Chapter 5

Introduction to XAML

Introduction

In a .NET MAUI app, the **user interface** (**UI**) is basically constructed as a tree of objects. The UI is generally created using a markup language instead of C#. This markup language is an XML-based language and is called **eXtensible Application Markup Language** (**XAML**). The markup defines the .NET MAUI classes and property values. At runtime, the XAML is parsed, and the objects are instantiated and initialized.

Structure

In this chapter, we will discuss the following topics:

* Overview of XAML
* XAML syntax
* Namespaces
* Markup extensions
* Passing arguments
* Data binding

Objectives

By the end of this chapter, you will gain a clear understanding of:

* Basic concepts of XAML
* The syntax of XAML
* Different types of constructs like object-element syntax, property-element syntax, attached property syntax, and markup extensions.

Overview of XAML

XAML was originally developed forSilverlight by *Rob Relyea* and the team at Microsoft. As it is a very powerful and productive way for developers to build rich user interfaces, it has been adopted in many other technologies, including WP7, WP8, UWP, WinUI, Xamarin, and also in .NET MAUI. Though apps can be developed completely using C#, defining the UI in XAML is recommended because of several advantages.

The major advantages that XAML provides are the following:

* It allows you to separate the UI design from the code behavior.
* XAML is more compact when compared to implementing the same functionality in C#.
* XAML provides a visual representation of the UI structure.
* XAML has a great tooling support.
* XAML makes it possible to use powerful concepts such as data binding.

Each Page/View in a .NET MAUI file can be defined in XAML, and a companion **.cs** file is used to define the event handlers and methods containing the programming logic.

XAML syntax

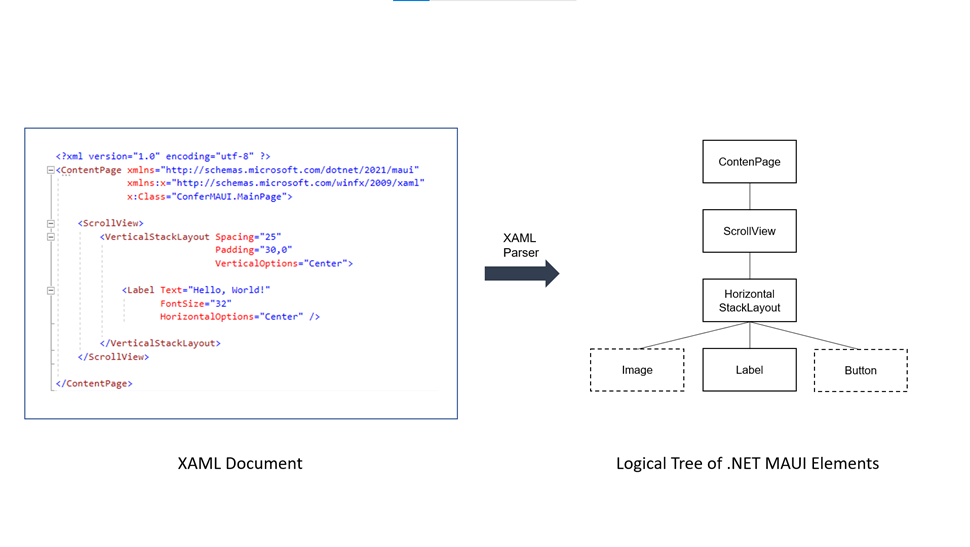
The syntax of XAML is similar to XML, with some additional features that follow these basic rules as in XML:

* The first line starts with an XML prolog, which is provided by default in all XAML files:

<?xml version="1.0" encoding="utf-8" ?>

* The rest of the XAML file consists of elements organized in a tree structure.

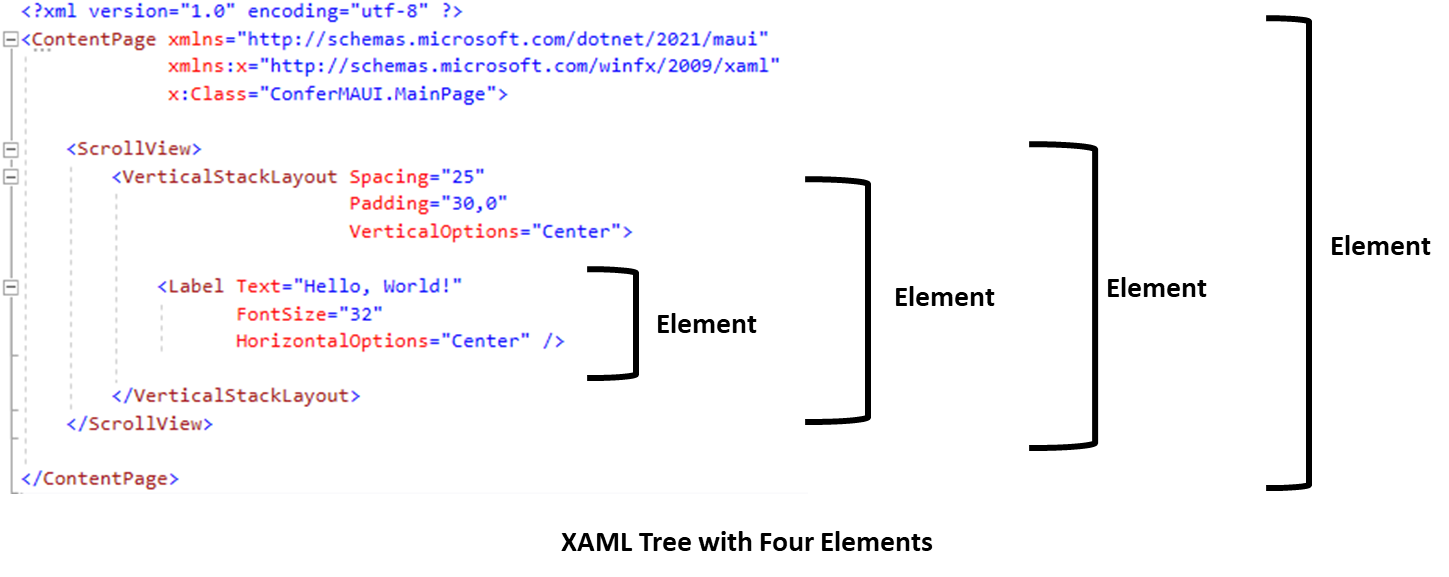
Refer to the following figure:



**Figure 5.1**:Logical Tree created from the XAML markup

* To create a user interface, the elements must be nested properly. The way the elements are nested determines the structure of the UI.
* The outermost element is called the root element and contains all the other elements.
* Each XAML file must contain exactly one root element, which is the parent of all other elements. In .NET MAUI, except for the **App.xaml** and **AppShell.xaml** files, the root element in all other XAML Files will be either a **ContentPage**, **ContentView** or **ResourceDictionary**.
* XAML elements represent objects in the UI. For example, a simple UI with four objects has four XAML elements that correspond to each of the objects in the logical tree.

Refer to the following figure:

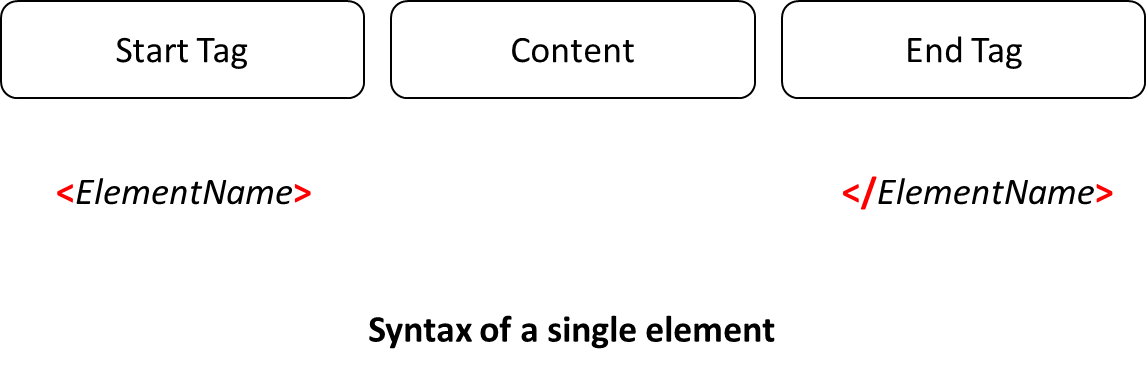


**Figure 5.2**: Nesting of elements in XAML

Object element syntax

Using XAML elements to represent .NET MAUI class objects is called object element syntax because each XAML element represents a .NET MAUI object.

The default syntax for an element has three parts: the start tag, the content section, and the end tag, as shown in the following figure:



**Figure 5.3**:Syntax of a single element

* The opening tag consists of the element name wrapped in angle brackets e.g. **<StackLayout>**
* The end tag will have a slash just before the element name e.g. **</StackLayout>**
* The content section can have text, other elements or white space

Comments

For commenting, use the following syntax:

1. *<!-- This is a comment -->*

Attribute syntax

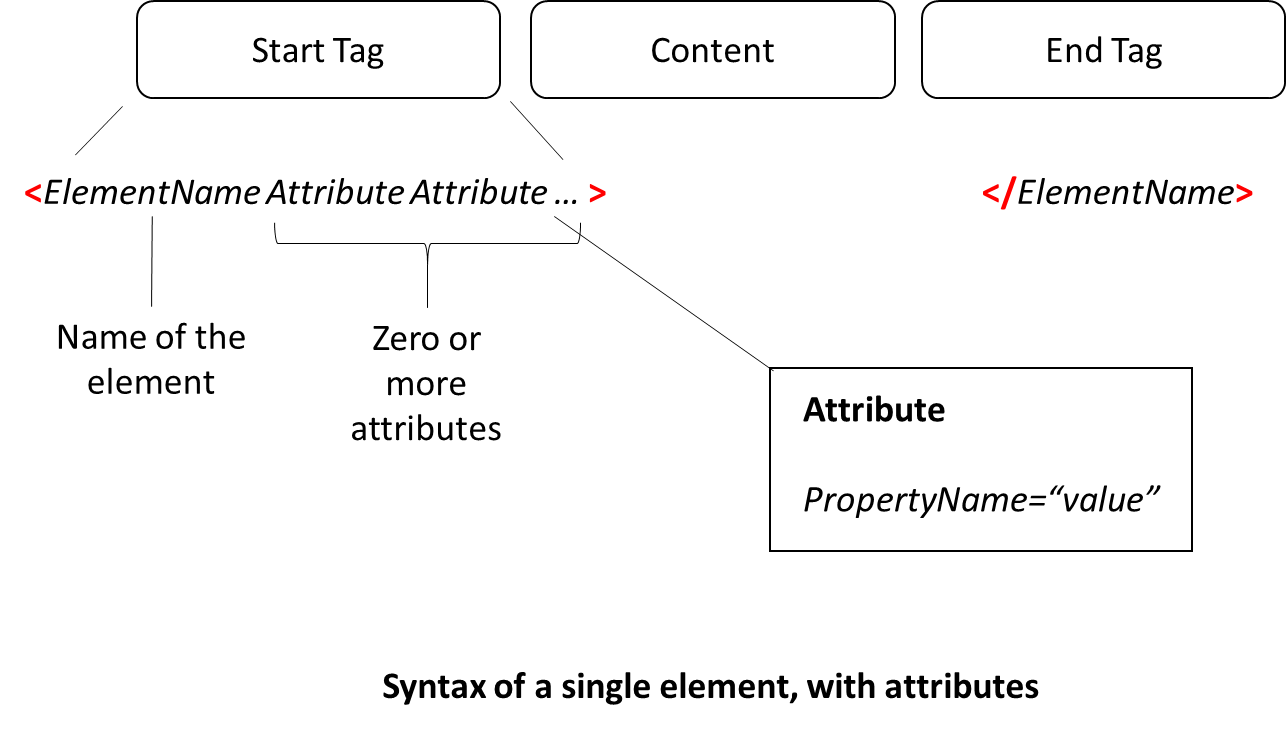
Remember each element corresponds to a class object. You can set the properties of the class object using attributes. For example:

1. <Button Width="75"  Height="25" />

In this example, it sets the values of two properties of a **Button** object: **Width** and **Height**:

* Attributes must be placed inside the start tag, after the element name, and cannot appear in the content section or end tag.
* The syntax of an attribute contains an identified followed by an = (equal to) sign.
* The property values must be set using double quotes.
* An element may have any number of attributes. If there are multiple attributes, they must be separated by a white space.

Refer to the following figure:



**Figure 5.4**:Syntax of a single element, with attributes

Empty element syntax

In case the element does not have any content, you can specify it in two ways:

* You can leave the content section blank or provide whitespace between the start and end tags:

1. <Button Width=”100” Height=”50”></Button>

Or,

* Use the empty element syntax, in which you have only a single tag, instead of the start and end tag. It resembles the start tag but, at the end of the tag, it uses **/|**instead of **>|**as the delimiter.

For example, refer to this command:

1. <Button Width=”100” Height=”50” />

Except the XML prolog, all elements must have either an end tag or a self-closing tag using the empty element syntax.

Property value syntax

For simple property values, attributes are helpful. For setting complex property values, we can use the property element syntax in the following way:

The property is not mentioned as an attribute but nested in the content section of the element.

For example, refer to this command:

1. <ElementName.PropertyName> Value </ElementName.PropertyName>

* This uses dot-syntax notation; the first part is the name of the class, and the second part is the name of the property.
* The XAML parser does not instantiate an object in this case, but sets the property on the object.
* The property element tag cannot contain attributes.

The following example shows how this is useful.

* Attribute syntax:

1. <VerticalStackLayout Background="LightBlue">
2. </VerticalStackLayout>

Property-element syntax:

1. <VerticalStackLayout>
2. <VerticalStackLayout.Background>
3. LightBlue
4. </VerticalStackLayout.Background>
5. </VerticalStackLayout>

In the above example, when the Background color property value is simple, attribute syntax is easier. However, if the property value is complex, like the following example, then property element syntax will have to be used:

In .NET MAUI we can apply a gradient color to the **Background** property using a **LinearGradientBrush**. This requires you to set two **GradientStops** with different colors, and .NET MAUI will take care of providing a smooth gradient between the two colors. Refer to the following code:

1. <VerticalStackLayout>
2. <VerticalStackLayout.Background>
3. <LinearGradientBrush StartPoint="0,0"
4. EndPoint="1,1">
5. <GradientStopCollection>
6. <GradientStop Offset="0"
7. Color="Red" />
8. <GradientStop Offset="1"
9. Color="Yellow" />
10. </GradientStopCollection>
11. </LinearGradientBrush>
12. </VerticalStackLayout.Background>
13. </VerticalStackLayout>

Attached property syntax

Attached properties are special in that it is defined in one class but used in another. Let us look at an example:

1. <Grid>
2. <Grid.RowDefinitions>
3. <RowDefinition Height="\*" />
4. <RowDefinition Height="\*" />
5. </Grid.RowDefinitions>
6. <Button Grid.Row="1" />
7. </Grid>

In this above example, the **Button** has an attribute assignment for **Grid.Row** property. This is not part of the **Button** class, but the **Row** property is defined in the **Grid** class. This is what is called an attached property.

When in doubt, observe the class name. If it is different from the class being instantiated, it is an attached property. In property element syntax, the class name is the same as the object being instantiated.

Namespaces

As mentioned earlier in the chapter, each part of an XAML tree corresponds to a specific .NET MAUI class. When the XAML parser needs to create objects of these types, it must know where to find their definitions. This information is communicated to the parser through something called XAML namespaces.

A XAML namespace is like a special name that represents a group of .NET MAUI classes. If you want to use a class as an element in your XAML markup, you need to tell the XAML parser which namespace the class belongs to.

Using namespaces in XAML is not like using namespaces in C#. In C#, you include namespaces with a bunch of using statements at the top of the code file, and then you can use the classes from those namespaces in your code (as long as you have a reference to the assembly containing the namespace).

In XAML, it is a bit different. Only one namespace can be the default. You can set the default namespace using the **xmlns** keyword followed by an equal sign and then a string that is the name of the default namespace. The strings look like website addresses, but they are not.

Other namespaces can be declared but need to be provided with a prefix. For this, you use the keyword **xmlns**, followed by a colon and a character or string, followed by an equal sign and the name of the namespace. The prefix that you declare will have to be mentioned wherever the classes or types from the respective namespace are being used.

In the following code, you can see the namespaces defined in the beginning of every XAML file:

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
3. xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4. x:Class="Some Class Name"
5. Title="Some Title">
6. </ContentPage>

Besides these, you may also add other namespaces, or from your own project. If you want to use a namespace that is not a XAML namespace, there is a special syntax to tell the XAML parser to use a **common language runtime** (**CLR**) namespace like this:

1. xmlns:local="clr-namespace:MyNameSpace"

Alternatively, you can write it as the following:

1. xmlns:local="using:MyNameSpace"

In case the namespace is in another assembly, then you can include the assembly name as well:

1. xmlns:local="clr-namespace:MyNameSpace; assembly=MyAssembly"

Object names

As mentioned earlier in the chapter, XAML is used to instantiate the objects in the UI of a .NET MAUI app. The actions performed by the objects are defined in a companion C# file, known as the code-behind file.

By default, XAML instantiates objects without a name, for example:

1. <Button Text=”Click”></Button>

However, in C#, whenever objects are created, a name has to be provided, for example:

1. Button btn = new Button();

The name is required to define further actions and behavior in the code-behind file.

When creating objects in XAML, you can provide a name by using the **x:Name** attribute:

1. <Button x:Name="btn" Text="Click" />

The **x** is the prefix mentioned in the standard namespace of the XAML document, which is used for elements and attributes that are intrinsic to XAML.

Markup extensions

While XAML is very powerful, there are some limitations. Let us understand these limitations and how we can deal with them:

* The first limitation is that XAML cannot evaluate values and perform conditional logic during runtime. For instance, let us imagine we have a Label in our UI, and we want to display AM if the time is before noon and PM if it is post-noon. In XAML, there is no direct way to perform this conditional logic at runtime to set the text accordingly.
* Another limitation is that when assigning a value to a property in XAML, we can use a value type or create a new object. XAML lacks a syntax for assigning an existing, non-static object to a property.

These are just a couple of limitations where pure XAML just falls short. To handle such situations and others requiring runtime evaluation, XAML needs a *hook* to an external code that is callable at runtime. The code must execute the necessary actions and return the value or object reference needed for XAML to complete its property assignment. This functionality is provided by something called a markup extension.

A markup extension acts as a connection to a class external to XAML, known as an extension class. The extension class is designed to be used by the markup extension.

In XAML, a markup extension is written by replacing the string on the right side of the equals sign in the attribute syntax. It begins and ends with curly braces.

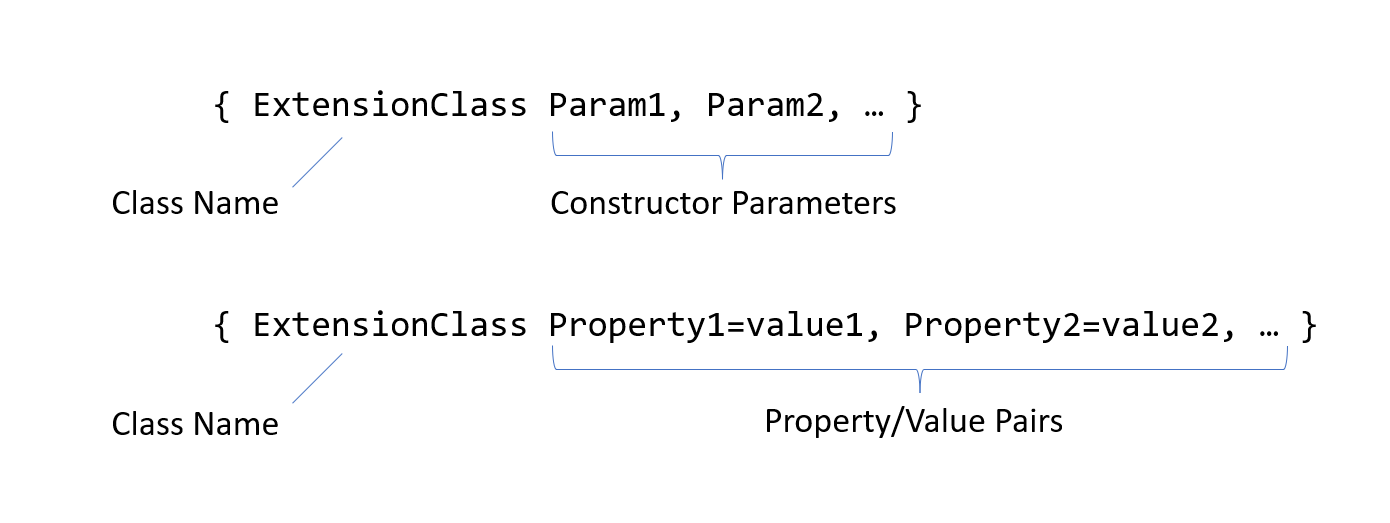
Here is a basic form of a markup extension:

1. <Button Style=”{StaticResource MyStyle}”

In the above example, we are setting the Style property of the Button using a markup extension called **StaticResource**. It searches the program’s resources for a style named **MyStyle** and, if found, returns its reference, which is then assigned to the **Style** property.

Markup extensions follow specific syntax forms, as shown in *Figure.5.5* The name of the extension class is the first string inside the open curly brace, and there is no comma following the class name. After the class name, there can be zero or more strings separated by commas. These strings can be either simple strings or property/value pairs.

Refer to the following figure:



**Figure 5.5**:Two syntax forms for markup extensions

Passing arguments

In .NET MAUI, sometimes, when creating objects in XAML, you might need to use constructors that need specific information. For this, you can make use of **x:Arguments**.

**x:Arguments** is used to tell the program what information to give to the constructor when creating the object. For example, when creating a color object, you can use the x**:Arguments** to provide details like how much red, green, and blue it should have. Your number should match the color needs, like grayscale or different color channels. The following XAML markup represents a **Grid** element with a background color defined by a **Color** object that takes three specified values (0.25, 0.4, 0.35) as arguments that define the RGB color components.

1. <Grid>
2. <Grid.Background>
3. <Color>
4. <x:Arguments>
5. <x:Color>0.25</x:Color>
6. <x:Color>0.4</x:Color>
7. <x:Color>0.35</x:Color>
8. </x:Arguments>
9. </Color>
10. </Grid.Background>
11. </Grid>

Additionally, if you are working with generic types (i.e., types that can work with different kinds of data), you can use x:TypeArguments to give specific details. Arguments can be passed to constructors using the following .NET MAUI XAML language primitives:

* **x:Array**, for **Array**.
* **x:Boolean**, for **Boolean**.
* **x:Byte**, for **Byte**.
* **x:Char**, for **Char**.
* **x:DateTime**, for **DateTime**.
* **x:Decimal**, for **Decimal**.
* **x:Double**, for **Double**.
* **x:Int16**, for **Int16**.
* **x:Int32**, for **Int32**.
* **x:Int64**, for **Int64**.
* **x:Object**, for the **Object**.
* **x:Single**, for **Single**.
* **x:String**, for **String**.
* **x:TimeSpan**, for **TimeSpan**.
* **x:FactoryMethod** is used to specify a special method that can be used to set up an object. This special method must be a public static method that returns objects or values of the same type as the class that defines the method.

Let us look at an example:

1. <Grid>
2. <Grid.Background>
3. <Color x:FactoryMethod="FromRgba">
4. <x:Arguments>
5. <x:Byte>192</x:Byte>
6. <x:Byte>75</x:Byte>
7. <x:Byte>150</x:Byte>
8. <x:Byte>128</x:Byte>
9. </x:Arguments>
10. </Color>
11. </Grid.Background>
12. </Grid>

Data binding

In most .NET MAUI apps, the UI represents some data and allows users to modify it. Moreover, when the underlying data changes, we would like the UI to be updated as well. In .NET MAUI, this is made possible easily using data binding.

Data binding, as the name suggests, is a way to connect the data in your application with the user interface elements. It ensures that the changes in data automatically reflect in the UI and vice versa without writing a lot of extra code. It makes the app more responsive and dynamic.

Let us first discuss a situation where you want to bind a **TextBox** with a Slider control. You would like the **TextBox** to show the value of the Slider, and when you move the slider, the **TextBox** should update to match.

One way to do this is to write code in the code-behind file to handle events when the Slider changes. Instead, data binding provides a much simpler approach and does the heavy lifting for you.

For data binding to work, we need a **Source** property and a **Target** property, and then we can implement a data binding between the two:

1. public Demo1()
2. {
3. InitializeComponent();
4. label.BindingContext = slider;
5. label.SetBinding(Label.ScaleProperty, "Value");
6. *// Alternatively, you can write a single statement like this:*
7. label.SetBinding(Label.ScaleProperty, new Binding("Value", source: slider));
8. }

It is even simpler to implement the data binding in XAML like the markup shown in the following code:

1. <StackLayout Padding="10, 0">
2. <Label x:Name="label"
3. Text="TEXT"
4. FontSize="48"
5. HorizontalOptions="Center"
6. VerticalOptions="Center">
7. <Label.Scale>
8. <Binding Source="{x:Reference slider}"
9. Path="Value" />
10. </Label.Scale>
11. </Label>
12. <Slider x:Name="slider"
13. Maximum="360"
14. VerticalOptions="Center" />
15. </StackLayout>

Data binding is not just useful for binding UI elements to each other. It can also be used to bind the UI element to the model. To illustrate this, let us consider a **Person** class with two properties:

1. public class Person
2. {
3. public string Name { get; set; }
4. public string Email { get; set; }
5. }

If we want to display the details of an object of the **Person** class in the UI, let us do it by adding two **Entry** elements to display the values:

1. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
2. xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
3. x:Class="DataBindingSessionDemos.Demo"
4. Title="Demo">
5. <VerticalStackLayout Spacing="20">
6. <Entry x:Name="lblName"
7. VerticalOptions="Center"
8. HorizontalOptions="Center" />
9. <Entry x:Name="lblEmail"
10. VerticalOptions="Center"
11. HorizontalOptions="Center" />
12. </VerticalStackLayout>
13. </ContentPage>

First,we will see how the data binding is done in C# in the code-behind file. Follow these steps:

1. For each UI element that we want to bind to data, we need a **Binding** object to be created by assigning the **Path** and the **Source** properties.
2. Then, we can invoke the **SetBinding** method of the elements and pass the required arguments—first, the property of the target to which we want to bind and second, the binding object that we just created in the previous step.

Refer to the following code:

1. public partial class Demo : ContentPage
2. {
3. Person p1;
4. public Demo()
5. {
6. InitializeComponent();
7. p1 = new Person { Name = "John Doe", Email = "j.doe@msn.com" };
8. var nameBinding = new Binding
9. {
10. Path = "Name",
11. Source = p1
12. };
13. lblName.SetBinding(Entry.TextProperty, nameBinding);
14. var emailBinding = new Binding
15. {
16. Path = "Email",
17. Source = p1
18. };
19. lblEmail.SetBinding(Entry.TextProperty, emailBinding);
20. }
21. }

Now, let us do the data binding in XAML. You will see how XAML syntax really shines through and makes the code simpler.

In the UI markup, only one change is required. In the two **Entry** elements, use the data binding syntax to map the two properties of the object to the **Text** properties of the **Entry** elements.

Refer to the following code:

1. <?xml version="1.0" encoding="utf-8" ?>
2. <ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
3. xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4. x:Class="DataBindingSessionDemos.ObjectDataBinding"
5. Title="ObjectDataBinding">
6. <VerticalStackLayout Spacing="25">
7. <Entry Text="{Binding Name}"
8. VerticalOptions="Center"
9. HorizontalOptions="Center" />
10. <Entry Text="{Binding Email}"
11. VerticalOptions="Center"
12. HorizontalOptions="Center" />
13. </VerticalStackLayout>
14. </ContentPage>

In the code-behind file, all you have to do is to create the **Person** object and assign it to the **BindingContext** property of the UI. Refer to the following code:

1. public partial class Demo : ContentPage
2. {
3. public ObjectDataBinding()
4. {
5. InitializeComponent();
6. this.BindingContext = new Person { Name="John Doe", Email = "j.doe@msn.com" };
7. }
8. }

In the above code snippet, you can see that the content in the code-behind file is much shorter, for the same functionality.

Note: Please note that this simple example was chosen to give a foundational understanding of XAML in data binding. The correct way is to extend the model class within the INotifyPropertyChanged interface. The following code is for your reference:

1. public class Person : INotifyPropertyChanged
2. {
3. private string name;
4. public string Name
5. {
6. get { return name; }
7. set
8. {
9. if (name != value)
10. {
11. name = value;
12. OnPropertyChanged("Name");
13. }
14. }
15. }
16. private string email;
17. public string Email
18. {
19. get { return email; }
20. set
21. {
22. if (email != value)
23. {
24. email = value;
25. OnPropertyChanged("Email");
26. }
27. }
28. }
29. public event PropertyChangedEventHandler PropertyChanged;
30. private void OnPropertyChanged(string propertyName)
31. {
32. if (this.PropertyChanged != null)
33. this.PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
34. }
35. }

We will discuss this in detail in the upcoming chapters when we use data binding in our projects.

Conclusion

In this chapter, you have learned how a .NET MAUI user interface is constructed like a tree. You can build this tree either by writing code in C# or using a language called XAML. We discussed some of the key points about XAML, including its syntax rules, attributes, markup extensions, namespaces, data binding, etc.

In the next chapter, we will apply the knowledge acquired so far to work on our first project and build a Color Picker app.

Points to remember

Here are some key takeaways from this chapter:

* XAML tree is a group of elements arranged like a family tree.
* There is always one element at the top of the tree called the root element.
* Each element in the tree corresponds to a class in .NET MAUI.
* Elements can have attributes which are just properties of the classes they represent.
* XAML syntax provides four types of constructs:
* Object element syntax
* Property-element syntax
* Empty element syntax
* Attached property syntax

Let us review the different XAML syntax forms that we discussed above:

|  |  |  |
| --- | --- | --- |
| **Name** | **Description** | **Example** |
| Object element syntax | Creates .NET MAUI object | <Button></Button> |
| Attribute syntax | Sets the properties of a .NET MAUI class object | <Button Text=”Click”>  </Button> |
| Empty element syntax | Creates .NET MAUI object without any content section | <Button Background=”Red” /> |
| Property element syntax | Sets the properties of a .NET MAUI class object | <Button.Background>  Red  </Button.Background> |
| Attached property syntax | Sets an attached property on a .NET MAUI class object | <Button Grid.Row=”1” /> |