Chapter 11

The bindable infrastructure

One of the most basic language constructs of C# is the class member known as the *property*. All of us very early on in our first encounters with C# learned the general routine of defining a property. The property is often backed by a private field and includes `set` and `get` accessors that reference the private field and do something with a new value:

```
public class MyClass
{
    …
    double quality;

    public double Quality
    {
        set
        {
            quality = value;
            // Do something with the new value
        }
        get
        {
            return quality;
        }
    }
    …
}
```

Properties are sometimes referred to as *smart fields*. Syntactically, code that accesses a property resembles code that accesses a field. Yet the property can execute some of its own code when the property is accessed.

Properties are also like methods. Indeed, C# code is compiled into intermediate language that implements a property such as `Quality` with a pair of methods named `set_Quality` and `get_Quality`. Yet despite the close functional resemblance between properties and a pair of *set* and *get* methods, the property syntax reveals itself to be much more suitable when moving from code to markup. It's hard to imagine XAML built on an underlying API that is missing properties.

So you may be surprised to learn that Xamarin.Forms implements an enhanced property definition that builds upon C# properties. Or maybe you won't be surprised. If you already have experience with Microsoft's XAML-based platforms, you'll encounter some familiar concepts in this chapter.

The property definition shown above is known as a *CLR property* because it's supported by the .NET common language runtime. The enhanced property definition in Xamarin.Forms builds upon the CLR property and is called a *bindable property,* encapsulated by the `BindableProperty` class and supported by the `BindableObject` class.

The Xamarin.Forms class hierarchy

Before exploring the details of the important `BindableObject` class, let's first discover how `BindableObject` fits into the overall Xamarin.Forms architecture by constructing a class hierarchy.

In an object-oriented programming framework such as Xamarin.Forms, a class hierarchy can often reveal important inner structures of the environment. The class hierarchy shows how various classes relate to one another and the properties, methods, and events that they share, including how bindable properties are supported.

You can construct such a class hierarchy by laboriously going through the online documentation and taking note of what classes derive from what other classes. Or you can write a Xamarin.Forms program to do the work for you and display the class hierarchy on the phone. Such a program makes use of .NET reflection to obtain all the public classes, structures, and enumerations in the **Xamarin.Forms.Core** and **Xamarin.Forms.Xaml** assemblies and arrange them in a tree. The **ClassHierarchy** application demonstrates this technique.

As usual, the **ClassHierarchy** project contains a class that derives from `ContentPage`, named `ClassHierarchyPage`, but it also contains two additional classes, named `TypeInformation` and `ClassAndSubclasses`.

The program creates one `TypeInformation` instance for every public class (and structure and enumeration) in the **Xamarin.Forms.Core** and **Xamarin.Forms.Xaml** assemblies, plus any .NET class that serves as a base class for any Xamarin.Forms class, with the exception of `Object`. (These .NET classes are `Attribute`, `Delegate`, `Enum`, `EventArgs`, `Exception`, `MulticastDelegate`, and `ValueType`.) The `TypeInformation` constructor requires a `Type` object identifying a type but also obtains some other information:

```
class TypeInformation
{
    bool isBaseGenericType;
    Type baseGenericTypeDef;

    public TypeInformation(Type type, bool isXamarinForms)
    {
        Type = type;
        IsXamarinForms = isXamarinForms;
        TypeInfo typeInfo = type.GetTypeInfo();
        BaseType = typeInfo.BaseType;

        if (BaseType != null)
        {
            TypeInfo baseTypeInfo = BaseType.GetTypeInfo();
            isBaseGenericType = baseTypeInfo.IsGenericType;

            if (isBaseGenericType)
            {
                baseGenericTypeDef = baseTypeInfo.GetGenericTypeDefinition();
```

```
            }
        }
    }

    public Type Type { private set; get; }
    public Type BaseType { private set; get; }
    public bool IsXamarinForms { private set; get; }

    public bool IsDerivedDirectlyFrom(Type parentType)
    {
        if (BaseType != null && isBaseGenericType)
        {
            if (baseGenericTypeDef == parentType)
            {
                return true;
            }
        }
        else if (BaseType == parentType)
        {
            return true;
        }
        return false;
    }
}
```

A very important part of this class is the `IsDerivedDirectlyFrom` method, which will return `true` if passed an argument that is this type's base type. This determination is complicated if generic classes are involved, and that issue largely accounts for the complexity of the class.

The `ClassAndSubclasses` class is considerably shorter:

```
class ClassAndSubclasses
{
    public ClassAndSubclasses(Type parent, bool isXamarinForms)
    {
        Type = parent;
        IsXamarinForms = isXamarinForms;
        Subclasses = new List<ClassAndSubclasses>();
    }

    public Type Type { private set; get; }
    public bool IsXamarinForms { private set; get; }
    public List<ClassAndSubclasses> Subclasses { private set; get; }
}
```

The program creates one instance of this class for every `Type` displayed in the class hierarchy, including `Object`, so the program creates one more `ClassAndSubclasses` instance than the number of `TypeInformation` instances. The `ClassAndSubclasses` instance associated with `Object` contains a collection of all the classes that derive directly from `Object`, and each of those `ClassAndSubclasses` instances contains a collection of all the classes that derive from that one, and so forth for the remainder of the hierarchy tree.

The `ClassHierarchyPage` class consists of a XAML file and a code-behind file, but the XAML file contains little more than a scrollable `StackLayout` ready for some `Label` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ClassHierarchy.ClassHierarchyPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="5, 20, 0, 0"
                    Android="5, 0, 0, 0"
                    WinPhone="5, 0, 0, 0" />
    </ContentPage.Padding>

    <ScrollView>
        <StackLayout x:Name="stackLayout"
                     Spacing="0" />
    </ScrollView>
</ContentPage>
```

The code-behind file obtains references to the two Xamarin.Forms `Assembly` objects and then accumulates all the public classes, structures, and enumerations in the `classList` collection. It then checks for the necessity of including any base classes from the .NET assemblies, sorts the result, and then calls two recursive methods, `AddChildrenToParent` and `AddItemToStackLayout`:

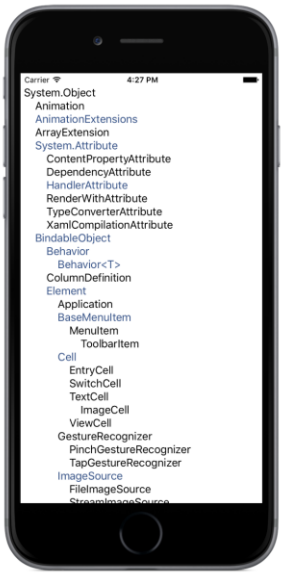
```
public partial class ClassHierarchyPage : ContentPage
{
    public ClassHierarchyPage()
    {
        InitializeComponent();

        List<TypeInformation> classList = new List<TypeInformation>();

        // Get types in Xamarin.Forms.Core assembly.
        GetPublicTypes(typeof(View).GetTypeInfo().Assembly, classList);

        // Get types in Xamarin.Forms.Xaml assembly.
        GetPublicTypes(typeof(Extensions).GetTypeInfo().Assembly, classList);

        // Ensure that all classes have a base type in the list.
        //  (i.e., add Attribute, ValueType, Enum, EventArgs, etc.)
        int index = 0;

        // Watch out! Loops through expanding classList!
        do
        {
            // Get a child type from the list.
            TypeInformation childType = classList[index];

            if (childType.Type != typeof(Object))
            {
                bool hasBaseType = false;
```

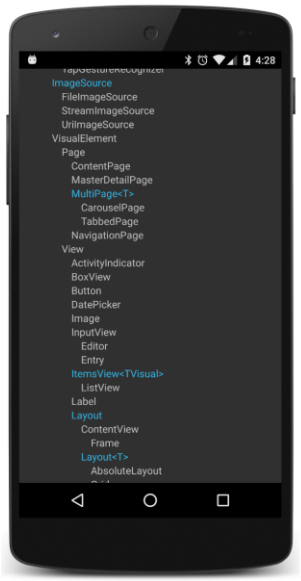
```
                // Loop through the list looking for a base type.
                foreach (TypeInformation parentType in classList)
                {
                    if (childType.IsDerivedDirectlyFrom(parentType.Type))
                    {
                        hasBaseType = true;
                    }
                }

                // If there's no base type, add it.
                if (!hasBaseType && childType.BaseType != typeof(Object))
                {
                    classList.Add(new TypeInformation(childType.BaseType, false));
                }
            }
            index++;
        }
        while (index < classList.Count);

        // Now sort the list.
        classList.Sort((t1, t2) =>
        {
            return String.Compare(t1.Type.Name, t2.Type.Name);
        });

        // Start the display with System.Object.
        ClassAndSubclasses rootClass = new ClassAndSubclasses(typeof(Object), false);

        // Recursive method to build the hierarchy tree.
        AddChildrenToParent(rootClass, classList);

        // Recursive method for adding items to StackLayout.
        AddItemToStackLayout(rootClass, 0);
    }

    void GetPublicTypes(Assembly assembly,
                        List<TypeInformation> classList)
    {
        // Loop through all the types.
        foreach (Type type in assembly.ExportedTypes)
        {
            TypeInfo typeInfo = type.GetTypeInfo();

            // Public types only but exclude interfaces.
            if (typeInfo.IsPublic && !typeInfo.IsInterface)
            {
                // Add type to list.
                classList.Add(new TypeInformation(type, true));
            }
        }
    }

    void AddChildrenToParent(ClassAndSubclasses parentClass,
                             List<TypeInformation> classList)
```

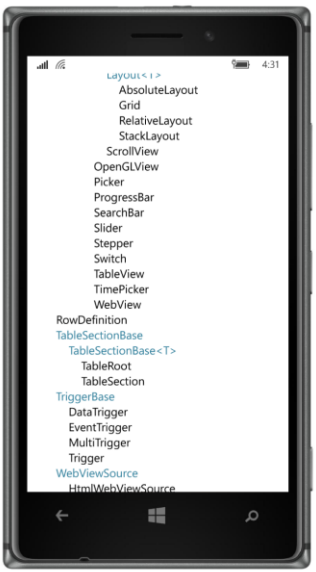
```
{
    foreach (TypeInformation typeInformation in classList)
    {
        if (typeInformation.IsDerivedDirectlyFrom(parentClass.Type))
        {
            ClassAndSubclasses subClass =
                new ClassAndSubclasses(typeInformation.Type,
                                       typeInformation.IsXamarinForms);
            parentClass.Subclasses.Add(subClass);
            AddChildrenToParent(subClass, classList);
        }
    }
}

void AddItemToStackLayout(ClassAndSubclasses parentClass, int level)
{
    // If assembly is not Xamarin.Forms, display full name.
    string name = parentClass.IsXamarinForms ? parentClass.Type.Name :
                                               parentClass.Type.FullName;

    TypeInfo typeInfo = parentClass.Type.GetTypeInfo();

    // If generic, display angle brackets and parameters.
    if (typeInfo.IsGenericType)
    {
        Type[] parameters = typeInfo.GenericTypeParameters;
        name = name.Substring(0, name.Length - 2);
        name += "<";

        for (int i = 0; i < parameters.Length; i++)
        {
            name += parameters[i].Name;
            if (i < parameters.Length - 1)
            {
                name += ", ";
            }
        }
        name += ">";
    }

    // Create Label and add to StackLayout.
    Label label = new Label
    {
        Text = String.Format("{0}{1}", new string(' ', 4 * level), name),
        TextColor = parentClass.Type.GetTypeInfo().IsAbstract ?
                        Color.Accent : Color.Default
    };

    stackLayout.Children.Add(label);

    // Now display nested types.
    foreach (ClassAndSubclasses subclass in parentClass.Subclasses)
    {
        AddItemToStackLayout(subclass, level + 1);
```

```
        }
    }
}
```

The recursive `AddChildrenToParent` method assembles the linked list of `ClassAndSubclasses` instances from the flat `classList` collection. The `AddItemToStackLayout` method is also recursive because it is responsible for adding the `ClassesAndSubclasses` linked list to the `StackLayout` object by creating a `Label` view for each class, with a little blank space at the beginning for the proper indentation. The method displays the Xamarin.Forms types with just the class names, but the .NET types include the fully qualified name to distinguish them. The method uses the platform accent color for classes that are not instantiable because they are abstract or static:







Overall, you'll see that the Xamarin.Forms visual elements have the following general hierarchy:

```
System.Object
    BindableObject
        Element
            VisualElement
                View
                    ...
                    Layout
                        ...
                        Layout<T>
                            ...
                Page
                    ...
```

Aside from `Object`, all the classes in this abbreviated class hierarchy are implemented in the Xamarin.Forms.Core.dll assembly and associated with a namespace of `Xamarin.Forms`.

Let's examine some of these major classes in detail.

As the name of the `BindableObject` class implies, the primary function of this class is to support data binding—the linking of two properties of two objects so that they maintain the same value. But `BindableObject` also supports styles and the `DynamicResource` markup extension as well. It does this in two ways: through `BindableObject` property definitions in the form of `BindableProperty` objects and also by implementing the .NET `INotifyPropertyChanged` interface. All of this will be discussed in much more detail in this chapter and future chapters.

Let's continue down the hierarchy: as you've seen, user-interface objects in Xamarin.Forms are often arranged on the page in a parent-child hierarchy, and the `Element` class includes support for parent and child relationships.

`VisualElement` is an exceptionally important class in Xamarin.Forms. A visual element is anything in Xamarin.Forms that occupies an area on the screen. The `VisualElement` class defines 28 public properties related to size, location, background color, and other visual and functional characteristics, such as `IsEnabled` and `IsVisible`.

In Xamarin.Forms the word *view* is often used to refer to individual visual objects such as buttons, sliders, and text-entry boxes, but you can see that the `View` class is the parent to the layout classes as well. Interestingly, `View` adds only three public members to what it inherits from `VisualElement`. These are `HorizontalOptions` and `VerticalOptions`—which make sense because these properties don't apply to pages—and `GestureRecognizers` to support touch input.

The descendants of `Layout` are capable of having children views. A child view appears on the screen visually within the boundaries of its parent. Classes that derive from `Layout` can have only one child of type `View`, but the generic `Layout<T>` class defines a `Children` property, which is a collection of multiple child views, including other layouts. You've already seen the `StackLayout`, which arranges its children in a horizontal or vertical stack. Although the `Layout` class derives from `View`, layouts are so important in Xamarin.Forms that they are often considered a category in themselves.

ClassHierarchy lists all the public classes, structures, and enumerations defined in the **Xamarin.Forms.Core** and **Xamarin.Forms.Xaml** assemblies, but it does not list interfaces. Those are important as well, but you'll just have to explore them on your own. (Or enhance the program to list them.)

Nor does **ClassHierarchy** list the many public classes that help implement Xamarin.Forms on the various platforms. In the final chapter of this book, you'll see a version that does.

A peek into BindableObject and BindableProperty

The existence of classes named `BindableObject` and `BindableProperty` is likely to be a little confusing at first. Keep in mind that `BindableObject` is much like `Object` in that it serves as a base class to a large chunk of the Xamarin.Forms API, and particularly to `Element` and hence `VisualElement`.

`BindableObject` provides support for objects of type `BindableProperty`. A `BindableProperty` object extends a CLR property. The best insights into bindable properties come when you create a few of your own—as you'll be doing before the end of this chapter—but you can also glean some understanding by exploring the existing bindable properties.

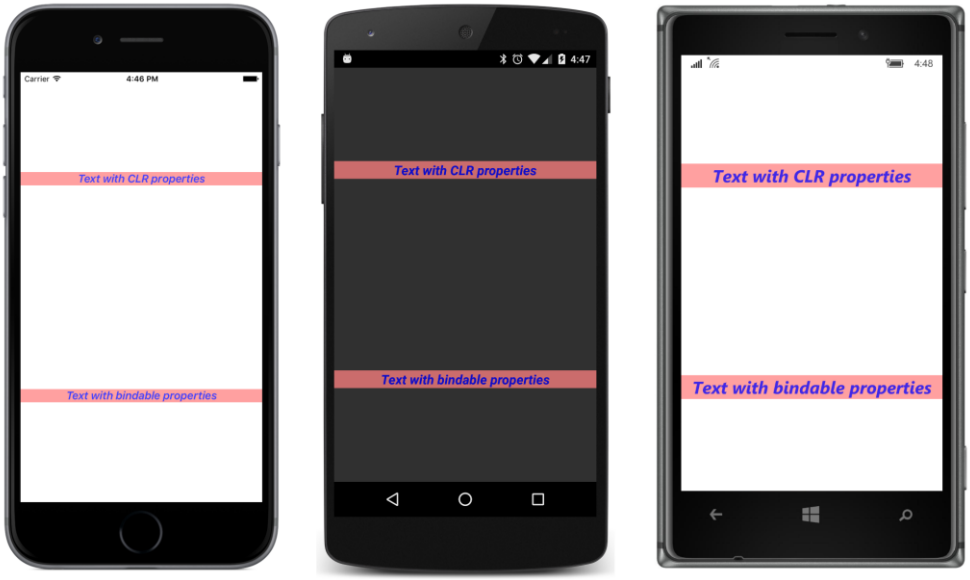
Toward the beginning of Chapter 7, "XAML vs. code," two buttons were created with many of the same property settings, except that the properties of one button were set in code using the C# 3.0 object initialization syntax and the other button was instantiated and initialized in XAML.

Here's a similar (but code-only) program named **PropertySettings** that also creates and initializes two buttons in two different ways. The properties of the first `Label` are set the old-fashioned way, while the properties of the second `Label` are set with a more verbose technique:

```
public class PropertySettingsPage : ContentPage
{
    public PropertySettingsPage()
    {
        Label label1 = new Label();
        label1.Text = "Text with CLR properties";
        label1.IsVisible = true;
        label1.Opacity = 0.75;
        label1.HorizontalTextAlignment = TextAlignment.Center;
        label1.VerticalOptions = LayoutOptions.CenterAndExpand;
        label1.TextColor = Color.Blue;
        label1.BackgroundColor = Color.FromRgb(255, 128, 128);
        label1.FontSize = Device.GetNamedSize(NamedSize.Medium, new Label());
        label1.FontAttributes = FontAttributes.Bold | FontAttributes.Italic;

        Label label2 = new Label();
        label2.SetValue(Label.TextProperty, "Text with bindable properties");
        label2.SetValue(Label.IsVisibleProperty, true);
        label2.SetValue(Label.OpacityProperty, 0.75);
        label2.SetValue(Label.HorizontalTextAlignmentProperty, TextAlignment.Center);
        label2.SetValue(Label.VerticalOptionsProperty, LayoutOptions.CenterAndExpand);
        label2.SetValue(Label.TextColorProperty, Color.Blue);
        label2.SetValue(Label.BackgroundColorProperty, Color.FromRgb(255, 128, 128));
        label2.SetValue(Label.FontSizeProperty,
                        Device.GetNamedSize(NamedSize.Medium, new Label()));
        label2.SetValue(Label.FontAttributesProperty,
                        FontAttributes.Bold | FontAttributes.Italic);

        Content = new StackLayout
        {
```

```
            Children =
            {
                label1,
                label2
            }
        };
    }
}
```

These two ways to set properties are entirely consistent:



Yet the alternative syntax seems very odd. For example:

```
label2.SetValue(Label.TextProperty, "Text with bindable properties");
```

What is that `SetValue` method? `SetValue` is defined by `BindableObject`, from which every visual object derives. `BindableObject` also defines a `GetValue` method.

That first argument to `SetValue` has the name `Label.TextProperty`, which indicates that `TextProperty` is static, but despite its name, it's not a property at all. It's a static *field* of the `Label` class. `TextProperty` is also read-only, and it's defined in the `Label` class something like this:

```
public static readonly BindableProperty TextProperty;
```

That's an object of type `BindableProperty`. Of course, it may seem a little disturbing that a field is named `TextProperty`, but there it is. Because it's static, however, it exists independently of any `Label` objects that might or might not exist.

If you look in the documentation of the `Label` class, you'll see that it defines 10 properties, including `Text`, `TextColor`, `FontSize`, `FontAttributes`, and others. You'll also see 10 corresponding

public static read-only fields of type `BindableProperty` with the names `TextProperty`, `TextColorProperty`, `FontSizeProperty`, `FontAttributesProperty`, and so forth.

These properties and fields are closely related. Indeed, internal to the `Label` class, the `Text` CLR property is defined like this to reference the corresponding `TextProperty` object:

```
public string Text
{
    set { SetValue(Label.TextProperty, value); }
    get { return (string)GetValue(Label.TextProperty); }
}
```

So you see why it is that your application calling `SetValue` with a `Label.TextProperty` argument is exactly equivalent to setting the `Text` property directly, and perhaps just a tinier bit faster!

The internal definition of the `Text` property in `Label` isn't secret information. This is standard code. Although any class can define a `BindableProperty` object, only a class that derives from `BindableObject` can call the `SetValue` and `GetValue` methods that actually implement the property in the class. Casting is required for the `GetValue` method because it's defined as returning `object`.

All the real work involved with maintaining the `Text` property is going on in those `SetValue` and `GetValue` calls. The `BindableObject` and `BindableProperty` objects effectively extend the functionality of standard CLR properties to provide systematic ways to:

- Define properties
- Give properties default values
- Store their current values
- Provide mechanisms for validating property values
- Maintain consistency among related properties in a single class
- Respond to property changes
- Trigger notifications when a property is about to change and has changed
- Support data binding
- Support styles
- Support dynamic resources

The close relationship of a property named `Text` with a `BindableProperty` named `TextProperty` is reflected in the way that programmers speak about these properties: Sometimes a programmer says that the `Text` property is "backed by" a `BindableProperty` named `TextProperty` because `TextProperty` provides infrastructure support for `Text`. But a common shortcut is to say that `Text` is itself a "bindable property," and generally no one will be confused.

Not every Xamarin.Forms property is a bindable property. Neither the `Content` property of `ContentPage` nor the `Children` property of `Layout<T>` is a bindable property. Of the 28 properties defined by `VisualElement`, 26 are backed by bindable properties, but the `Bounds` property and the `Resources` properties are not.

The `Span` class used in connection with `FormattedString` does not derive from `BindableObject`. Therefore, `Span` does not inherit `SetValue` and `GetValue` methods, and it cannot implement `BindableProperty` objects.

This means that the `Text` property of `Label` is backed by a bindable property, but the `Text` property of `Span` is not. Does it make a difference?

Of course it makes a difference! If you recall the **DynamicVsStatic** program in the previous chapter, you discovered that `DynamicResource` worked on the `Text` property of `Label` but not the `Text` property of `Span`. Can it be that `DynamicResource` works only with bindable properties?

This supposition is pretty much confirmed by the definition of the following public method defined by `Element`:

```
public void SetDynamicResource(BindableProperty property, string key);
```

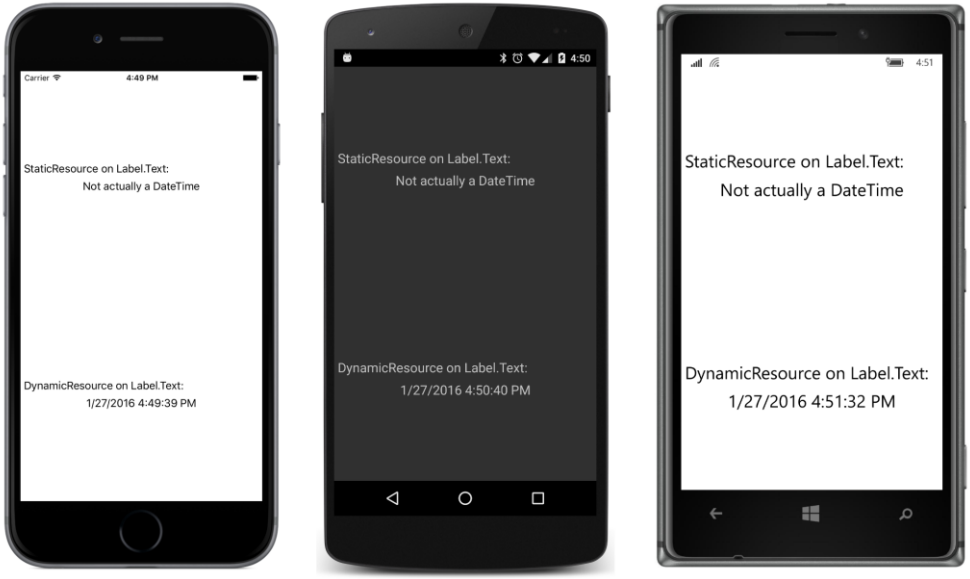
This is how a dictionary key is associated with a particular property of an element when that property is the target of a `DynamicResource` markup extension.

This `SetDynamicResource` method also allows you to set a dynamic resource link on a property in code. Here's the page class from a code-only version of **DynamicVsStatic** called **DynamicVsStaticCode**. It's somewhat simplified to exclude the use of a `FormattedString` and `Span` object, but otherwise it pretty accurately mimics how the previous XAML file is parsed and, in particular, how the `Text` properties of the `Label` elements are set by the XAML parser:

```
public class DynamicVsStaticCodePage : ContentPage
{
    public DynamicVsStaticCodePage()
    {
        Padding = new Thickness(5, 0);

        // Create resource dictionary and add item.
        Resources = new ResourceDictionary
        {
            { "currentDateTime", "Not actually a DateTime" }
        };

        Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "StaticResource on Label.Text:",
                    VerticalOptions = LayoutOptions.EndAndExpand,
```

```
                    FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label))
                },

                new Label
                {
                    Text = (string)Resources["currentDateTime"],
                    VerticalOptions = LayoutOptions.StartAndExpand,
                    HorizontalTextAlignment = TextAlignment.Center,
                    FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label))
                },

                new Label
                {
                    Text = "DynamicResource on Label.Text:",
                    VerticalOptions = LayoutOptions.EndAndExpand,
                    FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label))
                }
            }
        };

        // Create the final label with the dynamic resource.
        Label label = new Label
        {
            VerticalOptions = LayoutOptions.StartAndExpand,
            HorizontalTextAlignment = TextAlignment.Center,
            FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label))
        };

        label.SetDynamicResource(Label.TextProperty, "currentDateTime");

        ((StackLayout)Content).Children.Add(label);

        // Start the timer going.
        Device.StartTimer(TimeSpan.FromSeconds(1),
            () =>
            {
                Resources["currentDateTime"] = DateTime.Now.ToString();
                return true;
            });
    }
}
```

The `Text` property of the second `Label` is set directly from the dictionary entry and makes the use of the dictionary seem a little pointless in this context. But the `Text` property of the last `Label` is bound to the dictionary key through a call to `SetDynamicResource`, which allows the property to be updated when the dictionary contents change:



Consider this: What would the signature of this `SetDynamicResource` method be if it could not refer to a property using the `BindableProperty` object? It's easy to reference a property *value* in method calls, but not the property itself. There are a couple of ways, such as the `PropertyInfo` class in the `System.Reflection` namespace or the LINQ `Expression` object. But the `BindableProperty` object is designed specifically for this purpose, as well as the essential job of handling the underlying link between the property and the dictionary key.

Similarly, when we explore styles in the next chapter, you'll encounter a `Setter` class used in connection with styles. `Setter` defines a property named `Property` of type `BindableProperty`, which mandates that any property targeted by a style must be backed by a bindable property. This allows a style to be defined prior to the elements targeted by the style.

Likewise for data bindings. The `BindableObject` class defines a `SetBinding` method that is very similar to the `SetDynamicResource` method defined on `Element`:

```
public void SetBinding(BindableProperty targetProperty, BindingBase binding);
```

Again, notice the type of the first argument. Any property targeted by a data binding must be backed by a bindable property.

For these reasons, whenever you create a custom view and need to define public properties, your default inclination should be to define them as bindable properties. Only if after careful consideration you conclude that it is not necessary or appropriate for the property to be targeted by a style or a data binding should you retreat and define an ordinary CLR property instead.

So whenever you create a class that derives from `BindableObject`, one of the first pieces of code you should be typing in that class begins "public static readonly BindableProperty"—perhaps the most characteristic sequence of four words in all of Xamarin.Forms programming.

Defining bindable properties

Suppose you'd like an enhanced `Label` class that lets you specify font sizes in units of points. Let's call this class `AltLabel` for "alternative `Label`." It derives from `Label` and includes a new property named `PointSize`.

Should `PointSize` be backed by a bindable property? Of course! (Although the real advantages of doing so won't be demonstrated until upcoming chapters.)

The code-only `AltLabel` class is included in the **Xamarin.FormsBook.Toolkit** library, so it's accessible to multiple applications. The new `PointSize` property is implemented with a `BindableProperty` object named `PointSizeProperty` and a CLR property named `PointSize` that references `PointSizeProperty`:

```
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty … ;
    …
    public double PointSize
    {
        set { SetValue(PointSizeProperty, value); }
        get { return (double)GetValue(PointSizeProperty); }
    }
    …
}
```

Both the field and the property definition must be public.

Because `PointSizeProperty` is defined as `static` and `readonly`, it must be assigned either in a static constructor or right in the field definition, after which it cannot be changed. Generally, a `BindableProperty` object is assigned in the field definition by using the static `BindableProperty.Create` method. Four arguments are required (shown here with the argument names):

- `propertyName` The text name of the property (in this case "PointSize")
- `returnType` The type of the property (a `double` in this example)
- `declaringType` The type of the class defining the property (`AltLabel`)
- `defaultValue` A default value (let's say 8 points)

The second and third arguments are generally defined with `typeof` expressions. Here's the assignment statement with these four arguments passed to `BindableProperty.Create`:

```
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize",        // propertyName
                                typeof(double),     // returnType
                                typeof(AltLabel),   // declaringType
                                8.0,                // defaultValue
                                …);
    …
}
```

Notice that the default value is specified as 8.0 rather than just 8. Because `BindableProperty.Create` is designed to handle properties of any type, the `defaultValue` parameter is defined as `object`. When the C# compiler encounters just an 8 as that argument, it will assume that the 8 is an `int` and pass an `int` to the method. The problem won't be revealed until run time, however, when the `BindableProperty.Create` method will be expecting the default value to be of type `double` and respond by raising a `TypeInitializationException`.

You must be explicit about the type of the value you're specifying as the default. Not doing so is a very common error in defining bindable properties. A *very* common error.

`BindableProperty.Create` also has six optional arguments. Here they are with the argument names and their purpose:

- `defaultBindingMode` Used in connection with data binding
- `validateValue` A callback to check for a valid value
- `propertyChanged` A callback to indicate when the property has changed
- `propertyChanging` A callback to indicate when the property is about to change
- `coerceValue` A callback to coerce a set value to another value (for example, to restrict the values to a range)
- `defaultValueCreator` A callback to create a default value. This is generally used to instantiate a default object that can't be shared among all instances of the class; for example, a collection object such as `List` or `Dictionary`.

Do not perform any validation, coercion, or property-changed handling in the CLR property. The CLR property should be restricted to `SetValue` and `GetValue` calls. Everything else should be done in the callbacks provided by the bindable property infrastructure.

It is very rare that a particular call to `BindableProperty.Create` would need all of these optional arguments. For that reason, these optional arguments are commonly indicated with the named argument feature introduced in C# 4.0. To specify a particular optional argument, use the argument name followed by a colon. For example:

```
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize",        // propertyName
                                typeof(double),     // returnType
                                typeof(AltLabel),   // declaringType
                                8.0,                // defaultValue
                                propertyChanged: OnPointSizeChanged);
    …
}
```

Without a doubt, `propertyChanged` is the most important of the optional arguments because the class uses this callback to be notified when the property changes, either directly from a call to `SetValue` or through the CLR property.

In this example, the property-changed handler is called `OnPointSizeChanged`. It will be called only when the property truly changes and not when it's simply set to the same value. However, because `OnPointSizeChanged` is referenced from a static field, the method itself must also be static. Here's what it looks like:

```
public class AltLabel : Label
{
    …
    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        …
    }
    …
}
```

This seems a little odd. We might have multiple `AltLabel` instances in a program, yet whenever the `PointSize` property changes in any one of these instances, this same static method is called. How does the method know exactly which `AltLabel` instance has changed?

The method can tell which instance's property has changed because that instance is always the first argument to the property-changed handler. Although that first argument is defined as a `BindableObject`, in this case it's actually of type `AltLabel` and indicates which `AltLabel` instance's property has changed. This means that you can safely cast the first argument to an `AltLabel` instance:

```
static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
{
    AltLabel altLabel = (AltLabel)bindable;
    …
}
```

You can then reference anything in the particular instance of `AltLabel` whose property has changed. The second and third arguments are actually of type `double` for this example and indicate the previous value and the new value.

Often it's convenient for this static method to call an instance method with the arguments converted to their actual types:

```
public class AltLabel : Label
{
    …
    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ((AltLabel)bindable).OnPointSizeChanged((double)oldValue, (double)newValue);
    }

    void OnPointSizeChanged(double oldValue, double newValue)
    {
        …
    }
}
```

The instance method can then make use of any instance properties or methods of the underlying base class as it would normally.

For this class, this `OnPointSizeChanged` method needs to set the `FontSize` property based on the new point size and a conversion factor. In addition, the constructor needs to initialize the `FontSize` property based on the default `PointSize` value. This is done through a simple `SetLabelFontSize` method. Here's the final complete class:

```
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize",        // propertyName
                                typeof(double),     // returnType
                                typeof(AltLabel),   // declaringType
                                8.0,                // defaultValue
                                propertyChanged: OnPointSizeChanged);

    public AltLabel()
    {
        SetLabelFontSize((double)PointSizeProperty.DefaultValue);
    }

    public double PointSize
    {
        set { SetValue(PointSizeProperty, value); }
        get { return (double)GetValue(PointSizeProperty); }
    }

    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ((AltLabel)bindable).OnPointSizeChanged((double)oldValue, (double)newValue);
    }

    void OnPointSizeChanged(double oldValue, double newValue)
    {
        SetLabelFontSize(newValue);
    }

    void SetLabelFontSize(double pointSize)
```

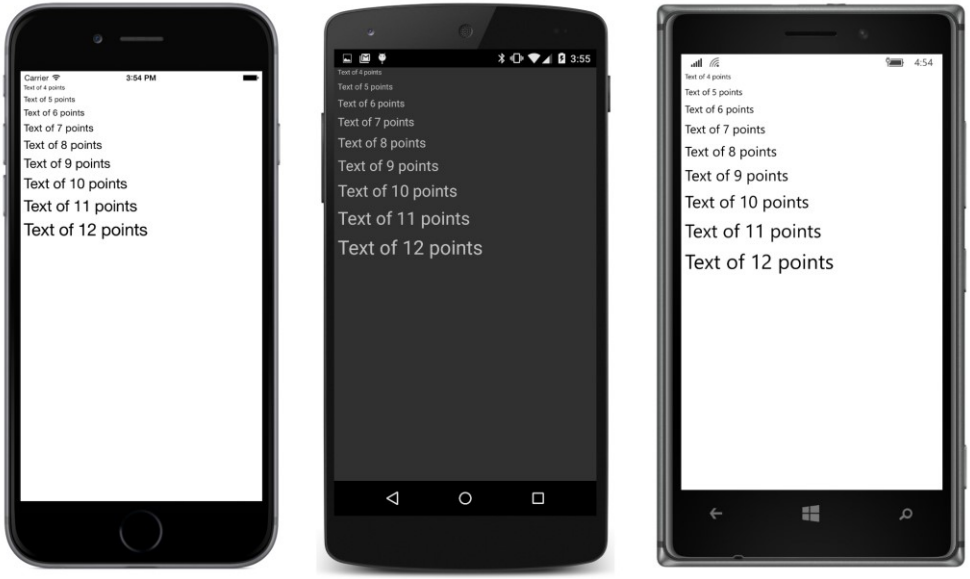
```
    {
        FontSize = 160 * pointSize / 72;
    }
}
```

It is also possible for the instance `OnPointSizeChanged` property to access the `PointSize` property directly rather than use `newValue`. By the time the property-changed handler is called, the underlying property value has already been changed. However, you don't have direct access to that underlying value, as you do when a private field backs a CLR property. That underlying value is private to `BindableObject` and accessible only through the `GetValue` call.

Of course, nothing prevents code that's using `AltLabel` from setting the `FontSize` property and overriding the `PointSize` setting, but let's hope such code is aware of that. Here's some code that is—a program called **PointSizedText,** which uses `AltLabel` to display point sizes from 4 through 12:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="PointSizedText.PointSizedTextPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="5, 20, 0, 0"
                    Android="5, 0, 0, 0"
                    WinPhone="5, 0, 0, 0" />
    </ContentPage.Padding>

    <StackLayout x:Name="stackLayout">
        <toolkit:AltLabel Text="Text of 4 points" PointSize="4" />
        <toolkit:AltLabel Text="Text of 5 points" PointSize="5" />
        <toolkit:AltLabel Text="Text of 6 points" PointSize="6" />
        <toolkit:AltLabel Text="Text of 7 points" PointSize="7" />
        <toolkit:AltLabel Text="Text of 8 points" PointSize="8" />
        <toolkit:AltLabel Text="Text of 9 points" PointSize="9" />
        <toolkit:AltLabel Text="Text of 10 points" PointSize="10" />
        <toolkit:AltLabel Text="Text of 11 points" PointSize="11" />
        <toolkit:AltLabel Text="Text of 12 points" PointSize="12" />
    </StackLayout>
</ContentPage>
```

And here are the screenshots:



The read-only bindable property

Suppose you're working with an application in which it's convenient to know the number of words in the text that is displayed by a `Label` element. Perhaps you'd like to build that facility right into a class that derives from `Label`. Let's call this new class `CountedLabel`.

By now, your first thought should be to define a `BindableProperty` object named `WordCountProperty` and a corresponding CLR property named `WordCount`.

But wait: It only makes sense for this `WordCount` property to be set from within the `CountedLabel` class. That means the `WordCount` CLR property should not have a public `set` accessor. It should be defined this way:

```
public int WordCount
{
    private set { SetValue(WordCountProperty, value); }
    get { return (double)GetValue(WordCountProperty); }
}
```

The `get` accessor is still public, but the `set` accessor is private. Is that sufficient?

Not exactly. Despite the private `set` accessor in the CLR property, code external to `CountedLabel` can still call `SetValue` with the `CountedLabel.WordCountProperty` bindable property object. That type of property setting should be prohibited as well. But how can that work if the `WordCountProperty` object is public?

The solution is to make a *read-only* bindable property by using the `BindableProperty.CreateReadOnly` method. The

Xamarin.Forms API itself defines several read-only bindable properties—for example, the `Width` and `Height` properties defined by `VisualElement`.

Here's how you can make one of your own:

The first step is to call `BindableProperty.CreateReadOnly` with the same arguments as for `BindableProperty.Create`. However, the `CreateReadOnly` method returns an object of `BindablePropertyKey` rather than `BindableProperty`. Define this object as `static` and `readonly`, as with the `BindableProperty`, but make it be private to the class:

```
public class CountedLabel : Label
{
    static readonly BindablePropertyKey WordCountKey =
        BindableProperty.CreateReadOnly("WordCount",            // propertyName
                                        typeof(int),            // returnType
                                        typeof(CountedLabel),   // declaringType
                                        0);                     // defaultValue
    …
}
```

Don't think of this `BindablePropertyKey` object as an encryption key or anything like that. It's much simpler—really just an object that is private to the class.

The second step is to make a public `BindableProperty` object by using the `BindableProperty` property of the `BindablePropertyKey`:

```
public class CountedLabel : Label
{
    …
    public static readonly BindableProperty WordCountProperty = WordCountKey.BindableProperty;
    …
}
```

This `BindableProperty` object is public, but it's a special kind of `BindableProperty`: It cannot be used in a `SetValue` call. Attempting to do so will raise an `InvalidOperationException`.

However, there is an overload of the `SetValue` method that accepts a `BindablePropertyKey` object. The CLR `set` accessor can call `SetValue` using this object, but this `set` accessor must be private to prevent the property from being set outside the class:

```
public class CountedLabel : Label
{
    …
    public int WordCount
    {
        private set { SetValue(WordCountKey, value); }
        get { return (int)GetValue(WordCountProperty); }
    }
    …
}
```

The `WordCount` property can now be set from within the `CountedLabel` class. But when should the

class set it? This `CountedLabel` class derives from `Label`, but it needs to detect when the `Text` property has changed so that it can count up the words.

Does `Label` have a `TextChanged` event? No it does not. However, `BindableObject` implements the `INotifyPropertyChanged` interface. This is a very important .NET interface, particularly for applications that implement the Model-View-ViewModel (MVVM) architecture. In Chapter 18 you'll see how to use it in your own data classes.

The `INotifyPropertyChanged` interface is defined in the `System.ComponentModel` namespace like so:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Every class that derives from `BindableObject` automatically fires this `PropertyChanged` event whenever any property backed by a `BindableProperty` changes. The `PropertyChangedEventArgs` object that accompanies this event includes a property named `PropertyName` of type `string` that identifies the property that has changed.

So all that's necessary is for `CountedLabel` to attach a handler for the `PropertyChanged` event and check for a property name of "Text". From there it can use whatever technique it wants for calculating a word count. The complete `CountedLabel` class uses a lambda function on the `PropertyChanged` event. The handler calls `Split` to break the string into words and see how many pieces result. The `Split` method splits the text based on spaces, dashes, and em dashes (Unicode \u2014):

```
public class CountedLabel : Label
{
    static readonly BindablePropertyKey WordCountKey =
        BindableProperty.CreateReadOnly("WordCount",            // propertyName
                                        typeof(int),            // returnType
                                        typeof(CountedLabel),   // declaringType
                                        0);                     // defaultValue

    public static readonly BindableProperty WordCountProperty = WordCountKey.BindableProperty;

    public CountedLabel()
    {
        // Set the WordCount property when the Text property changes.
        PropertyChanged += (object sender, PropertyChangedEventArgs args) =>
            {
                if (args.PropertyName == "Text")
                {
                    if (String.IsNullOrEmpty(Text))
                    {
                        WordCount = 0;
                    }
                    else
                    {
                        WordCount = Text.Split(' ', '-', '\u2014').Length;
```

```
                    }
                }
            };
    }

    public int WordCount
    {
        private set { SetValue(WordCountKey, value); }
        get { return (int)GetValue(WordCountProperty); }
    }
}
```

The class includes a `using` directive for the `System.ComponentModel` namespace for the `PropertyChangedEventArgs` argument to the handler. Watch out: Xamarin.Forms defines a class named `PropertyChangingEventArgs` (present tense). That's not what you want for the `PropertyChanged` handler. You want `PropertyChangedEventArgs` (past tense).

Because this call of the `Split` method splits the text at blank characters, dashes, and em dashes, you might assume that `CountedLabel` will be demonstrated with text that contains some dashes and em dashes. This is true. The **BaskervillesCount** program is a variation of the **Baskervilles** program from Chapter 3, but here the paragraph of text is displayed with a `CountedLabel`, and a regular `Label` is included to display the word count:

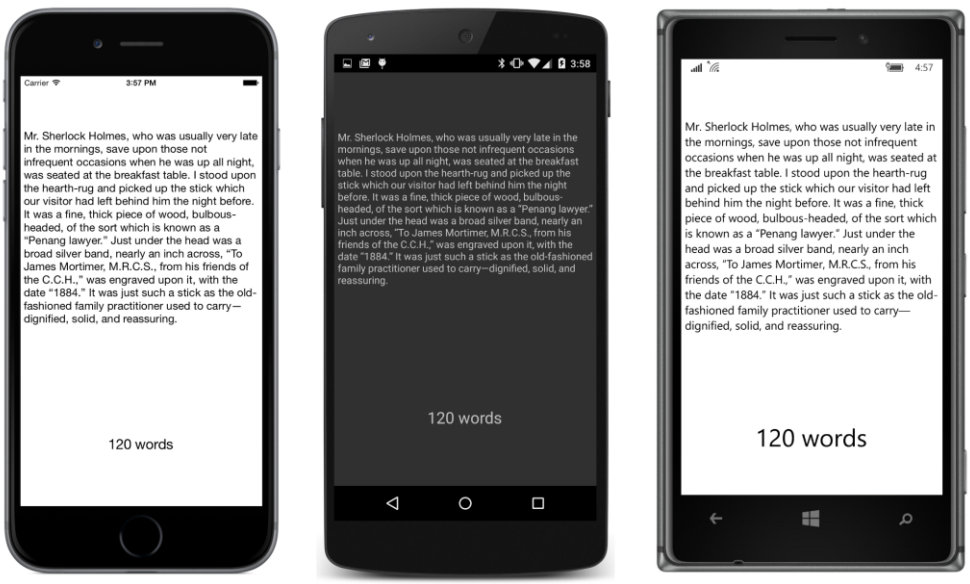
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="BaskervillesCount.BaskervillesCountPage"
             Padding="5, 0">

    <StackLayout>
        <toolkit:CountedLabel x:Name="countedLabel"
                              VerticalOptions="CenterAndExpand"
                              Text=
"Mr. Sherlock Holmes, who was usually very late in
the mornings, save upon those not infrequent
occasions when he was up all night, was seated at
the breakfast table. I stood upon the hearth-rug
and picked up the stick which our visitor had left
behind him the night before. It was a fine, thick
piece of wood, bulbous-headed, of the sort which
is known as a &#x201C;Penang lawyer.&#x201D; Just
under the head was a broad silver band, nearly an
inch across, &#x201C;To James Mortimer, M.R.C.S.,
from his friends of the C.C.H.,&#x201D; was engraved
upon it, with the date &#x201C;1884.&#x201D; It was
just such a stick as the old-fashioned family
practitioner used to carry&#x2014;dignified, solid,
and reassuring." />

        <Label x:Name="wordCountLabel"
               Text="???"
```

```
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center" />

    </StackLayout>
</ContentPage>
```

That regular `Label` is set in the code-behind file:

```
public partial class BaskervillesCountPage : ContentPage
{
    public BaskervillesCountPage()
    {
        InitializeComponent();

        int wordCount = countedLabel.WordCount;
        wordCountLabel.Text = wordCount + " words";
    }
}
```

The word count that it calculates is based on the assumption that all hyphens in the text separate two words and that "hearth-rug" and "bulbous-headed" should be counted as two words each. That's not always true, of course, but word counts are not quite as algorithmically simple as this code might imply:



How would the program be structured if the text changed dynamically while the program was running? In that case, it would be necessary to update the word count whenever the `WordCount` property of the `CountedLabel` object changed. You could attach a `PropertyChanged` handler on the `CountedLabel` object and check for the property named "WordCount".

However, exercise caution if you try to set such an event handler from XAML—for example, like so:

```
<toolkit:CountedLabel x:Name="countedLabel"
                      VerticalOptions="CenterAndExpand"
                      PropertyChanged="OnCountedLabelPropertyChanged"
                      Text=" … " />
```

You'll probably want to code the event handler in the code-behind file like this:

```
void OnCountedLabelPropertyChanged(object sender,
                                   PropertyChangedEventArgs args)
{
    wordCountLabel.Text = countedLabel.WordCount + " words";
}
```

That handler will fire when the `Text` property is set by the XAML parser, but the event handler is trying to set the `Text` property of the second `Label`, which hasn't been instantiated yet, which means that the `wordCountLabel` field is still set to `null`. This is an issue that will come up again in Chapter 15 when working with interactive controls, but it will be pretty much solved when we work with data binding in Chapter 16.

There is another variation of a bindable property coming up in Chapter 14 on the `AbsoluteLayout`: this is the *attached bindable property,* and it is very useful in implementing certain types of layouts, as you'll also discover in Chapter 26, "Custom layouts."

Meanwhile, let's look at one of the most important applications of bindable properties: styles.

Chapter 12

Styles

Xamarin.Forms applications often contain multiple elements with identical property settings. For example, you might have several buttons with the same colors, font sizes, and layout options. In code, you can assign identical properties to multiple buttons in a loop, but loops aren't available in XAML. If you want to avoid a lot of repetitious markup, another solution is required.

The solution is the `Style` class, which is a collection of property settings consolidated in one convenient object. You can set a `Style` object to the `Style` property of any class that derives from `VisualElement`. Generally, you'll apply the same `Style` object to multiple elements, and the style is shared among these elements.

The `Style` is the primary tool for giving visual elements a consistent appearance in your Xamarin.Forms applications. Styles help reduce repetitious markup in XAML files and allow applications to be more easily changed and maintained.

Styles were designed primarily with XAML in mind, and they probably wouldn't have been invented in a code-only environment. However, you'll see in this chapter how to define and use styles in code and how to combine code and markup to change program styling dynamically at run time.

The basic Style

In Chapter 10, "XAML markup extensions," you saw a trio of buttons that contained a lot of identical markup. Here they are again:

```
<StackLayout>
    <Button Text=" Carpe diem "
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
            FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>
```

```
    <Button Text=" Sapere aude "
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
                FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>

    <Button Text=" Discere faciendo "
            HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand"
            BorderWidth="3"
            TextColor="Red"
            FontSize="Large">
        <Button.BackgroundColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="#404040" />
        </Button.BackgroundColor>
        <Button.BorderColor>
            <OnPlatform x:TypeArguments="Color"
                        Android="White"
                        WinPhone="Black" />
        </Button.BorderColor>
    </Button>
</StackLayout>
```

With the exception of the `Text` property, all three buttons have the same property settings.

One partial solution to this repetitious markup involves defining property values in a resource dictionary and referencing them with the `StaticResource` markup extension. As you saw in the **ResourceSharing** project in Chapter 10, this technique doesn't reduce the markup bulk, but it does consolidate the values in one place.

To reduce the markup bulk, you'll need a `Style`. A `Style` object is almost always defined in a `ResourceDictionary`. Generally, you'll begin with a `Resources` section at the top of the page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BasicStyle.BasicStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            …
        </ResourceDictionary>
```

```
    </ContentPage.Resources>
    …
</ContentPage>
```

Instantiate a `Style` with separate start and end tags:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BasicStyle.BasicStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                …
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    …
</ContentPage>
```

Because the `Style` is an object in a `ResourceDictionary`, you'll need an `x:Key` attribute to give it a descriptive dictionary key. You must also set the `TargetType` property. This is the type of the visual element that the style is designed for, which in this case is `Button`.

As you'll see in the next section of this chapter, you can also define a `Style` in code, in which case the `Style` constructor requires an object of type `Type` for the `TargetType` property. The `TargetType` property does not have a public `set` accessor; hence the `TargetType` property cannot be changed after the `Style` is created.

`Style` also defines another important get-only property named `Setters` of type `IList<Setter>`, which is a collection of `Setter` objects. Each `Setter` is responsible for defining a property setting in the style. The `Setter` class defines just two properties:

- `Property` of type `BindableProperty`
- `Value` of type `Object`

Properties set in the `Style` must be backed by bindable properties! But when you set the `Property` property in XAML, don't use the entire fully qualified bindable property name. Just specify the text name, which is the same as the name of the related CLR property. Here's an example:

```
<Setter Property="HorizontalOptions" Value="Center" />
```

The XAML parser uses the familiar `TypeConverter` classes when parsing the `Value` settings of these `Setter` instances, so you can use the same property settings that you use normally.

`Setters` is the content property of `Style`, so you don't need the `Style.Setters` tags to add `Setter` objects to the `Style`:

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

```
             x:Class="BasicStyle.BasicStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BorderWidth" Value="3" />
                <Setter Property="TextColor" Value="Red" />
                <Setter Property="FontSize" Value="Large" />
                …
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    …
</ContentPage>
```

Two more `Setter` objects are required for `BackgroundColor` and `BorderColor`. These involve `OnPlatform` and might at first seem to be impossible to express in markup. However, it's possible to express the `Value` property of `Setter` as a property element, with the `OnPlatform` markup between the property element tags:

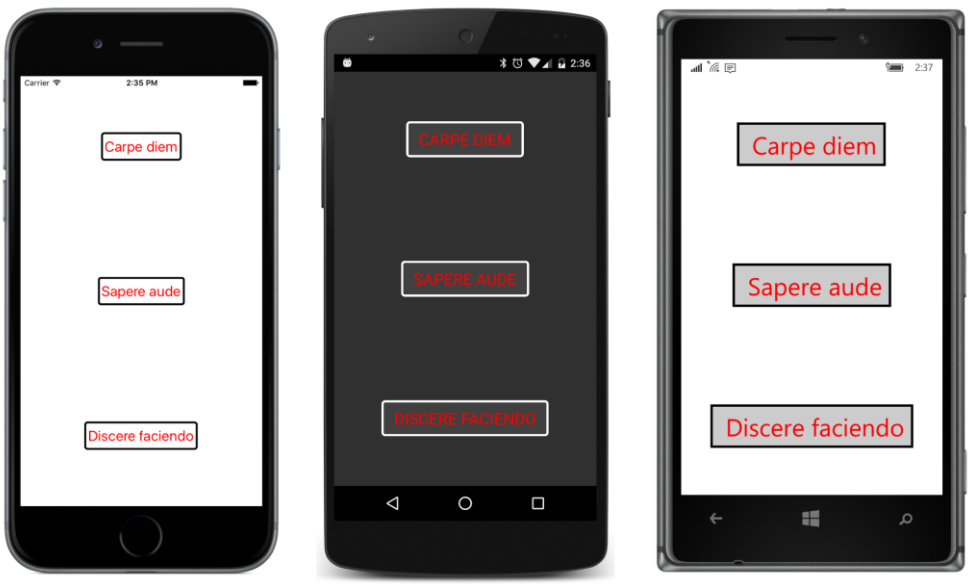
```
                <Setter Property="BackgroundColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="#404040" />
                    </Setter.Value>
                </Setter>
                <Setter Property="BorderColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="White"
                                    WinPhone="Black" />
                    </Setter.Value>
                </Setter>
```

The final step is to set this `Style` object to the `Style` property of each `Button`. Use the familiar `StaticResource` markup extension to reference the dictionary key. Here is the complete XAML file in the **BasicStyle** project:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BasicStyle.BasicStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BorderWidth" Value="3" />
                <Setter Property="TextColor" Value="Red" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="BackgroundColor">
```

```
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="#404040" />
                    </Setter.Value>
                </Setter>
                <Setter Property="BorderColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="White"
                                    WinPhone="Black" />
                    </Setter.Value>
                </Setter>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Carpe diem "
                Style="{StaticResource buttonStyle}" />

        <Button Text=" Sapere aude "
                Style="{StaticResource buttonStyle}" />

        <Button Text=" Discere faciendo "
                Style="{StaticResource buttonStyle}" />
    </StackLayout>
</ContentPage>
```

Now all these property settings are in one `Style` object that is shared among multiple `Button` elements:



The visuals are the same as those in the **ResourceSharing** program in Chapter 10, but the markup is a lot more concise.

Even after working with `Style` objects in markup, it's easy to be flummoxed with an unwieldy `Value` property. Suppose you'd like to define a `Setter` for the `TextColor` using the `Color.FromHsla` static method. You can define such a color by using the `x:FactoryMethod` attribute, but how can you possibly set such an unwieldy chunk of markup to the `Value` property of the `Setter` object? As you saw earlier, the solution is almost always property-element syntax:

```
<ResourceDictionary>
    <Style x:Key="buttonStyle" TargetType="Button">
        …
        <Setter Property="TextColor">
            <Setter.Value>
                <Color x:FactoryMethod="FromHsla">
                    <x:Arguments>
                        <x:Double>0.83</x:Double>
                        <x:Double>1</x:Double>
                        <x:Double>0.75</x:Double>
                        <x:Double>1</x:Double>
                    </x:Arguments>
                </Color>
            </Setter.Value>
        </Setter>
        …
    </Style>
</ResourceDictionary>
```

Here's another way to do it: Define the `Color` value as a separate item in the resource dictionary, and then use `StaticResource` to set it to the `Value` property of the `Setter`:

```
<ResourceDictionary>
    <Color x:Key="btnTextColor"
           x:FactoryMethod="FromHsla">
        <x:Arguments>
            <x:Double>0.83</x:Double>
            <x:Double>1</x:Double>
            <x:Double>0.75</x:Double>
            <x:Double>1</x:Double>
        </x:Arguments>
    </Color>

    <Style x:Key="buttonStyle" TargetType="Button">
        …
        <Setter Property="TextColor" Value="{StaticResource btnTextColor}" />
        …
    </Style>
</ResourceDictionary>
```

This is a good technique if you're sharing the same `Color` value among multiple styles or multiple setters.

You can override a property setting from a `Style` by setting a property directly in the visual element. Notice that the second `Button` has its `TextColor` property set to `Maroon`:

```
<StackLayout>
    <Button Text=" Carpe diem "
            Style="{StaticResource buttonStyle}" />

    <Button Text=" Sapere aude "
            TextColor="Maroon"
            Style="{StaticResource buttonStyle}" />

    <Button Text=" Discere faciendo "
            Style="{StaticResource buttonStyle}" />
</StackLayout>
```

The center `Button` will have maroon text while the other two buttons get their `TextColor` settings from the `Style`. A property directly set on the visual element is sometimes called a *local setting* or a *manual setting*, and it always overrides the property setting from the `Style`.

The `Style` object in the **BasicStyle** program is shared among the three buttons. The sharing of styles has an important implication for the `Setter` objects. Any object set to the `Value` property of a `Setter` must be shareable. Don't try to do something like this:

```
<!-- Invalid XAML! -->
<Style x:Key="frameStyle" TargetType="Frame">
    <Setter Property="OutlineColor" Value="Accent" />
    <Setter Property="Content">
        <Setter.Value>
            <Label Text="Text in a Frame" />
        </Setter.Value>
    </Setter>
</Style>
```

This XAML doesn't work for two reasons: `Content` is not backed by a `BindableProperty` and therefore cannot be used in a `Setter`. But the obvious intent here is for every `Frame`—or at least every `Frame` on which this style is applied—to get that same `Label` object as content. A single `Label` object can't appear in multiple places on the page. A much better way to do something like this is to derive a class from `Frame` and set a `Label` as the `Content` property, or to derive a class from `ContentView` that includes a `Frame` and `Label`.

You might want to use a style to set an event handler for an event such as `Clicked`. That would be useful and convenient, but it is not supported. Event handlers must be set on the elements themselves. (However, the `Style` class does support objects called *triggers*, which can respond to events or property changes. Triggers are discussed in Chapter 23, "Triggers and behaviors.")

You cannot set the `GestureRecognizers` property in a style. That would be useful as well, but `GestureRecognizers` is not backed by a bindable property.

If a bindable property is a reference type, and if the default value is `null`, you can use a style to set the property to a non-`null` object. But you might also want to override that style setting with a local

setting that sets the property back to `null`. You can set a property to `null` in XAML with the `{x:Null}` markup extension.

Styles in code

Although styles are mostly defined and used in XAML, you should know what they look like when defined and used in code. Here's the page class for the code-only **BasicStyleCode** project. The constructor of the `BasicStyleCodePage` class uses object-initialization syntax to mimic the XAML syntax in defining the `Style` object and applying it to three buttons:

```
public class BasicStyleCodePage : ContentPage
{
    public BasicStyleCodePage()
    {
        Resources = new ResourceDictionary
        {
            { "buttonStyle", new Style(typeof(Button))
                {
                    Setters =
                    {
                        new Setter
                        {
                            Property = View.HorizontalOptionsProperty,
                            Value = LayoutOptions.Center
                        },
                        new Setter
                        {
                            Property = View.VerticalOptionsProperty,
                            Value = LayoutOptions.CenterAndExpand
                        },
                        new Setter
                        {
                            Property = Button.BorderWidthProperty,
                            Value = 3
                        },
                        new Setter
                        {
                            Property = Button.TextColorProperty,
                            Value = Color.Red
                        },
                        new Setter
                        {
                            Property = Button.FontSizeProperty,
                            Value = Device.GetNamedSize(NamedSize.Large, typeof(Button))
                        },
                        new Setter
                        {
                            Property = VisualElement.BackgroundColorProperty,
                            Value = Device.OnPlatform(Color.Default,
                                                      Color.FromRgb(0x40, 0x40, 0x40),
```

```
                                                Color.Default)
                        },
                        new Setter
                        {
                            Property = Button.BorderColorProperty,
                            Value = Device.OnPlatform(Color.Default,
                                                Color.White,
                                                Color.Black)
                        }
                    }
                }
            }
        };

        Content = new StackLayout
        {
            Children =
            {
                new Button
                {
                    Text = " Carpe diem ",
                    Style = (Style)Resources["buttonStyle"]
                },
                new Button
                {
                    Text = " Sapere aude ",
                    Style = (Style)Resources["buttonStyle"]
                },
                new Button
                {
                    Text = " Discere faciendo ",
                    Style = (Style)Resources["buttonStyle"]
                }
            }
        };
    }
}
```

It's much more obvious in code than in XAML that the `Property` property of the `Setter` is of type `BindableProperty`.

The first two `Setter` objects in this example are initialized with the `BindableProperties` objects named `View.HorizontalOptionsProperty` and `View.VerticalOptionsProperty`. You could use `Button.HorizontalOptionsProperty` and `Button.VerticalOptionsProperty` instead because `Button` inherits these properties from `View`. Or you can change the class name to any other class that derives from `View`.

As usual, the use of a `ResourceDictionary` in code seems pointless. You could eliminate the dictionary and just assign the `Style` objects directly to the `Style` properties of the buttons. However, even in code, the `Style` is a convenient way to bundle all the property settings together into one compact package.

Style inheritance

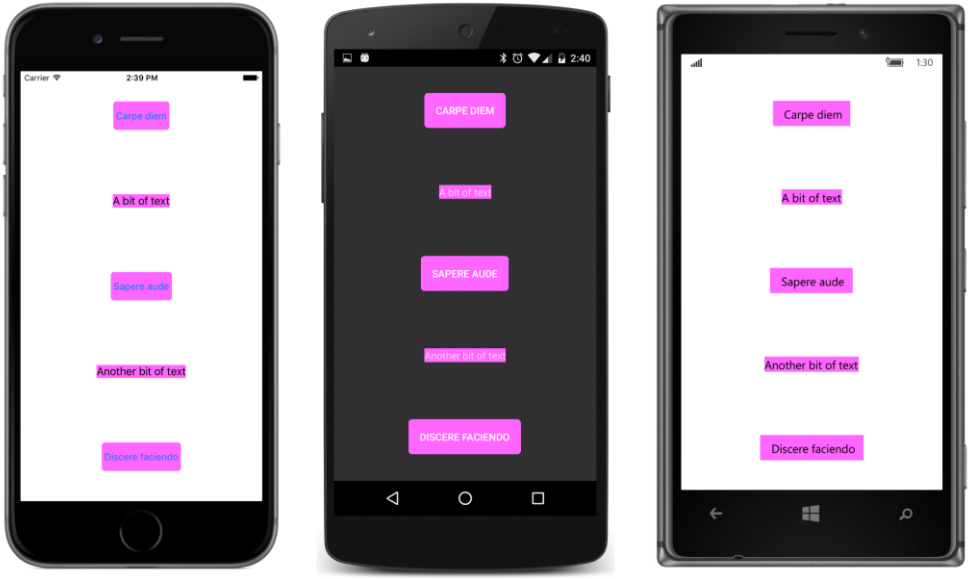
The `TargetType` of the `Style` serves two different functions: One of these functions is described in the next section on implicit styles. The other function is for the benefit of the XAML parser. The XAML parser must be able to resolve the property names in the `Setter` objects, and for that it needs a class name provided by the `TargetType`.

All the properties in the style must be defined by or inherited by the class specified in the `TargetType` property. The type of the visual element on which the `Style` is set must be the same as the `TargetType` or a derived class of the `TargetType`.

If you need a `Style` only for properties defined by `View`, you can set the `TargetType` to `View` and still use the style on buttons or any other `View` derivative, as in this modified version of the **BasicStyle** program:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BasicStyle.BasicStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="viewStyle" TargetType="View">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BackgroundColor" Value="Pink" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Carpe diem "
                Style="{StaticResource viewStyle}" />

        <Label Text ="A bit of text"
               Style="{StaticResource viewStyle}" />

        <Button Text=" Sapere aude "
                Style="{StaticResource viewStyle}" />

        <Label Text ="Another bit of text"
               Style="{StaticResource viewStyle}" />

        <Button Text=" Discere faciendo "
                Style="{StaticResource viewStyle}" />
    </StackLayout>
</ContentPage>
```

As you can see, the same style is applied to all the `Button` and `Label` children of the `StackLayout`:



But suppose you now want to expand on this style, but differently for `Button` and `Label`. Is that possible?

Yes, it is. Styles can derive from other styles. The `Style` class includes a property named `BasedOn` of type `Style`. In code, you can set this `BasedOn` property directly to another `Style` object. In XAML you set the `BasedOn` attribute to a `StaticResource` markup extension that references a previously created `Style`. The new `Style` can include `Setter` objects for new properties or use them to override properties in the earlier `Style`. The `BasedOn` style must target the same class or an ancestor class of the new style's `TargetType`.

Here's the XAML file for a project named **StyleInheritance**. The application has a reference to the **Xamarin.FormsBook.Toolkit** assembly for two purposes: It uses the `HslColor` markup extension to demonstrate that markup extensions are legitimate value settings in `Setter` objects and to demonstrate that a style can be defined for a custom class, in this case `AltLabel`.

The `ResourceDictionary` contains four styles: The first has a dictionary key of "visualStyle". The `Style` with the dictionary key of "baseStyle" derives from "visualStyle". The styles with keys of "labelStyle" and "buttonStyle" derive from "baseStyle":

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="StyleInheritance.StyleInheritancePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="visualStyle" TargetType="VisualElement">
```

```
                <Setter Property="BackgroundColor"
                        Value="{toolkit:HslColor H=0, S=1, L=0.8}" />
            </Style>

            <Style x:Key="baseStyle" TargetType="View"
                   BasedOn="{StaticResource visualStyle}">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style x:Key="labelStyle" TargetType="toolkit:AltLabel"
                   BasedOn="{StaticResource baseStyle}">
                <Setter Property="TextColor" Value="Black" />
                <Setter Property="PointSize" Value="12" />
            </Style>

            <Style x:Key="buttonStyle" TargetType="Button"
                   BasedOn="{StaticResource baseStyle}">
                <Setter Property="TextColor" Value="Blue" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="BorderColor" Value="Blue" />
                <Setter Property="BorderWidth" Value="2" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ContentPage.Style>
        <StaticResourceExtension Key="visualStyle" />
    </ContentPage.Style>

    <StackLayout>
        <Button Text=" Carpe diem "
                Style="{StaticResource buttonStyle}" />

        <toolkit:AltLabel Text ="A bit of text"
                          Style="{StaticResource labelStyle}" />

        <Button Text=" Sapere aude "
                Style="{StaticResource buttonStyle}" />

        <toolkit:AltLabel Text ="Another bit of text"
                          Style="{StaticResource labelStyle}" />

        <Button Text=" Discere faciendo "
                Style="{StaticResource buttonStyle}" />
    </StackLayout>
</ContentPage>
```

Immediately after the `Resources` section is some markup that sets the `Style` property of the page itself to the "visualStyle" `Style`:

```
    <ContentPage.Style>
        <StaticResourceExtension Key="visualStyle" />
    </ContentPage.Style>
```

Because `Page` derives from `VisualElement` but not `View`, this is the only style in the resource dictionary that can be applied to the page. However, the style can't be applied to the page until after the `Resources` section, so using the element form of `StaticResource` is a good solution here. The entire background of the page is colored based on this style, and the style is also inherited by all the other styles:

If the `Style` for the `AltLabel` only included `Setter` objects for properties defined by `Label`, the `TargetType` could be `Label` instead of `AltLabel`. But the `Style` has a `Setter` for the `PointSize` property. That property is defined by `AltLabel`, so the `TargetType` must be `toolkit:AltLabel`.

A `Setter` can be defined for the `PointSize` property because `PointSize` is backed by a bindable property. If you change the accessibility of the `BindableProperty` object in `AltLabel` from `public` to `private`, the property will still work for many routine uses of `AltLabel`, but now `PointSize` cannot be set in a style `Setter`. The XAML parser will complain that it cannot find `PointSizeProperty`, which is the bindable property that backs the `PointSize` property.

You discovered in Chapter 10 how `StaticResource` works: When the XAML parser encounters a `StaticResource` markup extension, it searches up the visual tree for a matching dictionary key. This process has implications for styles. You can define a style in one `Resources` section and then override it with another style with the same dictionary key in a different `Resources` section lower in the visual tree. When you set the `BasedOn` property to a `StaticResource` markup extension, the style you're deriving from must be defined in the same `Resources` section (as demonstrated in the **StyleInheritance** program) or a `Resources` section higher in the visual tree.

This means that you can structure your styles in XAML in two hierarchical ways: You can use `BasedOn` to derive styles from other styles, and you can define styles at different levels in the visual

tree that derive from styles higher in the visual tree or replace them entirely.

For larger applications with multiple pages and lots of markup, the recommendation for defining styles is very simple—define your styles as close as possible to the elements that use those styles.

Adhering to this recommendation aids in maintaining the program and becomes particularly important when working with *implicit styles*.

Implicit styles

Every entry in a `ResourceDictionary` requires a dictionary key. This is an indisputable fact. If you try to pass a `null` key to the `Add` method of a `ResourceDictionary` object, you'll raise an `ArgumentNullException`.

However, there is one special case where a programmer is not required to supply this dictionary key. A dictionary key is instead generated automatically.

This special case is for a `Style` object added to a `ResourceDictionary` without an `x:Key` setting. The `ResourceDictionary` generates a key based on the `TargetType`, which is always required. (A little exploration will reveal that this special dictionary key is the fully qualified name associated with the `TargetType` of the `Style`. For a `TargetType` of `Button`, for example, the dictionary key is "Xamarin.Forms.Button". But you don't need to know that.)

You can also add a `Style` to a `ResourceDictionary` without a dictionary key in code: an overload of the `Add` method accepts an argument of type `Style` but doesn't require anything else.

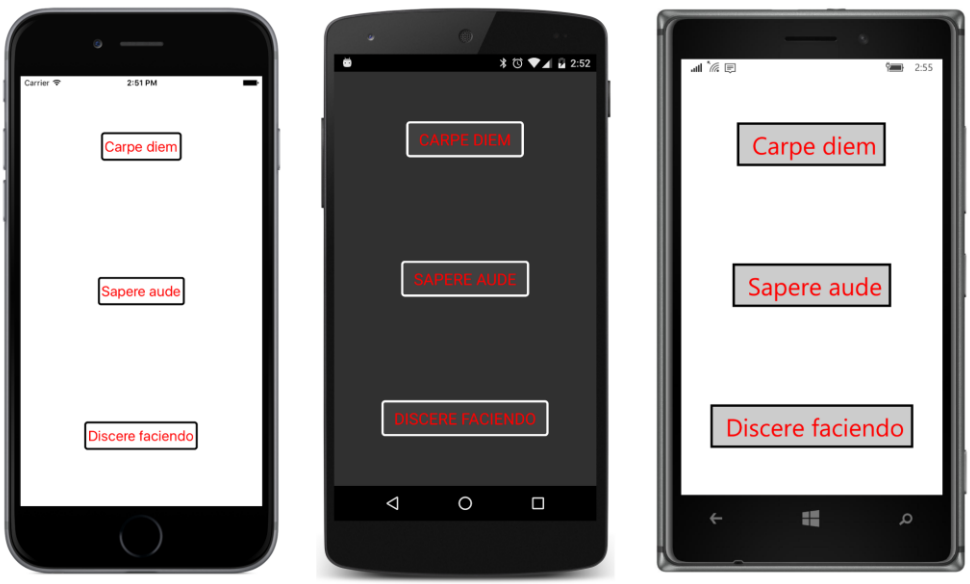
A `Style` object in a `ResourceDictionary` that has one of these generated keys is known as an *implicit style*, and the generated dictionary key is very special. You can't refer to this key directly using `StaticResource`. However, if an element within the scope of the `ResourceDictionary` has the same type as the dictionary key, and if that element does not have its `Style` property explicitly set to another `Style` object, then this implicit style is automatically applied.

The following XAML from the **ImplicitStyle** project demonstrates this. It is the same as the **BasicStyle** XAML file except that the `Style` has no `x:Key` setting and the `Style` properties on the buttons aren't set using `StaticResource`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ImplicitStyle.ImplicitStylePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BorderWidth" Value="3" />
                <Setter Property="TextColor" Value="Red" />
```

```
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="BackgroundColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="#404040" />
                    </Setter.Value>
                </Setter>

                <Setter Property="BorderColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    Android="White"
                                    WinPhone="Black" />
                    </Setter.Value>
                </Setter>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Carpe diem " />

        <Button Text=" Sapere aude " />

        <Button Text=" Discere faciendo " />
    </StackLayout>
</ContentPage>
```

Despite the absence of any explicit connection between the buttons and the style, the style is definitely applied:



An implicit style is applied only when the class of the element matches the `TargetType` of the `Style` exactly. If you include an element that derives from `Button` in the `StackLayout`, it would not have the `Style` applied.

You can use local property settings to override properties set through the implicit style, just as you can override property settings in a style set with `StaticResource`.

You will find implicit styles to be very powerful and extremely useful. Whenever you have several views of the same type and you determine that you want them all to have an identical property setting or two, it's very easy to quickly define an implicit style. You don't have to touch the elements themselves.

However, with great power comes at least *some* programmer responsibility. Because no style is referenced in the elements themselves, it can be confusing when simply examining the XAML to determine whether some elements are styled or not. Sometimes the appearance of a page indicates that an implicit style is applied to some elements, but it's not quite obvious where the implicit style is defined. If you then want to change that implicit style, you have to manually search for it up the visual tree.

For this reason, you should define implicit styles *as close as possible* to the elements they are applied to. If the views getting the implicit style are in a particular `StackLayout`, then define the implicit style in the `Resources` section on that `StackLayout`. A comment or two might help avoid confusion as well.

Interestingly, implicit styles have a built-in restriction that might persuade you to keep them close to the elements they are applied to. Here's the restriction: You can derive an implicit style from a `Style` with an explicit dictionary key, but you can't go the other way around. You can't use `BasedOn` to reference an implicit style.

If you define a chain of styles that use `BasedOn` to derive from one another, the implicit style (if any) is always at the end of the chain. No further derivations are possible.

This implies that you can structure your styles with three types of hierarchies:

- From styles defined on the `Application` and `Page` down to styles defined on layouts lower in the visual tree.
- From styles defined for base classes such as `VisualElement` and `View` to styles defined for specific classes.
- From styles with explicit dictionary keys to implicit styles.

This is demonstrated in the **StyleHierarchy** project, which uses a similar (but somewhat simplified) set of styles as you saw earlier in the **StyleInheritance** project. However, these styles are now spread out over three `Resources` sections.

Using a technique you saw in the **ResourceTrees** program in Chapter 10, the **StyleHierarchy** project was given a XAML-based `App` class. The App.xaml class has a `ResourceDictionary` containing a style with just one property setter:

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StyleHierarchy.App">

    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="visualStyle" TargetType="VisualElement">
                <Setter Property="BackgroundColor" Value="Pink" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

In a multipage application, this style would be used throughout the application.

The code-behind file for the `App` class calls `InitializeComponent` to process the XAML file and sets the `MainPage` property:

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        MainPage = new StyleHierarchyPage();
    }
    …
}
```

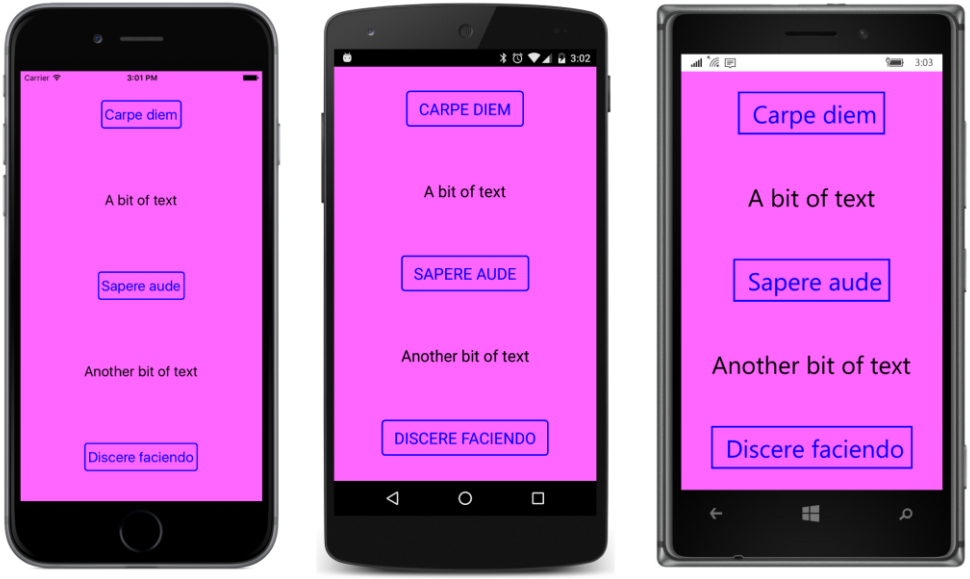
The XAML file for the page class defines one `Style` for the whole page that derives from the style in the `App` class and also two implicit styles that derive from the `Style` for the page. Notice that the `Style` property of the page is set to the `Style` defined in the `App` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StyleHierarchy.StyleHierarchyPage"
             Style="{StaticResource visualStyle}">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View"
                   BasedOn="{StaticResource visualStyle}">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <StackLayout.Resources>
```

```
            <ResourceDictionary>
                <Style TargetType="Label"
                       BasedOn="{StaticResource baseStyle}">
                    <Setter Property="TextColor" Value="Black" />
                    <Setter Property="FontSize" Value="Large" />
                </Style>

                <Style TargetType="Button"
                       BasedOn="{StaticResource baseStyle}">
                    <Setter Property="TextColor" Value="Blue" />
                    <Setter Property="FontSize" Value="Large" />
                    <Setter Property="BorderColor" Value="Blue" />
                    <Setter Property="BorderWidth" Value="2" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>

        <Button Text=" Carpe diem " />

        <Label Text ="A bit of text" />

        <Button Text=" Sapere aude " />

        <Label Text ="Another bit of text" />

        <Button Text=" Discere faciendo " />
    </StackLayout>
</ContentPage>
```

The implicit styles are defined as close to the target elements as possible.

Here's the result:



The incentive to separate `Style` objects into separate dictionaries doesn't make a lot of sense for very tiny programs like this one, but for larger programs, it becomes just as important to have a structured hierarchy of style definitions as it is to have a structured hierarchy of class definitions.

Sometimes you'll have a `Style` with an explicit dictionary key (for example "myButtonStyle"), but you'll want that same style to be implicit as well. Simply define a style based on that key with no key or setters of its own:

```
<Style TargetType="Button"
       BasedOn="{StaticResource myButtonStyle}" />
```

That's an implicit style that is identical to `myButtonStyle`.

Dynamic styles

A `Style` is generally a static object that is created and initialized in XAML or code and then remains unchanged for the duration of the application. The `Style` class does not derive from `BindableObject` and does not internally respond to changes in its properties. For example, if you assign a `Style` object to an element and then modify one of the `Setter` objects by giving it a new value, the new value won't show up in the element. Similarly, the target element won't change if you add a `Setter` or remove a `Setter` from the `Setters` collection. For these new property setters to take effect, you need to use code to detach the style from the element by setting the `Style` property to `null` and then reattach the style to the element.

However, your application can respond to style changes dynamically at run time through the use of `DynamicResource`. You'll recall that `DynamicResource` is similar to `StaticResource` in that it uses a dictionary key to fetch an object or a value from a resource dictionary. The difference is that `StaticResource` is a one-time dictionary lookup while `DynamicResource` maintains a link to the actual dictionary key. If the dictionary entry associated with that key is replaced with a new object, that change is propagated to the element.

This facility allows an application to implement a feature sometimes called *dynamic styles*. For example, you might include a facility in your program for stylistic themes (involving fonts and colors, perhaps), and you might make these themes selectable by the user. The application can switch between these themes because they are implemented with styles.

There's nothing in a style itself that indicates a dynamic style. A style becomes dynamic solely by being referenced using `DynamicResource` rather than `StaticResource`.

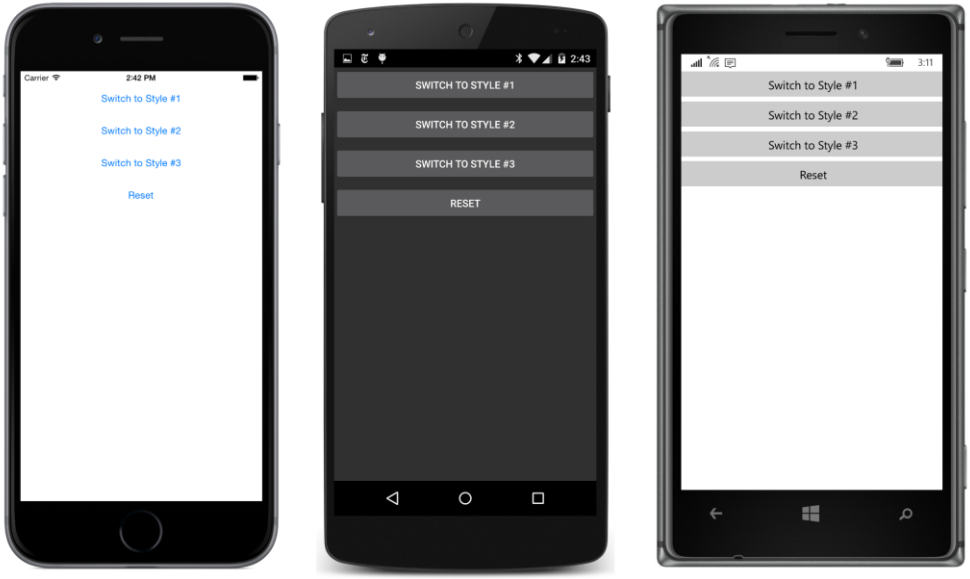
The **DynamicStyles** project demonstrates the mechanics of this process. Here is the XAML file for the `DynamicStylesPage` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DynamicStyles.DynamicStylesPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0"
                    Android="0"
                    WinPhone="0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseButtonStyle" TargetType="Button">
                <Setter Property="FontSize" Value="Large" />
            </Style>

            <Style x:Key="buttonStyle1" TargetType="Button"
                   BasedOn="{StaticResource baseButtonStyle}">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="TextColor" Value="Red" />
            </Style>

            <Style x:Key="buttonStyle2" TargetType="Button"
                   BasedOn="{StaticResource baseButtonStyle}">
                <Setter Property="HorizontalOptions" Value="Start" />
                <Setter Property="VerticalOptions" Value="EndAndExpand" />
                <Setter Property="TextColor" Value="Green" />
                <Setter Property="FontAttributes" Value="Italic" />
            </Style>

            <Style x:Key="buttonStyle3" TargetType="Button"
```

```
                BasedOn="{StaticResource baseButtonStyle}">
            <Setter Property="HorizontalOptions" Value="End" />
            <Setter Property="VerticalOptions" Value="StartAndExpand" />
            <Setter Property="TextColor" Value="Blue" />
            <Setter Property="FontAttributes" Value="Bold" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<StackLayout>
    <Button Text=" Switch to Style #1 "
            Style="{DynamicResource buttonStyle}"
            Clicked="OnButton1Clicked" />

    <Button Text=" Switch to Style #2 "
            Style="{DynamicResource buttonStyle}"
            Clicked="OnButton2Clicked" />

    <Button Text=" Switch to Style #3 "
            Style="{DynamicResource buttonStyle}"
            Clicked="OnButton3Clicked" />

    <Button Text=" Reset "
            Style="{DynamicResource buttonStyle}"
            Clicked="OnResetButtonClicked" />
</StackLayout>
</ContentPage>
```

The `Resources` section defines four styles: a simple style with the key "baseButtonStyle", and then three styles that derive from that style with the keys "buttonStyle1", "buttonStyle2", and "buttonStyle3".

However, the four `Button` elements toward the bottom of the XAML file all use `DynamicResource` to reference a style with the simpler key "buttonStyle". Where is the `Style` with that key? It does not exist. However, because the four button `Style` properties are set with `DynamicResource`, the missing dictionary key is not a problem. No exception is raised. But no `Style` is applied, which means that the buttons have a default appearance:



Each of the four `Button` elements has a `Clicked` handler attached, and in the code-behind file, the first three handlers set a dictionary entry with the key "buttonStyle" to one of the three numbered styles already defined in the dictionary:

```
public partial class DynamicStylesPage : ContentPage
{
    public DynamicStylesPage()
    {
        InitializeComponent();
    }

    void OnButton1Clicked(object sender, EventArgs args)
    {
        Resources["buttonStyle"] = Resources["buttonStyle1"];
    }

    void OnButton2Clicked(object sender, EventArgs args)
    {
        Resources["buttonStyle"] = Resources["buttonStyle2"];
    }

    void OnButton3Clicked(object sender, EventArgs args)
    {
        Resources["buttonStyle"] = Resources["buttonStyle3"];
    }

    void OnResetButtonClicked(object sender, EventArgs args)
    {
        Resources["buttonStyle"] = null;
    }
}
```

When you press one of the first three buttons, all four buttons get the selected style. Here's the program running on all three platforms showing the results (from left to right) when buttons 1, 2, and 3 are pressed:

Pressing the fourth button returns everything to the initial conditions by setting the value associated with the "buttonStyle" key to `null`. (You might also consider calling `Remove` or `Clear` on the `ResourceDictionary` object to remove the key entirely, but that doesn't work in the version of Xamarin.Forms used for this chapter.)

Suppose you want to derive another `Style` from the `Style` with the key "buttonStyle". How do you do this in XAML, considering that the "buttonStyle" dictionary entry doesn't exist until one of the first three buttons is pressed?

You can't do it like this:

```
<!-- This won't work! -->
<Style x:Key="newButtonStyle" TargetType="Button"
       BasedOn="{StaticResource buttonStyle}">
    ...
</Style>
```

`StaticResource` will raise an exception if the "buttonStyle" key does not exist, and even if the key does exist, the use of `StaticResource` won't allow changes in the dictionary entry to be reflected in this new style.

However, changing `StaticResource` to `DynamicResource` won't work either:

```
<!-- This won't work either! -->
<Style x:Key="newButtonStyle" TargetType="Button"
```

```
        BasedOn="{DynamicResource buttonStyle}">
    …
</Style>
```

`DynamicResource` works only with properties backed by bindable properties, and that is not the case here. `Style` doesn't derive from `BindableObject`, so it can't support bindable properties.

Instead, `Style` defines a property specifically for the purpose of inheriting dynamic styles. The property is `BaseResourceKey`, which is intended to be set directly to a dictionary key that might not yet exist or whose value might change dynamically, which is the case with the "buttonStyle" key:

```
<!-- This works!! -->
<Style x:Key="newButtonStyle" TargetType="Button"
       BaseResourceKey="buttonStyle">
    …
</Style>
```

The use of `BaseResourceKey` is demonstrated by the **DynamicStylesInheritance** project, which is very similar to the **DynamicStyles** project. Indeed, the code-behind processing is identical. Toward the bottom of the `Resources` section, a new `Style` is defined with a key of "newButtonStyle" that uses `BaseResourceKey` to reference the "buttonStyle" entry and add a couple of properties, including one that uses `OnPlatform`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DynamicStylesInheritance.DynamicStylesInheritancePage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0"
                    Android="0"
                    WinPhone="0" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseButtonStyle" TargetType="Button">
                <Setter Property="FontSize" Value="Large" />
            </Style>

            <Style x:Key="buttonStyle1" TargetType="Button"
                   BasedOn="{StaticResource baseButtonStyle}">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="TextColor" Value="Red" />
            </Style>

            <Style x:Key="buttonStyle2" TargetType="Button"
                   BasedOn="{StaticResource baseButtonStyle}">
                <Setter Property="HorizontalOptions" Value="Start" />
                <Setter Property="VerticalOptions" Value="EndAndExpand" />
                <Setter Property="TextColor" Value="Green" />
                <Setter Property="FontAttributes" Value="Italic" />
```

```
            </Style>

            <Style x:Key="buttonStyle3" TargetType="Button"
                   BasedOn="{StaticResource baseButtonStyle}">
                <Setter Property="HorizontalOptions" Value="End" />
                <Setter Property="VerticalOptions" Value="StartAndExpand" />
                <Setter Property="TextColor" Value="Blue" />
                <Setter Property="FontAttributes" Value="Bold" />
            </Style>

            <!-- New style definition. -->
            <Style x:Key="newButtonStyle" TargetType="Button"
                   BaseResourceKey="buttonStyle">
                <Setter Property="BackgroundColor">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="Color"
                                    iOS="#C0C0C0"
                                    Android="#404040"
                                    WinPhone="Gray" />
                    </Setter.Value>
                </Setter>
                <Setter Property="BorderColor" Value="Red" />
                <Setter Property="BorderWidth" Value="3" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text=" Switch to Style #1 "
                Style="{StaticResource newButtonStyle}"
                Clicked="OnButton1Clicked" />

        <Button Text=" Switch to Style #2 "
                Style="{StaticResource newButtonStyle}"
                Clicked="OnButton2Clicked" />

        <Button Text=" Switch to Style #3 "
                Style="{StaticResource newButtonStyle}"
                Clicked="OnButton3Clicked" />

        <Button Text=" Reset "
                Style="{DynamicResource buttonStyle}"
                Clicked="OnResetButtonClicked" />
    </StackLayout>
</ContentPage>
```

Notice that the first three `Button` elements reference the "newButtonStyle" dictionary entry with `StaticResource`. `DynamicResource` is not needed here because the `Style` object associated with the "newButtonStyle" will not itself change except for the `Style` that it derives from. The `Style` with the key "newButtonStyle" maintains a link with "buttonStyle" and internally alters itself when that underlying style changes. When the program begins to run, only the properties defined in the "newButtonStyle" are applied to those three buttons:

The **Reset** button continues to reference the "buttonStyle" entry.

As in the **DynamicStyles** program, the code-behind file sets that dictionary entry when you click one of the first three buttons, so all the buttons pick up the "buttonStyle" properties as well. Here are the results for (from left to right) clicks of buttons 3, 2, and 1:

Device styles

Xamarin.Forms includes six built-in dynamic styles. These are known as *device styles*, and they are members of a nested class of `Device` named `Styles`. This `Styles` class defines 12 `static` and `readonly` fields that help reference these six styles in code:

- `BodyStyle` of type `Style`.
- `BodyStyleKey` of type `string` and equal to "BodyStyle."
- `TitleStyle` of type `Style`.
- `TitleStyleKey` of type `string` and equal to "TitleStyle."
- `SubtitleStyle` of type `Style`.
- `SubtitleStyleKey` of type `string` and equal to "SubtitleStyle."
- `CaptionStyle` of type `Style`.
- `CaptionStyleKey` of type `string` and equal to "CaptionStyle."
- `ListItemTextStyle` of type `Style`.
- `ListItemTextStyleKey` of type `string` and equal to "ListItemTextStyle."
- `ListItemDetailTextStyle` of type `Style`.
- `ListItemDetailTextStyleKey` of type `string` and equal to "ListItemDetailTextStyle."

All six styles have a `TargetType` of `Label` and are stored in a dictionary—but not a dictionary that application programs can access directly.

In code, you use the fields in this list for accessing the device styles. For example, you can set the `Device.Styles.BodyStyle` object directly to the `Style` property of a `Label` for text that might be appropriate for the body of a paragraph. If you're defining a style in code that derives from one of these device styles, set the `BaseResourceKey` to `Device.Styles.BodyStyleKey` or simply "BodyStyle" if you're not afraid of misspelling it.

In XAML, you'll simply use the text key "BodyStyle" with `DynamicResource` for setting this style to the `Style` property of a `Label` or to set `BaseResourceKey` when deriving a style from `Device.Styles.BodyStyle`.

The **DeviceStylesList** program demonstrates how to access these styles—and to define a new style that inherits from `SubtitleStyle`—both in XAML and in code. Here's the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DeviceStylesList.DeviceStylesListPage">
```

```
<ContentPage.Padding>
    <OnPlatform x:TypeArguments="Thickness"
                iOS="10, 20, 10, 0"
                Android="10, 0"
                WinPhone="10, 0" />
</ContentPage.Padding>

<ContentPage.Resources>
    <ResourceDictionary>
        <Style x:Key="newSubtitleStyle" TargetType="Label"
               BaseResourceKey="SubtitleStyle">
            <Setter Property="TextColor" Value="Accent" />
            <Setter Property="FontAttributes" Value="Italic" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<ScrollView>
    <StackLayout Spacing="20">

        <!-- Device styles set with DynamicResource -->
        <StackLayout>
            <StackLayout HorizontalOptions="Start">
                <Label Text="Device styles set with DynamicResource" />
                <BoxView Color="Accent" HeightRequest="3" />
            </StackLayout>

            <Label Text="No Style whatsoever" />

            <Label Text="Body Style"
                   Style="{DynamicResource BodyStyle}" />

            <Label Text="Title Style"
                   Style="{DynamicResource TitleStyle}" />

            <Label Text="Subtitle Style"
                   Style="{DynamicResource SubtitleStyle}" />

            <!-- Uses style derived from device style. -->
            <Label Text="New Subtitle Style"
                   Style="{StaticResource newSubtitleStyle}" />

            <Label Text="Caption Style"
                   Style="{DynamicResource CaptionStyle}" />

            <Label Text="List Item Text Style"
                   Style="{DynamicResource ListItemTextStyle}" />

            <Label Text="List Item Detail Text Style"
                   Style="{DynamicResource ListItemDetailTextStyle}" />
        </StackLayout>

        <!-- Device styles set in code -->
        <StackLayout x:Name="codeLabelStack">
```

```
                <StackLayout HorizontalOptions="Start">
                    <Label Text="Device styles set in code:" />
                    <BoxView Color="Accent" HeightRequest="3" />
                </StackLayout>
            </StackLayout>
        </StackLayout>
    </ScrollView>
</ContentPage>
```

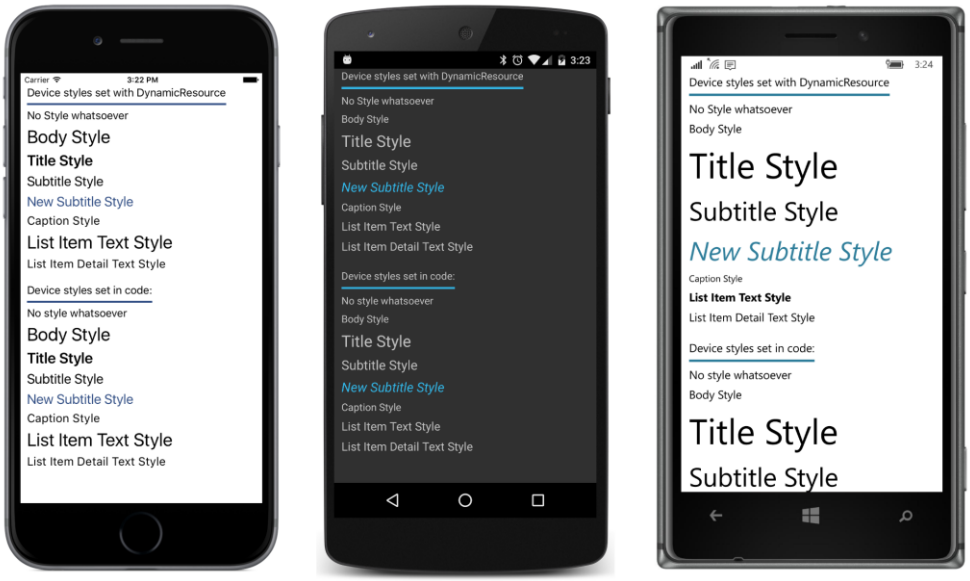
The `StackLayout` contains two `Label` and `BoxView` combinations (one at the top and one at the bottom) to display underlined headers. Following the first of these headers, `Label` elements reference the device styles with `DynamicResource`. The new subtitle style is defined in the `Resources` dictionary for the page.

The code-behind file accesses the device styles by using the properties in the `Device.Styles` class and creates a new style by deriving from `SubtitleStyle`:

```
public partial class DeviceStylesListPage : ContentPage
{
    public DeviceStylesListPage()
    {
        InitializeComponent();

        var styleItems = new[]
        {
            new { style = (Style)null, name = "No style whatsoever" },
            new { style = Device.Styles.BodyStyle, name = "Body Style" },
            new { style = Device.Styles.TitleStyle, name = "Title Style" },
            new { style = Device.Styles.SubtitleStyle, name = "Subtitle Style" },

            // Derived style
            new { style = new Style(typeof(Label))
            {
                BaseResourceKey = Device.Styles.SubtitleStyleKey,
                Setters =
                {
                    new Setter
                    {
                        Property = Label.TextColorProperty,
                        Value = Color.Accent
                    },
                    new Setter
                    {
                        Property = Label.FontAttributesProperty,
                        Value = FontAttributes.Italic
                    }
                }
            }, name = "New Subtitle Style" },

            new { style = Device.Styles.CaptionStyle, name = "Caption Style" },
            new { style = Device.Styles.ListItemTextStyle, name = "List Item Text Style" },
            new { style = Device.Styles.ListItemDetailTextStyle,
                  name = "List Item Detail Text Style" },
```

```
        };

        foreach (var styleItem in styleItems)
        {
            codeLabelStack.Children.Add(new Label
                {
                    Text = styleItem.name,
                    Style = styleItem.style
                });
        }
    }
}
```

The code and XAML result in identical styles, of course, but each platform implements these device styles in a different way:



The dynamic nature of these styles is easily demonstrated on iOS: While the **DeviceStyles** program is running, tap the **Home** button and run **Settings**. Pick the **General** item, then **Accessibility**, and **Larger Text**. A slider is available to make text smaller or larger. Change that slider, double tap the **Home** button to show the current applications, and select **DeviceStyles** again. You'll see the text set from device styles (or the styles that derive from device styles) change size, but none of the unstyled text in the application changes size. New objects have replaced the device styles in the dictionary.

The dynamic nature of device styles is not quite as obvious on Android because changes to the **Font size** item of the **Display** section in **Settings** affect all font sizes in a Xamarin.Forms program.

On a Windows 10 Mobile device, the **Text scaling** item in the **Ease of Access** and **More Options** section of **Settings** also affects all text.

The next chapter includes a program that demonstrates how to make a little e-book reader that lets you read a chapter of *Alice in Wonderland*. This program uses device styles for controlling the formatting of all the text, including the book and chapter titles.

But what this little e-book reader also includes are illustrations, and that requires an exploration into the subject of bitmaps.

Chapter 13

Bitmaps

The visual elements of a graphical user interface can be roughly divided between elements used for presentation (such as text) and those capable of interaction with the user, such as buttons, sliders, and list boxes.

Text is essential for presentation, but pictures are often just as important as a way to supplement text and convey crucial information. The web, for example, would be inconceivable without pictures. These pictures are often in the form of rectangular arrays of picture elements (or pixels) known as *bitmaps*.

Just as a view named `Label` displays text, a view named `Image` displays bitmaps. The bitmap formats supported by iOS, Android, and the Windows Runtime are a little different, but if you stick to JPEG, PNG, GIF, and BMP in your Xamarin.Forms applications, you'll probably not experience any problems.

`Image` defines a `Source` property that you set to an object of type `ImageSource`, which references the bitmap displayed by `Image`. Bitmaps can come from a variety of sources, so the `ImageSource` class defines four static creation methods that return an `ImageSource` object:

- `ImageSource.FromUri` for accessing a bitmap over the web.
- `ImageSource.FromResource` for a bitmap stored as an embedded resource in the application PCL.
- `ImageSource.FromFile` for a bitmap stored as content in an individual platform project.
- `ImageSource.FromStream` for loading a bitmap by using a .NET `Stream` object.

`ImageSource` also has three descendant classes, named `UriImageSource`, `FileImageSource`, and `StreamImageSource`, that you can use instead of the first, third, and fourth static creation methods. Generally, the static methods are easier to use in code, but the descendant classes are sometimes required in XAML.

In general, you'll use the `ImageSource.FromUri` and `ImageSource.FromResource` methods to obtain platform-independent bitmaps for presentation purposes and `ImageSource.FromFile` to load platform-specific bitmaps for user-interface objects. Small bitmaps play a crucial role in `MenuItem` and `ToolbarItem` objects, and you can also add a bitmap to a `Button`.

This chapter begins with the use of platform-independent bitmaps obtained from the `ImageSource.FromUri` and `ImageSource.FromResource` methods. It then explores some uses of the `ImageSource.FromStream` method. The chapter concludes with the use of `ImageSource.FromFile` to obtain platform-specific bitmaps for toolbars and buttons.

Platform-independent bitmaps

Here's a code-only program named **WebBitmapCode** with a page class that uses `ImageSource.FromUri` to access a bitmap from the Xamarin website:

```
public class WebBitmapCodePage : ContentPage
{
    public WebBitmapCodePage()
    {
        string uri = "https://developer.xamarin.com/demo/IMG_1415.JPG";

        Content = new Image
        {
            Source = ImageSource.FromUri(new Uri(uri))
        };
    }
}
```

If the URI passed to `ImageSource.FromUri` does not point to a valid bitmap, no exception is raised.

Even this tiny program can be simplified. `ImageSource` defines an implicit conversion from `string` or `Uri` to an `ImageSource` object, so you can set the string with the URI directly to the `Source` property of `Image`:

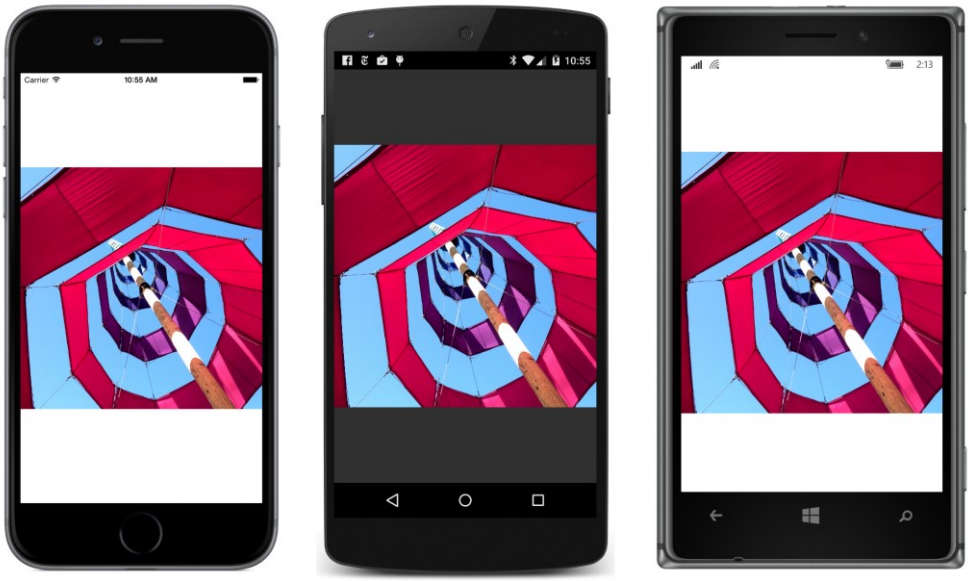
```
public class WebBitmapCodePage : ContentPage
{
    public WebBitmapCodePage()
    {
        Content = new Image
        {
            Source = "https://developer.xamarin.com/demo/IMG_1415.JPG"
        };
    }
}
```

Or, to make it more verbose, you can set the `Source` property of `Image` to a `UriImageSource` object with its `Uri` property set to a `Uri` object:

```
public class WebBitmapCodePage : ContentPage
{
    public WebBitmapCodePage()
    {
        Content = new Image
        {
            Source = new UriImageSource
            {
                Uri = new Uri("https://developer.xamarin.com/demo/IMG_1415.JPG")
            }
        };
    }
}
```

The `UriImageSource` class might be preferred if you want to control the caching of web-based images. The class implements its own caching that uses the application's private storage area available on each platform. `UriImageSource` defines a `CachingEnabled` property that has a default value of `true` and a `CachingValidity` property of type `TimeSpan` that has a default value of one day. This means that if the image is reaccessed within a day, the cached image is used. You can disable caching entirely by setting `CachingEnabled` to `false`, or you can change the caching expiry time by setting the `CachingValidity` property to another `TimeSpan` value.

Regardless which way you do it, by default the bitmap displayed by the `Image` view is stretched to the size of its container—the `ContentPage` in this case—while respecting the bitmap's aspect ratio:

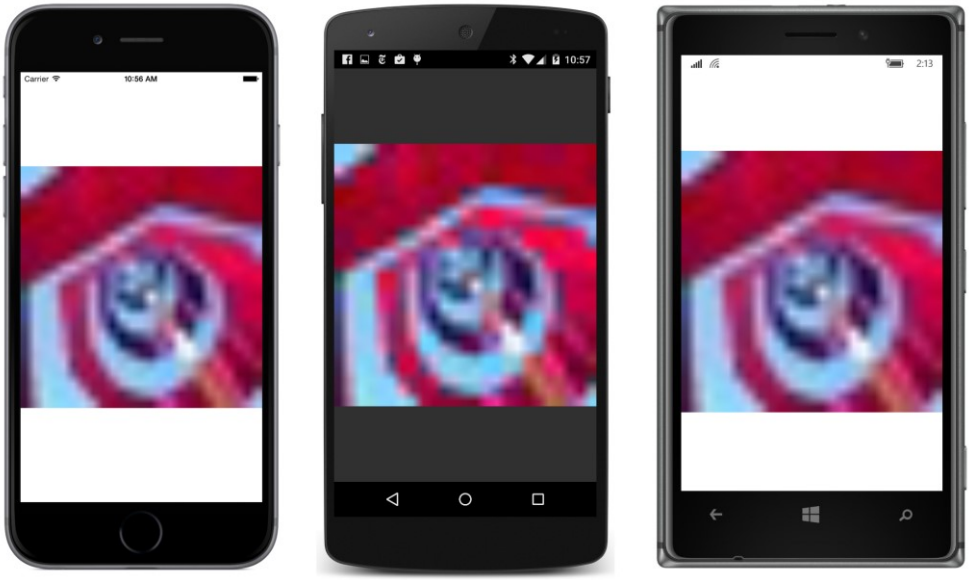


This bitmap is square, so blank areas appear above and below the image. As you turn your phone or emulator between portrait and landscape mode, a rendered bitmap can change size, and you'll see some blank space at the top and bottom or the left and right, where the bitmap doesn't reach. You can color that area by using the `BackgroundColor` property that `Image` inherits from `VisualElement`.

The bitmap referenced in the **WebBitmapCode** program is 4,096 pixels square, but a utility is installed on the Xamarin website that lets you download a much smaller bitmap file by specifying the URI like so:

```
Content = new Image
{
    Source = "https://developer.xamarin.com/demo/IMG_1415.JPG?width=25"
};
```

Now the downloaded bitmap is 25 pixels square, but it is again stretched to the size of its container. Each platform implements an interpolation algorithm in an attempt to smooth the pixels as the image is expanded to fit the page:



However, if you now set `HorizontalOptions` and `VerticalOptions` on the `Image` to `Center`—or put the `Image` element in a `StackLayout`—this 25-pixel bitmap collapses into a very tiny image. This phenomenon is discussed in more detail later in this chapter.

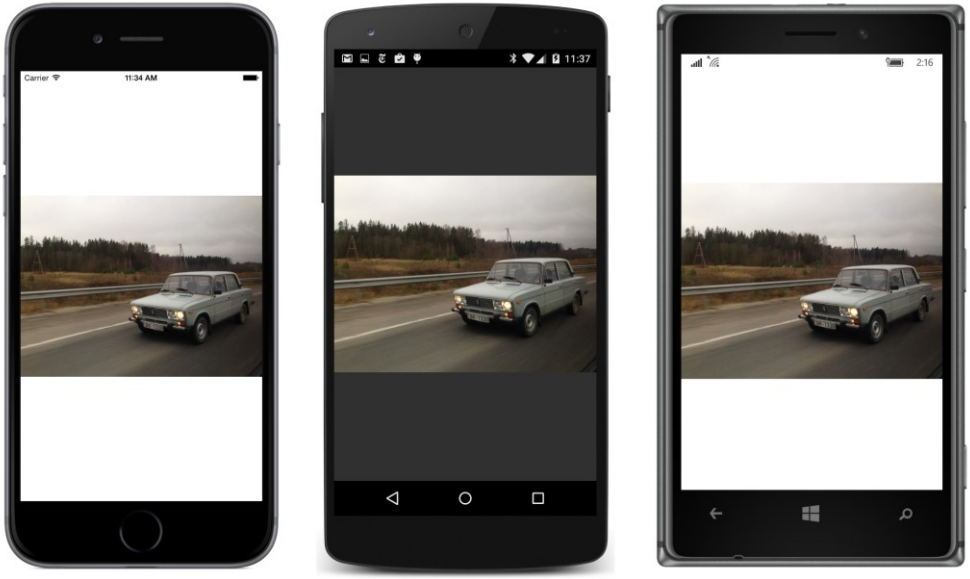
You can also instantiate an `Image` element in XAML and load a bitmap from a URL by setting the `Source` property directly to a web address. Here's the XAML file from the **WebBitmapXaml** program:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="WebBitmapXaml.WebBitmapXamlPage">

    <Image Source="https://developer.xamarin.com/demo/IMG_3256.JPG" />

</ContentPage>
```

A more verbose approach involves explicitly instantiating a `UriImageSource` object and setting the `Uri` property:

```
<Image>
    <Image.Source>
        <UriImageSource Uri="https://developer.xamarin.com/demo/IMG_3256.JPG" />
    </Image.Source>
</Image>
```

Regardless, here's how it looks on the screen:



Fit and fill

If you set the `BackgroundColor` property of `Image` on any of the previous code and XAML examples, you'll see that `Image` actually occupies the entire rectangular area of the page. `Image` defines an `Aspect` property that controls how the bitmap is rendered within this rectangle. You set this property to a member of the `Aspect` enumeration:

- `AspectFit` — the default
- `Fill` — stretches without preserving the aspect ratio
- `AspectFill` — preserves the aspect ratio but crops the image

The default setting is the enumeration member `Aspect.AspectFit`, meaning that the bitmap fits into its container's boundaries while preserving the bitmap's aspect ratio. As you've already seen, the relationship between the bitmap's dimensions and the container's dimensions can result in background areas at the top and bottom or at the right and left.

Try this in the **WebBitmapXaml** project:

```
<Image Source="https://developer.xamarin.com/demo/IMG_3256.JPG"
       Aspect="Fill" />
```

Now the bitmap is expanded to the dimensions of the page. This results in the picture being stretched vertically, so the car appears rather short and stocky:

If you turn the phone sideways, the image is stretched horizontally, but the result isn't quite as extreme because the picture's aspect ratio is somewhat landscape to begin with.

The third option is `AspectFill`:

```
<Image Source="https://developer.xamarin.com/demo/IMG_3256.JPG"
       Aspect="AspectFill" />
```

With this option the bitmap completely fills the container, but the bitmap's aspect ratio is maintained at the same time. The only way this is possible is by cropping part of the image, and you'll see that the image is indeed cropped, but in a different way on the three platforms. On iOS and Android, the image is cropped on either the top and bottom or the left and right, leaving only the central part of the bitmap visible. On the Windows Runtime platforms, the image is cropped on the right or bottom, leaving the upper-left corner visible:

Embedded resources

Accessing bitmaps over the Internet is convenient, but sometimes it's not optimum. The process requires an Internet connection, an assurance that the bitmaps haven't been moved, and some time for downloading. For fast and guaranteed access to bitmaps, they can be bound right into the application.

If you need access to images that are not platform specific, you can include bitmaps as embedded resources in the shared Portable Class Library project and access them with the `ImageSource.FromResource` method. The **ResourceBitmapCode** solution demonstrates how to do it.

The **ResourceBitmapCode** PCL project within this solution has a folder named **Images** that contains two bitmaps, named ModernUserInterface.jpg (a very large bitmap) and ModernUserInterface256.jpg (the same picture but with a 256-pixel width).

When adding any type of embedded resource to a PCL project, make sure to set the **Build Action** of the resource to **EmbeddedResource**. This is crucial.

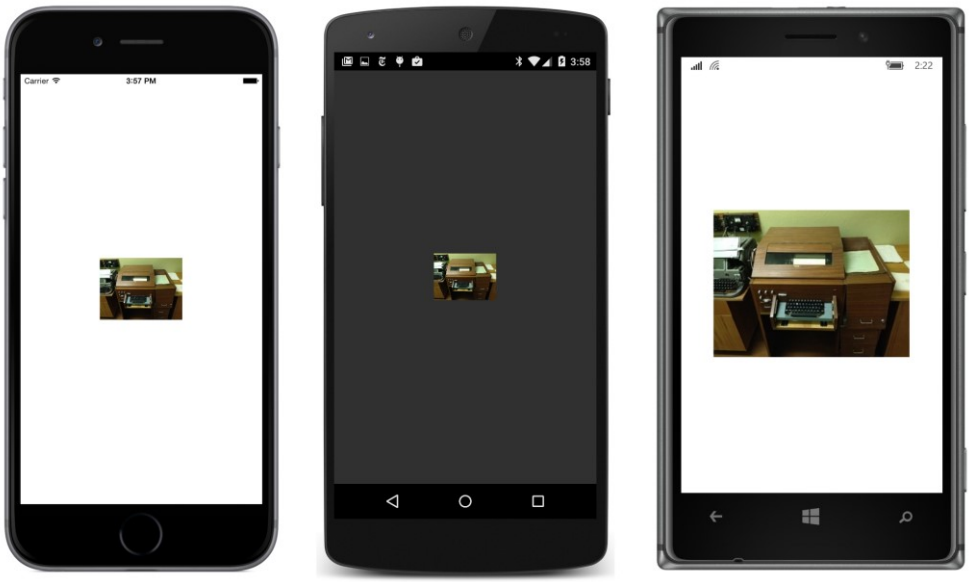
In code, you set the `Source` property of an `Image` element to the `ImageSource` object returned from the static `ImageSource.FromResource` method. This method requires the resource ID. The resource ID consists of the assembly name followed by a period, then the folder name followed by another period, and then the filename, which contains another period for the filename extension. For this example, the resource ID for accessing the smaller of the two bitmaps in the **ResourceBitmapCode** program is:

```
ResourceBitmapCode.Images.ModernUserInterface256.jpg
```

The code in this program references that smaller bitmap and also sets the `HorizontalOptions` and `VerticalOptions` on the `Image` element to `Center`:

```
public class ResourceBitmapCodePage : ContentPage
{
    public ResourceBitmapCodePage()
    {
        Content = new Image
        {
            Source = ImageSource.FromResource(
                        "ResourceBitmapCode.Images.ModernUserInterface256.jpg"),
            VerticalOptions = LayoutOptions.Center,
            HorizontalOptions = LayoutOptions.Center
        };
    }
}
```

As you can see, the bitmap in this instance is *not* stretched to fill the page:



A bitmap is not stretched to fill its container if:

- it is smaller than the container, and
- the `VerticalOptions` and `HorizontalOptions` properties of the `Image` element are not set to `Fill`, or if `Image` is a child of a `StackLayout`.

If you comment out the `VerticalOptions` and `HorizontalOptions` settings, or if you reference the large bitmap (which does not have the "256" at the end of its filename), the image will again stretch to fill the container.

When a bitmap is not stretched to fit its container, it must be displayed in a particular size. What is that size?

On iOS and Android, the bitmap is displayed in its pixel size. In other words, the bitmap is rendered with a one-to-one mapping between the pixels of the bitmap and the pixels of the video display. The iPhone 6 simulator used for these screenshots has a screen width of 750 pixels, and you can see that the 256-pixel width of the bitmap is about one-third that width. The Android phone here is a Nexus 5, which has a pixel width of 1080, and the bitmap is about one-quarter that width.

On the Windows Runtime platforms, however, the bitmap is displayed in device-independent units—in this example, 256 device-independent units. The Nokia Lumia 925 used for these screenshots has a pixel width of 768, which is approximately the same as the iPhone 6. However, the screen width of this Windows 10 Mobile phone in device-independent units is 341, and you can see that the rendered bitmap is much wider than on the other platforms.

This discussion on sizing bitmaps continues in the next section.

How would you reference a bitmap stored as an embedded resource from XAML? Unfortunately, there is no `ResourceImageSource` class. If there were, you would probably try instantiating that class in XAML between `Image.Source` tags. But that's not an option.

You might consider using `x:FactoryMethod` to call `ImageSource.FromResource`, but that won't work. As currently implemented, the `ImageSource.FromResource` method requires that the bitmap resource be in the same assembly as the code that calls the method. When you use `x:FactoryMethod` to call `ImageSource.FromResource`, the call is made from the **Xamarin.Forms.Xaml** assembly.

What *will* work is a very simple XAML markup extension. Here's one in a project named **StackedBitmap**:

```
namespace StackedBitmap
{
    [ContentProperty ("Source")]
    public class ImageResourceExtension : IMarkupExtension
    {
        public string Source { get; set; }

        public object ProvideValue (IServiceProvider serviceProvider)
        {
            if (Source == null)
                return null;

            return ImageSource.FromResource(Source);
        }
    }
}
```

`ImageResourceExtension` has a single property named `Source` that you set to the resource ID. The `ProvideValue` method simply calls `ImageSource.FromResource` with the `Source` property. As is common for single-property markup extensions, `Source` is also the content property of the class. That

means that you don't need to explicitly include "Source=" when you're using the curly-braces syntax for XAML markup extensions.

But watch out: You cannot move this `ImageResourceExtension` class to a library such as **Xamarin.FormsBook.Toolkit**. The class must be part of the same assembly that contains the embedded resources you want to load, which is generally the application's Portable Class Library.

Here's the XAML file from the **StackedBitmap** project. An `Image` element shares a `StackLayout` with two `Label` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:StackedBitmap "
             x:Class="StackedBitmap.StackedBitmapPage">

    <StackLayout>
        <Label Text="320 x 240 Pixel Bitmap"
               FontSize="Medium"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
               BackgroundColor="Aqua"
               SizeChanged="OnImageSizeChanged" />

        <Label x:Name="label"
               FontSize="Medium"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

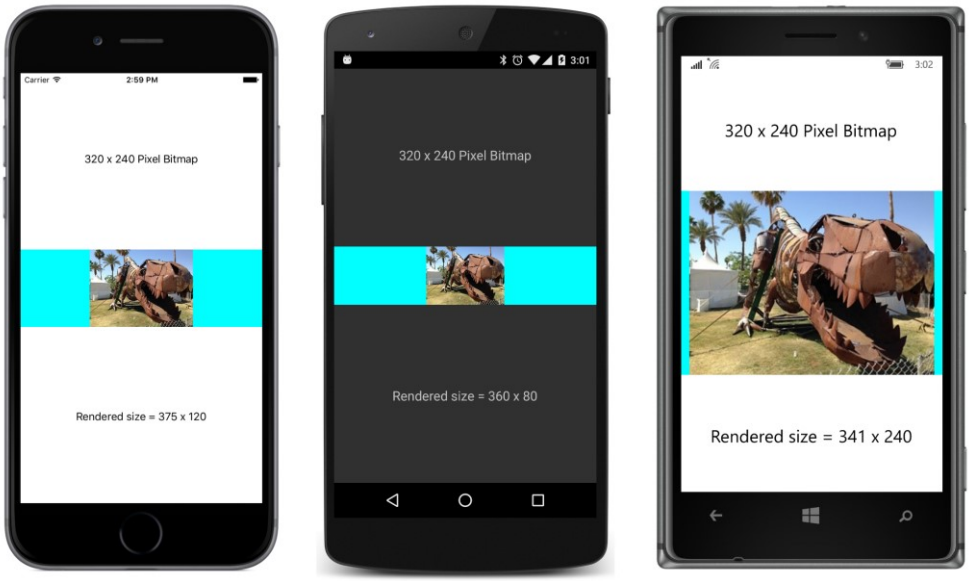
The `local` prefix refers to the `StackedBitmap` namespace in the **StackedBitmap** assembly. The `Source` property of the `Image` element is set to the `ImageResource` markup extension, which references a bitmap stored in the **Images** folder of the PCL project and flagged as an **EmbeddedResource**. The bitmap is 320 pixels wide and 240 pixels high. The `Image` also has its `BackgroundColor` property set; this will allow us to see the entire size of `Image` within the `StackLayout`.

The `Image` element has its `SizeChanged` event set to a handler in the code-behind file:

```
public partial class StackedBitmapPage : ContentPage
{
    public StackedBitmapPage()
    {
        InitializeComponent();
    }

    void OnImageSizeChanged(object sender, EventArgs args)
    {
        Image image = (Image)sender;
        label.Text = String.Format("Render size = {0:F0} x {1:F0}",
                                   image.Width, image.Height);
```

```
    }
}
```

The size of the `Image` element is constrained vertically by the `StackLayout`, so the bitmap is displayed in its pixel size (on iOS and Android) and in device-independent units on Windows Phone. The `Label` displays the size of the `Image` element in device-independent units, which differ on each platform:

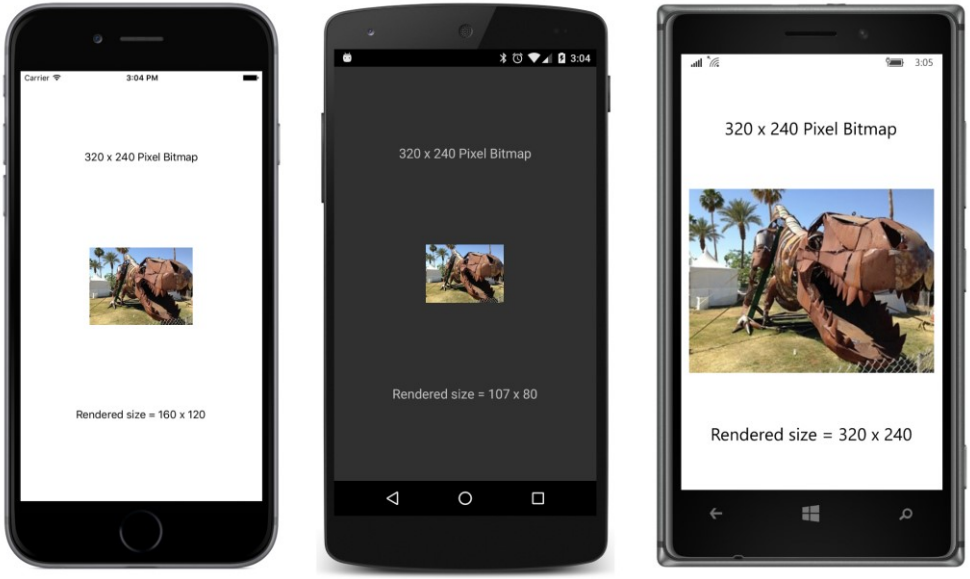


The width of the `Image` element displayed by the bottom `Label` includes the aqua background and equals the width of the page in device-independent units. You can use `Aspect` settings of `Fill` or `AspectFill` to make the bitmap fill that entire aqua area.

If you prefer that the size of the `Image` element be the same size as the rendered bitmap in device-independent units, you can set the `HorizontalOptions` property of the `Image` to something other than the default value of `Fill`:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       HorizontalOptions="Center"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

Now the bottom `Label` displays only the width of the rendered bitmap. Settings of the `Aspect` property have no effect:



Let's refer to this rendered `Image` size as its *natural size* because it is based on the size of the bitmap being displayed.

The iPhone 6 has a pixel width of 750 pixels, but as you discovered when running the **WhatSize** program in Chapter 5, applications perceive a screen width of 375. There are two pixels to the device-independent unit, so a bitmap with a width of 320 pixels is displayed with a width of 160 units.

The Nexus 5 has a pixel width of 1080, but applications perceive a width of 360, so there are three pixels to the device-independent unit, as the `Image` width of 107 units confirms.

On both iOS and Android devices, when a bitmap is displayed in its natural size, there is a one-to-one mapping between the pixels of the bitmap and the pixels of the display. On Windows Runtime devices, however, that's not the case. The Nokia Lumia 925 used for these screenshots has a pixel width of 768. When running the Windows 10 Mobile operating system, there are 2.25 pixels to the device-independent unit, so applications perceive a screen width of 341. But the 320 × 240 pixel bitmap is displayed in a size of 320 × 240 device-independent units.

This inconsistency between the Windows Runtime and the other two platforms is actually beneficial when you're accessing bitmaps from the individual platform projects. As you'll see, iOS and Android include a feature that lets you supply different sizes of bitmaps for different device resolutions. In effect, this allows you to specify bitmap sizes in device-independent units, which means that Windows devices are consistent with those schemes.

But when using platform-independent bitmaps, you'll probably want to size the bitmaps consistently on all three platforms, and that requires a deeper plunge into the subject.

More on sizing

So far, you've seen two ways to size `Image` elements:

If the `Image` element is not constrained in any way, it will fill its container while maintaining the bitmap's aspect ratio, or fill the area entirely if you set the `Aspect` property to `Fill` or `AspectFill`.

If the bitmap is less than the size of its container and the `Image` is constrained horizontally or vertically by setting `HorizontalOptions` or `VerticalOptions` to something other than `Fill`, or if the `Image` is put in a `StackLayout`, the bitmap is displayed in its natural size. That's the pixel size on iOS and Android devices, but the size in device-independent units on Windows devices.

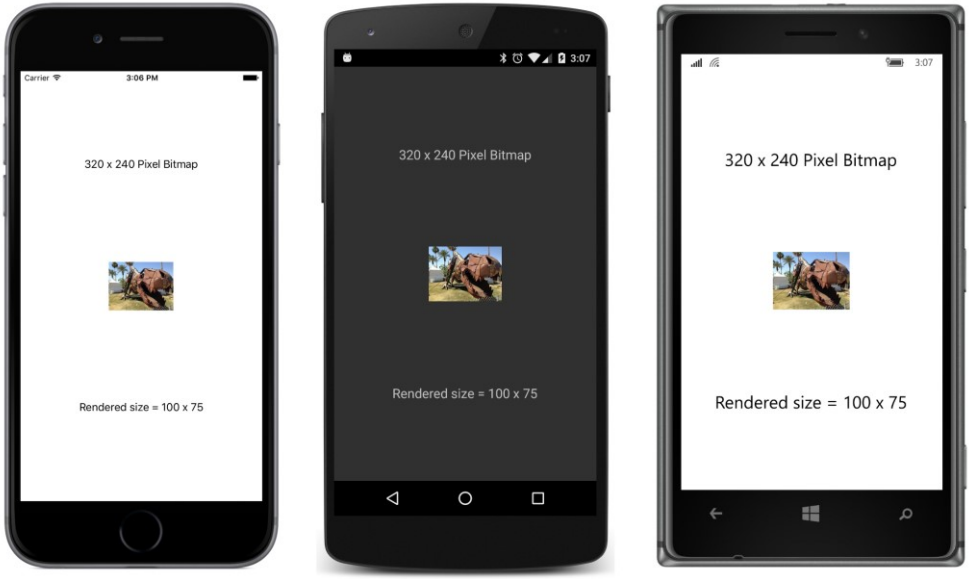
You can also control size by setting `WidthRequest` or `HeightRequest` to an explicit dimension in device-independent units. However, there are some restrictions.

The following discussion is based on experimentation with the **StackedBitmap** sample. It pertains to `Image` elements that are vertically constrained by being a child of a vertical `StackLayout` or having the `VerticalOptions` property set to something other than `Fill`. The same principles apply to an `Image` element that is horizontally constrained.

If an `Image` element is vertically constrained, you can use `WidthRequest` to reduce the size of the bitmap from its natural size, but you cannot use it to increase the size. For example, try setting `WidthRequest` to 100:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       WidthRequest="100"
       HorizontalOptions="Center"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

The resultant height of the bitmap is governed by the specified width and the bitmap's aspect ratio, so now the `Image` is displayed with a size of 100 × 75 device-independent units on all three platforms:

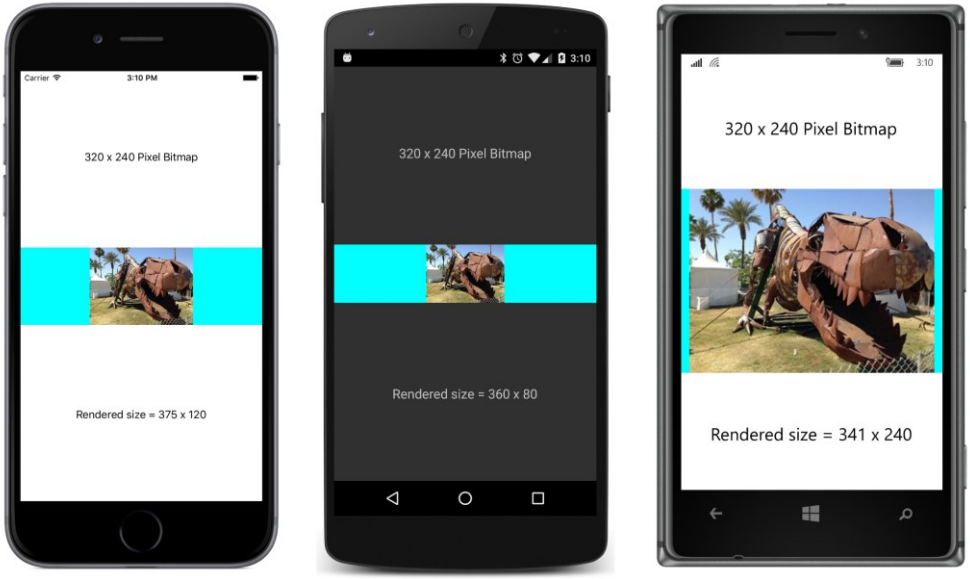


The `HorizontalOptions` setting of `Center` does not affect the size of the rendered bitmap. If you remove that line, the `Image` element will be as wide as the screen (as the aqua background color will demonstrate), but the bitmap will remain the same size.

You cannot use `WidthRequest` to increase the size of the rendered bitmap beyond its natural size. For example, try setting `WidthRequest` to 1000:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       WidthRequest="1000"
       HorizontalOptions="Center"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

Even with `HorizontalOptions` set to `Center`, the resultant `Image` element is now wider than the rendered bitmap, as indicated by the background color:



But the bitmap itself is displayed in its natural size. The vertical `StackLayout` is effectively preventing the height of the rendered bitmap from exceeding its natural height.

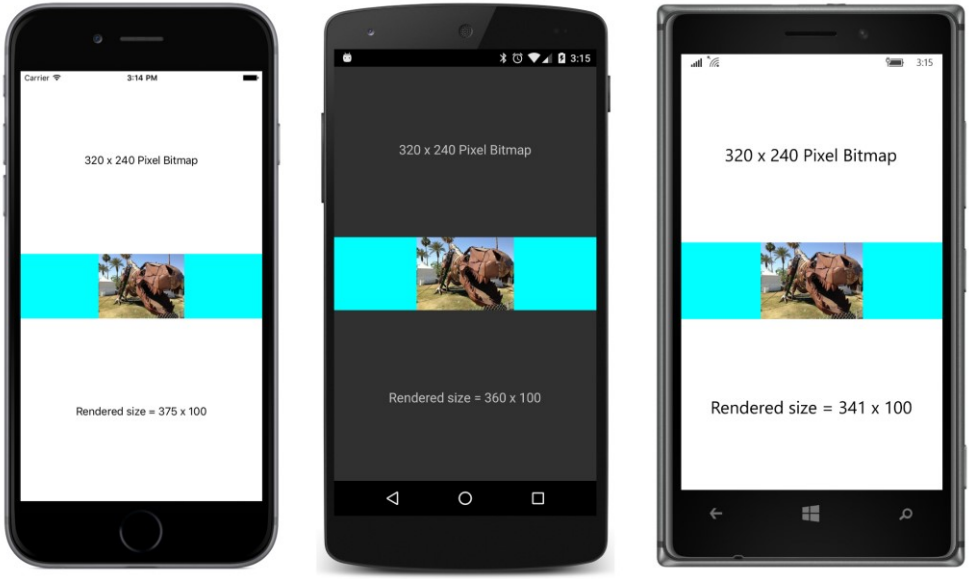
To overcome that constraint of the vertical `StackLayout`, you need to set `HeightRequest`. However, you'll also want to leave `HorizontalOptions` at its default value of `Fill`. Otherwise, the `HorizontalOptions` setting will prevent the width of the rendered bitmap from exceeding its natural size.

Just as with `WidthRequest`, you can set `HeightRequest` to reduce the size of the rendered bitmap. The following code sets `HeightRequest` to 100 device-independent units:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       HeightRequest="100"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

Notice also that the `HorizontalOptions` setting has been removed.

The rendered bitmap is now 100 device-independent units high with a width governed by the aspect ratio. The `Image` element itself stretches to the sides of the `StackLayout`:

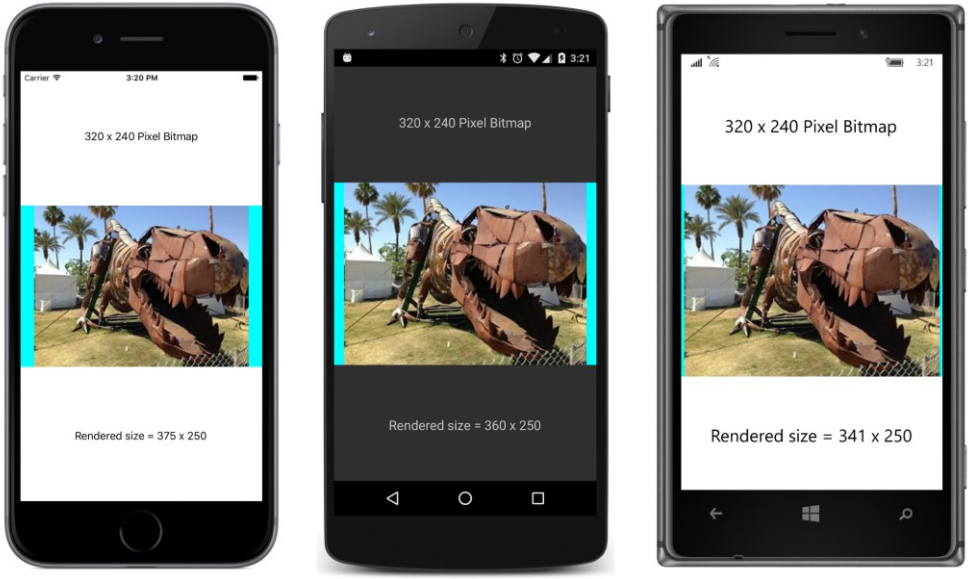


In this particular case, you can set `HorizontalOptions` to `Center` without changing the size of the rendered bitmap. The `Image` element will then be the size of the bitmap (133 × 100), and the aqua background will disappear.

It's important to leave `HorizontalOptions` at its default setting of `Fill` when setting the `HeightRequest` to a value greater than the bitmap's natural height, for example 250:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       HeightRequest="250"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

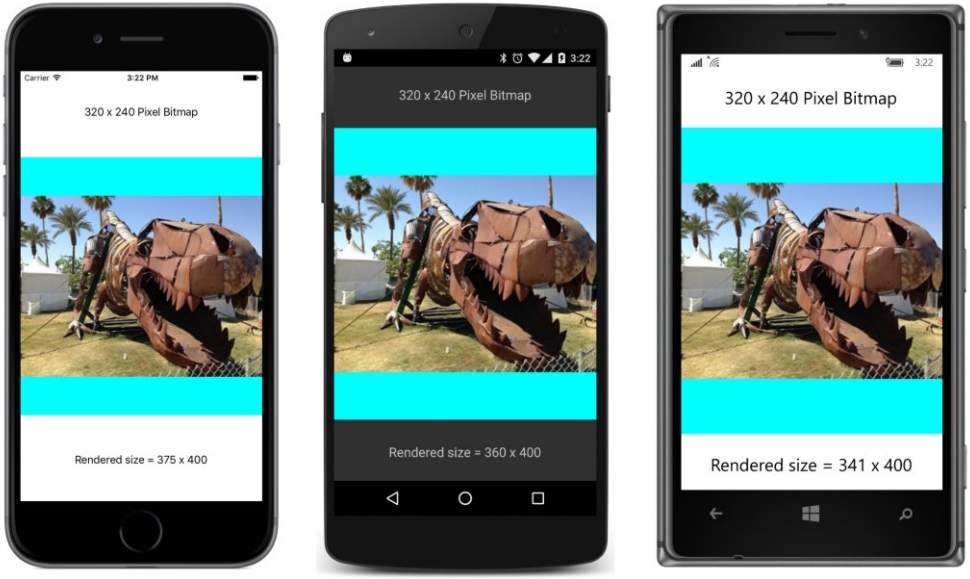
Now the rendered bitmap is larger than its natural size:



However, this technique has a built-in danger, which is revealed when you set the `HeightRequest` to 400:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       HeightRequest="400"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

Here's what happens: The `Image` element does indeed get a height of 400 device-independent units. But the width of the rendered bitmap in that `Image` element is limited by the width of the screen, which means that the height of the rendered bitmap is less than the height of the `Image` element:



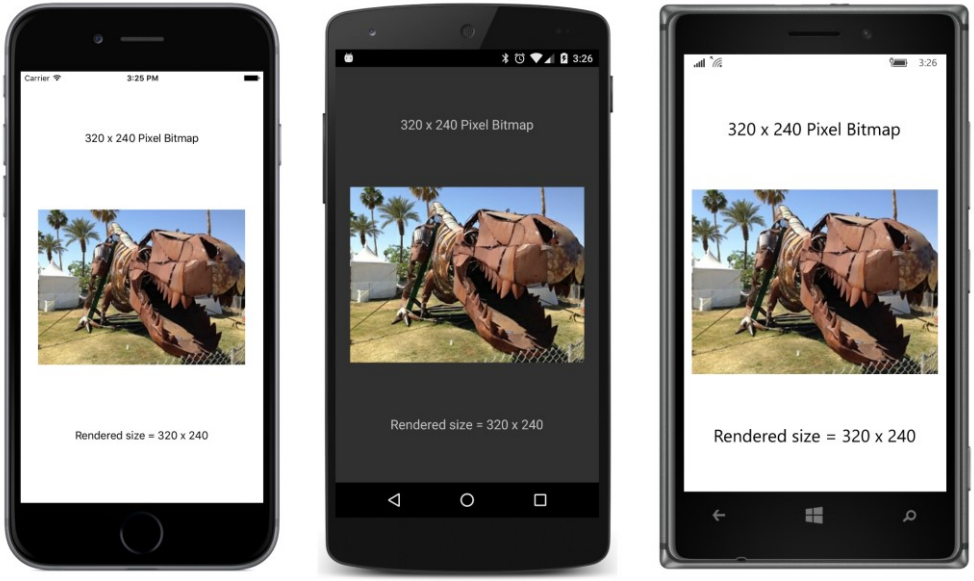
In a real program you probably wouldn't have the `BackgroundColor` property set, and instead a wasteland of blank screen will occupy the area at the top and bottom of the rendered bitmap.

What this implies is that you should not use `HeightRequest` to control the size of bitmaps in a vertical `StackLayout` unless you write code that ensures that `HeightRequest` is limited to the width of the `StackLayout` times the ratio of the bitmap's height to width.

If you know the pixel size of the bitmap that you'll be displaying, one easy approach is to set `WidthRequest` and `HeightRequest` to that size:

```
<Image Source="{local:ImageResource StackedBitmap.Images.Sculpture_320x240.jpg}"
       WidthRequest="320"
       HeightRequest="240"
       HorizontalOptions="Center"
       BackgroundColor="Aqua"
       SizeChanged="OnImageSizeChanged" />
```

Now the bitmap is displayed in that size in device-independent units on all the platforms:



The problem here is that the bitmap is not being displayed at its optimal resolution. Each pixel of the bitmap occupies at least two pixels of the screen, depending on the device.

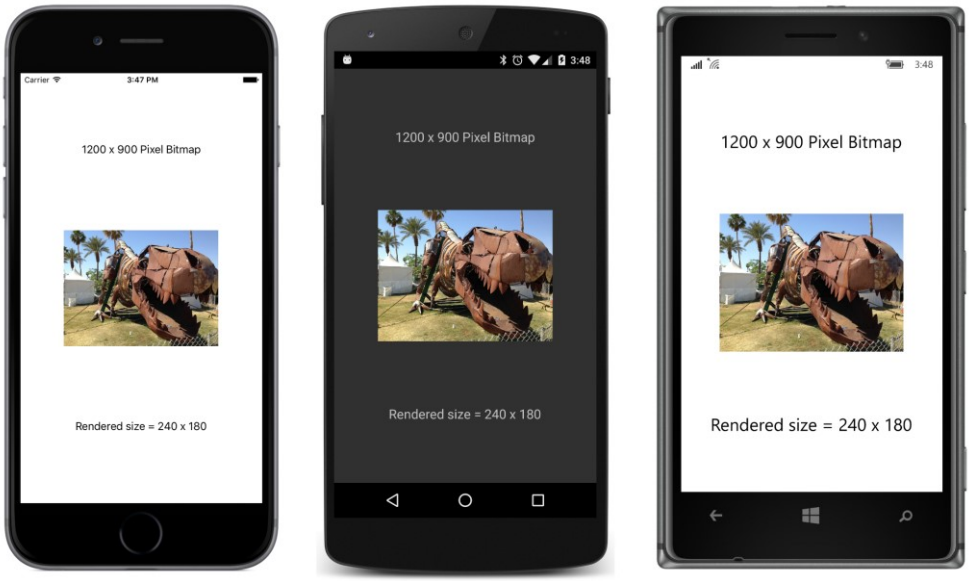
If you want to size bitmaps in a vertical `StackLayout` so that they look approximately the same size on a variety of devices, use `WidthRequest` rather than `HeightRequest`. You've seen that `WidthRequest` in a vertical `StackLayout` can only decrease the size of bitmaps. This means that you should use bitmaps that are larger than the size at which they will be rendered. This will give you a more optimal resolution when the image is sized in device-independent units. You can size the bitmap by using a desired metrical size in inches together with the number of device-independent units to the inch for the particular device, which we found to be 160 for these three devices.

Here's a project very similar to **StackedBitmap** called **DeviceIndBitmapSize**. It's the same bitmap but now 1200 × 900 pixels, which is wider than the portrait-mode width of even high-resolution 1920 × 1080 displays. The platform-specific requested width of the bitmap corresponds to 1.5 inches:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DeviceIndBitmapSize"
             x:Class="DeviceIndBitmapSize.DeviceIndBitmapSizePage">

    <StackLayout>
        <Label Text="1200 x 900 Pixel Bitmap"
               FontSize="Medium"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <!-- 1.5 inch image width -->
        <Image Source="{local:ImageResource DeviceIndBitmapSize.Images.Sculpture_1200x900.jpg}"
               WidthRequest="240"
```

```
                HorizontalOptions="Center"
                SizeChanged="OnImageSizeChanged" />
        </Image>

        <Label x:Name="label"
                FontSize="Medium"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

If the preceding analysis about sizing is correct and all goes well, this bitmap should look approximately the same size on all three platforms relative to the width of the screen, as well as provide higher fidelity resolution than the previous program:



With this knowledge about sizing bitmaps, it is now possible to make a little e-book reader with pictures, because what is the use of a book without pictures?

This e-book reader displays a scrollable `StackLayout` with the complete text of Chapter 7 of Lewis Carroll's *Alice's Adventures in Wonderland,* including three of John Tenniel's original illustrations. The text and illustrations were downloaded from the University of Adelaide's website. The illustrations are included as embedded resources in the **MadTeaParty** project. They have the same names and sizes as those on the website. The names refer to page numbers in the original book:

- image113.jpg — 709 × 553
- image122.jpg — 485 × 545
- image129.jpg — 670 × 596

Recall that the use of `WidthRequest` for `Image` elements in a `StackLayout` can only shrink the size of rendered bitmaps. These bitmaps are not wide enough to ensure that they will all shrink to a proper size on all three platforms, but it's worthwhile examining the results anyway because this is much closer to a real-life example.

The **MadTeaParty** program uses an implicit style for `Image` to set the `WidthRequest` property to a value corresponding to 1.5 inches. Just as in the previous example, this value is 240.

For the three devices used for these screenshots, this width corresponds to:

- 480 pixels on the iPhone 6
- 720 pixels on the Android Nexus 5
- 540 pixels on the Nokia Lumia 925 running Windows 10 Mobile

This means that all three images will shrink in size on the iPhone 6, and they will all have a rendered width of 240 device-independent units.

However, none of the three images will shrink in size on the Nexus 5 because they all have narrower pixel widths than the number of pixels in 1.5 inches. The three images will have a rendered width of (respectively) 236, 162, and 223 device-independent units on the Nexus 5. (That's the pixel width divided by 3.)

On the Windows 10 Mobile device, two will shrink and one will not.

Let's see if the predictions are correct. The XAML file includes a `BackgroundColor` setting on the root element that colors the entire page white, as is appropriate for a book. The `Style` definitions are confined to a `Resources` dictionary in the `StackLayout`. A style for the book title is based on the device `TitleStyle` but with black text and centered, and two implicit styles for `Label` and `Image` serve to style most of the `Label` elements and all three `Image` elements. Only the first and last paragraphs of the chapter's text are shown in this listing of the XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             xmlns:local="clr-namespace:MadTeaParty"
             x:Class="MadTeaParty.MadTeaPartyPage"
             BackgroundColor="White">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="5, 20, 5, 0"
                    Android="5, 0"
                    WinPhone="5, 0" />
    </ContentPage.Padding>

    <ScrollView>
        <StackLayout Spacing="10">
            <StackLayout.Resources>
```

```
            <ResourceDictionary>
                <Style x:Key="titleLabel"
                       TargetType="Label"
                       BaseResourceKey="TitleStyle">
                    <Setter Property="TextColor" Value="Black" />
                    <Setter Property="HorizontalTextAlignment" Value="Center" />
                </Style>

                <!-- Implicit styles -->
                <Style TargetType="Label"
                       BaseResourceKey="BodyStyle">
                    <Setter Property="TextColor" Value="Black" />
                </Style>

                <Style TargetType="Image">
                    <Setter Property="WidthRequest" Value="240" />
                </Style>

                <!-- 1/4 inch indent for poetry -->
                <Thickness x:Key="poemIndent">40, 0, 0, 0</Thickness>
            </ResourceDictionary>
        </StackLayout.Resources>

        <!-- Text and images from http://ebooks.adelaide.edu.au/c/carroll/lewis/alice/ -->
        <StackLayout Spacing="0">
            <Label Text="Alice's Adventures in Wonderland"
                   Style="{DynamicResource titleLabel}"
                   FontAttributes="Italic" />

            <Label Text="by Lewis Carroll"
                   Style="{DynamicResource titleLabel}" />
        </StackLayout>

        <Label Style="{DynamicResource SubtitleStyle}"
               TextColor="Black"
               HorizontalTextAlignment="Center">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Chapter VII" />
                    <Span Text="{x:Static sys:Environment.NewLine}" />
                    <Span Text="A Mad Tea-Party" />
                </FormattedString>
            </Label.FormattedText>
        </Label>

        <Label Text=
"There was a table set out under a tree in front of the
house, and the March Hare and the Hatter were having tea at
it: a Dormouse was sitting between them, fast asleep, and
the other two were using it as a cushion, resting their
elbows on it, and talking over its head. 'Very uncomfortable
for the Dormouse,' thought Alice; 'only, as it's asleep, I
suppose it doesn't mind.'" />
        …
```

```
            …
            …
            <Label>
                <Label.FormattedText>
                    <FormattedString>
                        <Span Text=
"Once more she found herself in the long hall, and close to
the little glass table. 'Now, I'll manage better this time,'
she said to herself, and began by taking the little golden
key, and unlocking the door that led into the garden. Then
she went to work nibbling at the mushroom (she had kept a
piece of it in her pocket) till she was about a foot high:
then she walked down the little passage: and " />
                        <Span Text="then" FontAttributes="Italic" />
                        <Span Text=
" – she found herself at last in the beautiful garden,
among the bright flower-beds and the cool fountains." />
                    </FormattedString>
                </Label.FormattedText>
            </Label>
        </StackLayout>
    </ScrollView>
</ContentPage>
```

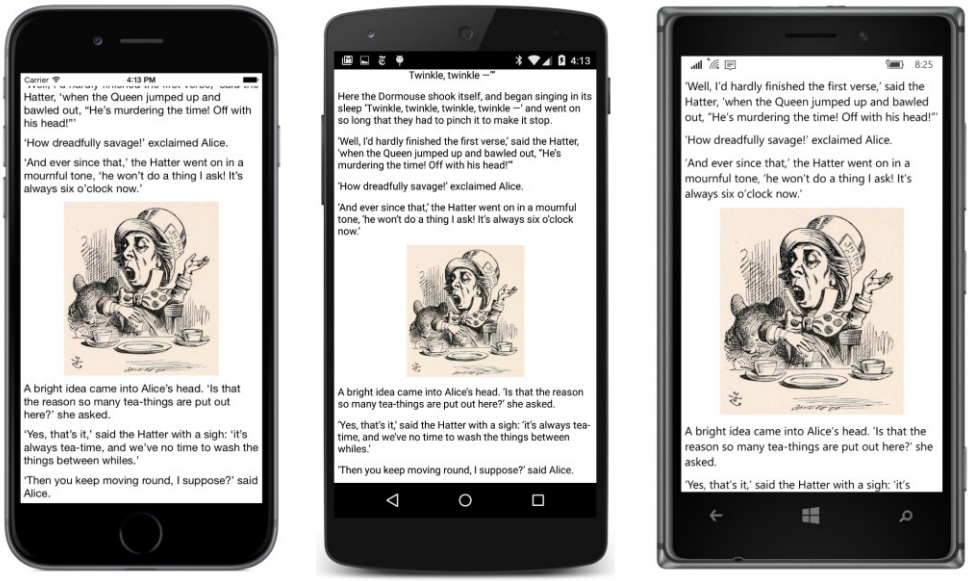
The three `Image` elements simply reference the three embedded resources and are given a setting of the `WidthRequest` property through the implicit style:

```
<Image Source="{local:ImageResource MadTeaParty.Images.image113.jpg}" />
…
<Image Source="{local:ImageResource MadTeaParty.Images.image122.jpg}" />
…
<Image Source="{local:ImageResource MadTeaParty.Images.image129.jpg}" />
```

Here's the first picture:

It's fairly consistent among the three platforms, even though it's displayed in its natural width of 709 pixels on the Nexus 5, but that's very close to the 720 pixels that a width of 240 device-independent units implies.

The difference is much greater with the second image:



This is displayed in its pixel size on the Nexus 5, which corresponds to 162 device-independent units, but is displayed with a width of 240 units on the iPhone 6 and the Nokia Lumia 925.

Although the pictures don't look bad on any of the platforms, getting them all about the same size would require starting out with larger bitmaps.

Browsing and waiting

Another feature of `Image` is demonstrated in the **ImageBrowser** program, which lets you browse the stock photos used for some of the samples in this book. As you can see in the following XAML file, an `Image` element shares the screen with a `Label` and two `Button` views. Notice that a `PropertyChanged` handler is set on the `Image`. You learned in Chapter 11, "The bindable infrastructure," that the `PropertyChanged` handler is implemented by `BindableObject` and is fired whenever a bindable property changes value.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ImageBrowser.ImageBrowserPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Image x:Name="image"
               VerticalOptions="CenterAndExpand"
               PropertyChanged="OnImagePropertyChanged" />

        <Label x:Name="filenameLabel"
               HorizontalOptions="Center" />

        <ActivityIndicator x:Name="activityIndicator" />

        <StackLayout Orientation="Horizontal">
            <Button x:Name="prevButton"
                    Text="Previous"
                    IsEnabled="false"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnPreviousButtonClicked" />

            <Button x:Name="nextButton"
                    Text="Next"
                    IsEnabled="false"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnNextButtonClicked" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

Also on this page is an `ActivityIndicator`. You generally use this element when a program is waiting for a long operation to complete (such as downloading a bitmap) but can't provide any information about the progress of the operation. If your program knows what fraction of the operation has completed, you can use a `ProgressBar` instead. (`ProgressBar` is demonstrated in the next chapter.)

The `ActivityIndicator` has a Boolean property named `IsRunning`. Normally, that property is `false` and the `ActivityIndicator` is invisible. Set the property to `true` to make the `ActivityIndicator` visible. All three platforms implement an animated visual to indicate that the program is working, but it looks a little different on each platform. On iOS it's a spinning wheel, and on Android it's a spinning partial circle. On Windows devices, a series of dots moves across the screen.

To provide browsing access to the stock images, the **ImageBrowser** needs to download a JSON file with a list of all the filenames. Over the years, various versions of .NET have introduced several classes capable of downloading objects over the web. However, not all of these are available in the version of .NET that is available in a Portable Class Library that has the profile compatible with Xamarin.Forms. A class that is available is `WebRequest` and its descendent class `HttpWebRequest`.

The `WebRequest.Create` method returns a `WebRequest` method based on a URI. (The return value is actually an `HttpWebRequest` object.) The `BeginGetResponse` method requires a callback function that is called when the `Stream` referencing the URI is available for access. The `Stream` is accessible from a call to `EndGetResponse` and `GetResponseStream`.

Once the program gets access to the `Stream` object in the following code, it uses the `DataContractJsonSerializer` class together with the embedded `ImageList` class defined near the top of the `ImageBrowserPage` class to convert the JSON file to an `ImageList` object:

```
public partial class ImageBrowserPage : ContentPage
{
    [DataContract]
    class ImageList
    {
        [DataMember(Name = "photos")]
        public List<string> Photos = null;
    }

    WebRequest request;
    ImageList imageList;
    int imageListIndex = 0;

    public ImageBrowserPage()
    {
        InitializeComponent();

        // Get list of stock photos.
        Uri uri = new Uri("https://developer.xamarin.com/demo/stock.json");
        request = WebRequest.Create(uri);
        request.BeginGetResponse(WebRequestCallback, null);
    }

    void WebRequestCallback(IAsyncResult result)
    {
        Device.BeginInvokeOnMainThread(() =>
        {
            try
            {
```

```
            Stream stream = request.EndGetResponse(result).GetResponseStream();

            // Deserialize the JSON into imageList;
            var jsonSerializer = new DataContractJsonSerializer(typeof(ImageList));
            imageList = (ImageList)jsonSerializer.ReadObject(stream);

            if (imageList.Photos.Count > 0)
                FetchPhoto();
        }
        catch (Exception exc)
        {
            filenameLabel.Text = exc.Message;
        }
    });
}

void OnPreviousButtonClicked(object sender, EventArgs args)
{
    imageListIndex--;
    FetchPhoto();
}

void OnNextButtonClicked(object sender, EventArgs args)
{
    imageListIndex++;
    FetchPhoto();
}

void FetchPhoto()
{
    // Prepare for new image.
    image.Source = null;
    string url = imageList.Photos[imageListIndex];

    // Set the filename.
    filenameLabel.Text = url.Substring(url.LastIndexOf('/') + 1);

    // Create the UriImageSource.
    UriImageSource imageSource = new UriImageSource
    {
        Uri = new Uri(url + "?Width=1080"),
        CacheValidity = TimeSpan.FromDays(30)
    };

    // Set the Image source.
    image.Source = imageSource;

    // Enable or disable buttons.
    prevButton.IsEnabled = imageListIndex > 0;
    nextButton.IsEnabled = imageListIndex < imageList.Photos.Count - 1;
}

void OnImagePropertyChanged(object sender, PropertyChangedEventArgs args)
{
```

```
        if (args.PropertyName == "IsLoading")
        {
            activityIndicator.IsRunning = ((Image)sender).IsLoading;
        }
    }
}
```

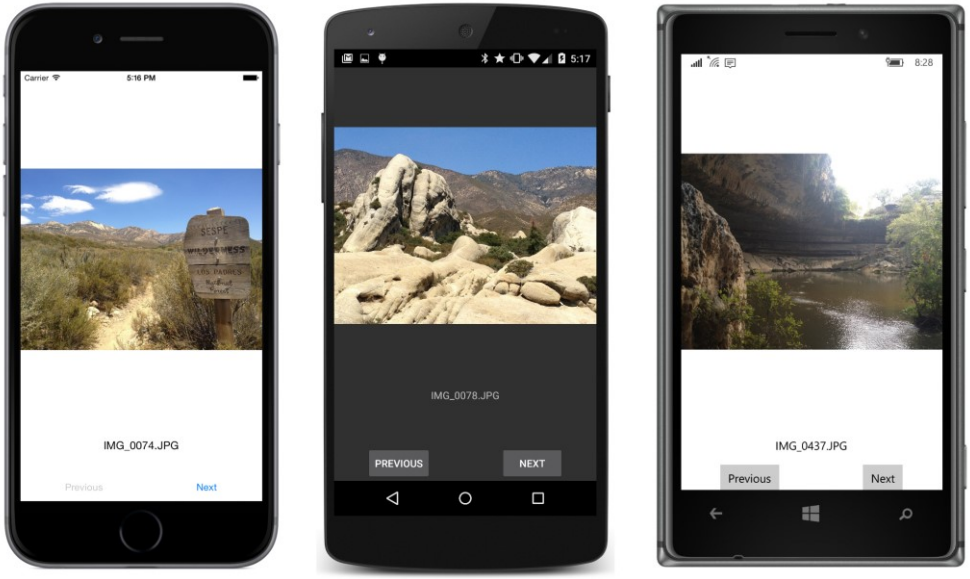
The entire body of the `WebRequestCallback` method is enclosed in a lambda function that is the argument to the `Device.BeginInvokeOnMainThread` method. `WebRequest` downloads the file referenced by the URI in a secondary thread of execution. This ensures that the operation doesn't block the program's main thread, which is handling the user interface. The callback method also executes in this secondary thread. However, user-interface objects in a Xamarin.Forms application can be accessed only from the main thread.

The purpose of the `Device.BeginInvokeOnMainThread` method is to get around this problem. The argument to this method is queued to run in the program's main thread and can safely access user-interface objects.

As you click the two buttons, calls to `FetchPhoto` use `UriImageSource` to download a new bitmap. This might take a second or so. The `Image` class defines a Boolean property named `IsLoading` that is `true` when `Image` is in the process of loading (or downloading) a bitmap. `IsLoading` is backed by the bindable property `IsLoadingProperty`. That also means that whenever `IsLoading` changes value, a `PropertyChanged` event is fired. The program uses the `PropertyChanged` event handler—the `OnImagePropertyChanged` method at the very bottom of the class—to set the `IsRunning` property of the `ActivityIndicator` to the same value as the `IsLoading` property of `Image`.

You'll see in Chapter 16, "Data binding," how your applications can link properties like `IsLoading` and `IsRunning` so that they maintain the same value without any explicit event handlers.

Here's **ImageBrowser** in action:



Some of the images have an EXIF orientation flag set, and if the particular platform ignores that flag, the image is displayed sideways.

If you run this program in landscape mode, you'll discover that the buttons disappear. A better layout option for this program is a `Grid`, which is demonstrated in Chapter 17.

Streaming bitmaps

If the `ImageSource` class didn't have `FromUri` or `FromResource` methods, you would still be able to access bitmaps over the web or stored as resources in the PCL. You can do both of these jobs—as well as several others—with `ImageSource.FromStream` or the `StreamImageSource` class.

The `ImageSource.FromStream` method is somewhat easier to use than `StreamImageSource`, but both are a little odd. The argument to `ImageSource.FromStream` is not a `Stream` object but a `Func` object (a method with no arguments) that returns a `Stream` object. The `Stream` property of `StreamImageSource` is likewise not a `Stream` object but a `Func` object that has a `CancellationToken` argument and returns a `Task<Stream>` object.

Accessing the streams

The **BitmapStreams** program contains a XAML file with two `Image` elements waiting for bitmaps, each of which is set in the code-behind file by using `ImageSource.FromStream`:

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

```
            x:Class="BitmapStreams.BitmapStreamsPage">
    <StackLayout>
        <Image x:Name="image1"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Image x:Name="image2"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

The first `Image` is set from an embedded resource in the PCL; the second is set from a bitmap accessed over the web.

In the **BlackCat** program in Chapter 4, "Scrolling the stack," you saw how to obtain a `Stream` object for any resource stored with a **Build Action** of **EmbeddedResource** in the PCL. You can use this same technique for accessing a bitmap stored as an embedded resource:

```
public partial class BitmapStreamsPage : ContentPage
{
    public BitmapStreamsPage()
    {
        InitializeComponent();

        // Load embedded resource bitmap.
        string resourceID = "BitmapStreams.Images.IMG_0722_512.jpg";
        image1.Source = ImageSource.FromStream(() =>
            {
                Assembly assembly = GetType().GetTypeInfo().Assembly;
                Stream stream = assembly.GetManifestResourceStream(resourceID);
                return stream;
            });
        …
    }
}
```

The argument to `ImageSource.FromStream` is defined as a function that returns a `Stream` object, so that argument is here expressed as a lambda function. The call to the `GetType` method returns the type of the `BitmapStreamsPage` class, and `GetTypeInfo` provides more information about that type, including the `Assembly` object containing the type. That's the **BitmapStream** PCL assembly, which is the assembly with the embedded resource. `GetManifestResourceStream` returns a `Stream` object, which is the return value that `ImageSource.FromStream` wants.

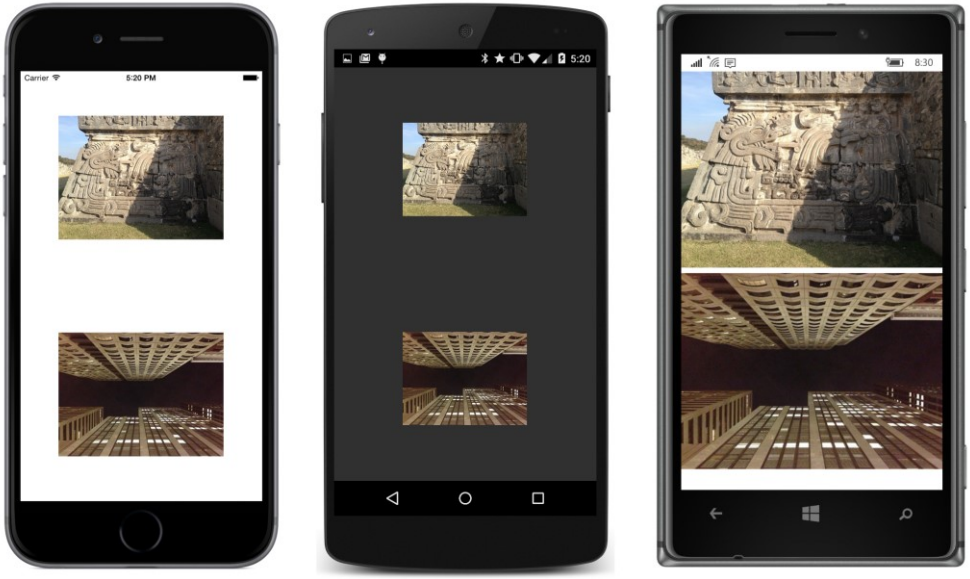
If you ever need a little help with the names of these resources, the `GetManifestResourceNames` returns an array of string objects with all the resource IDs in the PCL. If you can't figure out why your `GetManifestResourceStream` isn't working, first check to make sure your resources have a **Build Action** of **EmbeddedResource**, and then call `GetManifestResourceNames` to get all the resource IDs.

To download a bitmap over the web, you can use the same `WebRequest` method demonstrated earlier in the **ImageBrowser** program. In this program, the `BeginGetResponse` callback is a lambda function:

```
public partial class BitmapStreamsPage : ContentPage
{
    public BitmapStreamsPage()
    {
        …
        // Load web bitmap.
        Uri uri = new Uri("https://developer.xamarin.com/demo/IMG_0925.JPG?width=512");
        WebRequest request = WebRequest.Create (uri);
        request.BeginGetResponse((IAsyncResult arg) =>
            {
                Stream stream = request.EndGetResponse(arg).GetResponseStream();

                if (Device.OS == TargetPlatform.WinPhone ||
                    Device.OS == TargetPlatform.Windows)
                {
                    MemoryStream memStream = new MemoryStream();
                    stream.CopyTo(memStream);
                    memStream.Seek(0, SeekOrigin.Begin);
                    stream = memStream;
                }
                ImageSource imageSource = ImageSource.FromStream(() => stream);
                Device.BeginInvokeOnMainThread(() => image2.Source = imageSource);
            }, null);
    }
}
```

The `BeginGetResponse` callback also contains two more embedded lambda functions! The first line of the callback obtains the `Stream` object for the bitmap. This `Stream` object is not quite suitable for Windows Runtime so the contents are copied to a `MemoryStream`.

The next statement uses a short lambda function as the argument to `ImageSource.FromStream` to define a function that returns that stream. The last line of the `BeginGetResponse` callback is a call to `Device.BeginInvokeOnMainThread` to set the `ImageSource` object to the `Source` property of the `Image`.



It might seem as though you have more control over the downloading of images by using `WebRequest` and `ImageSource.FromStream` than with `ImageSource.FromUri`, but the `ImageSource.FromUri` method has a big advantage: it caches the downloaded bitmaps in a storage area private to the application. As you've seen, you can turn off the caching, but if you're using `ImageSource.FromStream` instead of `ImageSource.FromUri`, you might find the need to cache the images, and that would be a much bigger job.

Generating bitmaps at run time

All three platforms support the BMP file format, which dates back to the very beginning of Microsoft Windows. Despite its ancient heritage, the BMP file format is now fairly standardized with more extensive header information.

Although there are some BMP options that allow some rudimentary compression, most BMP files are uncompressed. This lack of compression is usually regarded as a disadvantage of the BMP file format, but in some cases it's not a disadvantage at all. For example, if you want to generate a bitmap algorithmically at run time, it's *much* easier to generate an uncompressed bitmap instead of one of the compressed file formats. (Indeed, even if you had a library function to create a JPEG or PNG file, you'd apply that function to the uncompressed pixel data.)

You can create a bitmap algorithmically at run time by filling a `MemoryStream` with the BMP file headers and pixel data and then passing that `MemoryStream` to the `ImageSource.FromStream` method. The `BmpMaker` class in the **Xamarin.FormsBook.Toolkit** library demonstrates this. It creates a BMP in memory using a 32-bit pixel format—8 bits each for red, green, blue, and alpha (opacity) chan-

nels. The `BmpMaker` class was coded with performance in mind, in hopes that it might be used for animation. Maybe someday it will be, but in this chapter the only demonstration is a simple color gradient.

The constructor creates a `byte` array named `buffer` that stores the entire BMP file beginning with the header information and followed by the pixel bits. The constructor then uses a `MemoryStream` for writing the header information to the beginning of this buffer:

```
public class BmpMaker
{
    const int headerSize = 54;
    readonly byte[] buffer;

    public BmpMaker(int width, int height)
    {
        Width = width;
        Height = height;

        int numPixels = Width * Height;
        int numPixelBytes = 4 * numPixels;
        int fileSize = headerSize + numPixelBytes;
        buffer = new byte[fileSize];

        // Write headers in MemoryStream and hence the buffer.
        using (MemoryStream memoryStream = new MemoryStream(buffer))
        {
            using (BinaryWriter writer = new BinaryWriter(memoryStream, Encoding.UTF8))
            {
                // Construct BMP header (14 bytes).
                writer.Write(new char[] { 'B', 'M' });  // Signature
                writer.Write(fileSize);                 // File size
                writer.Write((short)0);                 // Reserved
                writer.Write((short)0);                 // Reserved
                writer.Write(headerSize);               // Offset to pixels

                // Construct BitmapInfoHeader (40 bytes).
                writer.Write(40);                       // Header size
                writer.Write(Width);                    // Pixel width
                writer.Write(Height);                   // Pixel height
                writer.Write((short)1);                 // Planes
                writer.Write((short)32);                // Bits per pixel
                writer.Write(0);                        // Compression
                writer.Write(numPixelBytes);            // Image size in bytes
                writer.Write(0);                        // X pixels per meter
                writer.Write(0);                        // Y pixels per meter
                writer.Write(0);                        // Number colors in color table
                writer.Write(0);                        // Important color count
            }
        }
    }

    public int Width
    {
```

```
        private set;
        get;
    }

    public int Height
    {
        private set;
        get;
    }

    public void SetPixel(int row, int col, Color color)
    {
        SetPixel(row, col, (int)(255 * color.R),
                           (int)(255 * color.G),
                           (int)(255 * color.B),
                           (int)(255 * color.A));
    }

    public void SetPixel(int row, int col, int r, int g, int b, int a = 255)
    {
        int index = (row * Width + col) * 4 + headerSize;
        buffer[index + 0] = (byte)b;
        buffer[index + 1] = (byte)g;
        buffer[index + 2] = (byte)r;
        buffer[index + 3] = (byte)a;
    }

    public ImageSource Generate()
    {
        // Create MemoryStream from buffer with bitmap.
        MemoryStream memoryStream = new MemoryStream(buffer);

        // Convert to StreamImageSource.
        ImageSource imageSource = ImageSource.FromStream(() =>
        {
            return memoryStream;
        });
        return imageSource;
    }
}
```

After creating a `BmpMaker` object, a program can then call one of the two `SetPixel` methods to set a color at a particular row and column. When making very many calls, the `SetPixel` call that uses a `Color` value is significantly slower than the one that accepts explicit red, green, and blue values.

The last step is to call the `Generate` method. This method instantiates another `MemoryStream` object based on the `buffer` array and uses it to create a `FileImageSource` object. You can call `Generate` multiple times after setting new pixel data. The method creates a new `MemoryStream` each time because `ImageSource.FromStream` closes the `Stream` object when it's finished with it.

The **DiyGradientBitmap** program—"DIY" stands for "Do It Yourself"—demonstrates how to use

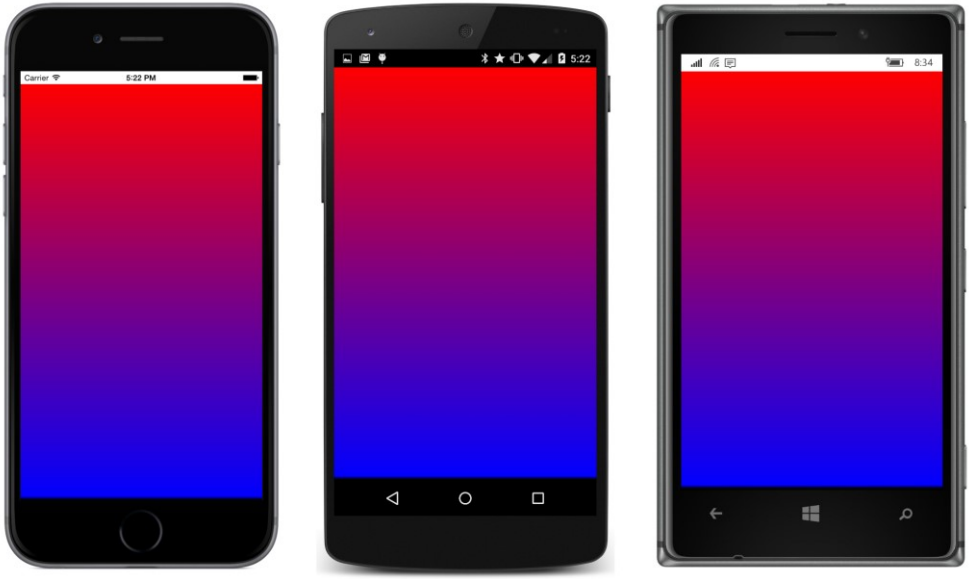
`BmpMaker` to make a bitmap with a simple gradient and display it to fill the page. The XAML file includes the `Image` element:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DiyGradientBitmap.DiyGradientBitmapPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <Image x:Name="image"
           Aspect="Fill" />

</ContentPage>
```

The code-behind file instantiates a `BmpMaker` and loops through the rows and columns of the bitmap to create a gradient that ranges from red at the top to blue at the bottom:

```
public partial class DiyGradientBitmapPage : ContentPage
{
    public DiyGradientBitmapPage()
    {
        InitializeComponent();

        int rows = 128;
        int cols = 64;
        BmpMaker bmpMaker = new BmpMaker(cols, rows);

        for (int row = 0; row < rows; row++)
            for (int col = 0; col < cols; col++)
            {
                bmpMaker.SetPixel(row, col, 2 * row, 0, 2 * (128 - row));
            }

        ImageSource imageSource = bmpMaker.Generate();
        image.Source = imageSource;
    }
}
```

Here's the result:



Now use your imagination and see what you can do with `BmpMaker`.

Platform-specific bitmaps

As you've seen, you can load bitmaps over the web or from the shared PCL project. You can also load bitmaps stored as resources in the individual platform projects. The tools for this job are the `ImageSource.FromFile` static method and the corresponding `FileImageSource` class.

You'll probably use this facility mostly for bitmaps connected with user-interface elements. The `Icon` property in `MenuItem` and `ToolBarItem` is of type `FileImageSource`. The `Image` property in `Button` is also of type `FileImageSource`.

Two other uses of `FileImageSource` won't be discussed in this chapter: the `Page` class defines an `Icon` property of type `FileImageSource` and a `BackgroundImage` property of type `string`, but which is assumed to be the name of a bitmap stored in the platform project.

The storage of bitmaps in the individual platform projects allows a high level of platform specificity. You might think you can get the same degree of platform specificity by storing bitmaps for each platform in the PCL project and using the `Device.OnPlatform` method or the `OnPlatform` class to select them. However, as you'll soon discover, all three platforms have provisions for storing bitmaps of different pixel resolutions and then automatically accessing the optimum one. You can take advantage of this valuable feature only if the individual platforms themselves load the bitmaps, and this is the case only when you use `ImageSource.FromFile` and `FileImageSource`.

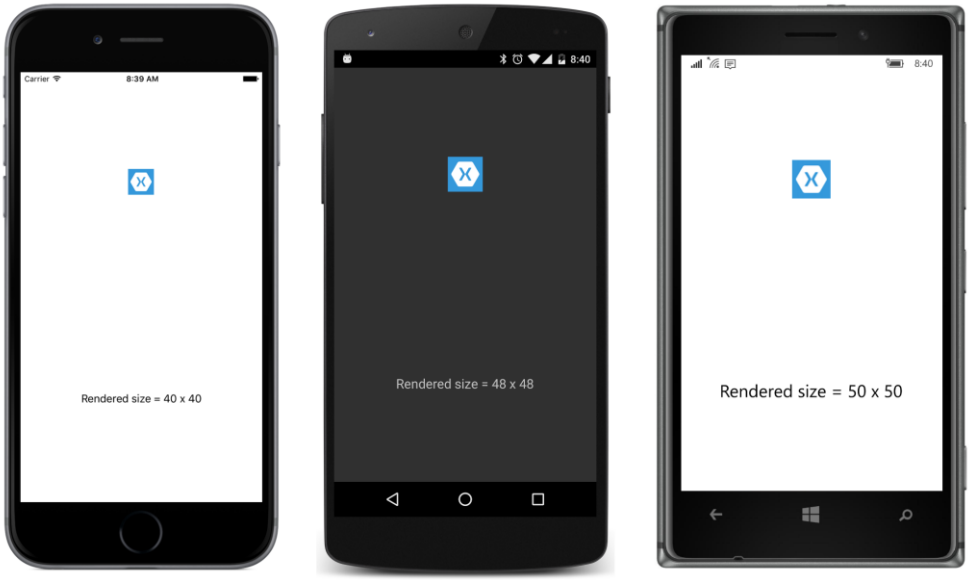
The platform projects in a newly created Xamarin.Forms solution already contain several bitmaps. In the iOS project, you'll find these in the **Resources** folder. In the Android project, they're in subfolders of the **Resources** folder. In the various Windows projects, they're in the **Assets** folder and subfolders. These bitmaps are application icons and splash screens, and you'll want to replace them when you prepare to bring an application to market.

Let's write a small project called **PlatformBitmaps** that accesses an application icon from each platform project and displays the rendered size of the `Image` element. If you're using `FileImageSource` to load the bitmap (as this program does), you need to set the `File` property to a `string` with the bitmap's filename. Almost always, you'll be using `Device.OnPlatform` in code or `OnPlatform` in XAML to specify the three filenames:

```
public class PlatformBitmapsPage : ContentPage
{
    public PlatformBitmapsPage()
    {
        Image image = new Image
        {
            Source = new FileImageSource
            {
                File = Device.OnPlatform(iOS: "Icon-Small-40.png",
                                         Android: "icon.png",
                                         WinPhone: "Assets/StoreLogo.png")
            },
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Label label = new Label
        {
            FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        image.SizeChanged += (sender, args) =>
            {
                label.Text = String.Format("Rendered size = {0} x {1}",
                                           image.Width, image.Height);
            };

        Content = new StackLayout
        {
            Children =
            {
                image,
                label
            }
        };
    }
}
```

When you access a bitmap stored in the **Resources** folder of the iOS project or the **Resources** folder (or subfolders) of the Android project, do not preface the filename with a folder name. These folders are the standard repositories for bitmaps on these platforms. But bitmaps can be anywhere in the Windows or Windows Phone project (including the project root), so the folder name (if any) is required.

In all three cases, the default icon is the famous hexagonal Xamarin logo (fondly known as the Xamagon), but each platform has different conventions for its icon size, so the rendered sizes are different:



If you begin exploring the icon bitmaps in the iOS and Android projects, you might be a little confused: there seem to be multiple bitmaps with the same names (or similar names) in the iOS and Android projects.

It's time to dive deeper into the subject of bitmap resolution.

Bitmap resolutions

The iOS bitmap filename specified in **PlatformBitmaps** is Icon-Small-40.png, but if you look in the **Resources** folder of the iOS project, you'll see three files with variations of that name. They all have different sizes:

- Icon-Small-40.png — 40 pixels square
- Icon-Small-40@2x.png — 80 pixels square
- Icon-Small-40@3x.png — 120 pixels square

As you discovered earlier in this chapter, when an `Image` is a child of a `StackLayout`, iOS displays the bitmap in its pixel size with a one-to-one mapping between the pixels of the bitmap and the pixels of the screen. This is the optimum display of a bitmap.

However, on the iPhone 6 simulator used in the screenshot, the `Image` has a rendered size of 40 device-independent units. On the iPhone 6 there are two pixels per device-independent unit, which means that the actual bitmap being displayed in that screenshot is not Icon-Small-40.png but Icon-Small-40@2x.png, which is two times 40, or 80 pixels square.

If you instead run the program on the iPhone 6 Plus—which has a device-independent unit equal to three pixels—you'll again see a rendered size of 40 pixels, which means that the Icon-Small-40@3x.png bitmap is displayed. Now try it on the iPad 2 simulator. The iPad 2 has a screen size of just 768 × 1024, and device-independent units are the same as pixels. Now the Icon-Small-40.png bitmap is displayed, and the rendered size is still 40 pixels.

This is what you want. You want to be able to control the rendered size of bitmaps in device-independent units because that's how you can achieve perceptibly similar bitmap sizes on different devices and platforms. When you specify the Icon-Small-40.png bitmap, you want that bitmap to be rendered as 40 device-independent units—or about one-quarter inch—on all iOS devices. But if the program is running on an Apple Retina device, you don't want a 40-pixel-square bitmap stretched to be 40 device-independent units. For maximum visual fidelity, you want a higher resolution bitmap displayed, with a one-to-one mapping of bitmap pixels to screen pixels.

If you look in the Android **Resources** directory, you'll find four different versions of a bitmap named icon.png. These are stored in different subfolders of **Resources**:

- drawable/icon.png — 72 pixels square
- drawable-hdpi/icon.png — 72 pixels square
- drawable-xdpi/icon.png — 96 pixels square
- drawable-xxdpi/icon.png — 144 pixels square

Regardless of the Android device, the icon is rendered with a size of 48 device-independent units. On the Nexus 5 used in the screenshot, there are three pixels to the device-independent unit, which means that the bitmap actually displayed on that screen is the one in the **drawable-xxdpi** folder, which is 144 pixels square.

What's nice about both iOS and Android is that you only need to supply bitmaps of various sizes—and give them the correct names or store them in the correct folders—and the operating system chooses the optimum image for the particular resolution of the device.

The Windows Runtime platform has a similar facility. In the **UWP** project you'll see filenames that include scale-200; for example, Square150x150Logo.scale-200.png. The number after the word *scale* is a percentage, and although the filename seems to indicate that this is a 150×150 bitmap, the image is

actually twice as large: 300×300. In the **Windows** project you'll see filenames that include scale-100 and in the **WinPhone** project you'll see scale-240.

However, you've seen that Xamarin.Forms on the Windows Runtime displays bitmaps in their device-independent sizes, and you'll still need to treat the Windows platforms a little differently. But on all three platforms you can control the size of bitmaps in device-independent units.

When creating your own platform-specific images, follow the guidelines in the next three sections.

Device-independent bitmaps for iOS

The iOS naming scheme for bitmaps involves a suffix on the filename. The operating system fetches a particular bitmap with the underlying filename based on the approximate pixel resolution of the device:

- No suffix for 160 DPI devices (1 pixel to the device-independent unit)
- @2x suffix for 320 DPI devices (2 pixels to the DIU)
- @3x suffix: 480 DPI devices (3 pixels to the DIU)

For example, suppose you want a bitmap named MyImage.jpg to show up as about one inch square on the screen. You should supply three versions of this bitmap:

- MyImage.jpg — 160 pixels square
- MyImage@2x.jpg — 320 pixels square
- MyImage@3x.jpg — 480 pixels square

The bitmap will render as 160 device-independent units. For rendered sizes smaller than one inch, decrease the pixels proportionally.

When creating these bitmaps, start with the largest one. Then you can use any bitmap-editing utility to reduce the pixel size. For some images, you might want to fine-tune or completely redraw the smaller versions.

As you might have noticed when examining the various icon files that the Xamarin.Forms template includes with the iOS project, not every bitmap comes in all three resolutions. If iOS can't find a bitmap with the particular suffix it wants, it will fall back and use one of the others, scaling the bitmap up or down in the process.

Device-independent bitmaps for Android

For Android, bitmaps are stored in various subfolders of **Resources** that correspond to a pixel resolution of the screen. Android defines six different directory names for six different levels of device resolution:

- **drawable-ldpi** (low DPI) for 120 DPI devices (0.75 pixels to the DIU)

- **drawable-mdpi** (medium) for 160 DPI devices (1 pixel to the DIU)
- **drawable-hdpi** (high) for 240 DPI devices (1.5 pixels to the DIU))
- **drawable-xhdpi** (extra high) for 320 DPI devices (2 pixels to the DIU)
- **drawable-xxhdpi** (extra extra high) for 480 DPI devices (3 pixels to the DIU)
- **drawable-xxxhdpi** (three extra highs) for 640 DPI devices (4 pixels to the DIU)

If you want a bitmap named MyImage.jpg to render as a one-inch square on the screen, you can supply up to six versions of this bitmap using the same name in all these directories. The size of this one-inch-square bitmap in pixels is equal to the DPI associated with that directory:

- drawable-ldpi/MyImage.jpg — 120 pixels square
- drawable-mdpi/MyImage.jpg — 160 pixels square
- drawable-hdpi/MyImage.jpg — 240 pixels square
- drawable-xhdpi/MyImage.jpg — 320 pixels square
- drawable-xxdpi/MyImage.jpg — 480 pixels square
- drawable-xxxhdpi/MyImage.jpg — 640 pixels square

The bitmap will render as 160 device-independent units.

You are not required to create bitmaps for all six resolutions. The Android project created by the Xamarin.Forms template includes only **drawable-hdpi**, **drawable-xhdpi**, and **drawable-xxdpi**, as well as an unnecessary **drawable** folder with no suffix. These encompass the most common devices. If the Android operating system does not find a bitmap of the desired resolution, it will fall back to a size that is available and scale it.

Device-independent bitmaps for Windows Runtime platforms

The Windows Runtime supports a bitmap naming scheme that lets you embed a scaling factor of pixels per device-independent unit expressed as a percentage. For example, for a one-inch-square bitmap targeted to a device that has two pixels to the unit, use the name:

- MyImage.scale-200.jpg — 320 pixels square

The Windows documentation is unclear about the actual percentages you can use. When building a program, sometimes you'll see error messages in the **Output** window regarding percentages that are not supported on the particular platform.

However, given that Xamarin.Forms displays Windows Runtime bitmaps in their device-independent sizes, this facility is of limited use on these devices.

Let's look at a program that actually does supply custom bitmaps of various sizes for the three platforms. These bitmaps are intended to be rendered about one inch square, which is approximately half the width of the phone's screen in portrait mode.

This **ImageTap** program creates a pair of rudimentary, tappable button-like objects that display not text but a bitmap. The two buttons that **ImageTap** creates might substitute for traditional **OK** and **Cancel** buttons, but perhaps you want to use faces from famous paintings for the buttons. Perhaps you want the **OK** button to display the face of Botticelli's Venus and the **Cancel** button to display the distressed man in Edvard Munch's *The Scream*.

In the sample code for this chapter is a directory named **Images** that contains such images, named Venus_xxx.jpg and Scream_xxx.jpg, where the xxx indicates the pixel size. Each image is in eight different sizes: 60, 80, 120, 160, 240, 320, 480, and 640 pixels square. In addition, some of the files have names of Venus_xxx_id.jpg and Scream_xxx_id.jpg. These versions have the actual pixel size displayed in the lower-right corner of the image so that we can see on the screen exactly what bitmap the operating system has selected.

To avoid confusion, the bitmaps with the original names were added to the **ImageTap** project folders first, and then they were renamed within Visual Studio.

In the **Resources** folder of the iOS project, the following files were renamed:

- Venus_160_id.jpg became Venus.jpg
- Venus_320_id.jpg because Venus@2x.jpg
- Venus_480_id.jpg became Venus@3x.jpg

This was done similarly for the Scream.jpg bitmaps.

In the various subfolders of the Android project **Resources** folder, the following files were renamed:

- Venus_160_id.jpg became drawable-mdpi/Venus.jpg
- Venus_240_id.jpg became drawable-hdpi/Venus.jpg
- Venus_320_id.jpg became drawable-xhdpi/Venus.jpg
- Venus_480_id.jpg became drawable_xxhdpi/Venus.jpg

And similarly for the Scream.jpg bitmaps.

For the Windows Phone 8.1 project, the Venus_160_id.jpg and Scream_160_id.jpg files were copied to an **Images** folder and renamed Venus.jpg and Scream.jpg.

The Windows 8.1 project creates an executable that runs not on phones but on tablets and desktops. These devices have traditionally assumed a resolution of 96 units to the inch, so the Venus_100_id.jpg and Scream_100_id.jpg files were copied to an **Images** folder and renamed Venus.jpg and Scream.jpg.

The UWP project targets all the form factors, so several bitmaps were copied to an **Images** folder and renamed so that the 160-pixel square bitmaps would be used on phones, and the 100-pixel square bitmaps would be used on tablets and desktop screens:

- Venus_160_id.jpg became Venus.scale-200.jpg
- Venus_100_id.jpg became Venus.scale-100.jpg

And similarly for the Scream.jpg bitmaps.

Each of the projects requires a different **Build Action** for these bitmaps. This should be set automatically when you add the files to the projects, but you definitely want to double-check to make sure the **Build Action** is set correctly:

- iOS: **BundleResource**
- Android: **AndroidResource**
- Windows Runtime: **Content**

You don't have to memorize these. When in doubt, just check the **Build Action** for the bitmaps included by the Xamarin.Forms solution template in the platform projects.

The XAML file for the **ImageTap** program puts each of the two `Image` elements on a `ContentView` that is colored white from an implicit style. This white `ContentView` is entirely covered by the `Image`, but (as you'll see) it comes into play when the program flashes the picture to signal that it's been tapped.

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ImageTap.ImageTapPage">

    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="ContentView">
                    <Setter Property="BackgroundColor" Value="White" />
                    <Setter Property="HorizontalOptions" Value="Center" />
                    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>

        <ContentView>
            <Image>
                <Image.Source>
                    <OnPlatform x:TypeArguments="ImageSource"
                                iOS="Venus.jpg"
                                Android="Venus.jpg"
                                WinPhone="Images/Venus.jpg" />
                </Image.Source>
```

```
            <Image.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnImageTapped" />
            </Image.GestureRecognizers>
        </Image>
    </ContentView>

    <ContentView>
        <Image>
            <Image.Source>
                <OnPlatform x:TypeArguments="ImageSource"
                            iOS="Scream.jpg"
                            Android="Scream.jpg"
                            WinPhone="Images/Scream.jpg" />
            </Image.Source>

            <Image.GestureRecognizers>
                <TapGestureRecognizer Tapped="OnImageTapped" />
            </Image.GestureRecognizers>
        </Image>
    </ContentView>

    <Label x:Name="label"
           FontSize="Medium"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />

</StackLayout>
</ContentPage>
```

The XAML file uses `OnPlatform` to select the filenames of the platform resources. Notice that the `x:TypeArguments` attribute of `OnPlatform` is set to `ImageSource` because this type must exactly match the type of the target property, which is the `Source` property of `Image`. `ImageSource` defines an implicit conversion of `string` to itself, so specifying the filenames is sufficient. (The logic for this implicit conversion checks first whether the string has a URI prefix. If not, it assumes that the string is the name of an embedded file in the platform project.)

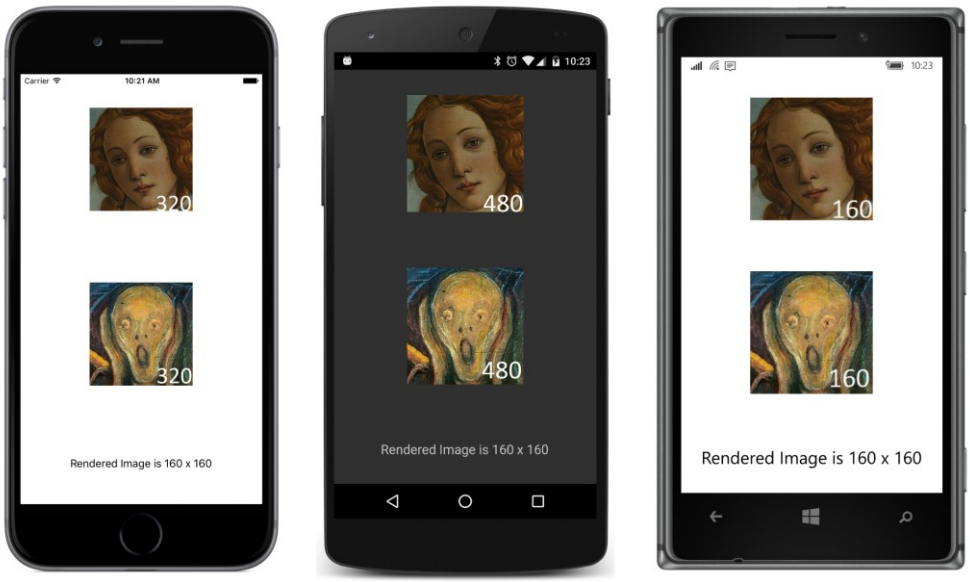
If you want to avoid using `OnPlatform` entirely in programs that use platform bitmaps, you can put the Windows bitmaps in the root directory of the project rather than in a folder.

Tapping one of these buttons does two things: The `Tapped` handler sets the `Opacity` property of the `Image` to 0.75, which results in partially revealing the white `ContentView` background and simulating a flash. A timer restores the `Opacity` to the default value of one-tenth of a second later. The `Tapped` handler also displays the rendered size of the `Image` element:

```
public partial class ImageTapPage : ContentPage
{
    public ImageTapPage()
    {
        InitializeComponent();
    }

    void OnImageTapped(object sender, EventArgs args)
```

```
    {
        Image image = (Image)sender;
        image.Opacity = 0.75;

        Device.StartTimer(TimeSpan.FromMilliseconds(100), () =>
            {
                image.Opacity = 1;
                return false;
            });

        label.Text = String.Format("Rendered Image is {0} x {1}",
                                   image.Width, image.Height);
    }
}
```

That rendered size compared with the pixel sizes on the bitmaps confirms that the three platforms have indeed selected the optimum bitmap:



These buttons occupy roughly half the width of the screen on all three platforms. This sizing is based entirely on the size of the bitmaps themselves, without any additional sizing information in the code or markup.

Toolbars and their icons

One of the primary uses of bitmaps in the user interface is the Xamarin.Forms toolbar, which appears at the top of the page on iOS and Android devices and at the bottom of the page on Windows Phone devices. Toolbar items are tappable and fire `Clicked` events much like `Button`.

There is no class for toolbar itself. Instead, you add objects of type `ToolbarItem` to the `ToolbarItems` collection property defined by `Page`.

The `ToolbarItem` class does not derive from `View` like `Label` and `Button`. It instead derives from `Element` by way of `MenuItemBase` and `MenuItem`. (`MenuItem` is used only in connection with the `TableView` and won't be discussed until Chapter 19.) To define the characteristics of a toolbar item, use the following properties:

- `Text` — the text that might appear (depending on the platform and `Order`)
- `Icon` — a `FileImageSource` object referencing a bitmap from the platform project
- `Order` — a member of the `ToolbarItemOrder` enumeration: `Default`, `Primary`, or `Secondary`

There is also a `Name` property, but it just duplicates the `Text` property and should be considered obsolete.

The `Order` property governs whether the `ToolbarItem` appears as an image (`Primary`) or text (`Secondary`). The Windows Phone and Windows 10 Mobile platforms are limited to four `Primary` items, and both the iPhone and Android devices start getting crowded with more than that, so that's a reasonable limitation. Additional `Secondary` items are text only. On the iPhone they appear underneath the `Primary` items; on Android and Windows Phone they aren't seen on the screen until the user taps a vertical or horizontal ellipsis.

The `Icon` property is crucial for `Primary` items, and the `Text` property is crucial for `Secondary` items, but the Windows Runtime also uses `Text` to display a short text hint underneath the icons for `Primary` items.

When the `ToolbarItem` is tapped, it fires a `Clicked` event. `ToolbarItem` also has `Command` and `CommandParameter` properties like the `Button`, but these are for data-binding purposes and will be demonstrated in a later chapter.

The `ToolbarItems` collection defined by `Page` is of type `IList<ToolbarItem>`. Once you add a `ToolbarItem` to this collection, the `ToolbarItem` properties cannot be changed. The property settings are instead used internally to construct platform-specific objects.

You can add `ToolbarItem` objects to a `ContentPage` in Windows Phone, but iOS and Android restrict toolbars to a `NavigationPage` or to a page navigated to from a `NavigationPage`. Fortunately, this requirement doesn't mean that the whole topic of page navigation needs to be discussed before you can use the toolbar. Instantiating a `NavigationPage` instead of a `ContentPage` simply involves calling the `NavigationPage` constructor with the newly created `ContentPage` object in the `App` class.

The **ToolbarDemo** program reproduces the toolbar that you saw on the screenshots in Chapter 1. The `ToolbarDemoPage` derives from `ContentPage`, but the `App` class passes the `ToolbarDemoPage` object to a `NavigationPage` constructor:

```
public class App : Application
{
    public App()
    {
        MainPage = new NavigationPage(new ToolbarDemoPage());
    }
    …
}
```

That's all that's necessary to get the toolbar to work on iOS and Android, and it has some other implications as well. A title that you can set with the `Title` property of `Page` is displayed at the top of the iOS and Android screens, and the application icon is also displayed on the Android screen. Another result of using `NavigationPage` is that you no longer need to set some padding at the top of the iOS screen. The status bar is now out of the range of the application's page.

Perhaps the most difficult aspect of using `ToolbarItem` is assembling the bitmap images for the `Icon` property. Each platform has different requirements for the color composition and size of these icons, and each platform has somewhat different conventions for the imagery. The standard icon for **Share**, for example, is different on all three platforms.

For these reasons, it makes sense for each of the platform projects to have its own collection of toolbar icons, and that's why `Icon` is of type `FileImageSource`.

Let's begin with the two platforms that provide collections of icons suitable for `ToolbarItem`.

Icons for Android

The Android website has a downloadable collection of toolbar icons at this URL:

http://developer.android.com/design/downloads

Download the ZIP file identified as **Action Bar Icon Pack**.

The unzipped contents are organized into two main directories: **Core_Icons** (23 images) and **Action Bar Icons** (144 images). These are all PNG files, and the **Action Bar Icons** come in four different sizes, indicated by the directory name:

- **drawable-mdpi** (medium DPI) — 32 pixels square
- **drawable-hdpi** (high DPI) — 48 pixels square
- **drawable-xhdpi** (extra high DPI) — 64 pixels square
- **drawable-xxhdpi** (extra extra high DPI) — 96 pixels square

These directory names are the same as the **Resources** folders in your Android project and imply that the toolbar icons render at 32 device-independent units, or about one-fifth of an inch.

The **Core_Icons** folder also arranges its icons into four directories with the same four sizes, but these directories are named **mdpi**, **hdpi**, **xdpi**, and **unscaled**.

The **Action Bar Icons** folder has an additional directory organization using the names **holo_dark** and **holo_light**:

- **holo_dark**—white foreground image on a transparent background
- **holo_light**—black foreground image on a transparent background

The word "holo" stands for "holographic" and refers to the name Android uses for its color themes. Although the **holo_light** icons are much easier to see in **Finder** and **Windows Explorer**, for most purposes (and especially for toolbar items) you should use the **holo_dark** icons. (Of course, if you know how to change your application theme in the AndroidManifest.xml file, then you probably also know to use the other icon collection.)

The **Core_Icons** folder contains only icons with white foregrounds on a transparent background.

For the **ToolbarDemo** program, three icons were chosen from the **holo_dark** directory in all four resolutions. These were copied to the appropriate subfolders of the **Resources** directory in the Android project:

- From the **01_core_edit** directory, the files named ic_action_edit.png
- From the **01_core_search** directory, the files named ic_action_search.png
- From the **01_core_refresh** directory, the files named ic_action_refresh.png

Check the properties of these PNG files. They must have a **Build Action** of **AndroidResource**.

Icons for Windows Runtime platforms

If you have a version of Visual Studio installed for Windows Phone 8, you can find a collection of PNG files suitable for `ToolbarItem` in the following directory on your hard drive:

C:\Program Files (x86)\Microsoft SDKs\Windows Phone\v8.0\Icons

You can use these for all the Windows Runtime platforms.

There are two subdirectories, **Dark** and **Light**, each containing the same 37 images. As with Android, the icons in the **Dark** directory have white foregrounds on transparent backgrounds, and the icons in the **Light** directory have black foregrounds on transparent backgrounds. You should use the ones in the **Dark** directory for Windows Phone 8.1 and the **Light** directory for Windows 10 Mobile.

The images are a uniform 76 pixels square but have been designed to appear inside a circle. Indeed, one of the files is named basecircle.png, which can serve as a guide if you'd like to design your own, so there are really only 36 usable icons in the collection and a couple of them are the same.

Generally, in a Windows Runtime project, files such as these are stored in the **Assets** folder (which already exists in the project) or a folder named **Images**. The following bitmaps were added to an **Images** folder in all three Windows platforms:

- edit.png
- feature.search.png
- refresh.png

For the Windows 8.1 platform (but not the Windows Phone 8.1 platform), icons are needed for all the toolbar items, so the following bitmaps were added to the **Images** folder of that project:

- Icon1F435.png
- Icon1F440.png
- Icon1F52D.png

These were generated in a Windows program from the Segoe UI Symbol font, which supports emoji characters. The five-digit hexadecimal number in the filename is the Unicode ID for those characters.

When you add icons to a Windows Runtime project, make sure the **Build Action** is **Content**.

Icons for iOS devices

This is the most problematic platform for `ToolbarItem`. If you're programming directly for the native iOS API, a bunch of constants let you select an image for `UIBarButtonItem`, which is the underlying iOS implementation of `ToolbarItem`. But for the Xamarin.Forms `ToolbarItem`, you'll need to obtain icons from another source—perhaps licensing a collection such as the one at glyphish.com—or make your own.

For best results, you should supply two or three image files for each toolbar item in the **Resources** folder. An image with a filename such as image.png should be 20 pixels square, while the same image should also be supplied in a 40-pixel-square dimension with the name image@2x.png and as a 60-pixel-square bitmap named image@3x.png.

Here's a collection of free, unrestricted-use icons used for the program in Chapter 1 and for the **ToolbarDemo** program in this chapter:

http://www.smashingmagazine.com/2010/07/14/gcons-free-all-purpose-icons-for-designers-and-developers-100-icons-psd/

However, they are uniformly 32 pixels square, and some basic ones are missing. Regardless, the following three bitmaps were copied to the **Resources** folder in the iOS project under the assumption that they will be properly scaled:

- edit.png
- search.png
- reload.png

Another option is to use Android icons from the **holo_light** directory and scale the largest image for the various iOS sizes.

For toolbar icons in an iOS project, the **Build Action** must be **BundleResource**.

Here's the **ToolbarDemo** XAML file showing the various `ToolbarItem` objects added to the `ToolbarItems` collection of the page. The `x:TypeArguments` attribute for `OnPlatform` must be `FileImageSource` in this case because that's the type of the `Icon` property of `ToolbarItem`. The three items flagged as `Secondary` have only the `Text` property set and not the `Icon` property.

The root element has a `Title` property set on the page. This is displayed on the iOS and Android screens when the page is instantiated as a `NavigationPage` (or navigated to from a `NavigationPage`):

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ToolbarDemo.ToolbarDemoPage"
             Title="Toolbar Demo">

    <Label x:Name="label"
           FontSize="Medium"
           HorizontalOptions="Center"
           VerticalOptions="Center" />

    <ContentPage.ToolbarItems>
        <ToolbarItem Text="edit"
                     Order="Primary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="edit.png"
                            Android="ic_action_edit.png"
                            WinPhone="Images/edit.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="search"
                     Order="Primary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="search.png"
                            Android="ic_action_search.png"
                            WinPhone="Images/feature.search.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="refresh"
                     Order="Primary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
```

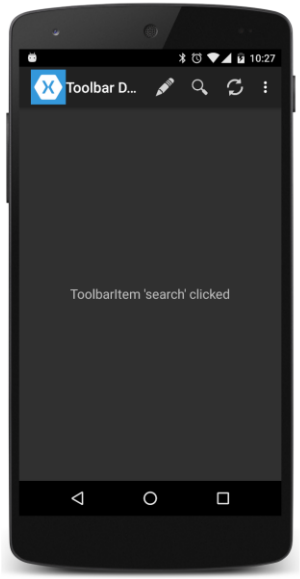
```
                        iOS="reload.png"
                        Android="ic_action_refresh.png"
                        WinPhone="Images/refresh.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="explore"
                     Order="Secondary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                        WinPhone="Images/Icon1F52D.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="discover"
                     Order="Secondary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                        WinPhone="Images/Icon1F440.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Text="evolve"
                     Order="Secondary"
                     Clicked="OnToolbarItemClicked">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                        WinPhone="Images/Icon1F435.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
</ContentPage>
```

Although the `OnPlatform` element implies that the secondary icons exist for all the Windows Runtime platforms, they do not, but nothing bad happens if the particular icon file is missing from the project.

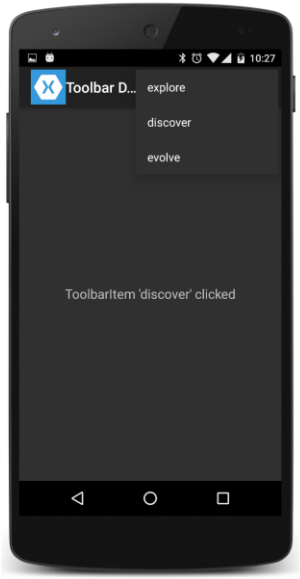
All the `Clicked` events have the same handler assigned. You can use unique handlers for the items, of course. This handler just displays the text of the `ToolbarItem` using the centered `Label`:

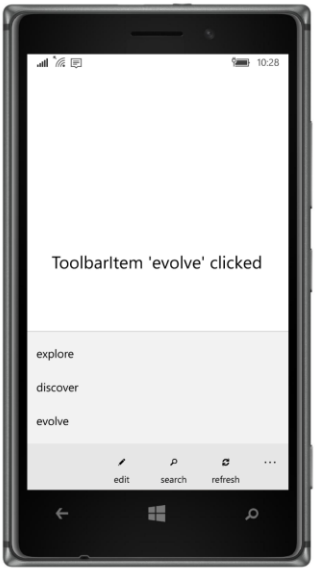
```
public partial class ToolbarDemoPage : ContentPage
{
    public ToolbarDemoPage()
    {
        InitializeComponent();
    }

    void OnToolbarItemClicked(object sender, EventArgs args)
    {
        ToolbarItem toolbarItem = (ToolbarItem)sender;
        label.Text = "ToolbarItem '" + toolbarItem.Text + "' clicked";
    }
}
```

The screenshots show the icon toolbar items (and for iOS, the text items) and the centered `Label` with the most recently clicked item:



If you tap the ellipsis at the top of the Android screen or the ellipsis at the lower-right corner of the Windows 10 Mobile screen, the text items are displayed and, in addition, the text items associated with the icons are also displayed on Windows 10 Mobile:





Regardless of the platform, the toolbar is the standard way to add common commands to a phone application.

Button images

`Button` defines an `Image` property of type `FileImageSource` that you can use to supply a small supplemental image that is displayed to the left of the button text. This feature is *not* intended for an image-only button; if that's what you want, the **ImageTap** program in this chapter is a good starting point.

You want the images to be about one-fifth inch in size. That means you want them to render at 32 device-independent units and to show up against the background of the `Button`. For iOS and the UWP, that means a black image against a white or transparent background. For Android, Windows 8.1, and Windows Phone 8.1, you'll want a white image against a transparent background.

All the bitmaps in the **ButtonImage** project are from the **Action Bar** directory of the **Android Design Icons** collection and the **03_rating_good** and **03_rating_bad** subdirectories. These are "thumbs up" and "thumbs down" images.

The iOS images are from the **holo_light** directory (black images on transparent backgrounds) with the following filename conversions:

- drawable-mdpi/ic_action_good.png not renamed
- drawable-xhdpi/ic_action_good.png renamed to ic_action_good@2x.png

And similarly for ic_action_bad.png.

The Android images are from the **holo_dark** directory (white images on transparent backgrounds) and include all four sizes from the subdirectories **drawable-mdpi** (32 pixels square), **drawable-hdpi** (48 pixels), **drawable-xhdpi** (64 pixels), and **drawable-xxhdpi** (96 pixels square).

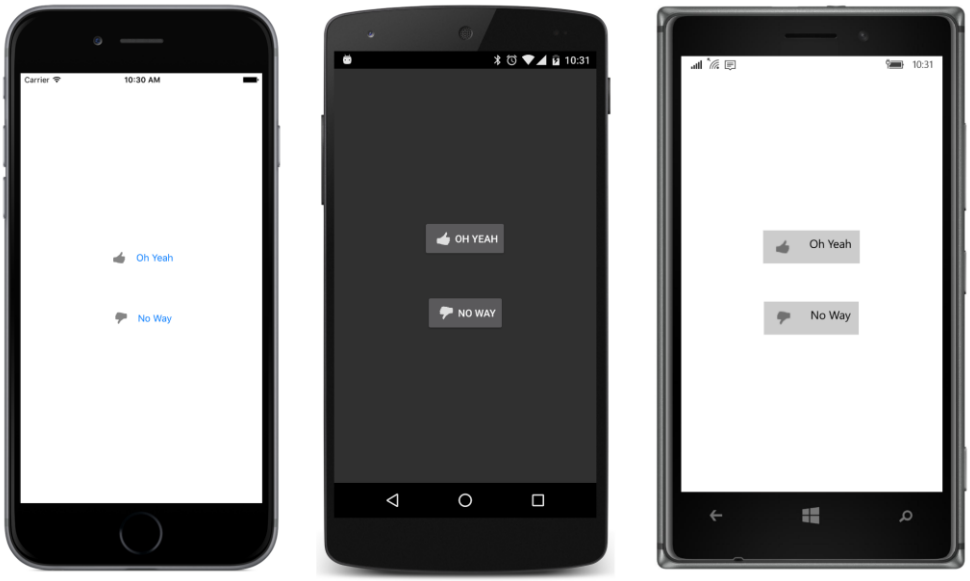
The images for the various Windows Runtime projects are all uniformly the 32-pixel bitmaps from the **drawable-mdpi** directories.

Here's the XAML file that sets the `Icon` property for two `Button` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonImage.ButtonImagePage">

    <StackLayout VerticalOptions="Center"
                 Spacing="50">

        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="Button">
                    <Setter Property="HorizontalOptions" Value="Center" />
                        </Setter.Value>
```

```
                </Setter>
            </Style>
        </ResourceDictionary>
    </StackLayout.Resources>

    <Button Text="Oh Yeah">
        <Button.Image>
            <OnPlatform x:TypeArguments="FileImageSource"
                        iOS="ic_action_good.png"
                        Android="ic_action_good.png"
                        WinPhone="Images/ic_action_good.png" />
        </Button.Image>
    </Button>

    <Button Text="No Way">
        <Button.Image>
            <OnPlatform x:TypeArguments="FileImageSource"
                        iOS="ic_action_bad.png"
                        Android="ic_action_bad.png"
                        WinPhone="Images/ic_action_bad.png" />
        </Button.Image>
    </Button>
  </StackLayout>
</ContentPage>
```

And here they are:



It's not much, but the bitmap adds a little panache to the normally text-only `Button`.

Another significant use for small bitmaps is the context menu available for items in the `TableView`. But a prerequisite for that is a deep exploration of the various views that contribute to the interactive interface of Xamarin.Forms. That's coming up in Chapter 15.

But first let's look at an alternative to `StackLayout` that lets you position child views in a completely flexible manner.