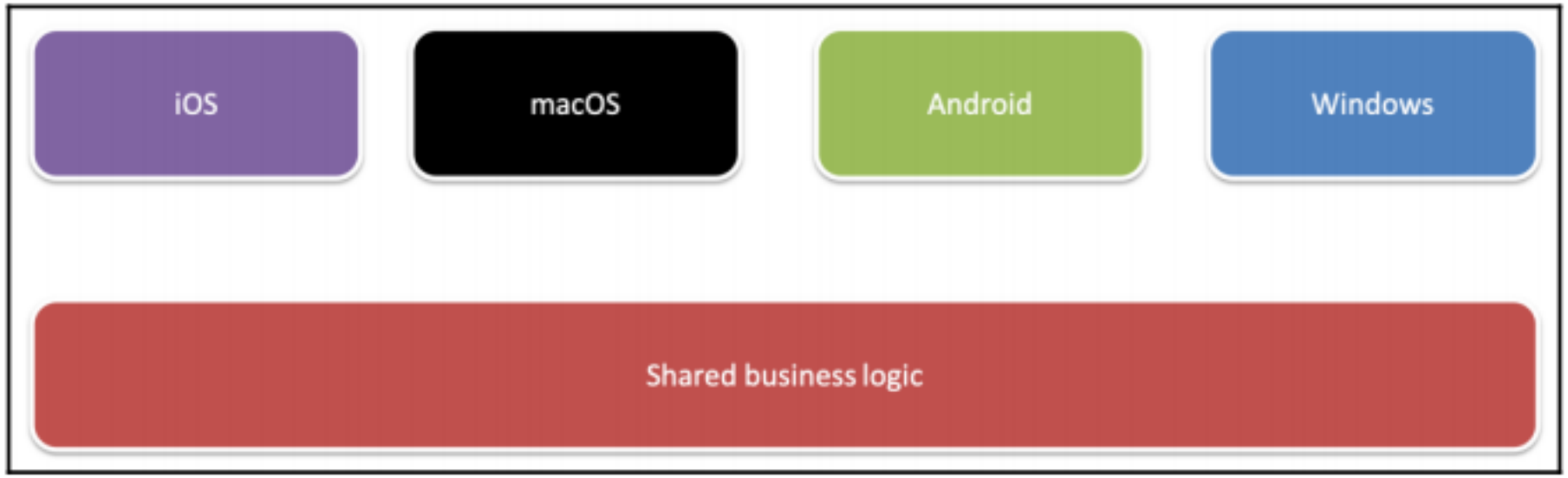
Finally, we need to assign StackLayout to the Content property of the page.

XAML or C#?

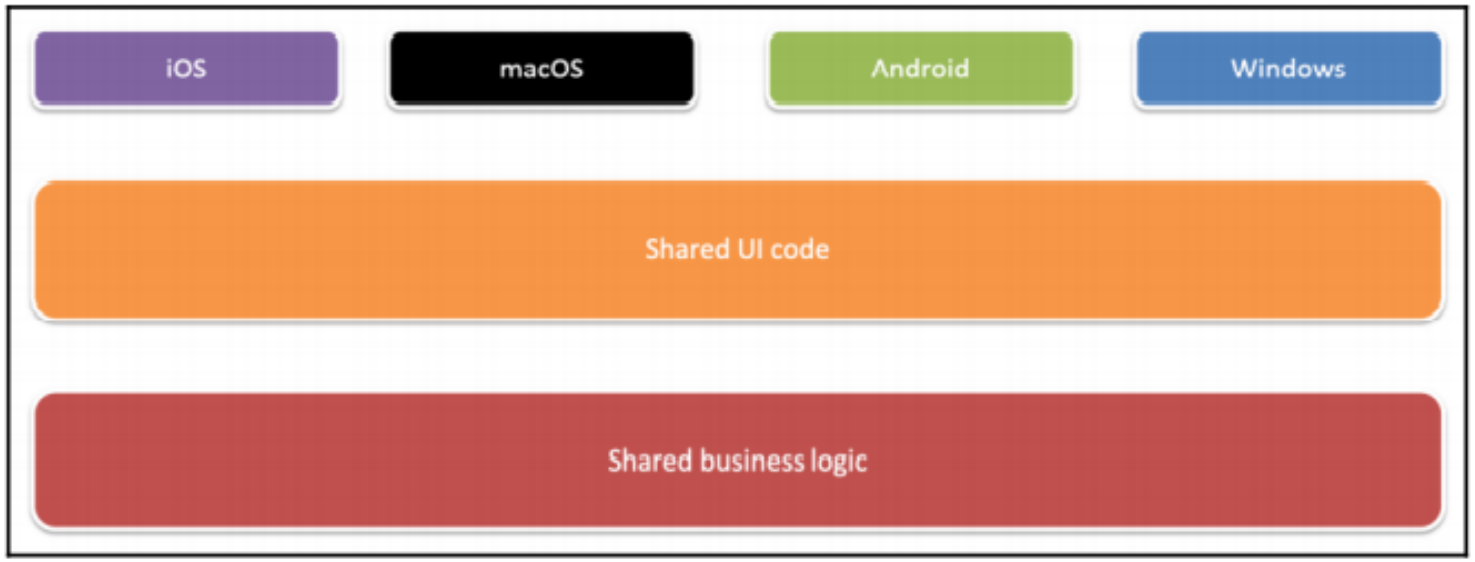
- Generally, using XAML provides a much better overview, since the page is a hierarchical structure of objects and XAML is a very nice way of defining that structure. In code, the structure is flipped around as we need to define the innermost object first, making it harder to read the structure of our page. This was demonstrated in the Creating a page in XAML section of this chapter. Having said that, it is generally a matter of preference as to how we decide to define the GUI. This book will use XAML rather than C# in the projects to come.

Xamarin.Forms versus traditional Xamarin

- While this book is about Xamarin.Forms, we will also highlight the differences between using traditional Xamarin and Xamarin.Forms. Traditional Xamarin is used when developing apps that use iOS and an Android Software Development Kit (SDK) without any means of abstraction. For example, we can create an iOS app that defines its UI in a storyboard or in the code directly. This code would not be reusable for other platforms, such as Android. Apps built using this approach can still share non-platform-specific code by simply referencing a .NET standard library. This relationship is shown in the following diagram:



Xamarin.Forms, on the other hand, is an abstraction of the GUI, which allows us to define UIs in a platform-agnostic way. It still builds on top of Xamarin.iOS, Xamarin.Android, and all the other supported platforms. The Xamarin.Forms app can be created as a .NET standard library or as a shared code project, where the source files are linked as copies and built within the same project as the platform we are currently building for. This relationship is shown in the following diagram:



Having said that, Xamarin.Forms cannot exist without traditional Xamarin since it's bootstrapped through an app for each platform. This gives us the ability to extend Xamarin.Forms on each platform using custom renderers and platform-specific code that can be exposed to our shared code base through interfaces. We'll look at these concepts in more detail later on in this chapter.

When to use Xamarin.Forms

- We can use Xamarin.Forms in most cases and for most types of apps. If we need to use controls that not are available in Xamarin.Forms, we can always use the platform-specific APIs. There are, however, cases where Xamarin.Forms is not useful. The most common situation where we might want to avoid using Xamarin.Forms is if we build an app that should look very different across our different target platforms.

Any Questions?