

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>
</StackLayout>
```

```

<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>

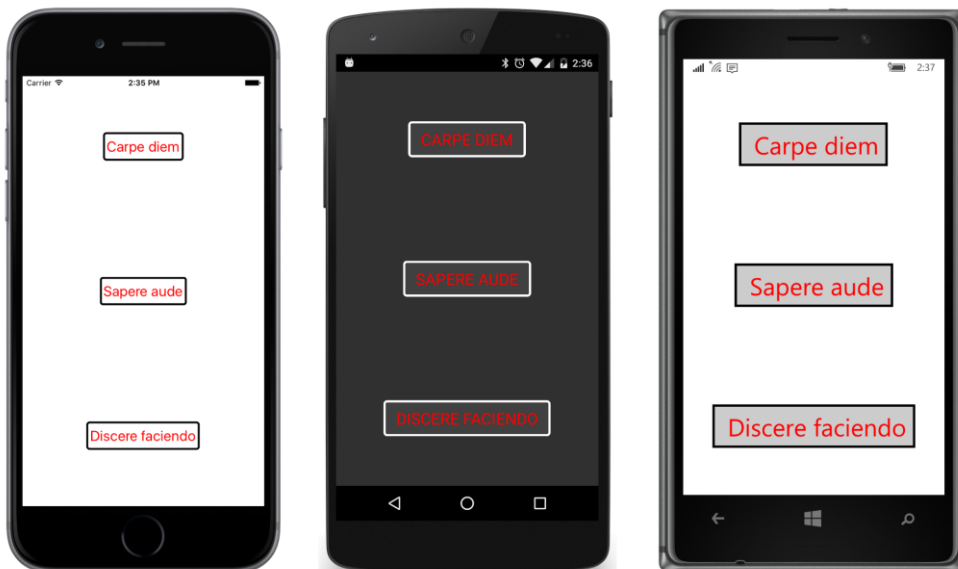
<StackLayout>
    <Button Text=" Carpe diem "
        Style="{StaticResource buttonStyle}" />

    <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:

[illegible]


```

                                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}" />

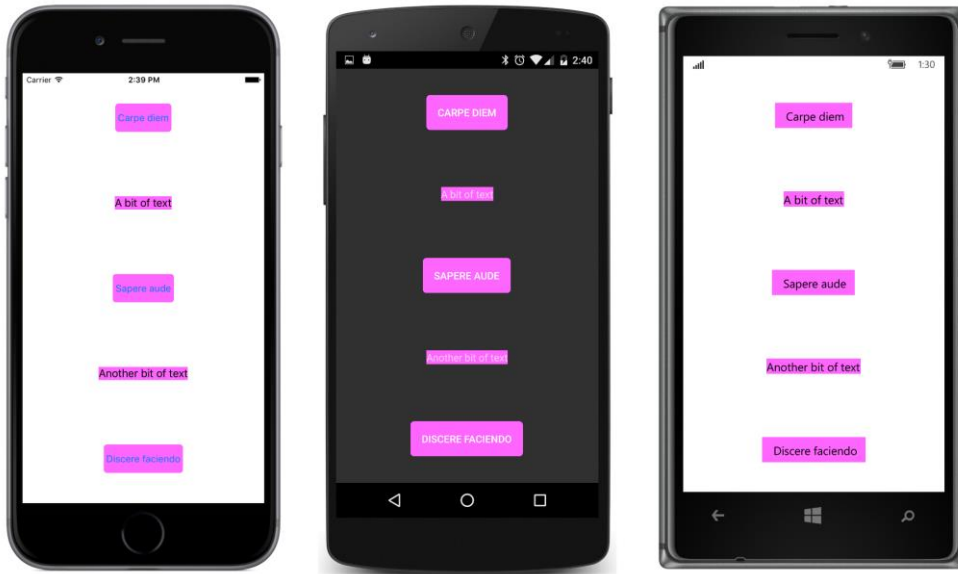
        <Label Text="A bit of text"
               Style="{StaticResource viewStyle}" />

        <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>

```

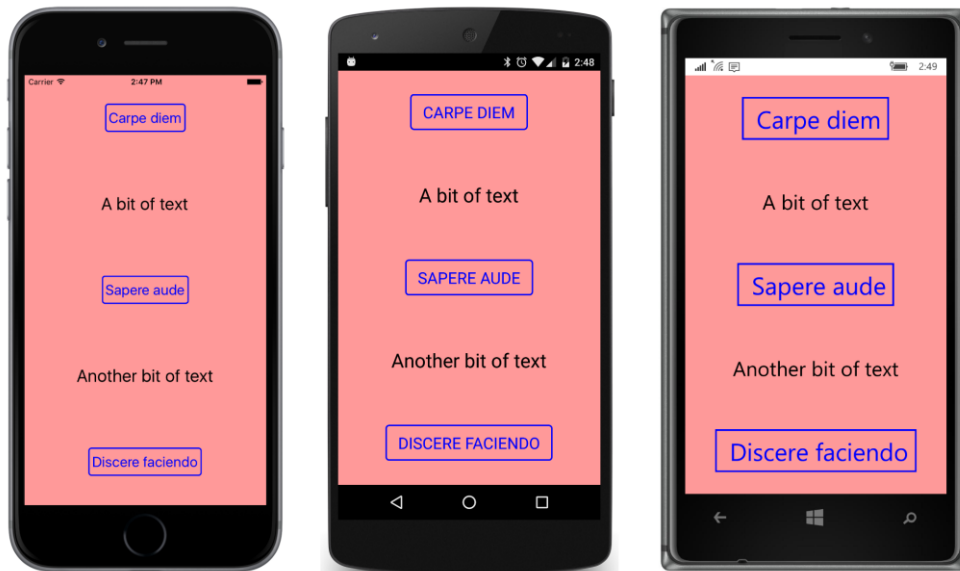
Immediately after the `Resources` section is some markup that sets the `Style` property of the page itself to the “visualStyle” style:

```

<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 `FontSize` property. That property is defined by `AltLabel`, so the `TargetType` must be `toolkit:AltLabel`.

A `Setter` can be defined for the `FontSize` property because `FontSize` 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 `FontSize` cannot be set in a style `Setter`. The XAML parser will complain that it cannot find `FontSizeProperty`, which is the bindable property that backs the `FontSize` 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" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
</ContentPage>
```

```

<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>

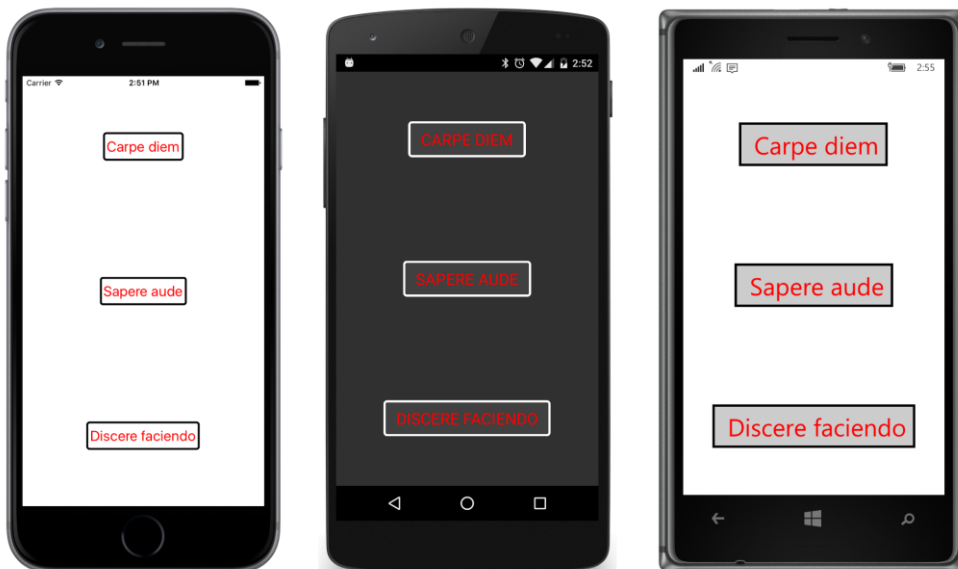
<StackLayout>
  <Button Text=" Carpe diem " />

  <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 " />

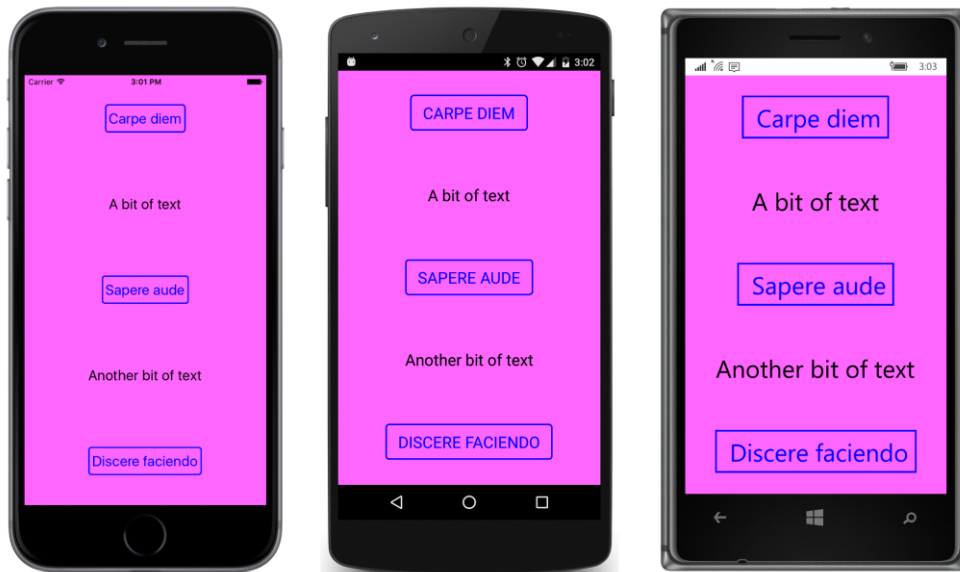
<Label Text ="Another bit of text" />

<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 re-attach 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>
        </ResourceDictionary>
    </ContentPage.Resources>
</ContentPage>

```



```

</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>

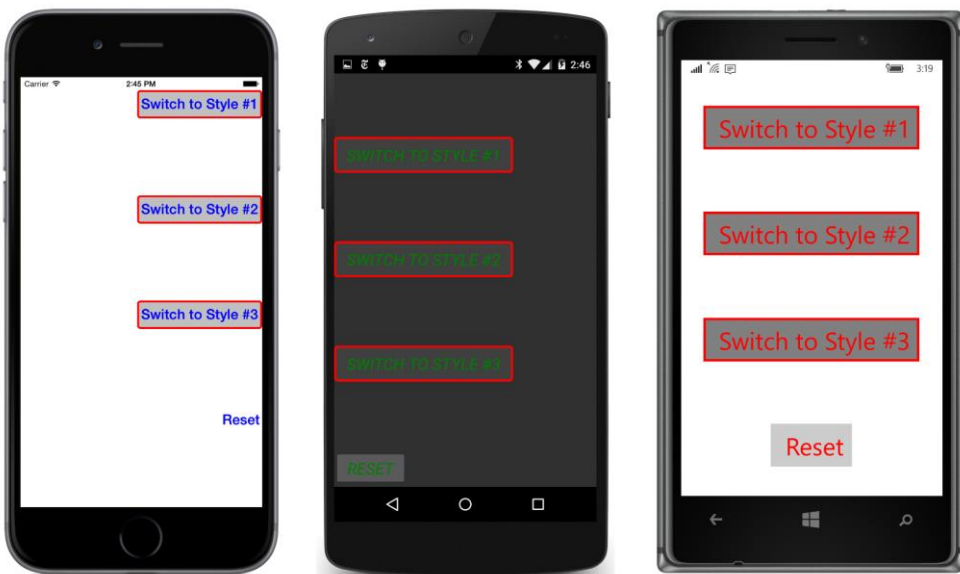
```

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" },

```

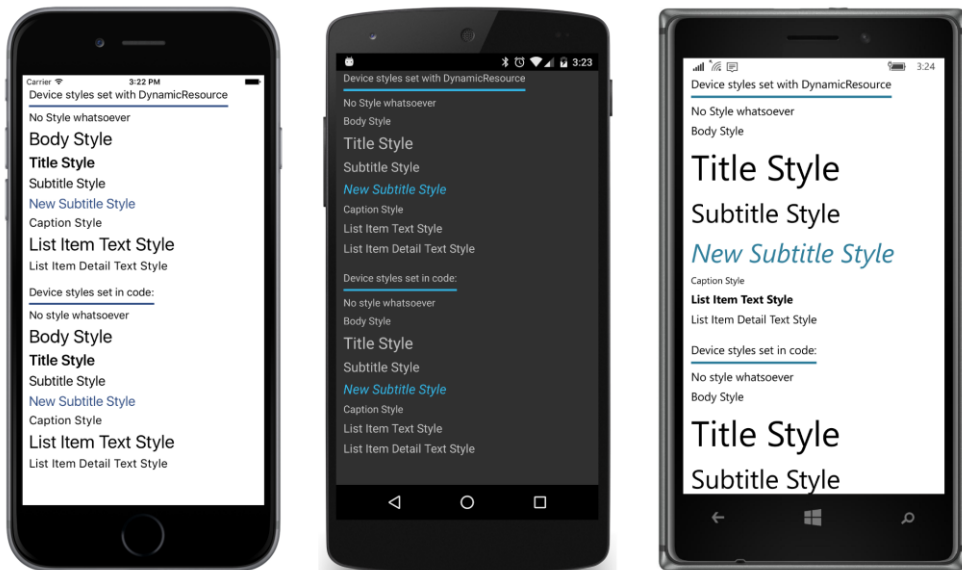
```

};

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.