CHAPTER 2

Building Xamarin.Forms Apps Using XAML

The eXtensible Application Markup Language (XAML, pronounced "zammel") is used to define user interfaces (UI) for frameworks such as the Windows Presentation Foundation (WPF), the Universal Windows Platform (UWP), and Xamarin.Forms. These XAML dialects share the same syntax based on the 2009 XAML specification but differ in their vocabularies, which may eventually be aligned under one XAML Standard.

Every XAML document is an XML document with one root element and nested child elements. In XAML, an element represents a corresponding C# class such as an application, a visual element, or control defined in Xamarin.Forms. The attributes of the elements represent the properties or events supported by the class. XAML provides two ways to assign values to properties and events—as an attribute of the element or as a child element. Either way, the attribute assigns the value of a property or wires an event to an event handler you write in C# in the code behind file.

As I mentioned in the intro, my other book, *Xamarin Mobile Application Development,* focused on creating UI for Xamarin.Forms using C#. This book is about creating UI using XAML. XAML helps you separate the visual design from the underlying business logic. XAML and accompanying code behind files are written using Visual Studio or Visual Studio for Mac.

In this chapter, we will focus primarily on XAML syntax to help you read and write XAML. We'll begin with basic XAML syntax: tags can form elements which can be decorated with attributes which are property/value pairs, all of which are nested into a hierarchy. XAML elements can represent real classes and their members. We'll use namespaces to extend the vocabulary available in an XML document. XAML syntax employs a number of approaches for the definition of elements and attributes ranging

D. Hermes and N. Mazloumi, *Building Xamarin.Forms Mobile Apps Using XAML*,
https://doi.org/10.1007/978-1-4842-4030-4_2

from the property element syntax to the collection syntax. Each XAML file has a C# code behind. The XAML Standard is the holy grail of XAML development so we'll touch on it.

Since XAML is based upon XML, let us first delve into basic XML syntax.

Basic Syntax

Xamarin.Forms XAML is based on XML and the 2009 XAML specification. A basic understanding of these two languages is essential to be able to read and write XAML effectively.

The XML syntax determines the basic structure of XAML files comprised of elements, attributes, and namespaces. The 2009 XAML specification applies XML to the realm of programming languages where elements represent classes and attributes class members. XAML adds basic data types, vocabulary to name and reference elements, and approaches to construct objects using constructors and factory methods of classes.

For some of you, the next few paragraphs may be a review, but if you're not up on your XML skills, then read carefully. Let's start with the basic structure of a XAML document based on XML.

XML Syntax

At the core of XAML is the eXtensible Markup Language (XML). The main building blocks of an XML document are elements, attributes, hierarchy, and namespaces. Elements are entities declared using begin and end tags and defined using tag-encased data or other tags. Attributes are properties assigned to an element. A hierarchy is the structure created using nested elements. Next we'll look at each in turn.

Element

The declaration of an element uses the *element syntax*, so it has a begin and end tag surrounding the element values. Use the element syntax to declare a `Label` view and to assign `"Some Text"`:

```
<Label>Some Text</Label>
```

In an empty `Label`, the end tag can be omitted by adding a forward slash at the end of the begin tag, like this:

```
<Label/>
```

Attribute

Ascribe metadata to elements using *attributes*, which can be assigned a value. The *attribute syntax* is used to assign primitive values to an attribute by placing the attribute name inside the begin tag of an element, and its value is stored in double or single quotes following an equal sign. Use the attribute syntax to assign a value to the `Text` property of `Label`:

```
<Label Text="Some Text"/>
```

Hierarchy

A typical XML document is comprised of many nested elements, referred to as a *hierarchy*. In Chapter 1, Listing 1-7, a sample page is defined, comprised of a `ContentPage` element, which includes a `StackLayout` element with several child views such as `Label` and `Button`. This makes XML particularly interesting for user interface design, where pages contain layouts and views. Use a `ContentPage` with a `StackLayout` that includes a `Label` and a `Button` to define the hierarchy of a page, as outlined in Listing 2-1.

Listing 2-1. Hierarchy of XML Elements

```
<ContentPage>
    <StackLayout>
        <Label Text="This control is great ..."/>
        <Button Text="Make It So"/>
    </StackLayout>
</ContentPage>
```

Tip In XAML, element and attribute names correspond to class and member names in C#.

XML Namespaces

Namespaces extend the vocabulary available in an XML document, allowing the use of more uniquely defined elements and attributes. Each namespace is given a prefix to avoid ambiguity within an XML document in case multiple namespaces are used that may have elements or attributes with identical names. Add a namespace to an XML document using the XML `xmlns` attribute with the syntax `xmlns:prefix="URI"`. An element can have unlimited `xmlns` attributes for as long as the prefix is unique. For one `xmlns` declaration in the XML document, the prefix can be omitted, which makes the vocabulary of that namespace the default. All elements in the XML without a prefix belong to that namespace. Listing 1-7, in Chapter 1, adds the XAML and the Xamarin.Forms namespaces to the `ContentPage` element using

```
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

In Xamarin the default namespace is reserved for the Xamarin.Forms namespace, which is why `ContentPage`, `StackLayout`, `Label`, and `Button` have no prefix. For XAML terms the prefix `x` needs to be added, e.g., `x:TypeArguments`, which is used in Chapter 1, Listing 1-6, to specify the platform-specific `Thickness`. Both XAML and Xamarin.Forms use as the Uniform Resource Identifier (URI) simply a Uniform Resource Locator (URL) for that matter, which is not further evaluated other than being unique.

Tip XML namespaces can be declared on any element. However, in Xamarin.Forms all namespaces must be defined in the root element, e.g., `ContentPage`.

Those are the key syntaxes in XML, so now let's move on to XAML.

XAML Syntax

The 2009 XAML specification gives us a way to describe classes and class members in a declarative way using XML elements and attributes. Namespaces behave similarly to the `using` keyword in C#, allowing class libraries to extend the vocabulary available in XAML. XAML already comes with its own vocabulary including basic data types, markup extensions to extend the basic syntax with classes backed by code, and approaches to name and reference elements and to specify to the runtime how to construct objects.

Tip XAML does not allow code or conditional expressions such as `for`, `while`, `do`, and `loop` inside the XML document.

At the end of this topic, the list of all XAML terms used in Xamarin.Forms is provided as a reference.

Classes and Members

In XAML, XML elements represent actual C# *classes* that are instantiated to objects at runtime. The *members* of a class are represented as XML attributes. At runtime, the assigned attribute value is used to set the value of the property of an object. The attribute name corresponds with the member name of a class. The `Label` element with the attribute `Text` in Listing 2-1 `<Label Text="This control is great ..."/>` represents a class `Label` that has a `public` member called `Text`. At runtime, an object of type `Label` will be instantiated and the value of its `Text` property will be set to `"This control is great ..."`. Use the attribute syntax to assign values of primitive types as `string`, `bool`, `double`, and `int` to an attribute. At runtime, these are projected to `String`, `Boolean`, `Double`, and `Int32` objects.

XAML Namespaces

Adding a namespace in XAML is equivalent to the `using` directive in C# and makes a C# namespace available to the XAML document, allowing any of the classes in that namespace to be used as elements in the XAML. XAML itself is added as a namespace to a `ContentPage` like this:

```
<ContentPage xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="FormsExample.ContentPageExample"/>
```

The URI points to Microsoft's web site, and the `x` prefix means that XAML elements and attributes have to use this prefix inside the document. Use `x:Class` to specify the C# name of a `ContentPage`, like this:

```
x:Class="FormsExample.ContentPageExample"
```

This defines that the class `ContentPageExample` in the namespace `FormsExample` is a subclass of `ContentPage`. This also means that there is an associated code behind file that contains your class definition for `ContentPageExample`, as demonstrated in Chapter 1, Listing 1-7.

In XAML the `xmlns` attribute in combination with the *Common Language Runtime* namespace (`clr-namespace`) and the `assembly` name can be used to load namespaces and libraries to the XAML document that are available within a project. For the sake of simplicity, we'll reference a system library, though typically we reference our local namespaces in the project. Listing 2-2 demonstrates how to use the .NET `System` library in the assembly `mscorlib.dll` in XAML in order to use `System.String` to assign a string literal to a `Label`.

Listing 2-2. Adding External Class Libraries

```
<ContentPage xmlns:sys="clr-namespace:System;assembly=mscorlib" ...>
    <Label><sys:String>Hello System.String</sys:String></Label>
</ContentPage>
```

The colon sign is used when specifying the namespace and the equal sign when specifying the assembly. The assembly name must correspond to the actual library that is referenced in your Xamarin project without the `.dll` file extension, which is the case for the majority of NuGet package names.

Markup Extensions

Markup extensions extend the basic XML syntax, are backed by code, and can perform specific tasks. You can use the attribute or element syntax to specify a markup extension. To distinguish a markup extension from a string literal, use curly braces when using the attribute syntax such as `{x:Static Color.Maroon}`.

Tip In Xamarin.Forms any class that implements the `IMarkupExtension` interface and its method `ProvideValue` is a markup extension. All XAML markup extensions are implemented through this mechanism.

The intrinsic XAML markup extensions also supported by Xamarin.Forms include

- `Static`
- `Array`
- `Type`
- `Reference`

Let's look at each of them in detail.

Static

The *Static* markup extension is used to access static fields, properties, and constant fields as well as enumeration members. These do not need to be `public` as long as they are in the same assembly. In Chapter 1, the declaration `<BoxView Color="Maroon"/>` uses the `Color Maroon`, which is a `static` member of the class `Color`. With `Static` we can achieve the same result:

```
<BoxView Color="{x:Static Color.Maroon}" WidthRequest="150"
HeightRequest="150"/>
```

Alternatively, to the attribute syntax, the element syntax can be used when working with markup extensions, as shown in Listing 2-3.

Listing 2-3. Markup Extensions Using Element Syntax

```
<BoxView WidthRequest="150" HeightRequest="150">
    <BoxView.Color>
        <x:Static>Color.Lime</x:Static>
    </BoxView.Color>
</BoxView>
```

Figure 2-1 shows the `Maroon` and `Lime` boxes on iOS and Android platforms.

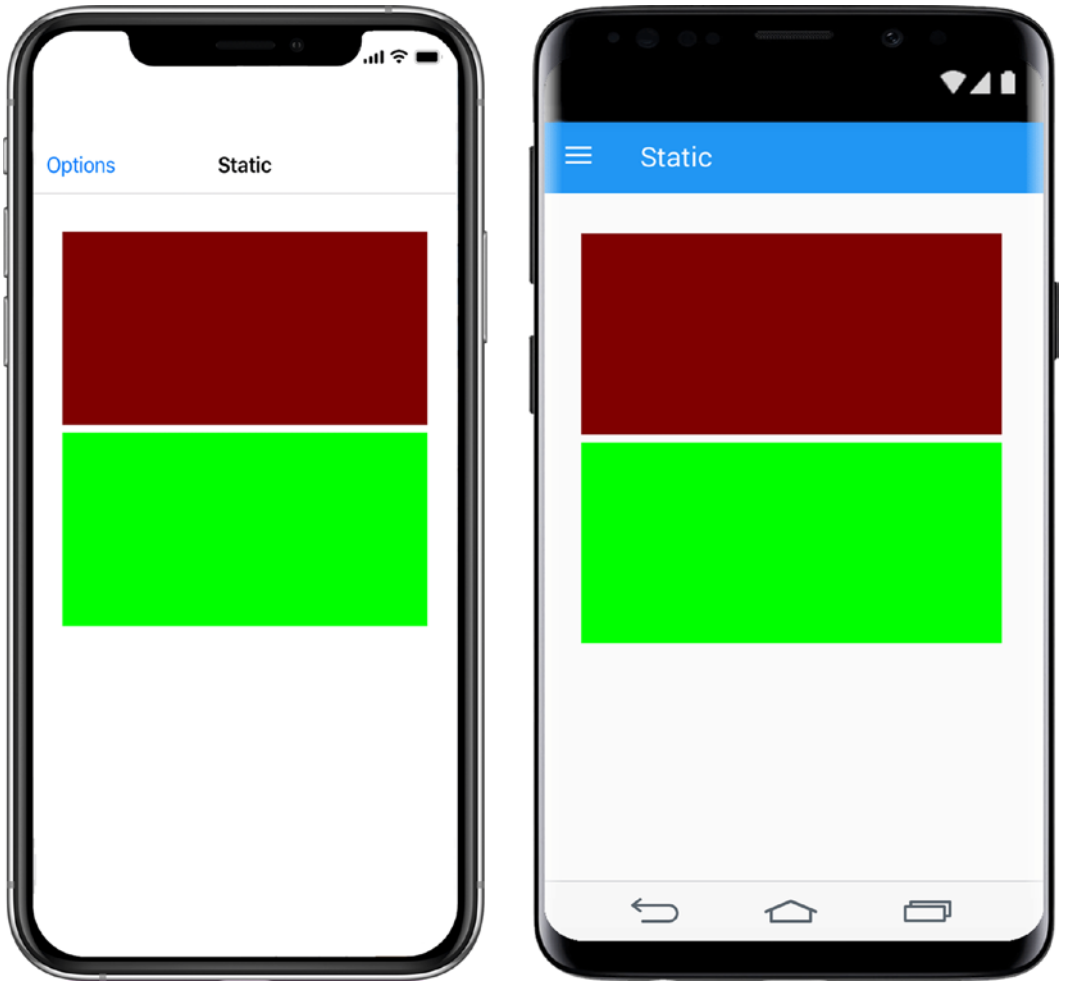

Figure 2-1. *Maroon and Lime BoxViews using Static to assign a value*

Array

Use the *Array* markup extension to define arrays with objects of a specific Type as shown in Listing 2-4 to create an Array of Strings.

Listing 2-4. Using Array

```
<x:Array Type="{x:Type x:String}">
    <x:String>A</x:String>
    <x:String>B</x:String>
</x:Array>
```

Use a `Picker` view to create a drop-down list, by assigning an `Array` to the `Picker's ItemsSource`, like this:

```
<Picker><Picker.ItemsSource><x:Array>...</x:Array></Picker.ItemsSource>
</Picker>
```

Figure 2-2 shows the result on both platforms.

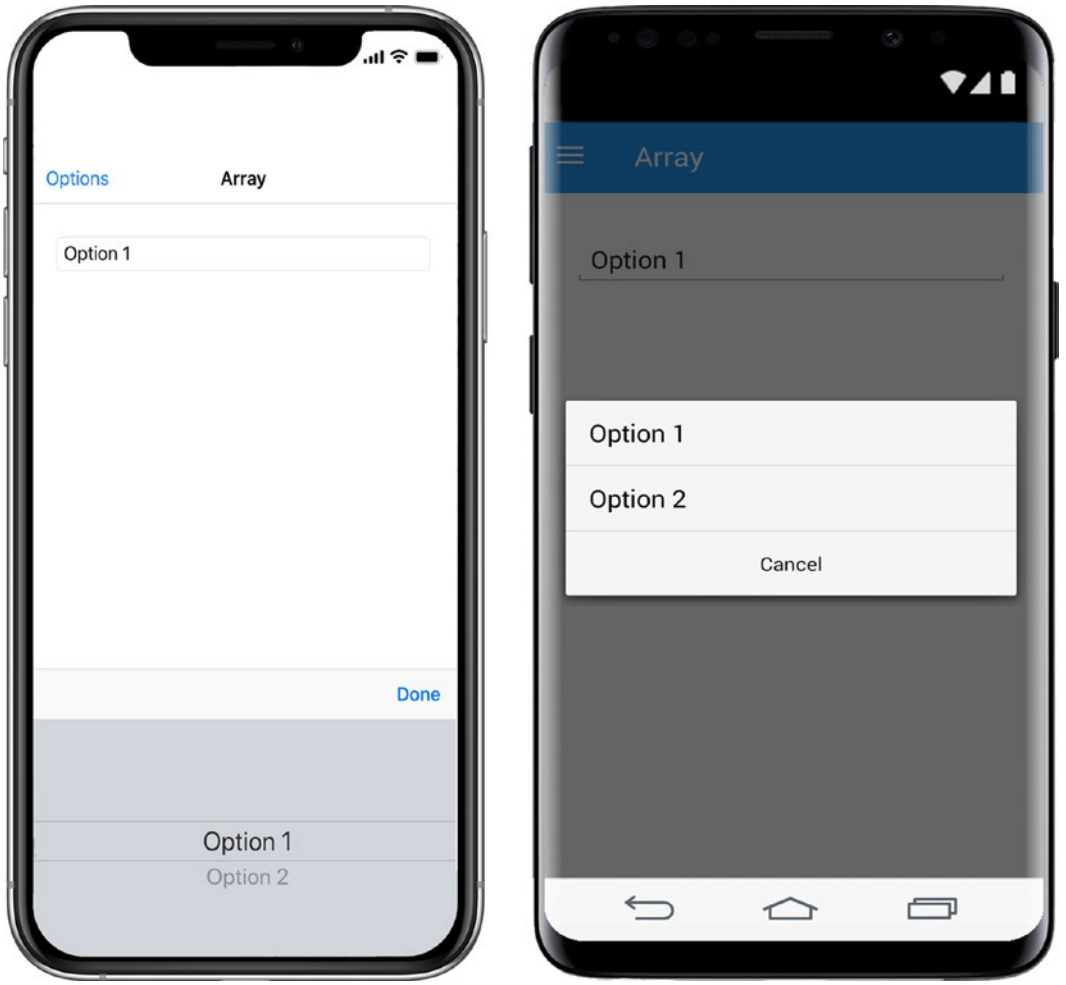

Figure 2-2. *Using Array as the ItemsSource of a Picker view*

CODE COMPLETE: Array Markup Extension

Listing 2-5 provides the complete code for creating a `Picker` that uses an `Array` as the `ItemsSource`.

Listing 2-5. Using Array

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage Title="Array"
xmlns:sys="clr-namespace:System;assembly=mscorlib"
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamlExamples.ArrayPage">
    <ContentPage.Content>
        <StackLayout Padding="30,30">
            <Picker>
                <Picker.ItemsSource>
                    <x:Array Type="{x:Type x:String}">
                        <x:String>Option 1</x:String>
                        <x:String>Option 2</x:String>
                    </x:Array>
                </Picker.ItemsSource>
            </Picker>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Type

Use *Type* to specify the data type of a value. The value of `Type` is the name of a `Type` object. Specify that objects in the `Array` are of type `String`:

```
<x:Array Type="{x:Type x:String}">
```

Reference

The *Reference* markup extension is used in combination with the `Name` directive to reference an object previously declared in the XAML. Use `Name` to assign a unique name to a `Label` view and `Reference` in an `Entry` control to reference the `Label` by its name in order to link the two `Text` properties, as shown in Listing 2-6.

Listing 2-6. Using x:Reference

```
<Label x:Name="MyLabel" Text="Hello Entry" />
<Entry Text="{Binding Path=Text, Source={x:Reference MyLabel}}" />
```

The example demonstrates the use of the Xamarin.Forms markup extension `Binding`, which is covered more in depth in Chapter 9. However, in the preceding example, we first use `Binding` to assign the `Label` view as the `Source` of the `Entry` control, that is, `Source={x:Reference MyLabel}`, and then link the `Text` property of `Label` to the `Text` attribute of the `Entry` through `Text="{Binding Text, ...}"`.

The `Binding` markup extension demonstrates two other concepts related to markup extensions:

1. *Multiple properties*: Markup extensions are essentially C# classes with public members. Use a comma to assign values to multiple members, e.g., `{Binding Path="", Source=""}`.
2. *Nesting*: The values assigned to the properties of a markup extension can be objects. Use nested curly braces to assign complex values to a property, e.g., `Source={x:Reference MyLabel}`. The `Reference` markup extension is nested inside the `Binding` markup extension. At runtime, the innermost markup extension is evaluated first.

Figure 2-3 shows the result on both platforms.

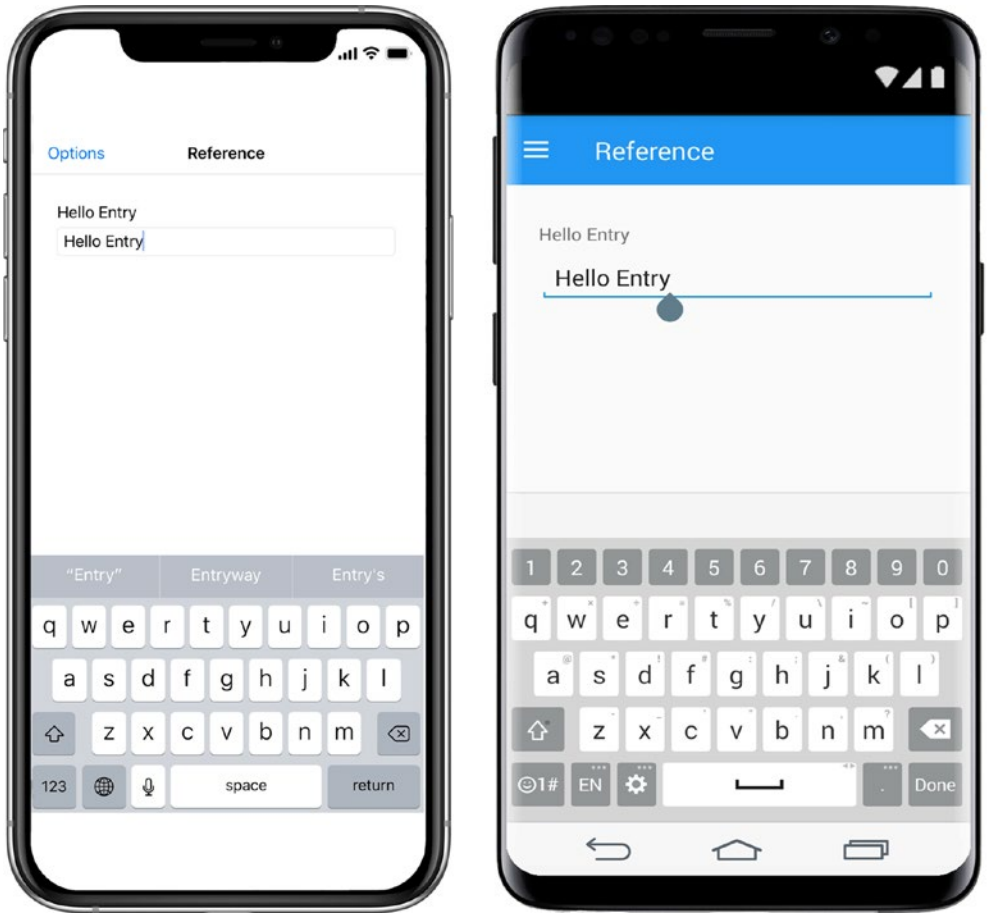

Figure 2-3. *Binding Label as the Source to Entry and linking the two Text properties*

CODE COMPLETE: Reference Markup Extension

Listing 2-7 provides the complete code for creating a `Label` that is referenced by an `Entry` as the `Source`.

Listing 2-7. Using Reference

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage Title="Reference"
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="XamlExamples.ReferencePage">
    <ContentPage.Content>
        <StackLayout Padding="30,30">
            <Label x:Name="MyLabel" Text="Hello Entry" />
            <Entry Text="{Binding Path=Text, Source={x:Reference
            MyLabel}}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

Tip If the default value of a property is not `null`, use the XAML `Null` markup extension to set the value of a property to `null`, e.g., `<Label Text="{x:Null}"/>`.

Constructors

Each Xamarin.Forms XAML element provides a built-in *default constructor* to allow the runtime to instantiate an object without depending on any particular property. Values specified to attributes are assigned to the object properties after the object is instantiated. Some classes also have constructors that expect arguments or even *factory methods,* which are `public static` methods that may accept arguments and return an object. Let's discuss each of these approaches.

Default Constructor

All views in Xamarin.Forms have a built-in default constructor. The empty element tag can be used without any attributes to create an instance of the class it represents. Use the empty element `DatePicker` to instruct the runtime to create an instance of the view to select a date, like this:

```
<DatePicker/>
```

Non-default Constructor

Some Xamarin.Forms classes have additional constructors that require passing in arguments, referred to as *non-default constructors*. The `Color` class in Xamarin.Forms has several non-default constructors. Use the `Arguments` element to pass arguments to a constructor. The number of arguments must match one of the `Color` constructors. A single `Double` argument is used for grayscale colors; three `Double` parameters are used to construct a `Color` from red, green, and blue values; and four `Double` values are used to create a `Color` also passing in the alpha channel, as shown in Listing 2-8, to set the `Color` for a `BoxView`.

Listing 2-8. Utilizing Constructors and Passing in Parameters Using x:Arguments

```
<BoxView>
    <BoxView.Color>
        <Color>
            <x:Arguments>
                <x:Double>0.25</x:Double>
                <x:Double>0.75</x:Double>
                <x:Double>0.2</x:Double>
                <x:Double>0.9</x:Double>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>
```

Factory Method

Some Xamarin.Forms classes provide publicly accessible `static` methods, also known as *factory methods,* to construct an object. XAML provides the `FactoryMethod` attribute to specify the factory method an element should use in order to construct an object. The `Color` class has several factory methods, that is, `FromRgb`, `FromRgba`, `FromHsla`, and `FromHex`, to create a `Color` instance. Use `FactoryMethod` attribute inside the `Color` element begin tag to specify the factory method followed by the `Arguments` element to provide the parameters, as shown in Listing 2-9.

Listing 2-9. Constructing Objects Using Factory Methods

```
<BoxView>
    <BoxView.Color>
        <Color x:FactoryMethod="FromHex">
            <x:Arguments>
                <x:String>#02dd52</x:String>
            </x:Arguments>
        </Color>
    </BoxView.Color>
</BoxView>
```

Figure 2-4 shows the result on both platforms.

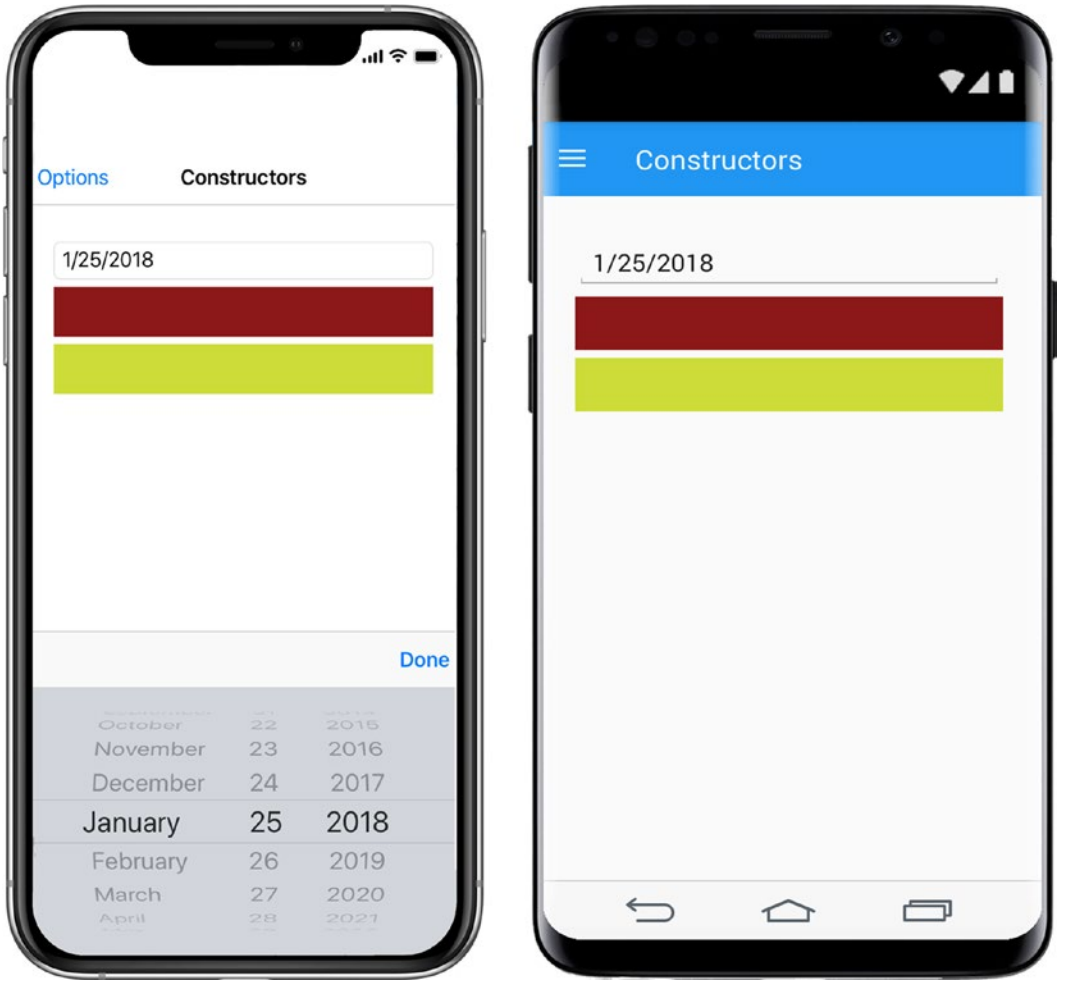

Figure 2-4. *Constructing views using the default constructors, non-default constructors, and factory methods*

CODE COMPLETE: XAML Constructors

Listing 2-10 provides the complete code for constructing objects using the default constructors, non-default constructors, and factory methods.

Listing 2-10. Default and Non-default Constructors and Factory Methods in XAML

```
<?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="XamlExamples.ConstructorsPage">
        <ContentPage.Content>
            <StackLayout Padding="30,30">
                <DatePicker />
                <BoxView>
                    <BoxView.Color>
                        <Color>
                            <x:Arguments>
                                <x:Double>0.5</x:Double>
                                <x:Double>0.0</x:Double>
                                <x:Double>0.0</x:Double>
                                <x:Double>0.9</x:Double>
                            </x:Arguments>
                        </Color>
                    </BoxView.Color>
                </BoxView>
                <BoxView>
                    <BoxView.Color>
                        <Color x:FactoryMethod="FromHex">
                            <x:Arguments>
                                <x:String>#CDDC39</x:String>
                            </x:Arguments>
                        </Color>
                    </BoxView.Color>
                </BoxView>
            </StackLayout>
        </ContentPage.Content>
    </ContentPage>
```

Let's complete the XAML syntax topic with an overview of all XAML terms available in Xamarin.Forms.

XAML Terms

Xamarin.Forms supports a subset of the terms defined in the 2009 XAML specification, the majority of which we have discussed in this chapter. The purpose of this section is to provide a summary as a reference. The terms can be grouped into basic types that represent the respective C# type defined in the System namespace, keywords used to identify and reference elements, and terms used to construct objects:

- *Simple data types*: The following XAML basic types are supported by Xamarin.Forms. Use these terms to represent their corresponding types defined in the `System` namespace.
  - *`Null`*: Use the XAML Null markup extension to set the value of a property to `null`.
  - *`Array`*: Use `Array` to define arrays with objects of a specific `Type`.
  - *`Type`*: Use `Type` to specify the data type of a value.
  - *`Object`*: Represents `System.Object` and is useful if you want to create an array that can expect any type, e.g.: `<x:Array Type="{x:Type x:Object}">...</x:Array>`
  - *`Boolean`*, *`Byte`*, *`Int16`*, *`Int32`*, *`Int64`*, *`Single`*, *`Double`*, *`Decimal`*, *`Char`*, *`String`*, and *`TimeSpan`*: These are mapped to the corresponding simple type in C#.
  - *`DateTime`*: This type does not exist in the 2009 XAML specification and was added by Xamarin.Forms. Use `DateTime` to specify a date and time of day.
- *Classes, Identifiers, and References*: Use terms in this category to identify classes, name elements and reference them:
  - *`Class`*: Use `Class` in the root element of a XAML document to wire the element with its underlying C# class.
  - *`Key`*: Use `Key` register and uniquely identify a resource in a dictionary.

    - *Name*: Use `Name` to assign a unique name to an element and have Xamarin.Forms create a local variable with this name created for you in the code behind.
    - *Reference*: Use `Reference` in XAML to reference a previously named element.
    - *Static*: Use `Static` to access static properties, fields, constants, or enumeration values.
- *Constructing objects*: Use the following terms to instantiate objects.
    - *Arguments*: Use this term to pass arguments to a non-default constructor or a factory method.
    - *TypeArguments*: Use TypeArguments to instantiate classes that use generics such as `List<T>` or `Dictionary<T,T>`. Using the System namespace, you can define your own dictionary in XAML `<sys:Dictionary x:TypeArguments="sys:String,sys:Object">` that instantiates a `Dictionary` object at runtime with `string` as the key type and `object` as the value type.
    - *FactoryMethod*: Use FactoryMethod for elements that have a static method defined in the C# class and return an instance of the element.

Now that we've covered the important aspects of XAML syntax, let's move on to Xamarin.Forms syntax.

Xamarin.Forms Syntax

Xamarin.Forms syntax uses the element and attribute syntax introduced in XML to extend the functionality available in XAML. Six approaches are made available:

- *Property element syntax*: Use the property element syntax if the value that is being assigned is a complex object and cannot be represented by a string literal. Property elements can also specify platform-specific values using the `OnPlatform` tag.

- *Content property syntax*: Classes can have one of their members defined as a content property, which serves as a default property for the view. For brevity, this property name can then be omitted in the XAML, and the property value can be declared between the element's begin and end tag.
- *Enumeration value syntax*: Use this syntax to pass or assign a constant name of an enumeration to a property.
- *Event handler syntax*: Use the event handler syntax to wire a property that represents an event to the event handler defined in the code behind.
- *Collection syntax*: Some properties represent collections. Use the collection syntax to assign elements as children of the collections.
- *Attached property syntax*: Extend the functionality of elements using attached properties to define new properties for an element that elements have not defined themselves.

Let's examine each approach.

Property Element Syntax

A common approach to assign values to object properties is to use XML element tags instead of an attribute using the `class.member` notation for the element name. This is referred to as *property element syntax*. Use `Label.Text` to assign to the `Text` attribute of the `Label` element, e.g.:

```
<Label>
    <Label.Text>Hello</Label.Text>
</Label>
```

Content Property Syntax

In Xamarin.Forms, each element can have a default property where its value is assigned between the element's begin and end tags. Views can declare one of their properties as a content property using the C# attribute `ContentProperty`, e.g.:

```
[ContentProperty("Text")]
public class Label : View {}
```

`ContentProperty` indicates that the property can be omitted when using the property element syntax, which is referred to as *content property syntax*. In the following example, `<Label.Text>` can be omitted entirely, that is:

```
<Label>Hello</Label>
```

The content property syntax reduces the verbosity of the XAML document. Most of the Xamarin.Forms views, layouts, and pages specify a content property, such as the `Content` property of `ContentPage`. This means that the start and end tags `<ContentPage.Content>` and `</ContentPage.Content>` can also be omitted entirely in Listing 2-10.

Enumeration Value Syntax

Many classes in Xamarin.Forms use enumerations to restrict the values a member can be assigned to. The *enumeration value syntax* is based on the attribute syntax where the string literal assigned represents the constant name in an enumeration. Use the `NamedSize` enumeration to assign a platform-specific size to the `FontSize` attribute of `Button`, e.g.:

```
<Button FontSize="Medium" Text="Medium Size Button" />
```

In the example, `Medium` is assigned as the size to the `FontSize` property. Xamarin.Forms uses the built-in value converter class `FontSizeConverter` to evaluate the string literal, first trying to convert it to a `Double` and if that fails calling the `Device.GetNamedSize` method to convert the constant name `Medium` to the device-specific double value.

Some attributes allow a combination of enumeration values. These are referred to as *flags attributes*, which indicates that the enumeration is treated as a bit field. Use a comma to assign multiple flags to the `FontAttributes` property of `Button`, that is:

```
<Button FontAttributes="Italic,Bold" Text="Italic Bold Button" />
```

Event Handler Syntax

The *event handler* syntax is based on the attribute syntax and provides the foundation of XAML behaviors, commands, and triggers. Write an event handler in the code behind and wire them to Xamarin.Forms views to respond to user interactions. Specify the name

of the event supported by a particular Xamarin.Forms view as attribute name and the name of the C# event handler as the attribute value, e.g.:

```
<Button Text="Make It So" Clicked="ButtonClicked" />
```

In Listing 1-7 `Clicked="ButtonClicked"` registers the event handler `ButtonClicked` with the event `Clicked` defined in the `Button` class. The runtime takes care of registering the handler to the event, and the garbage collection removes the handler when the `Button` view is destroyed. In the code behind, define the event handler to change the `Text` of the `Button` to `"It is so!"` once the user clicks the `Button`, as shown in Listing 2-11.

Listing 2-11. Code Behind Event Handler

```
protected void ButtonClicked(object sender, EventArgs e) {
    ((Button)sender).Text = "It is so!";
}
```

It is recommended to declare an event handler as `protected` or even `private`. The `sender` argument of type `object` refers to the `Button` view in the XAML that is wired to this event handler. You can cast it to `Button` object, e.g., `(Button)sender` or `sender as Button`. The second argument represents the event object.

Call an asynchronous event using the `async`/`await` syntax, as demonstrated in Listing 2-12.

Listing 2-12. Asynchronous Event Handler Using async/await

```
private async Task<bool> ButtonClicked(object sender, EventArgs e) {
    var b = sender as Button;
    b.Text = "It is so!";
    return await Task.FromResult(true);
}
```

The asynchronous method `ButtonClicked` returns `true` after the `Task.FromResult` method completes.

Collection Syntax

Xamarin.Forms `Layout` subclasses such as `StackLayout` or `Grid` act as containers and have a `Children` property that is declared as content property and is omitted in XAML. The *collection syntax* uses the content element syntax to add a `Label`, `Button`, and `Grid` to `StackLayout`, as shown in Listing 2-13.

Listing 2-13. Using Collection Syntax to Add Child Elements to a Container

```
<StackLayout Padding="30,30">
    <Label/>
    <Button/>
    <Grid/>
</StackLayout>
```

The `Children` collection is read-only. Xamarin.Forms uses the `Add` method internally for each object that is instantiated at runtime to add the object to the `Children` collection.

Attached Property Syntax

Some classes in Xamarin.Forms need to assign values to an element without the element even having that property. This is achieved using the *attached property syntax*, which is based on the property element syntax. The `Layout Grid` requires its `Children` to be positioned in rows and columns. Create `Grid.Row` and `Grid.Column` as new attributes of `Label` to place the view inside a cell, e.g.:

```
<Grid>
    <Label Grid.Row="1" Grid.Column="1" Text="Cell (1,1)" />
</Grid>
```

This positions the Label in the first row and column of the `Grid`. Attached properties can be simple or complex objects that encapsulate business logic.

Figure 2-5 shows the result on both platforms.

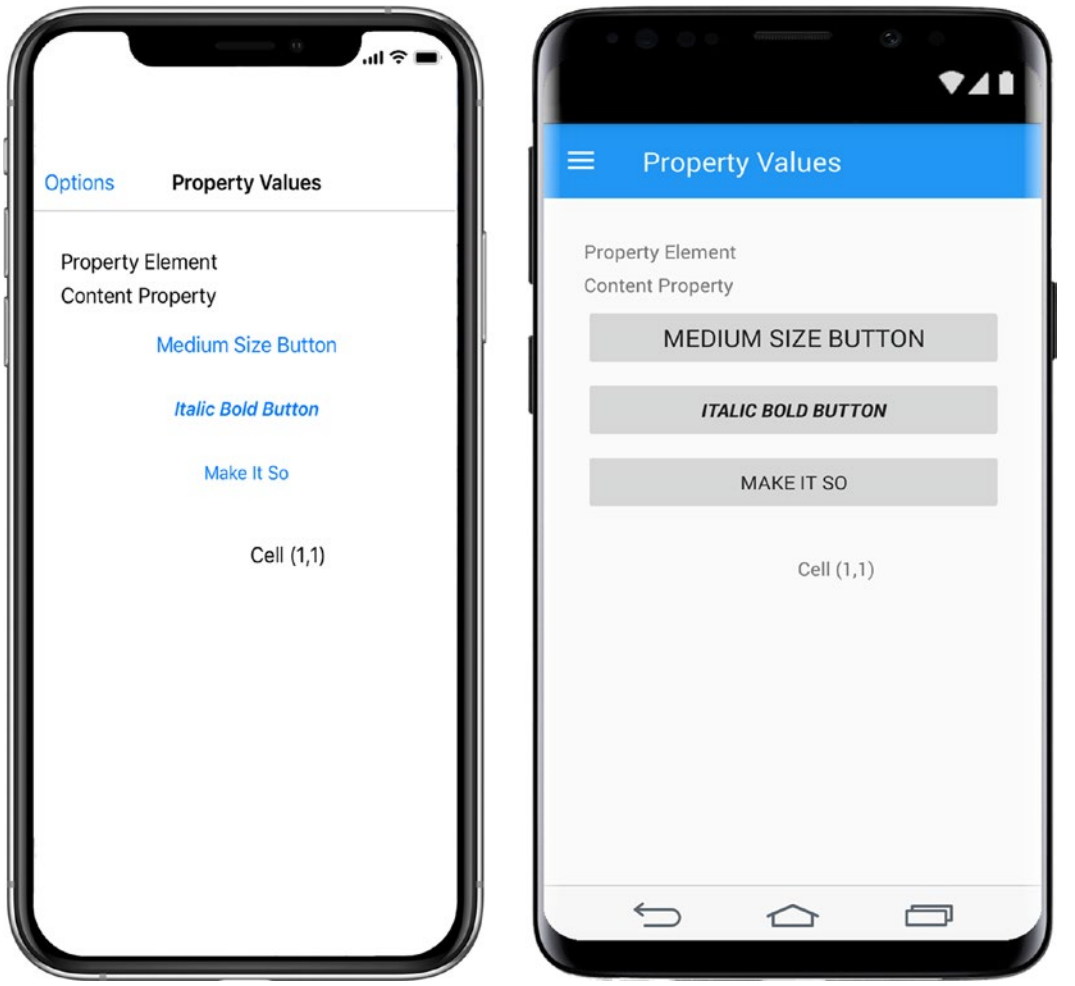

Figure 2-5. *Approaches for setting property values in Xamarin.Forms*

CODE COMPLETE: Setting Property Values

Listing 2-14 demonstrates the different approaches to assign values to properties.

Listing 2-14. Setting Property Values in XAML

```
<?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="XamlExamples.PropertyValuesPage">
    <ContentPage.Content>
        <StackLayout Padding="30,30">
            <Label>
                <Label.Text>Property Element</Label.Text>
            </Label>
            <Label>Content Property</Label>
            <Button FontSize="Medium" Text="Medium Size Button" />
            <Button FontAttributes="Italic,Bold" Text="Italic Bold
            Button" />
            <Button Text="Make It So" Clicked="ButtonClicked" />
            <Grid>
                <Label Grid.Row="1" Grid.Column="1" Text="Cell (1,1)" />
            </Grid>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

This completes the overview of the XAML syntax. Let's move on to the anatomy of XAML documents itself.

Anatomy of XAML Files

A XAML document is comprised of three files: the platform-independent XAML, the associated code behind file, and the generated file, which is used internally, as shown in Figure 2-6.

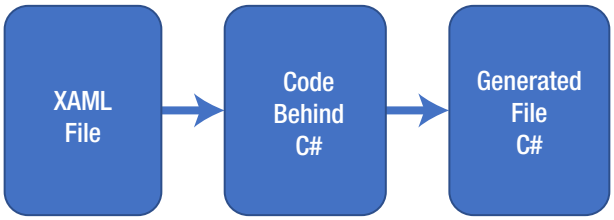

Figure 2-6. *XAML, code behind, and generated files*

The main file you interact with is the *XAML file* (`.xaml`). It contains the user interface definition.

The associated C# file (`.xaml.cs`) that has the corresponding business logic is called a c*ode behind*, functioning very much like its equivalent in web and desktop application development. The code behind contains a partial class definition with the same name specified in the `x:Class` attribute of the root element in the XAML. When starting an application, the platform-specific iOS or Droid project executes the `LoadApplication` method passing in an instance of the `App` class, which instantiates the XAML page using its default constructor. The constructor calls the `InitializeComponent` method to load the XAML into the application.

The XAML parser generates for each platform a *generated file* (`.xaml.g.cs`), which contains the constructors, classes, and properties to implement the XAML. It contains another partial class, now with the implementation of the `InitializeComponent` method. This method calls the `LoadFromXaml` method at runtime to load the actual user interface as an object graph when you run the application. The XAML parser uses, unless specified differently, the default constructor of the elements in the XAML to instantiate the objects and then set the values of the object properties if provided in the XAML.

The names of the event handlers specified in XAML must be instance methods that exist in the code behind. They cannot be `static`. Event handlers need to be used wisely, ideally only to enhance the controls they are serving and not to access services and business layer. Instead consider using other techniques such as behaviors, commands, and triggers (see Chapter 5) or data binding (see Chapter 9) for more reusable code.

The XAML parser generates for each named element in the XAML using the `x:Name` directive a local variable with the same name inside the generated file that can be accessed only from within the code behind. The local variable in the generated file is instantiated using the `FindByName` method. Local variables can be accessed only after the method `InitializeComponent` was called in the code behind.

That's XAML syntax. Next we'll look at the benefits and uses of XAML compilation.

XAML Compilation

XAML can be compiled in Visual Studio using the Xamarin compiler (XAMLC), which provides a performance improvement, compile-time error checking, and a smaller executable since the XAML files aren't needed at runtime. For backward compatibility, this feature is turned off by default. When XAML is set not to compile, then it is

interpreted upon execution and the execution takes longer, and runtime errors that could have been picked up at compile time will increase. Turn on compilation at both the assembly and the class level by adding the `XamlCompilation` attribute. Here is the usage at the assembly level:

```
[assembly: XamlCompilation (XamlCompilationOptions.Compile)]
namespace PhotoApp
{
  ...
}
```

The class level implementation is almost identical.

```
[XamlCompilation (XamlCompilationOptions.Compile)]
public class MyPage : ContentPage
{
  ...
}
```

Before moving on to Xamarin.Forms, here is an overview of how Xamarin.Forms XAML relates to other XAML dialects.

XAML Standard

Microsoft has initiated a process of aligning XAML dialects across multiple products such as Xamarin.Forms and non-Xamarin.Forms XAML like WPF. This could possibly result in name changes of Xamarin.Forms core classes, controls, layouts, and property enumerations. So far they've provided a mapping from XAML Standard elements to Xamarin.Forms equivalent in the form of aliases.

Developers can preview this by adding the `Xamarin.Forms.Alias` NuGet package to the Forms and platform projects and adding the namespace `Xamarin.Forms.Alias` to the XAML page, e.g.:

```
xmlns:a="clr-namespace:Xamarin.Forms.Alias;assembly=Xamarin.Forms.Alias"
```

Instead of `<Label Text="Xamarin.Forms"/>`, use the alias `<a:TextBlock Text="WPF"/>`.

Tables 2-1 and 2-2 list the aliases for Xamarin.Forms controls, properties, and enumerations available as a preview.

Table 2-1. *Xamarin.Forms Controls and Equivalent XAML Standard*

Xamarin.Forms Control	XAML Standard Alias
Frame	Border
Picker	ComboBox
ActivityIndicator	ProgressRing
StackLayout	StackPanel
Label	TextBlock
Entry	TextBox
Switch	ToggleSwitch
ContentView	UserControl

Table 2-2. *Xamarin.Forms Properties, Enumeration, and Equivalent XAML Standard*

Xamarin.Forms Control	Xamarin.Forms Property or Enum	XAML Standard
Button, Entry, Label, DatePicker, Editor, SearchBar, TimePicker	TextColor	Foreground
VisualElement	BackgroundColor	Background*
Picker, Button	BorderColor, OutlineColor	BorderBrush
Button	BorderWidth	BorderThickness
ProgressBar	Progress	Value
Button, Entry, Label, Editor, SearchBar, Span, Font	FontAttributes Bold, Italic, None	FontStyle Italic, Normal
		FontWeights* Bold, Normal
InputView	Keyboard Default, Url, Number, Telephone, Text, Chat, Email	InputScopeNameValue Default, Url, Number, TelephoneNumber, Text, Chat, EmailNameOrAddress
StackPanel	StackOrientation	Orientation*

Items marked with * are currently incomplete.

The future of XAML Standard is unclear. Such standardization is, as ever, desirable but problematic. Someday we may see an exodus to the XAML Standard syntax, but in the meantime use the XAML format provided in Xamarin.Forms and be aware of the `Xamarin.Forms.Alias` NuGet package.

That's XAML syntax in relation to the larger XAML universe and XAML Standard.

Summary

Xamarin.Forms XAML is based on XML and 2009 XAML syntax and is used to define cross-platform user interfaces. Pages, layouts, and controls provided by the Xamarin.Forms class library and the intrinsic 2009 XAML terms are made available to the XAML document through the `xmlns` namespace directive.

In this chapter, we have discussed how to declare elements, assign values to properties, use markup extensions to reference static members, create arrays, reference other elements inside the XAML, and use non-default constructors and factory methods to instantiate classes. We covered approaches Xamarin.Forms provides to assign values that can be simple data types, enumeration values, collections, event handlers, and even values to properties not defined in the element itself.

Using XAML offers an alternative to the C# approach of writing platform-specific iOS and Android user interfaces. This layer of abstraction allows creating truly cross-platform applications. The XAML files are stored inside the platform-independent .NET Standard project. You can increase the reusability and maintainability of the mobile application by following design patterns, such as MVVM, instead of allowing the code behind file to define your pattern for you.

Let's now move on to the Xamarin.Forms XAML vocabulary to build rich user interfaces.