Chapter 4
Scrolling the stack

If you're like most programmers, as soon as you saw that list of static Color properties in the previous chapter, you wanted to write a program to display them all, perhaps using the Text property of Label to identify the color, and the TextColor property to show the actual color.

Although you could do this with a single Label using a FormattedString object, it's much easier with multiple Label objects. Because multiple Label objects are involved, this job also requires some way to display all the Label objects on the screen.

The ContentPage class defines a Content property of type View that you can set to an object—but only one object. Displaying multiple views requires setting Content to an instance of a class that can have multiple children of type View. Such a class is Layout<T>, which defines a Children property of type IList<T>.

The Layout<T> class is abstract, but four classes derive from Layout<View>, a class that can have multiple children of type View. In alphabetical order, these four classes are:

- AbsoluteLayout
- Grid
- RelativeLayout
- StackLayout

Each of them arranges its children in a characteristic manner. This chapter focuses on StackLayout.

Stacks of views

The StackLayout class arranges its children in a stack. It defines only two properties on its own:

- Orientation of type StackOrientation, an enumeration with two members: Vertical (the default) and Horizontal.
- Spacing of type double, initialized to 6.0.

StackLayout seems ideal for the job of listing colors. You can use the Add method defined by IList<T> to add children to the Children collection of a StackLayout instance. Here's some code that creates multiple Label objects from two arrays and then adds each Label to the Children collection of a StackLayout:

```
Color[] colors =
```

```
{
    Color.White, Color.Silver, Color.Gray, Color.Black, Color.Red,
    Color.Maroon, Color.Yellow, Color.Olive, Color.Lime, Color.Green,
    Color.Aqua, Color.Teal, Color.Blue, Color.Navy, Color.Pink,
    Color.Fuchsia, Color.Purple
};

string[] colorNames =
{
    "White", "Silver", "Gray", "Black", "Red",
    "Maroon", "Yellow", "Olive", "Lime", "Green",
    "Aqua", "Teal", "Blue", "Navy", "Pink",
    "Fuchsia", "Purple"
};

StackLayout stackLayout = new StackLayout();

for (int i = 0; i < colors.Length; i++)
{
    Label label = new Label
    {
        Text = colorNames[i],
        TextColor = colors[i],
        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
    };
    stackLayout.Children.Add(label);
}
```

The `StackLayout` object can then be set to the `Content` property of the page.

But the technique of using parallel arrays is rather perilous. What if they're out of sync or have a different number of elements? A better approach is to keep the color and name together, perhaps in a tiny structure with `Color` and `Name` fields, or as an array of `Tuple<Color, string>` values, or as an anonymous type, as demonstrated in the **ColorLoop** program:

```
class ColorLoopPage : ContentPage
{
    public ColorLoopPage()
    {
        var colors = new[]
        {
            new { value = Color.White, name = "White" },
            new { value = Color.Silver, name = "Silver" },
            new { value = Color.Gray, name = "Gray" },
            new { value = Color.Black, name = "Black" },
            new { value = Color.Red, name = "Red" },
            new { value = Color.Maroon, name = "Maroon" },
            new { value = Color.Yellow, name = "Yellow" },
            new { value = Color.Olive, name = "Olive" },
            new { value = Color.Lime, name = "Lime" },
            new { value = Color.Green, name = "Green" },
            new { value = Color.Aqua, name = "Aqua" },
            new { value = Color.Teal, name = "Teal" },
```

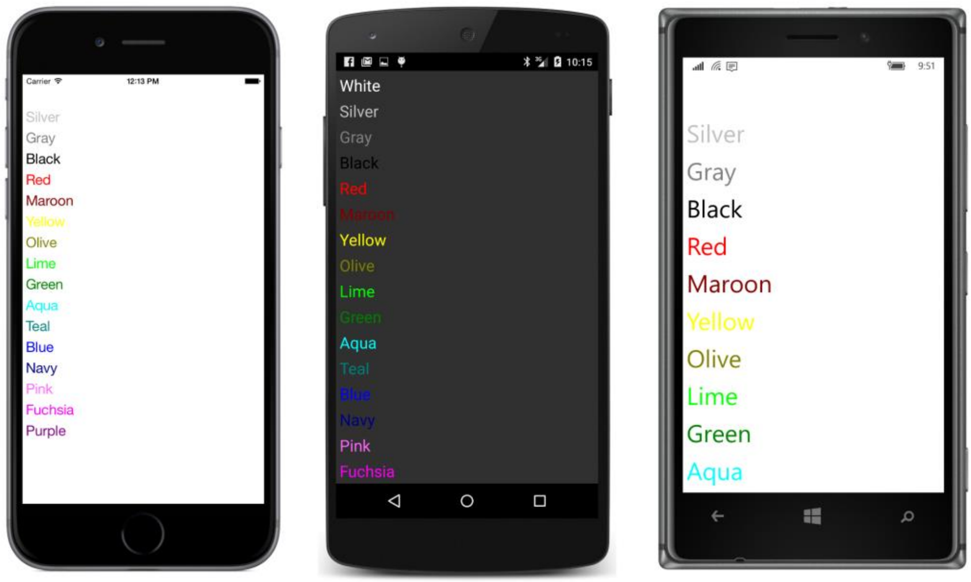
```
            new { value = Color.Blue, name = "Blue" },
            new { value = Color.Navy, name = "Navy" },
            new { value = Color.Pink, name = "Pink" },
            new { value = Color.Fuchsia, name = "Fuchsia" },
            new { value = Color.Purple, name = "Purple" }
        };

        StackLayout stackLayout = new StackLayout();

        foreach (var color in colors)
        {
            stackLayout.Children.Add(
                new Label
                {
                    Text = color.name,
                    TextColor = color.value,
                    FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
                });
        }

        Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);
        Content = stackLayout;
    }
}
```

Or you can initialize the `Children` property of `StackLayout` with an explicit collection of views (similar to the way the `Spans` collection of a `FormattedString` object was initialized in the previous chapter). The **ColorList** program sets the `Content` property of the page to a `StackLayout` object, which then has its `Children` property initialized with 17 `Label` views:

```
class ColorListPage : ContentPage
{
    public ColorListPage()
    {
        Padding = new Thickness (5, Device.OnPlatform (20, 5, 5), 5, 5);
        double fontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label));
        Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "White",
                    TextColor = Color.White,
                    FontSize = fontSize
                },
                new Label
                {
                    Text = "Silver",
                    TextColor = Color.Silver,
                    FontSize = fontSize
                },
```

```
                …
                new Label
                {
                    Text = "Fuchsia",
                    TextColor = Color.Fuchsia,
                    FontSize = fontSize
                },
                new Label
                {
                    Text = "Purple",
                    TextColor = Color.Purple,
                    FontSize = fontSize
                }
            }
        };
    }
}
```

You don't need to see the code for all 17 children to get the idea! Regardless of how you fill the `Children` collection, here's the result:



Obviously, this isn't optimum. Some colors aren't visible at all, and some of them are too faint to read well. Moreover, the list overflows the page on two platforms, and there's no way to scroll it up.

One solution is to reduce the text size. Instead of using `NamedSize.Large`, try one of the smaller values.

Another partial solution can be found in `StackLayout` itself: `StackLayout` defines a `Spacing` property of type `double` that indicates how much space to leave between the children. By default, it's

6.0, but you can set it to something smaller (for example, zero) to help ensure that all the items will fit:

```
Content = new StackLayout
{
    Spacing = 0,
    Children =
    {
        new Label
        {
            Text = "White",
            TextColor = Color.White,
            FontSize = fontSize
        },
        …
```

Now all the `Label` views occupy only as much vertical space as required for the text. You can even set `Spacing` to negative values to make the items overlap!

But the best solution is scrolling. Scrolling is not automatically supported by `StackLayout` and must be added with another element called `ScrollView`, as you'll see in the next section.

But there's another issue with the color programs shown so far: they need to either explicitly create an array of colors and names, or explicitly create `Label` views for each color. To programmers, this is somewhat tedious, and hence somewhat distasteful. Might it be automated?

Scrolling content

Keep in mind that a Xamarin.Forms program has access to the .NET base class libraries and can use .NET reflection to obtain information about all the classes and structures defined in an assembly, such as **Xamarin.Forms.Core**. This suggests that obtaining the static fields and properties of the `Color` structure can be automated.

Most .NET reflection begins with a `Type` object. You can obtain a `Type` object for any class or structure by using the C# `typeof` operator. For example, the expression `typeof(Color)` returns a `Type` object for the `Color` structure.

In the version of .NET available in the PCL, an extension method for the `Type` class, named `GetTypeInfo`, returns a `TypeInfo` object from which additional information can be obtained. Although that's not required in the program shown below; it needs other extension methods defined for the `Type` class, named `GetRuntimeFields` and `GetRuntimeProperties`. These return the fields and properties of the type in the form of collections of `FieldInfo` and `PropertyInfo` objects. From these, the names as well as the values of the properties can be obtained.

This is demonstrated by the **ReflectedColors** program. The ReflectedColorsPage.cs file requires a `using` directive for `System.Reflection`.

In two separate `foreach` statements, the `ReflectedColorsPage` class loops through all the fields and properties of the `Color` structure. For all the public static members that return `Color` values, the two loops call `CreateColorLabel` to create a `Label` with the `Color` value and name, and then add that `Label` to the `StackLayout`.

By including all the public static fields and properties, the program lists `Color.Transparent`, `Color.Default`, and `Color.Accent` along with the 17 static fields displayed in the earlier program. A separate `CreateColorLabel` method creates a `Label` view for each item. Here's the complete listing of the `ReflectedColorsPage` class:

```
public class ReflectedColorsPage : ContentPage
{
    public ReflectedColorsPage()
    {
        StackLayout stackLayout = new StackLayout();

        // Loop through the Color structure fields.
        foreach (FieldInfo info in typeof(Color).GetRuntimeFields())
        {
            // Skip the obsolete (i.e. misspelled) colors.
            if (info.GetCustomAttribute<ObsoleteAttribute>() != null)
                continue;

            if (info.IsPublic &&
                info.IsStatic &&
                info.FieldType == typeof(Color))
            {
                stackLayout.Children.Add(
                    CreateColorLabel((Color)info.GetValue(null), info.Name));
            }
        }

        // Loop through the Color structure properties.
        foreach (PropertyInfo info in typeof(Color).GetRuntimeProperties())
        {
            MethodInfo methodInfo = info.GetMethod;

            if (methodInfo.IsPublic &&
                methodInfo.IsStatic &&
                methodInfo.ReturnType == typeof(Color))
            {
                stackLayout.Children.Add(
                    CreateColorLabel((Color)info.GetValue(null), info.Name));
            }
        }

        Padding = new Thickness(5, Device.OnPlatform(20, 5, 5), 5, 5);

        // Put the StackLayout in a ScrollView.
        Content = new ScrollView
        {
            Content = stackLayout
        };
    }
```

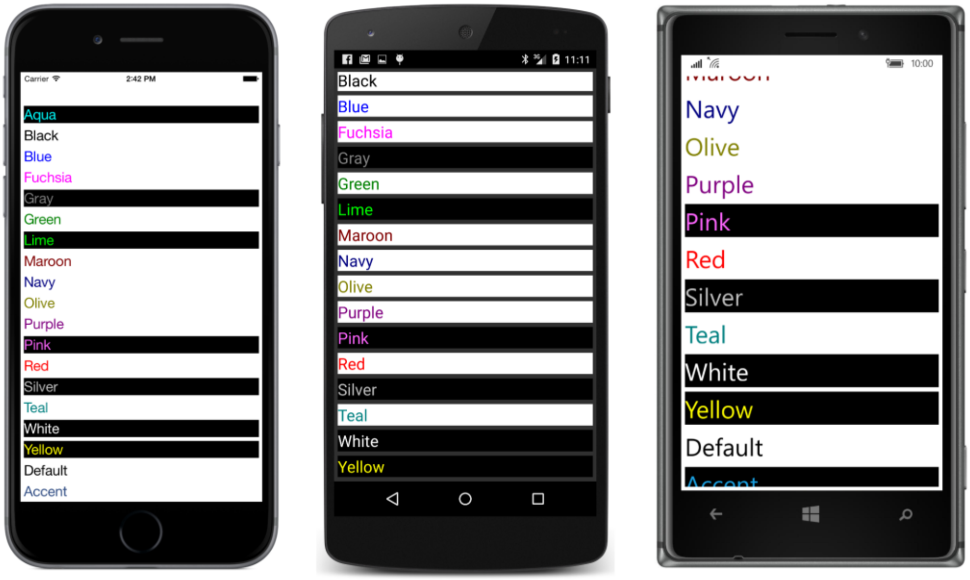
```
    Label CreateColorLabel(Color color, string name)
    {
        Color backgroundColor = Color.Default;

        if (color != Color.Default)
        {
            // Standard luminance calculation.
            double luminance = 0.30 * color.R +
                               0.59 * color.G +
                               0.11 * color.B;

            backgroundColor = luminance > 0.5 ? Color.Black : Color.White;
        }

        // Create the Label.
        return new Label
        {
            Text = name,
            TextColor = color,
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            BackgroundColor = backgroundColor
        };
    }
}
```

Toward the end of the constructor, the `StackLayout` is set to the `Content` property of a `ScrollView`, which is then set to the `Content` property of the page.

The `CreateColorLabel` method in the class attempts to make each color visible by setting a contrasting background. The method calculates a luminance value based on a standard weighted average of the red, green, and blue components and then selects a background of either white or black.

This technique won't work for `Transparent`, so that item can't be displayed at all, and the method treats `Color.Default` as a special case and displays that color (whatever it may be) against a `Color.Default` background.

Here are the results, which are still quite short of being aesthetically satisfying:



But you can scroll the display because the `StackLayout` is the child of a `ScrollView`.

`StackLayout` and `ScrollView` are related in the class hierarchy. `StackLayout` derives from `Layout<View>`, and you'll recall that the `Layout<T>` class defines the `Children` property that `StackLayout` inherits. The generic `Layout<T>` class derives from the nongeneric `Layout` class, and `ScrollView` also derives from this nongeneric `Layout`. Theoretically, `ScrollView` is a type of layout object—even though it has only one child.

As you can see from the screenshot, the background color of the `Label` extends to the full width of the `StackLayout`, which means that each `Label` is as wide as the `StackLayout`.

Let's experiment a bit to get a better understanding of Xamarin.Forms layout. For these experiments, you might want to temporarily give the `StackLayout` and the `ScrollView` distinct background colors:

```
public ReflectedColorsPage()
{
    StackLayout stackLayout = new StackLayout
    {
        BackgroundColor = Color.Blue
    };
    …
    Content = new ScrollView
    {
        BackgroundColor = Color.Red,
        Content = stackLayout
    };
}
```

Layout objects usually have transparent backgrounds by default. Although they occupy an area on the screen, they are not directly visible. Giving layout objects temporary colors is a great way to see exactly where they are on the screen. It's a good debugging technique for complex layouts.

You will discover that the blue `StackLayout` peeks out in the space between the individual `Label` views. This is a result of the default `Spacing` property of `StackLayout`. The `StackLayout` is also visible through the `Label` for `Color.Default`, which has a transparent background.

Try setting the `HorizontalOptions` property of all the `Label` views to `LayoutOptions.Start`:

```
return new Label
{
    Text = name,
    TextColor = color,
    FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
    BackgroundColor = backgroundColor,
    HorizontalOptions = LayoutOptions.Start
};
```

Now the blue background of the `StackLayout` is even more prominent because all the `Label` views occupy only as much horizontal space as the text requires, and they are all pushed over to the left side. Because each `Label` view is a different width, this display looks even uglier than the first version!

Now remove the `HorizontalOptions` setting from the `Label`, and instead set a `HorizontalOptions` on the `StackLayout`:

```
StackLayout stackLayout = new StackLayout
{
    BackgroundColor = Color.Blue,
    HorizontalOptions = LayoutOptions.Start
};
```

Now the `StackLayout` becomes only as wide as the widest `Label` (at least on iOS and Android) with the red background of the `ScrollView` now clearly in view.

As you begin constructing a tree of visual objects, these objects acquire a parent-child relationship. A parent object is sometimes referred to as the *container* of its child or children because the child's location and size is contained within its parent.

By default, `HorizontalOptions` and `VerticalOptions` are set to `LayoutOptions.Fill`, which means that each child view attempts to fill the parent container. (At least with the containers encountered so far. As you'll see, other layout classes have somewhat different behavior.) Even a `Label` fills its parent container by default, although without a background color, the `Label` appears to occupy only as much space as it requires.

Setting a view's `HorizontalOptions` or `VerticalOptions` property to `LayoutOptions.Start`, `Center`, or `End` effectively forces the view to shrink down—either horizontally, vertically, or both—to only the size the view requires.

A StackLayout has this same effect on its child's vertical size: every child in a StackLayout occupies only as much height as it requires. Setting the VerticalOptions property on a child of a StackLayout to Start, Center, or End has no effect! However, the child views still expand to fill the width of the StackLayout, except when the children are given a HorizontalOptions property other than LayoutOptions.Fill.

If a StackLayout is set to the Content property of a ContentPage, you can set HorizontalOptions or VerticalOptions on the StackLayout. These properties have two effects: first, they shrink the StackLayout width or height (or both) to the size of its children; and second, they govern where the StackLayout is positioned relative to the page.

If a StackLayout is in a ScrollView, the ScrollView causes the StackLayout to be only as tall as the sum of the heights of its children. This is how the ScrollView can determine how to vertically scroll the StackLayout. You can continue to set the HorizontalOptions property on the StackLayout to control the width and horizontal placement.

However, you should avoid setting VerticalOptions on the ScrollView to LayoutOptions.Start, Center, or End. The ScrollView must be able to scroll its child content, and the only way ScrollView can do that is by forcing its child (usually a StackLayout) to assume a height that reflects only what the child needs and then to use the height of this child and its own height to calculate how much to scroll that content. If you set VerticalOptions on the ScrollView to LayoutOptions.Start, Center, or End, you are effectively telling the ScrollView to be only as tall as it needs to be. But what is that height? Because ScrollView can scroll its contents, it doesn't need to be any particular height, so in theory it will shrink down to nothing. Xamarin.Forms protects against this eventuality, but it's best for you to avoid code that suggests something you don't want to happen.

Although putting a StackLayout in a ScrollView is normal, putting a ScrollView in a StackLayout doesn't seem quite right. In theory, the StackLayout will force the ScrollView to have a height of only what it requires, and that required height is basically zero. Again, Xamarin.Forms protects against this eventuality, but you should avoid such code.

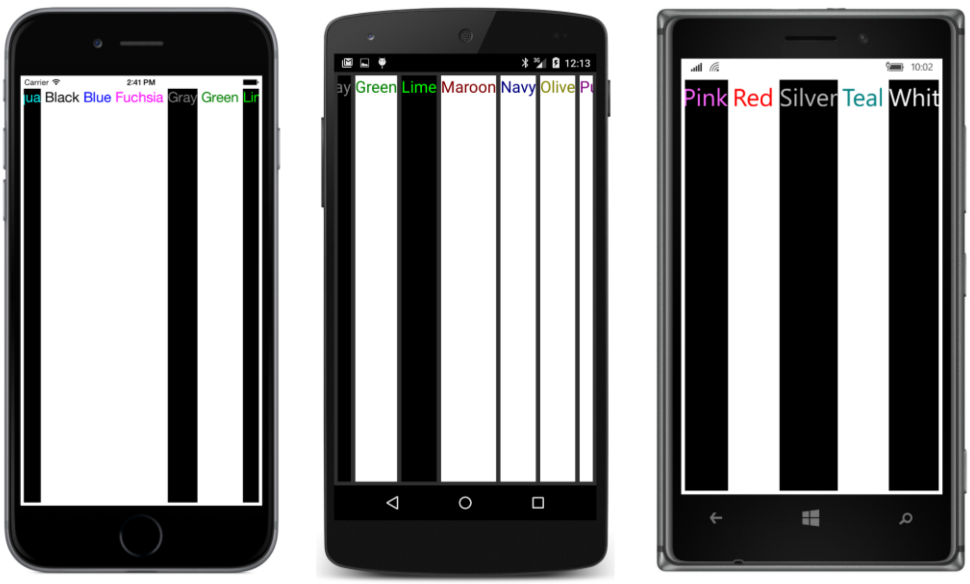
There is a proper way to put a ScrollView in a StackLayout that is in complete accordance with Xamarin.Forms layout principles, and that will be demonstrated shortly.

The preceding discussion applies to vertically oriented StackLayout and ScrollView elements. StackLayout has a property named Orientation that you can set to a member of the StackOrientation enumeration—Vertical (the default) or Horizontal. Similarly, ScrollView also has an Orientation property that you set to a member of the ScrollOrientation enumeration. Try this:

```
public ReflectedColorsPage()
{
    StackLayout stackLayout = new StackLayout
    {
        Orientation = StackOrientation.Horizontal
    };
    …
    Content = new ScrollView
```

```
    {
        Orientation = ScrollOrientation.Horizontal,
        Content = stackLayout
    };
}
```

Now the `Label` views are stacked horizontally, and the `ScrollView` fills the page vertically but allows horizontal scrolling of the `StackLayout`, which vertically fills the `ScrollView`:



It looks pretty weird with the default vertical layout options, but those could be fixed to make it look a little better.

The Expands option

You probably noticed that the `HorizontalOptions` and `VerticalOptions` properties are plurals, as if there's more than one option. These properties are generally set to a static field of the `LayoutOptions` structure—another plural.

The discussions so far have focused on the following static read-only `LayoutOptions` fields that returned predefined values of `LayoutOptions`:

- `LayoutOptions.Start`
- `LayoutOptions.Center`
- `LayoutOptions.End`

- `LayoutOptions.Fill`

The default—established by the `View` class—is `LayoutOptions.Fill`, which means that the view fills its container.

As you've seen, a `VerticalOptions` setting on a `Label` doesn't make a difference when the `Label` is a child of a vertical `StackLayout`. The `StackLayout` itself constrains the height of its children to only the height they require, so the child has no freedom to move vertically within that slot.

Be prepared for this rule to be slightly amended!

The `LayoutOptions` structure has four additional static read-only fields not discussed yet:

- `LayoutOptions.StartAndExpand`
- `LayoutOptions.CenterAndExpand`
- `LayoutOptions.EndAndExpand`
- `LayoutOptions.FillAndExpand`

`LayoutOptions` also defines two instance properties, named `Alignment` and `Expands`. The four instances of `LayoutOptions` returned by the static fields ending with `AndExpand` all have the `Expands` property set to `true`.

This `Expands` property is recognized only by `StackLayout`. It can be very useful for managing the layout of the page, but it can be confusing on first encounter. Here are the requirements for `Expands` to play a role in a vertical `StackLayout`:

- The contents of the `StackLayout` must have a total height that is less than the height of the `StackLayout` itself. In other words, some extra unused vertical space must exist in the `StackLayout`.
- That first requirement implies that the vertical `StackLayout` cannot have its own `VerticalOptions` property set to `Start`, `Center`, or `End` because that would cause the `StackLayout` to have a height equal to the aggregate height of its children, and it would have no extra space.
- At least one child of the `StackLayout` must have a `VerticalOptions` setting with the `Expands` property set to `true`.

If these conditions are satisfied, the `StackLayout` allocates the extra vertical space equally among all the children that have a `VerticalOptions` setting with `Expands` equal to `true`. Each of these children gets a larger slot in the `StackLayout` than normal. How the child occupies that slot depends on the `Alignment` setting of the `LayoutOptions` value: `Start`, `Center`, `End`, or `Fill`.

Here's a program, named **VerticalOptionsDemo,** that uses reflection to create `Label` objects with all the possible `VerticalOptions` settings in a vertical `StackLayout`. The background and foreground colors are alternated so that you can see exactly how much space each `Label` occupies. The program uses Language Integrated Query (LINQ) to sort the fields of the `LayoutOptions` structure in a visually more illuminating manner:

```
public class VerticalOptionsDemoPage : ContentPage
{
    public VerticalOptionsDemoPage()
    {
        Color[] colors = { Color.Yellow, Color.Blue };
        int flipFlopper = 0;

        // Create Labels sorted by LayoutAlignment property.
        IEnumerable<Label> labels =
            from field in typeof(LayoutOptions).GetRuntimeFields()
            where field.IsPublic && field.IsStatic
            orderby ((LayoutOptions)field.GetValue(null)).Alignment
            select new Label
            {
                Text = "VerticalOptions = " + field.Name,
                VerticalOptions = (LayoutOptions)field.GetValue(null),
                HorizontalTextAlignment = TextAlignment.Center,
                FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label)),
                TextColor = colors[flipFlopper],
                BackgroundColor = colors[flipFlopper = 1 - flipFlopper]
            };

        // Transfer to StackLayout.
        StackLayout stackLayout = new StackLayout();

        foreach (Label label in labels)
        {
            stackLayout.Children.Add(label);
        }

        Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
        Content = stackLayout;
    }
}
```

You might want to study the results a little:

The Label views with yellow text on blue backgrounds are those with VerticalOptions properties set to LayoutOptions values *without* the Expands flag set. If the Expands flag is not set on the LayoutOptions value of an item in a vertical StackLayout, the VerticalOptions setting is ignored. As you can see, the Label occupies only as much vertical space as it needs in the vertical StackLayout.

The total height of the children in this StackLayout is less than the height of the StackLayout, so the StackLayout has extra space. It contains four children with their VerticalOptions properties set to LayoutOptions values with the Expands flag set, so this extra space is allocated equally among those four children.

In these four cases—the Label views with blue text on yellow backgrounds—the Alignment property of the LayoutOptions value indicates how the child is aligned within the area that includes the extra space. The first one—with the VerticalOptions property set to LayoutOptions.StartAndExpand—is above this extra space. The second (CenterAndExpand) is in the middle of the extra space. The third (EndAndExpand) is below the extra space. However, in all these three cases, the Label is getting only as much vertical space as it needs, as indicated by the background color. The rest of the space belongs to the StackLayout, which shows the background color of the page.

The last Label has its VerticalOptions property set to LayoutOptions.FillAndExpand. In this case, the Label occupies the entire slot including the extra space, as the large area of yellow background indicates. The text is at the top of this area; that's because the default setting of VerticalTextAlignment is TextAlignment.Start. Set it to something else to position the text vertically within the area.

The `Expands` property of `LayoutOptions` plays a role only when the view is a child of a `StackLayout`. In other contexts, it's ignored.

Frame and BoxView

Two simple rectangular views are often useful for presentation purposes:

The `BoxView` is a filled rectangle. It derives from `View` and defines a `Color` property with a default setting of `Color.Default` that's transparent by default.

The `Frame` displays a rectangular border surrounding some content. `Frame` derives from `Layout` by way of `ContentView`, from which it inherits a `Content` property. The content of a `Frame` can be a single view or a layout containing a bunch of views. From `VisualElement`, `Frame` inherits a `BackgroundColor` property that's white on the iPhone but transparent on Android and Windows Phone. From `Layout`, `Frame` inherits a `Padding` property that it initializes to 20 units on all sides to give the content a little breathing room. `Frame` itself defines a `HasShadow` property that is `true` by default (but the shadow shows up only on iOS devices) and an `OutlineColor` property that is transparent by default but doesn't affect the iOS shadow, which is always black and always visible when `HasShadow` is set to `true`.

Both the `Frame` outline and the `BoxView` are transparent by default, so you might be a little uncertain how to color them without resorting to different colors for different platforms. One good choice is `Color.Accent`, which is guaranteed to show up regardless. Or, you can take control over coloring the background as well as the `Frame` outline and `BoxView`.

If the `BoxView` or `Frame` is not constrained in size in any way—that is, if it's not in a `StackLayout` and has its `HorizontalOptions` and `VerticalOptions` set to default values of `LayoutOptions.Fill`—these views expand to fill their containers.

For example, here's a program that has a centered `Label` set to the `Content` property of a `Frame`:

```
public class FramedTextPage : ContentPage
{
    public FramedTextPage()
    {
        Padding = new Thickness(20);
        Content = new Frame
        {
            OutlineColor = Color.Accent,
            Content = new Label
            {
                Text = "I've been framed!",
                FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.Center
            }
        };
```

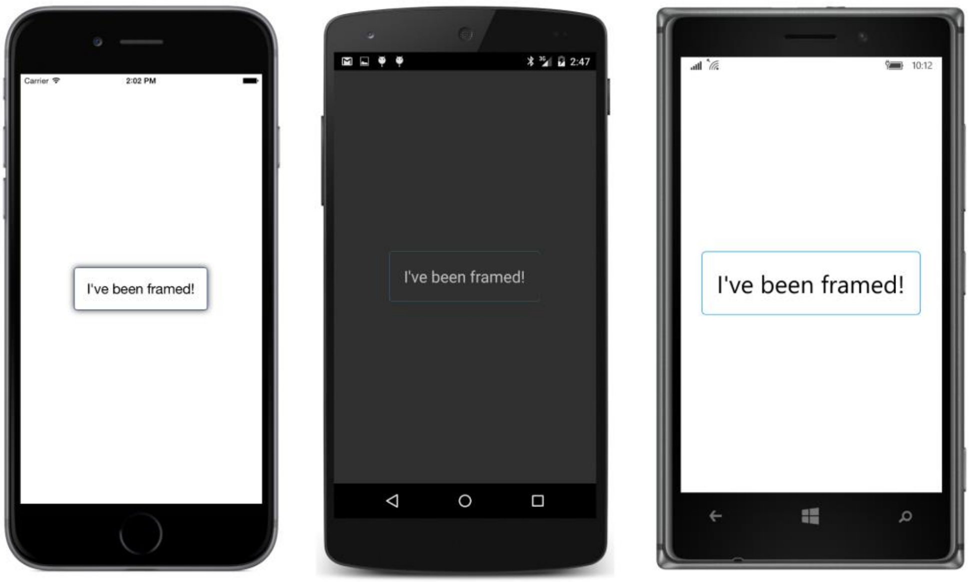
```
    }
}
```

The `Label` is centered in the `Frame`, but the `Frame` fills the whole page, and you might not even be able to see the `Frame` clearly if the page had not been given a `Padding` of 20 on all sides:

To display centered framed text, you want to set the `HorizontalOptions` and `VerticalOptions` properties on the `Frame` (rather than the `Label`) to `LayoutOptions.Center`:

```
public class FramedTextPage : ContentPage
{
    public FramedTextPage()
    {
        Padding = new Thickness(20);
        Content = new Frame
        {
            OutlineColor = Color.Accent,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            Content = new Label
            {
                Text = "I've been framed!",
                FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label))
            }
        };
    }
}
```

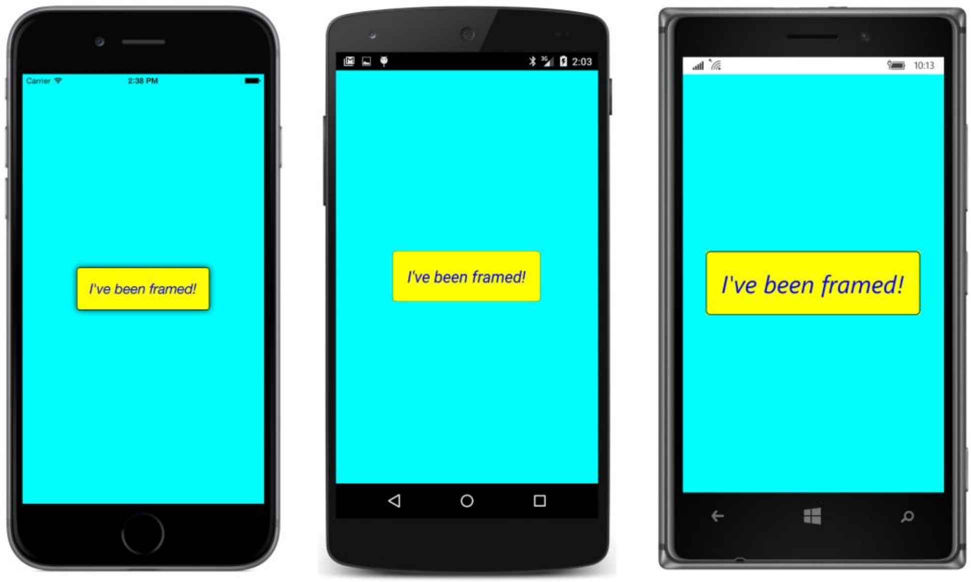
Now the `Frame` hugs the text (but with the frame's 20-unit default padding) in the center of the page:



The version of **FramedText** included with the sample code for this chapter exercises the freedom to give everything a custom color:

```
public class FramedTextPage : ContentPage
{
    public FramedTextPage()
    {
        BackgroundColor = Color.Aqua;

        Content = new Frame
        {
            OutlineColor = Color.Black,
            BackgroundColor = Color.Yellow,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,

            Content = new Label
            {
                Text = "I've been framed!",
                FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
                FontAttributes = FontAttributes.Italic,
                TextColor = Color.Blue
            }
        };
    }
}
```

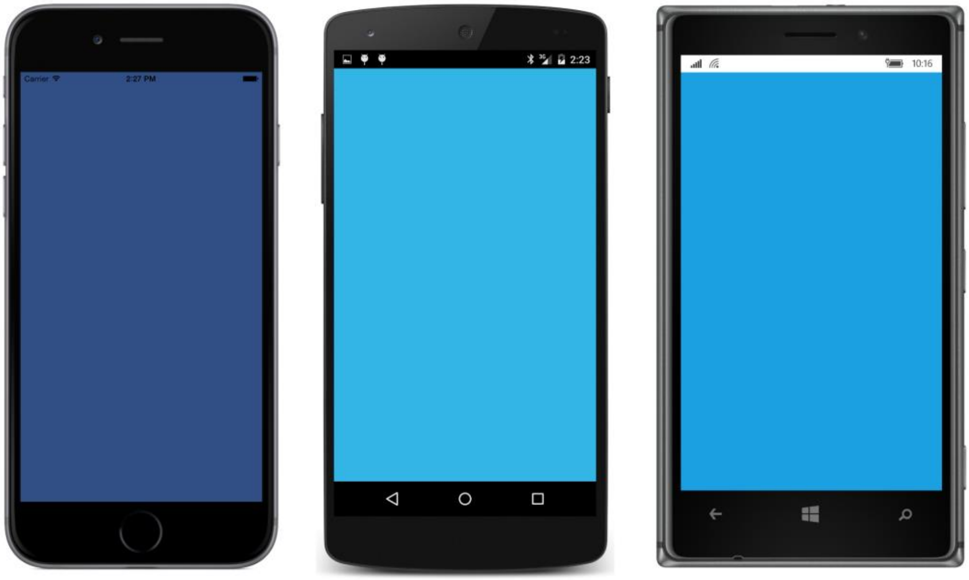
The result looks roughly the same on all three platforms:



Try setting a `BoxView` to the `Content` property of a `ContentPage`, like so:

```
public class SizedBoxViewPage : ContentPage
{
    public SizedBoxViewPage()
    {
        Content = new BoxView
        {
            Color = Color.Accent
        };
    }
}
```

Be sure to set the `Color` property so you can see it. The `BoxView` fills the whole area of its container, just as `Label` does with its default `HorizontalOptions` or `VerticalOptions` settings:

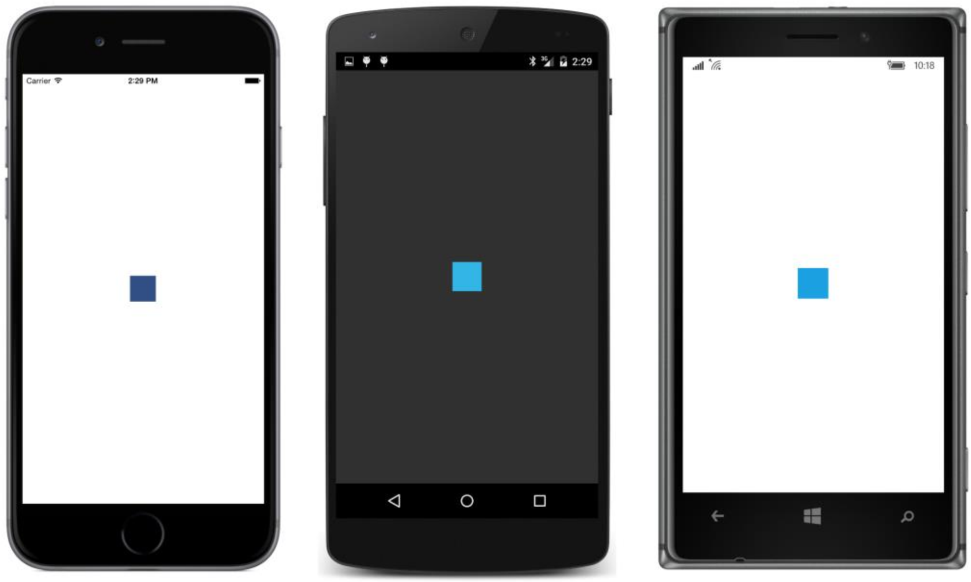


It's even underlying the iOS status bar!

Now try setting the `HorizontalOptions` and `VerticalOptions` properties of the `BoxView` to something other than `Fill`, as in this code sample:

```
public class SizedBoxViewPage : ContentPage
{
    public SizedBoxViewPage()
    {
        Content = new BoxView
        {
            Color = Color.Accent,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
    }
}
```

In this case, the `BoxView` will assume its default dimensions of 40 units square:



The BoxView is now 40 units square because the BoxView initializes its WidthRequest and HeightRequest properties to 40. These two properties require a little explanation:

VisualElement defines Width and Height properties, but these properties are read-only. VisualElement also defines WidthRequest and HeightRequest properties that are both settable and gettable. Normally, all these properties are initialized to –1 (which effectively means they are undefined), but some View derivatives, such as BoxView, set the WidthRequest and HeightRequest properties to specific values.

After a page has organized the layout of its children and rendered all the visuals, the Width and Height properties indicate actual dimensions of each view—the area that the view occupies on the screen. Because Width and Height are read-only, they are for informational purposes only. (Chapter 5, "Dealing with sizes," describes how to work with these values.)

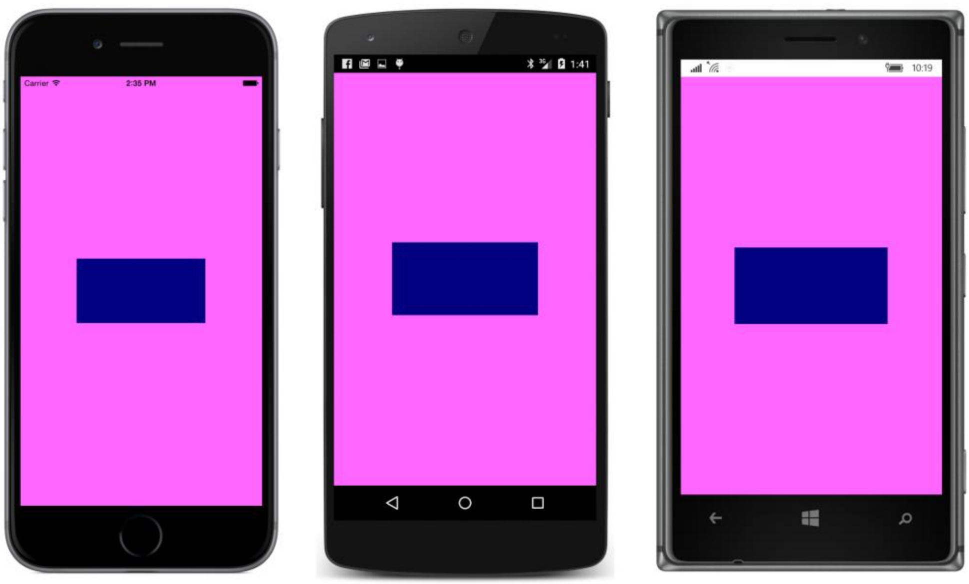
If you want a view to be a specific size, you can set the WidthRequest and HeightRequest properties. But these properties indicate (as their names suggest) a *requested* size or a *preferred* size. If the view is allowed to fill its container, these properties will be ignored.

BoxView sets its default size to values of 40 by overriding the OnSizeRequest method. You can think of these settings as a size that BoxView would like to be if nobody else has any opinions in the matter. You've already seen that WidthRequest and HeightRequest are ignored when the BoxView is allowed to fill the page. The WidthRequest kicks in if the HorizontalOptions is set to LayoutOptions.Left, Center, or Right, or if the BoxView is a child of a horizontal StackLayout. The HeightRequest behaves similarly.

Here's the version of the **SizedBoxView** program included with the code for this chapter:

```
public class SizedBoxViewPage : ContentPage
{
    public SizedBoxViewPage()
    {
        BackgroundColor = Color.Pink;

        Content = new BoxView
        {
            Color = Color.Navy,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center,
            WidthRequest = 200,
            HeightRequest = 100
        };
    }
}
```

Now we get a `BoxView` with that specific size and the colors explicitly set:



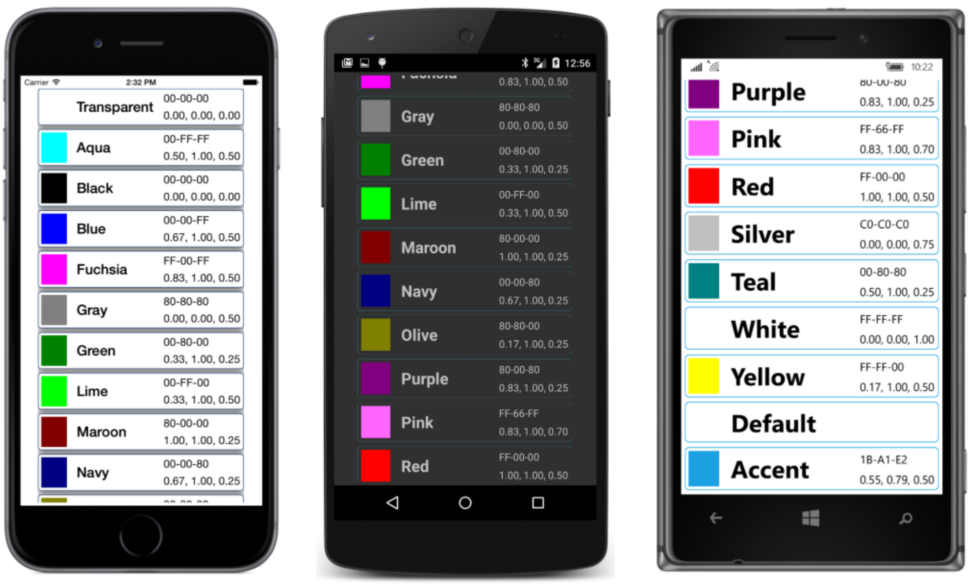
Let's use both `Frame` and `BoxView` in an enhanced color list. The **ColorBlocks** program has a page constructor that is virtually identical to the one in **ReflectedColors**, except that it calls a method named `CreateColorView` rather than `CreateColorLabel`. Here's that method:

```
class ColorBlocksPage : ContentPage
{
    ...

    View CreateColorView(Color color, string name)
    {
```

```
        return new Frame
        {
            OutlineColor = Color.Accent,
            Padding = new Thickness(5),
            Content = new StackLayout
            {
                Orientation = StackOrientation.Horizontal,
                Spacing = 15,
                Children =
                {
                    new BoxView
                    {
                        Color = color
                    },
                    new Label
                    {
                        Text = name,
                        FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
                        FontAttributes = FontAttributes.Bold,
                        VerticalOptions = LayoutOptions.Center,
                        HorizontalOptions = LayoutOptions.StartAndExpand
                    },
                    new StackLayout
                    {
                        Children =
                        {
                            new Label
                            {
                                Text = String.Format("{0:X2}-{1:X2}-{2:X2}",
                                                     (int)(255 * color.R),
                                                     (int)(255 * color.G),
                                                     (int)(255 * color.B)),
                                VerticalOptions = LayoutOptions.CenterAndExpand,
                                IsVisible = color != Color.Default
                            },
                            new Label
                            {
                                Text = String.Format("{0:F2}, {1:F2}, {2:F2}",
                                                     color.Hue,
                                                     color.Saturation,
                                                     color.Luminosity),
                                VerticalOptions = LayoutOptions.CenterAndExpand,
                                IsVisible = color != Color.Default
                            }
                        },
                        HorizontalOptions = LayoutOptions.End
                    }
                }
            }
        };
    }
}
```

The `CreateColorView` method returns a `Frame` containing a horizontal `StackLayout` with a `Box-View` indicating the color, a `Label` for the name of the color, and another `StackLayout` with two more `Label` views for the RGB composition and the `Hue`, `Saturation`, and `Luminosity` values. The RGB and HSL displays are meaningless for the `Color.Default` value, so that inner `StackLayout` has its `IsVisible` property set to `false` in that case. The `StackLayout` still exists, but it's ignored when the page is rendered.

The program doesn't know which element will determine the height of each color item—the `Box-View`, the `Label` with the color name, or the two `Label` views with the RGB and HSL values—so it centers all the `Label` views. As you can see, the `BoxView` expands in height to accommodate the height of the text:



Now this is a scrollable color list that's beginning to be something we can take a little pride in.

A ScrollView in a StackLayout?

It's common to put a `StackLayout` in a `ScrollView`, but can you put a `ScrollView` in a `StackLay-out`? And why would you even want to?

It's a general rule in layout systems like the one in Xamarin.Forms that you can't put a scroll in a stack. A `ScrollView` needs to have a specific height to compute the difference between the height of its content and its own height. That difference is the amount that the `ScrollView` can scroll its contents. If the `ScrollView` is in a `StackLayout`, it doesn't get that specific height. The `StackLayout` wants the `ScrollView` to be as short as possible, and that's either the height of the `ScrollView` contents or zero, and neither solution works.

So why would you want a `ScrollView` in a `StackLayout` anyway?

Sometimes it's precisely what you need. Consider a primitive e-book reader that implements scrolling. You might want a `Label` at the top of the page always displaying the book's title, followed by a `ScrollView` containing a `StackLayout` with the content of the book itself. It would be convenient for that `Label` and the `ScrollView` to be children of a `StackLayout` that fills the page.

With Xamarin.Forms, such a thing is possible. If you give the `ScrollView` a `VerticalOptions` setting of `LayoutOptions.FillAndExpand`, it can indeed be a child of a `StackLayout`. The `StackLayout` will give the `ScrollView` all the extra space not required by the other children, and the `ScrollView` will then have a specific height. Interestingly, Xamarin.Forms protects against other settings of that `VerticalOptions` property, so it works with whatever you set it to.

The **BlackCat** project displays the text of Edgar Allan Poe's short story "The Black Cat," which is stored in a text file named TheBlackCat.txt in a one-line-per-paragraph format.

How does the **BlackCat** program access the file with this short story? Perhaps the easiest approach is to embed the text file right in the program executable or—in the case of a Xamarin.Forms application—right in the Portable Class Library DLL. These files are known as *embedded resources*, and that's what TheBlackCat.txt file is in this program.

To make an embedded resource in either Visual Studio or Xamarin Studio, you'll probably first want to create a folder in the project by selecting the **Add > New Folder** option from the project menu. A folder for text files might be called **Texts**, for example. The folder is optional, but it helps organize program assets. Then, in that folder, you can select **Add > Existing Item** in Visual Studio or **Add > Add Files** in Xamarin Studio. Navigate to the file, select it, and click **Add** in Visual Studio or **Open** in Xamarin Studio.

Now here's the important part: Once the file is part of the project, bring up the **Properties** dialog from the menu associated with the file. Specify that the **Build Action** for the file is **EmbeddedResource**. This is an easy step to forget, but it is essential.

This was done for the **BlackCat** project, and consequently the TheBlackCat.txt file becomes embedded in the BlackCat.dll file.

In code, the file can be retrieved by calling the `GetManifestResourceStream` method defined by the `Assembly` class in the `System.Reflection` namespace. To get the assembly of the PCL, all you need to do is get the `Type` of any class defined in the assembly. You can use `typeof` with the page type you've derived from `ContentPage` or `GetType` on the instance of that class. Then call `GetTypeInfo` on this `Type` object. `Assembly` is a property of the resultant `TypeInfo` object:

```
Assembly assembly = GetType().GetTypeInfo().Assembly;
```

In the `GetManifestResourceStream` method of `Assembly`, you'll need to specify the name of the resource. For embedded resources, that name is not the filename of the resource but the *resource ID*. It's easy to confuse these because that ID might look vaguely like a fully qualified filename.

The resource ID begins with the default namespace of the assembly. This is not the .NET namespace! To get the default namespace of the assembly in Visual Studio, select **Properties** from the project menu, and in the properties dialog, select **Library** at the left and look for the **Default namespace** field. In Xamarin Studio, select **Options** from the project menu, and in the **Project Options** dialog, select **Main Settings** at the left, and look for a field labeled **Default Namespace**.

For the **BlackCat** project, that default namespace is the same as the assembly: "BlackCat". However, you can actually set that default namespace to whatever you want.

The resource ID begins with that default namespace, followed by a period, followed by the folder name you might have used, followed by another period and the filename. For this example, the resource ID is "BlackCat.Texts.TheBlackCat.txt"—and that's what you'll pass to the `GetManifestResourceStream` method in the code. The method returns a .NET `Stream` object, and from that a `StreamReader` can be created to read the lines of text.

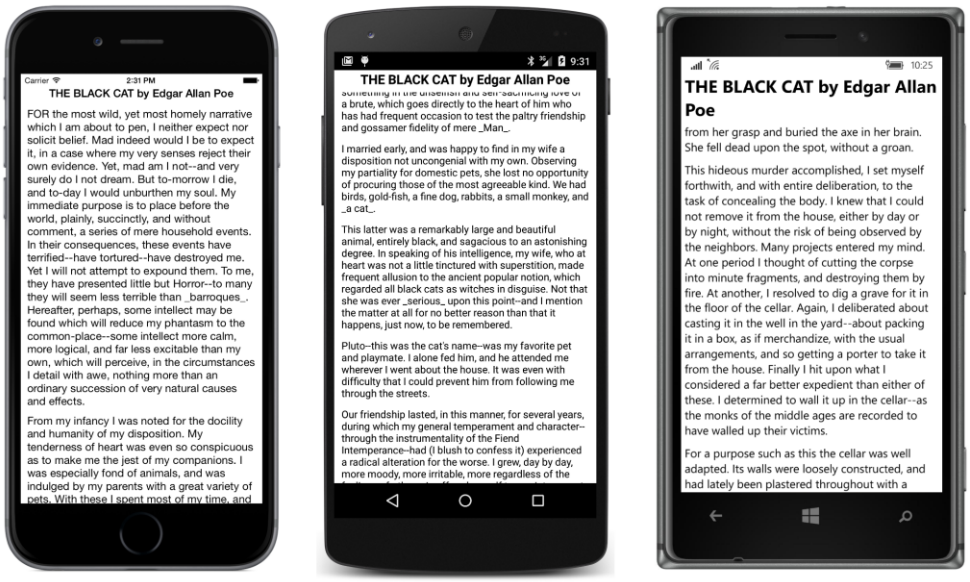
It's a good idea to use `using` statements with the `Stream` object returned from `GetManifestResourceStream` and the `StreamReader` object because that will properly dispose of the objects when they're no longer needed or if they raise exceptions.

For layout purposes, the `BlackCatPage` constructor creates two `StackLayout` objects: `mainStack` and `textStack`. The first line from the file (containing the story's title and author) becomes a bolded and centered `Label` in `mainStack`; all the subsequent lines go in `textStack`. The `mainStack` instance also contains a `ScrollView` with `textStack`.

```
class BlackCatPage : ContentPage
{
    public BlackCatPage()
    {
        StackLayout mainStack = new StackLayout();
        StackLayout textStack = new StackLayout
        {
            Padding = new Thickness(5),
            Spacing = 10
        };

        // Get access to the text resource.
        Assembly assembly = GetType().GetTypeInfo().Assembly;
        string resource = "BlackCat.Texts.TheBlackCat.txt";

        using (Stream stream = assembly.GetManifestResourceStream (resource))
        {
            using (StreamReader reader = new StreamReader (stream))
            {
                bool gotTitle = false;
                string line;

                // Read in a line (which is actually a paragraph).
```

```
                while (null != (line = reader.ReadLine()))
                {
                    Label label = new Label
                    {
                        Text = line,

                        // Black text for ebooks!
                        TextColor = Color.Black
                    };

                    if (!gotTitle)
                    {
                        // Add first label (the title) to mainStack.
                        label.HorizontalOptions = LayoutOptions.Center;
                        label.FontSize = Device.GetNamedSize(NamedSize.Medium, label);
                        label.FontAttributes = FontAttributes.Bold;
                        mainStack.Children.Add(label);
                        gotTitle = true;
                    }
                    else
                    {
                        // Add subsequent labels to textStack.
                        textStack.Children.Add(label);
                    }
                }
            }
        }

        // Put the textStack in a ScrollView with FillAndExpand.
        ScrollView scrollView = new ScrollView
        {
            Content = textStack,
            VerticalOptions = LayoutOptions.FillAndExpand,
            Padding = new Thickness(5, 0),
        };

        // Add the ScrollView as a second child of mainStack.
        mainStack.Children.Add(scrollView);

        // Set page content to mainStack.
        Content = mainStack;

        // White background for ebooks!
        BackgroundColor = Color.White;

        // Add some iOS padding for the page.
        Padding = new Thickness (0, Device.OnPlatform (20, 0, 0), 0, 0);
    }
}
```

Because this is basically an e-book reader, and humans have been reading black text on white paper for hundreds of years, the `BackgroundColor` of the page is set to white and the `TextColor` of each `Label` is set to black:



BlackCat is a PCL application. It is also possible to write this program using a Shared Asset Project rather than a PCL. To prove it, a **BlackCatSap** project is included with the code for this chapter. However, because the resource actually becomes part of the application project, you'll need the default namespace for the application, and that's different for each platform. The code to set the resource variable looks like this:

```
#if __IOS__
            string resource = "BlackCatSap.iOS.Texts.TheBlackCat.txt";
#elif __ANDROID__
            string resource = "BlackCatSap.Droid.Texts.TheBlackCat.txt";
#elif WINDOWS_UWP
            string resource = "BlackCatSap.UWP.Texts.TheBlackCat.txt";
#elif WINDOWS_APP
            string resource = "BlackCatSap.Windows.Texts.TheBlackCat.txt";
#elif WINDOWS_PHONE_APP
            string resource = "BlackCatSap.WinPhone.Texts.TheBlackCat.txt";
#endif
```

If you're having problems referencing an embedded resource, you might be using an incorrect name. Try calling `GetManifestResourceNames` on the `Assembly` object to get a list of the resource IDs of all embedded resources.