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" },
```

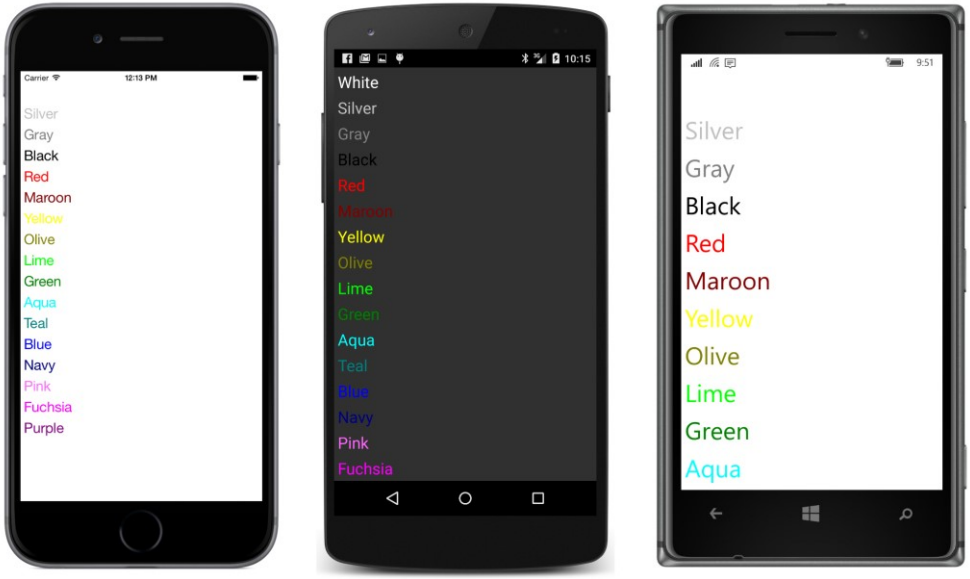
```
            new { value = Color.Blue, name = "Blue" },
            new { value = Color.Navy, name = "Navy" },
            new { value = Color.Pink, name = "Pink" },
            new { value = Color.Fuchsia, name = "Fuchsia" },
            new { value = Color.Purple, name = "Purple" }
        };

        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
        };
    }
```

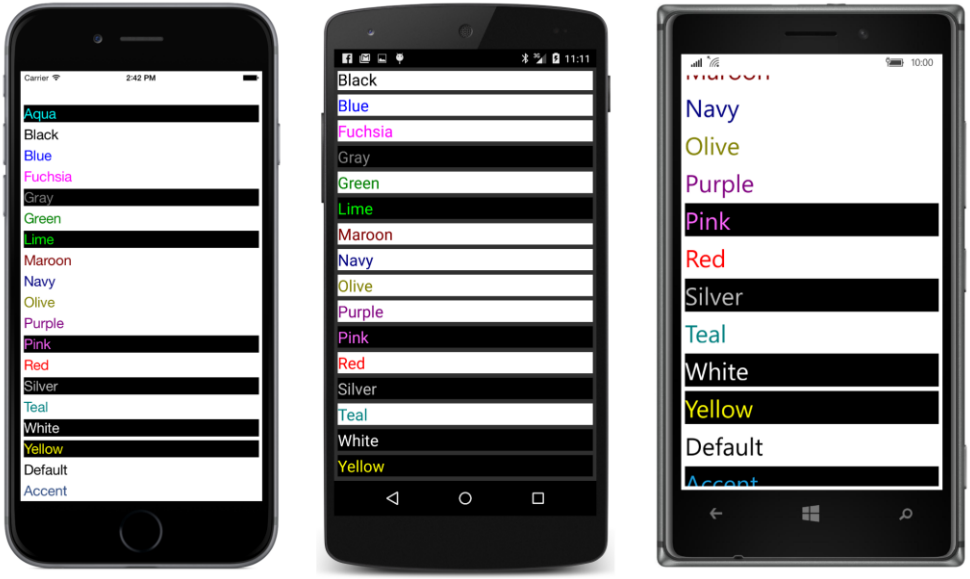
```
    Label CreateColorLabel(Color color, string name)
    {
        Color backgroundColor = Color.Default;

        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.

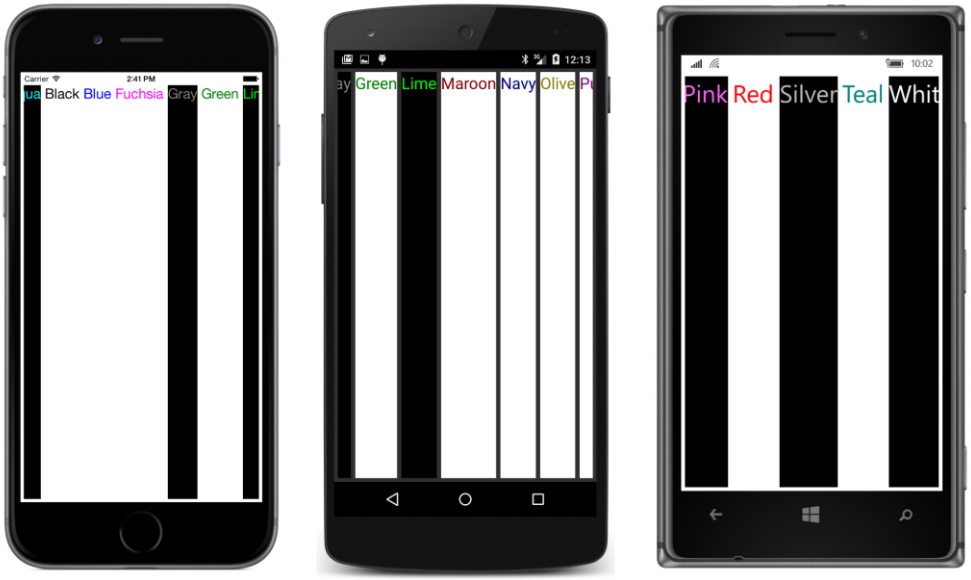
There is a proper way to put a ScrollView in a StackLayout that is in complete accordance with Xamarin.Forms layout principles, and that will be demonstrated shortly.

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`.

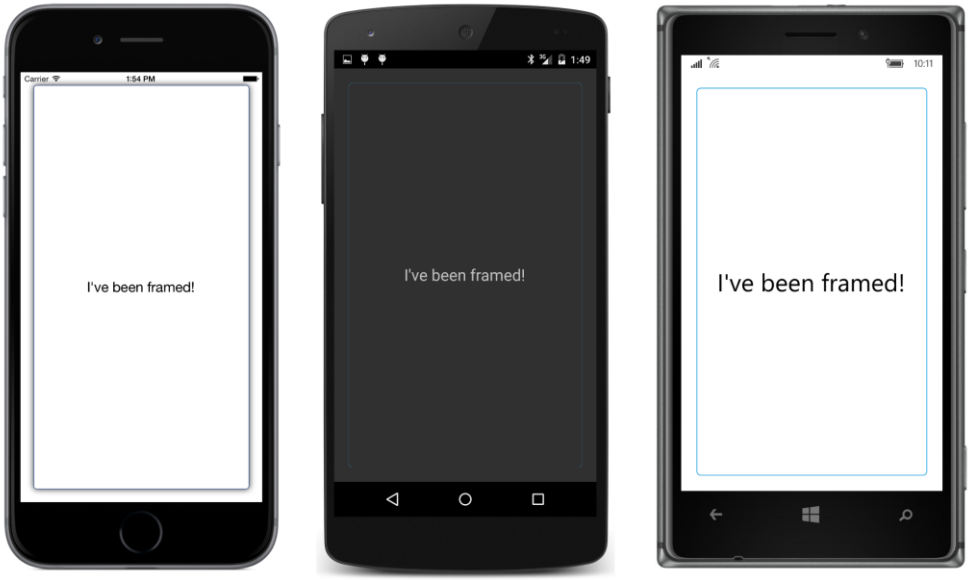
If the `BoxView` or `Frame` is not constrained in size in any way—that is, if it's not in a `StackLayout` and has its `HorizontalOptions` and `VerticalOptions` set to default values of `LayoutOptions.Fill`—these views expand to fill their containers.

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
            }
        };
```

```
    }
}
```

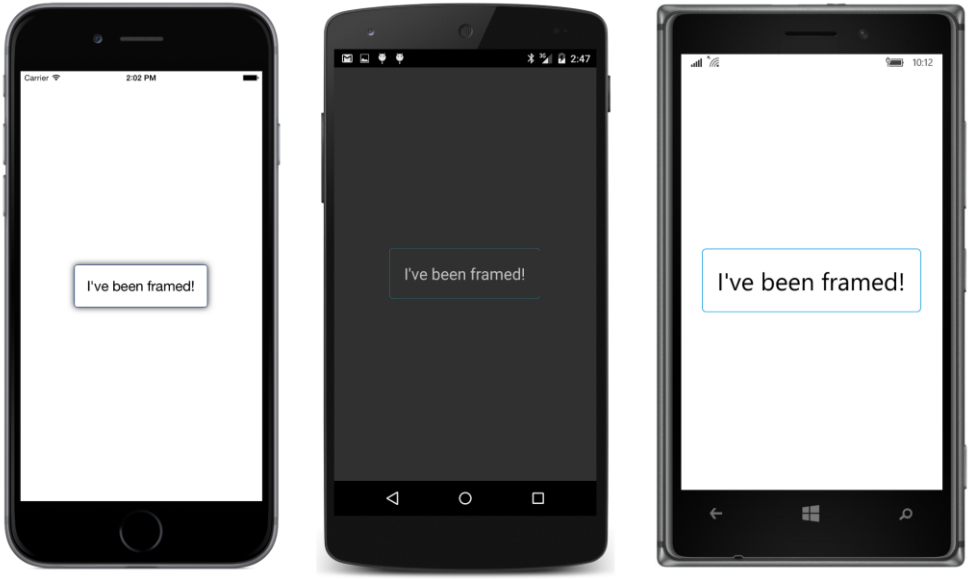
The `Label` is centered in the `Frame`, but the `Frame` fills the whole page, and you might not even be able to see the `Frame` clearly if the page had not been given a `Padding` of 20 on all sides:



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))
            }
        };
    }
}
```

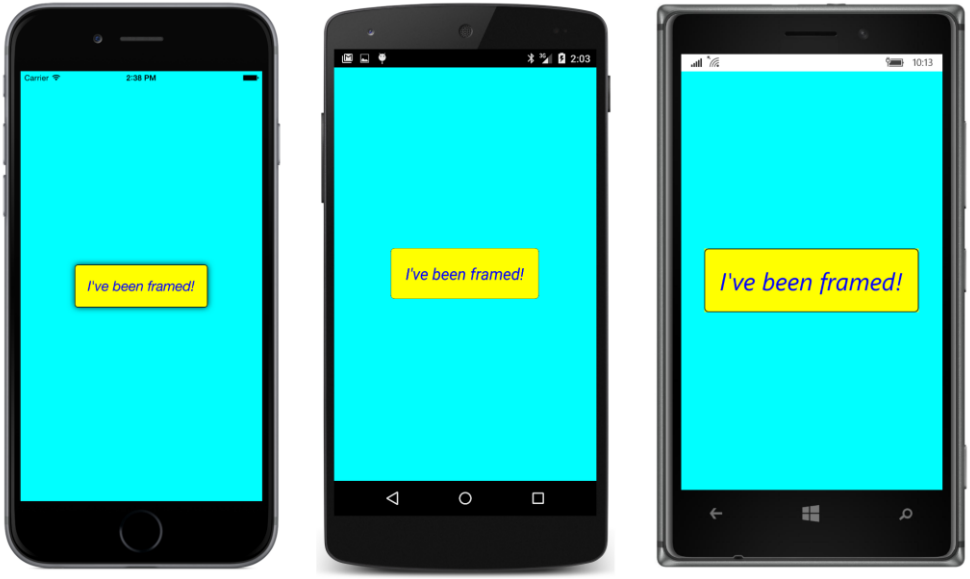
Now the `Frame` hugs the text (but with the frame's 20-unit default padding) in the center of the page:



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
            }
        };
    }
}
```

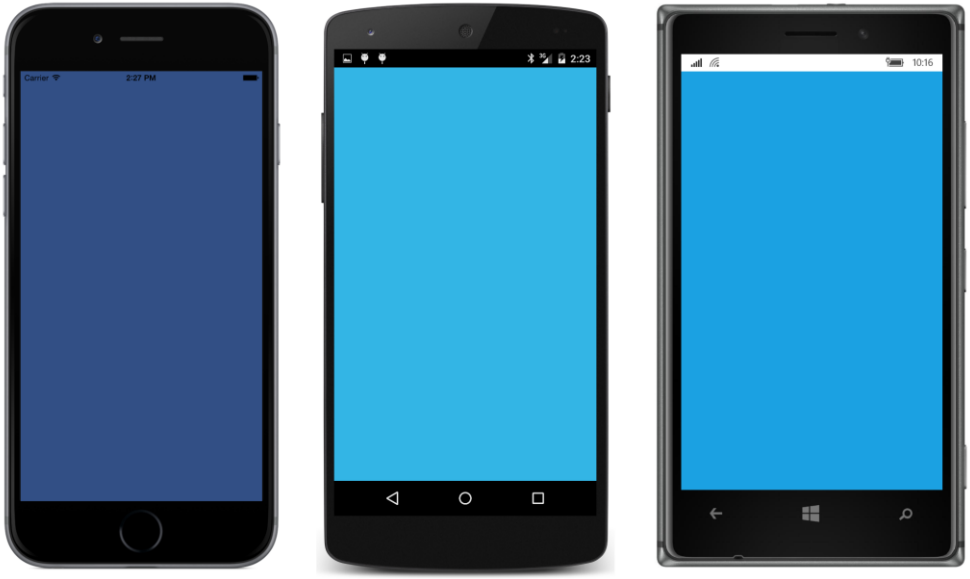
The result looks roughly the same on all three platforms:



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:

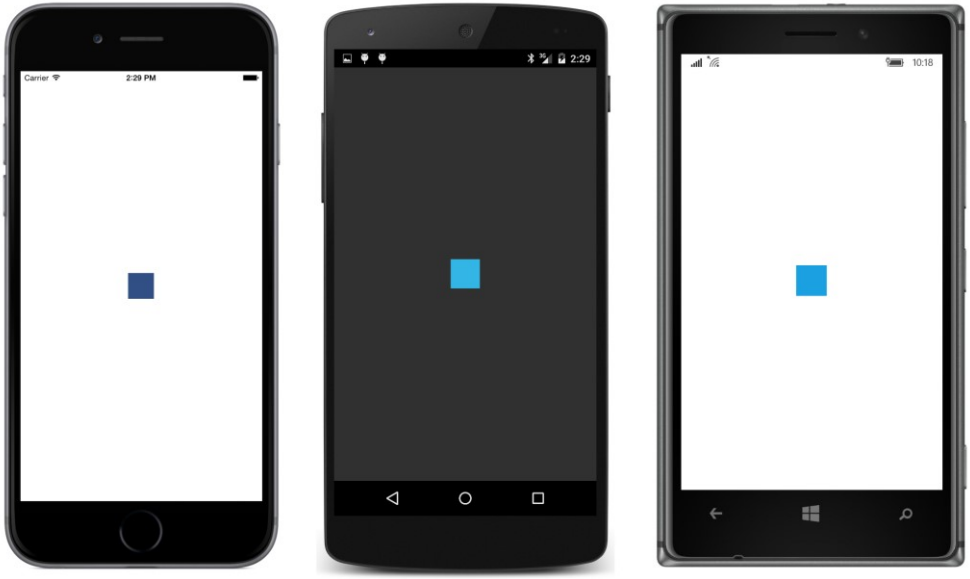


It's even underlying the iOS status bar!

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.)

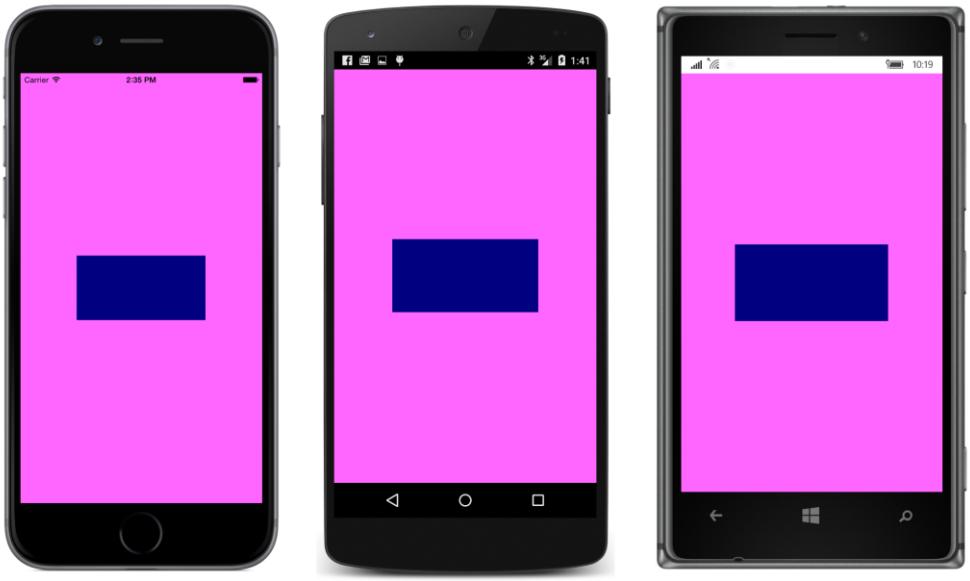
If you want a view to be a specific size, you can set the `WidthRequest` and `HeightRequest` properties. But these properties indicate (as their names suggest) a *requested* size or a *preferred* size. If the view is allowed to fill its container, these properties will be ignored.

`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:



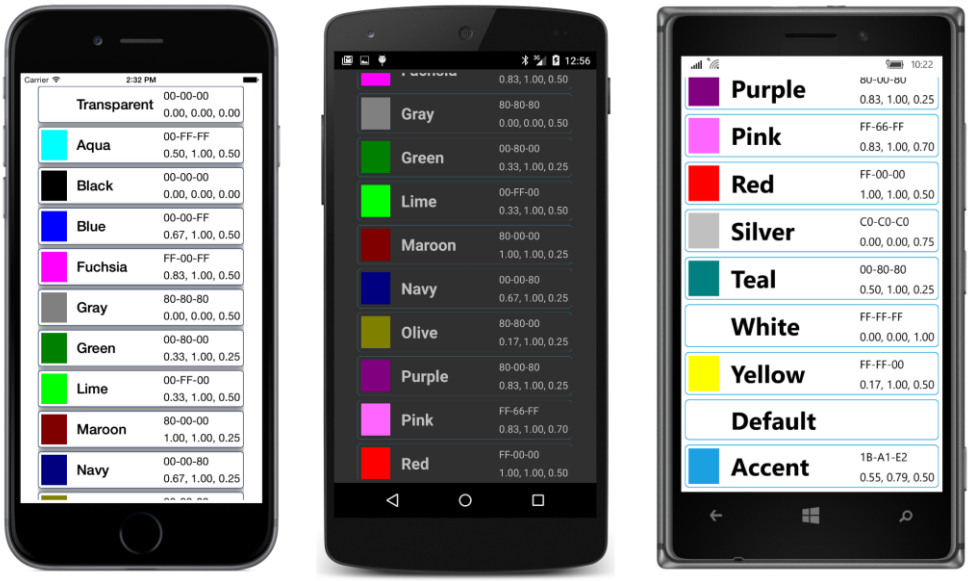
Let's use both `Frame` and `BoxView` in an enhanced color list. The **ColorBlocks** program has a page constructor that is virtually identical to the one in **ReflectedColors**, except that it calls a method named `CreateColorView` rather than `CreateColorLabel`. Here's that method:

```
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 `BoxView` 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 `BoxView`, 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 `StackLayout`? 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.

Chapter 5

Dealing with sizes

Already you've seen some references to sizes in connection with various visual elements:

- The iOS status bar has a height of 20, which you can adjust for with a `Padding` setting on the page.
- The `BoxView` sets its default width and height to 40.
- The default `Padding` within a `Frame` is 20.
- The default `Spacing` property on the `StackLayout` is 6.

And then there's `Device.GetNamedSize`, which for various members of the `NamedSize` enumeration returns a platform-dependent number appropriate for `FontSize` values for a `Label` or `Button`.

What are these numbers? What are their units? And how do we intelligently set properties requiring sizes to other values?

Good questions. As you've seen, the various platforms have different screen sizes and different text sizes, and all display a different quantity of text on the screen. Is that quantity of text something that a Xamarin.Forms application can anticipate or control? And even if it's possible, is it a proper programming practice? Should an application adjust font sizes to achieve a desired text density on the screen?

In general, when programming a Xamarin.Forms application, it's best not to get too close to the actual numeric dimensions of visual objects. It's preferable to trust Xamarin.Forms and the individual platforms to make the best default choices.

However, there are times when a programmer needs to know something about the size of particular visual objects and the size of the screen on which they appear.

Pixels, points, dps, DIPs, and DIUs

Video displays consist of a rectangular array of pixels. Any object displayed on the screen also has a pixel size. In the early days of personal computers, programmers sized and positioned visual objects in units of pixels. But as a greater variety of screen sizes and pixel densities became available, working with pixels became undesirable for programmers attempting to write applications that look roughly the same on many devices. Another solution was required.

These solutions began with operating systems for desktop computers and were then adapted for mobile devices. For this reason, it's illuminating to begin this exploration with the desktop.

Desktop video displays have a wide range of pixel dimensions, from the nearly obsolete 640 × 480 on up into the thousands. The aspect ratio of 4:3 was once standard for computer displays—and for movies and television as well—but the high-definition aspect ratio of 16:9 (or the similar 16:10) is now more common.

Desktop video displays also have a physical dimension usually measured along the diagonal of the screen in inches or centimeters. The pixel dimension combined with the physical dimension allows you to calculate the video display's resolution or pixel density in dots per inch (DPI), sometimes also referred to as pixels per inch (PPI). The display resolution can also be measured as a dot pitch, which is the distance between adjacent pixel centers, usually measured in millimeters.

For example, you can use the Pythagorean theorem to calculate that an ancient 800 × 600 display has a diagonal length of 1,000, the square root of 800 squared plus 600 squared. If this monitor has a 13-inch diagonal, that's a pixel density of 77 DPI, or a dot pitch of 0.33 millimeters. However, a 13-inch screen on a modern laptop might have pixel dimensions of 2560 × 1600, which is a pixel density of about 230 DPI, or a dot pitch of about 0.11 millimeters. A 100-pixel square object on this screen is one-third the size of the same object on the older screen.

Programmers should have a fighting chance when attempting to size visual elements correctly. For this reason, both Apple and Microsoft devised systems for desktop computing that allow programmers to work with the video display in some form of device-independent units instead of pixels. Most of the dimensions that a programmer encounters and specifies are in these device-independent units. It is the responsibility of the operating system to convert back and forth between these units and pixels.

In the Apple world, desktop video displays were traditionally assumed to have a resolution of 72 units to the inch. This number comes from typography, where many measurements are in units of *points*. In classical typography, there are approximately 72 points to the inch, but in digital typography the point has been standardized to be exactly one seventy-second of an inch. By working with points rather than pixels, a programmer has an intuitive sense of the relationship between numeric sizes and the area that visual objects occupy on the screen.

In the Windows world, a similar technique was developed, called *device-independent pixels* (DIPs) or *device-independent units* (DIUs). To a Windows programmer, desktop video displays are assumed to have a resolution of 96 DIUs, which is exactly one-third higher than 72 DPI, although it can be adjusted by the user.

Mobile devices, however, have somewhat different rules: The pixel densities achieved on modern phones are typically much higher than on desktop displays. This higher pixel density allows text and other visual objects to shrink much more in size before becoming illegible.

Phones are also typically held much closer to the user's face than is a desktop or laptop screen. This difference also implies that visual objects on the phone can be smaller than comparable objects on desktop or laptop screens. Because the physical dimensions of the phone are much smaller than desktop displays, shrinking down visual objects is very desirable because it allows much more to fit on the screen.

Apple continues to refer to the device-independent units on the iPhone as *points*. Until recently, all of Apple's high-density displays—which Apple refers to by the brand name Retina—have a conversion of two pixels to the point. This was true for the MacBook Pro, iPad, and iPhone. The recent exception is the iPhone 6 Plus, which has three pixels to the point.

For example, the 640 × 960 pixel dimension of the 3.5-inch screen of the iPhone 4 has an actual pixel density of about 320 DPI. There are two pixels to the point, so to an application program running on the iPhone 4, the screen appears to have a dimension of 320 × 480 points. The iPhone 3 actually did have a pixel dimension of 320 × 480, and points equaled pixels, so to a program running on these two devices, the displays of the iPhone 3 and iPhone 4 appear to be the same size. Despite the same perceived sizes, graphical objects and text are displayed in greater resolution on the iPhone 4 than the iPhone 3.

For the iPhone 3 and iPhone 4, the relationship between the screen size and point dimensions implies a conversion factor of 160 points to the inch rather than the desktop standard of 72.

The iPhone 5 has a 4-inch screen, but the pixel dimension is 640 × 1136. The pixel density is about the same as the iPhone 4. To a program, this screen has a size of 320 × 768 points.

The iPhone 6 has a 4.7-inch screen and a pixel dimension of 750 × 1334. The pixel density is also about 320 DPI. There are two pixels to the point, so to a program, the screen appears to have a point size of 375 × 667.

However, the iPhone 6 Plus has a 5.5-inch screen and a pixel dimension of 1080 × 1920, which is a pixel density of 400 DPI. This higher pixel density implies more pixels to the point, and for the iPhone 6 Plus, Apple has set the point equal to three pixels. That would normally imply a perceived screen size of 360 × 640 points, but to a program, the iPhone 6 Plus screen has a point size of 414 × 736, so the perceived resolution is about 150 points to the inch.

This information is summarized in the following table:

Model	iPhone 2, 3	iPhone 4	iPhone 5	iPhone 6	iPhone 6 Plus*
Pixel size	320 × 480	640 × 960	640 × 1136	750 × 1334	1080 × 1920
Screen diagonal	3.5 in.	3.5 in.	4 in.	4.7 in.	5.5 in.
Pixel density	165 DPI	330 DPI	326 DPI	326 DPI	401 DPI
Pixels per point	1	2	2	2	3
Point size	320 × 480	320 × 480	320 × 568	375 × 667	414 × 736
Points per inch	165	165	163	163	154

* Includes 115 percent downsampling.

Android does something quite similar: Android devices have a wide variety of sizes and pixel dimensions, but an Android programmer generally works in units of density-independent pixels (dps). The relationship between pixels and dps is set assuming 160 dps to the inch, which means that Apple and Android device-independent units are very similar.

Microsoft took a different approach with Windows Phone 7. The original Windows Phone 7 devices had a screen dimension of 480 × 800 pixels, which is often referred to as WVGA (Wide Video Graphics

Array). Applications worked with this display in units of pixels. If you assume an average screen size of 4 inches for a 480 × 800 Windows Phone 7 device, this means that Windows Phone 7 implicitly assumed a pixel density of about 240 DPI. That's 1.5 times the assumed pixel density of iPhone and Android devices. Eventually, several larger screen sizes were allowed: 768 × 1280 (WXGA or Wide Extended Graphics Array), 720 × 1280 (referred to using high-definition television lingo as 720p), and 1080 × 1920 (called 1080p). For these additional display sizes, programmers worked in device-independent units. An internal scaling factor translated between pixels and device-independent units so that the width of the screen in portrait mode always appeared to be 480 pixels.

With the Windows Runtime API in Windows Phone 8.1, different scaling factors were introduced based on both the screen's pixel size and the physical size of the screen. The following table was put together based on the Windows Phone 8.1 emulators using a program named **WhatSize,** which you'll see shortly:

Screen type	WVGA 4"	WXGA 4.5"	720p 4.7"	1080p 5.5"	1080p 6"
Pixel size	480 × 800	768 × 1280	720 × 1280	1080 × 1920	1080 × 1920
Size in DIUs	400 × 640	384 × 614.5	400 × 684	450 × 772	491 × 847
Scaling factor	1.2	2	1.8	2.4	2.2
DPI	194	161	169	167	167

The scaling factors were calculated from the width because the height in DIUs displayed by the **WhatSize** program excludes the Windows Phone status bar. The final DPI figures were calculated based on the full pixel size, the diagonal size of the screen in inches, and the scaling factor.

Aside from the WVGA outlier, the calculated DPI is close enough to the 160 DPI criterion associated with iOS and Android devices.

Windows 10 Mobile uses somewhat higher scaling factors, and in multiples of 0.25 rather than 0.2. The following table was put together based on the Windows 10 Mobile emulators:

Screen type	WVGA 4"	QHD 5.2"	WXGA 4.5"	720p 5"	1080p 6"
Pixel size	480 × 800	540 × 960	768 × 1280	720 × 1280	1080 × 1920
Size in DIUs	320 × 512	360 × 616	341 × 546	360 × 616	432 × 744
Scaling factor	1.5	1.5	2.25	2	2.5
DPI	155	141	147	147	141

You might conclude from this that a good average DPI for Windows 10 Mobile is 144 (rounded to the nearest multiple of 16) rather than 160. Or you might say that it's close enough to 160 to assume that it's consistent with iOS and Windows Phone.

Xamarin.Forms has a philosophy of using the conventions of the underlying platforms as much as possible. In accordance with this philosophy, a Xamarin.Forms programmer works with sizes defined by each particular platform. All sizes that the programmer encounters through the Xamarin.Forms API are in these platform-specific, device-independent units.

Xamarin.Forms programmers can generally treat the phone display in a device-independent manner, with the following resolution:

- 160 units to the inch
- 64 units to the centimeter

The `VisualElement` class defines two properties, named `Width` and `Height`, that provide the rendered dimensions of views, layouts, and pages in these device-independent units. However, the initial settings of `Width` and `Height` are "mock" values of –1. The values of these properties become valid only when the layout system has positioned and sized everything on the page. Also, keep in mind that the default `Fill` setting for `HorizontalOptions` or `VerticalOptions` often causes a view to occupy more space than it would otherwise. The `Width` and `Height` values reflect this extra space. The `Width` and `Height` values also include any `Padding` that may be set on the element and are consistent with the area colored by the view's `BackgroundColor` property.

`VisualElement` defines an event named `SizeChanged` that is fired whenever the `Width` or `Height` property of the visual element changes. This event is part of several notifications that occur when a page is laid out, a process that involves the various elements of the page being sized and positioned. This layout process occurs following the first definition of a page (generally in the page constructor), and a new layout pass takes place in response to any change that might affect layout—for example, when views are added to a `ContentPage` or a `StackLayout`, removed from these objects, or when properties are set on visual elements that might result in their sizes changing.

A new layout is also triggered when the screen size changes. This happens mostly when the phone is swiveled between portrait and landscape modes.

A full familiarity with the Xamarin.Forms layout system often accompanies the job of writing your own `Layout<View>` derivatives. This task awaits us in Chapter 26, "Custom layouts." Until then, simply knowing when `Width` and `Height` properties change is helpful for working with sizes of visual objects. You can attach a `SizeChanged` handler to any visual object on the page, including the page itself. The **WhatSize** program demonstrates how to obtain the page's size and display it:

```
public class WhatSizePage : ContentPage
{
    Label label;

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

        Content = label;

        SizeChanged += OnPageSizeChanged;
    }

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

```
    {
        label.Text = String.Format("{0} \u00D7 {1}", Width, Height);
    }
}
```

This is the first example of event handling in this book, and you can see that events are handled in the normal C# and .NET manner. The code at the end of the constructor attaches the `OnPageSizeChanged` event handler to the `SizeChanged` event of the page. The first argument to the event handler (customarily named `sender`) is the object firing the event, in this case the instance of `WhatSizePage`, but the event handler doesn't use that. Nor does the event handler use the second argument—the so-called *event arguments*—which sometimes provides more information about the event.

Instead, the event handler accesses the `Label` element (conveniently saved as a field) to display the `Width` and `Height` properties of the page. The Unicode character in the `String.Format` call is a times (×) symbol.

The `SizeChanged` event is not the only opportunity to obtain an element's size. `VisualElement` also defines a protected virtual method named `OnSizeAllocated` that indicates when the visual element is assigned a size. You can override this method in your `ContentPage` derivative rather than handling the `SizeChanged` event, but `OnSizeAllocated` is sometimes called when the size isn't actually changing.

Here's the program running on the three standard platforms:

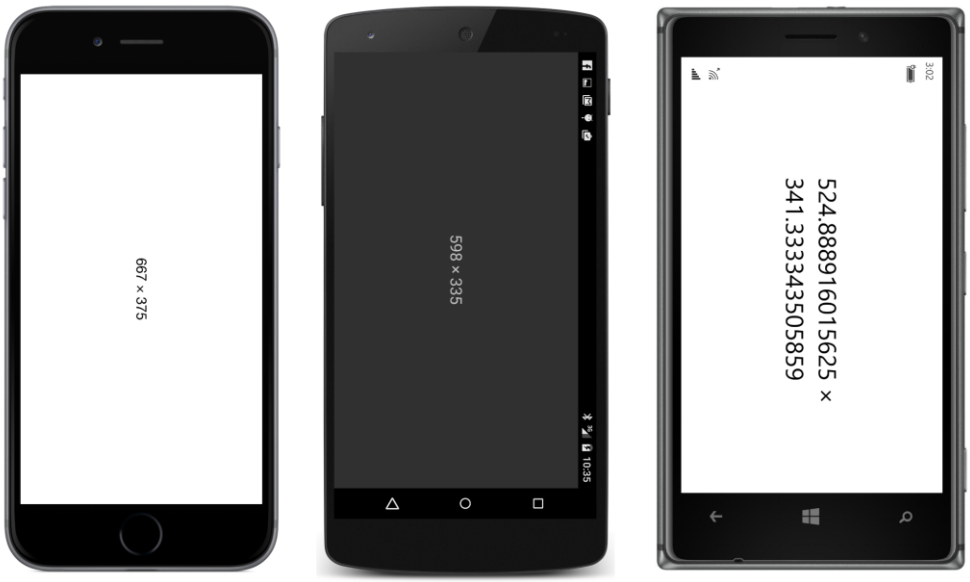
For the record, these are the sources of the screens in these three images:

- The iPhone 6 simulator, with pixel dimensions of 750 × 1334.

- An LG Nexus 5 with a screen size of 1080 × 1920 pixels.
- A Nokia Lumia 925 with a screen size of 768 × 1280 pixels.

Notice that the vertical size perceived by the program on the Android does not include the area occupied by the status bar or bottom buttons; the vertical size on the Windows 10 Mobile device does not include the area occupied by the status bar.

By default, all three platforms respond to device orientation changes. If you turn the phones (or emulators) 90 degrees counterclockwise, the phones display the following sizes:



The screenshots for this book are designed only for portrait mode, so you'll need to turn this book sideways to see what the program looks like in landscape. The 598-pixel width on the Android excludes the area for the buttons; the 335-pixel height excludes the status bar, which always appears above the page. On the Windows 10 Mobile device, the 728-pixel width excludes the area for the status bar, which appears in the same place but with rotated icons to reflect the new orientation.

Here's the program running on the iPad Air 2 simulator with a pixel dimension of 2048 × 1536.

Obviously, the scaling factor is 2. The screen is 9.7 inches in diagonal for a resolution of 132 DPI.

The Surface Pro 3 has a pixel dimension of 2160 × 1440. The scaling factor is selectable by the user to make everything on the screen larger or smaller, but the recommended scaling factor is 1.5:

The height displayed by **WhatSize** excludes the taskbar at the bottom of the screen. The screen is 12″ in diagonal for a resolution of 144 DPI.

A few notes on the **WhatSize** program itself:

WhatSize creates a single `Label` in its constructor and sets the `Text` property in the event handler. That's not the only way to write such a program. The program could use the `SizeChanged` handler to create a whole new `Label` with the new text and set that new `Label` as the content of the page, in which case the previous `Label` would become unreferenced and hence eligible for garbage collection. But creating new visual elements is unnecessary and wasteful in this program. It's best for the program to create only one `Label` view and just set the `Text` property to indicate the page's new size.

Monitoring size changes is the only way a Xamarin.Forms application can detect orientation changes without obtaining platform-specific information. Is the width greater than the height? That's landscape. Otherwise, it's portrait.

By default, the Visual Studio and Xamarin Studio templates for Xamarin.Forms solutions enable device orientation changes for all three platforms. If you want to disable orientation changes—for example, if you have an application that just doesn't work well in portrait or landscape mode—you can do so.

For iOS, first display the contents of Info.plist in Visual Studio or Xamarin Studio. In the **iPhone Deployment Info** section, use the **Supported Device Orientations** area to specify which orientations are allowed.

For Android, in the `Activity` attribute on the `MainActivity` class in the MainActivity.cs file, add:

```
ScreenOrientation = ScreenOrientation.Landscape
```

or

```
ScreenOrientation = ScreenOrientation.Portrait
```

The `Activity` attribute generated by the solution template contains a `ConfigurationChanges` argument that also refers to screen orientation, but the purpose of `ConfigurationChanges` is to inhibit a restart of the activity when the phone's orientation or screen size changes.

For the two Windows Phone projects, the class and enumeration to use is in the `Windows.Graphics.Display` namespace. In the `MainPage` constructor in the MainPage.xaml.cs file, set the static `DisplayInformation.AutoRotationPreferences` property to one or more members of the `DisplayOrientations` enumeration combined with the C# bitwise OR operation. To restrict the program to landscape or portrait, use:

```
DisplayInformation.AutoRotationPreferences = DisplayOrientations.Landscape
```

or:

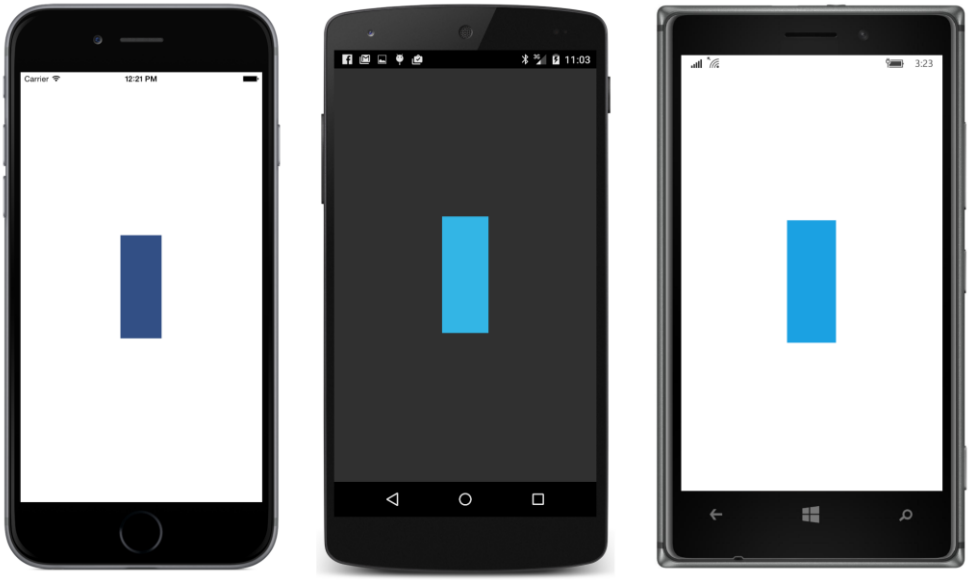
```
DisplayInformation.AutoRotationPreferences = DisplayOrientations.Portrait;
```

Metrical sizes

Now that you know how sizes in a Xamarin.Forms application approximately correspond to metrical dimensions of inches and centimeters, you can size elements so that they are approximately the same size on various devices. Here's a program called **MetricalBoxView** that displays a `BoxView` with a width of approximately one centimeter and a height of approximately one inch:

```
public class MetricalBoxViewPage : ContentPage
{
    public MetricalBoxViewPage()
    {
        Content = new BoxView
        {
            Color = Color.Accent,
            WidthRequest = 64,
            HeightRequest = 160,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
    }
}
```

If you actually take a ruler to the object on your phone's screen, you'll find that it's not exactly the desired size but certainly close to it, as these screenshots also confirm:



This program is intended to run on phones. If you want to run it on tablets as well, you might use the `Device.Idiom` property to set a somewhat smaller factor for the iPad and Windows tablets.

Estimated font sizes

The `FontSize` property on `Label` and `Button` specifies the approximate height of font characters from the bottom of descenders to the top of ascenders, often (depending on the font) including diacritical marks as well. In most cases you'll want to set this property to a value returned by the `Device.GetNamedSize` method. This allows you to specify a member of the `NamedSize` enumeration: `Default`, `Micro`, `Small`, `Medium`, or `Large`.

Alternatively, you can set the `FontSize` property to actual numeric font sizes, but there's a little problem involved (to be discussed in detail shortly). For the most part, you specify font sizes in the same device-independent units used throughout Xamarin.Forms, which means that you can calculate device-independent font sizes based on the platform resolution.

For example, suppose you want to use a 12-point font in your program. The first thing you should know is that while a 12-point font might be a comfortable size for printed material or a desktop screen, on a phone it's quite large. But let's continue.

There are 72 points to the inch, so a 12-point font is one-sixth of an inch. Multiply by the DPI resolution of 160 and that's about 27 device-independent units.

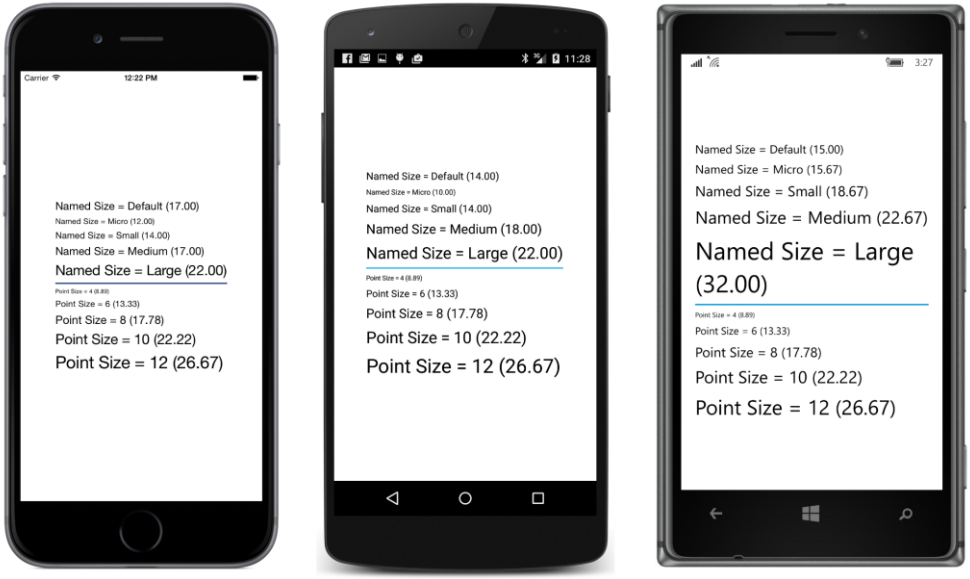
Let's write a little program called **FontSizes**, which begins with a display similar to the **NamedFontSizes** program in Chapter 3 but then displays some text with numeric point sizes, converted to device-independent units using the device resolution:

```
public class FontSizesPage : ContentPage
{
    public FontSizesPage()
    {
        BackgroundColor = Color.White;
        StackLayout stackLayout = new StackLayout
        {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        // Do the NamedSize values.
        NamedSize[] namedSizes =
        {
            NamedSize.Default, NamedSize.Micro, NamedSize.Small,
            NamedSize.Medium, NamedSize.Large
        };

        foreach (NamedSize namedSize in namedSizes)
        {
            double fontSize = Device.GetNamedSize(namedSize, typeof(Label));

            stackLayout.Children.Add(new Label
                {
                    Text = String.Format("Named Size = {0} ({1:F2})",
```

```
                                        namedSize, fontSize),
                    FontSize = fontSize,
                    TextColor = Color.Black
                });
        }

        // Resolution in device-independent units per inch.
        double resolution = 160;

        // Draw horizontal separator line.
        stackLayout.Children.Add(
            new BoxView
            {
                Color = Color.Accent,
                HeightRequest = resolution / 80
            });

        // Do some numeric point sizes.
        int[] ptSizes = { 4, 6, 8, 10, 12 };

        foreach (double ptSize in ptSizes)
        {
            double fontSize = resolution * ptSize / 72;

            stackLayout.Children.Add(new Label
                {
                    Text = String.Format("Point Size = {0} ({1:F2})",
                                         ptSize, fontSize),
                    FontSize = fontSize,
                    TextColor = Color.Black
                });
        }

        Content = stackLayout;
    }
}
```

To facilitate comparisons among the three screens, the backgrounds have been uniformly set to white and the labels to black. Notice the `BoxView` inserted into the `StackLayout` between the two `foreach` blocks: the `HeightRequest` setting gives it a device-independent height of approximately one-eightieth of an inch, and it resembles a horizontal rule.

Interestingly, the resultant visual sizes based on the calculation are more consistent among the platforms than the named sizes. The numbers in parentheses are the numeric `FontSize` values in device-independent units:



Fitting text to available size

You might need to fit a block of text to a particular rectangular area. It's possible to calculate a value for the `FontSize` property of `Label` based on the number of text characters, the size of the rectangular area, and just two numbers.

The first number is line spacing. This is the vertical height of a `Label` view per line of text. For the default fonts associated with the three platforms, it is roughly related to the `FontSize` property as follows:

- iOS: `lineSpacing` = 1.2 * `label.FontSize`
- Android: `lineSpacing` = 1.2 * `label.FontSize`
- Windows Phone: `lineSpacing` = 1.3 * `label.FontSize`

The second helpful number is average character width. For a normal mix of uppercase and lowercase letters for the default fonts, this average character width is about half of the font size, regardless of the platform:

- `averageCharacterWidth` = 0.5 * `label.FontSize`

For example, suppose you want to fit a text string containing 80 characters in a width of 320 units, and you'd like the font size to be as large as possible. Divide the width (320) by half the number of characters (40), and you get a font size of 8, which you can set to the `FontSize` property of `Label`. For

text that's somewhat indeterminate and can't be tested beforehand, you might want to make this calculation a little more conservative to avoid surprises.

The following program uses both line spacing and average character width to fit a paragraph of text on the page, minus the area at the top of the iPhone occupied by the status bar. To make the exclusion of the iOS status bar a bit easier in this program, the program uses a `ContentView`.

`ContentView` derives from `Layout` but only adds a `Content` property to what it inherits from `Layout`. `ContentView` is also the base class to `Frame`. Although `ContentView` has no functionality other than occupying a rectangular area of space, it is useful for two purposes: Most often, `ContentView` can be a parent to other views to define a new custom view. But `ContentView` can also simulate a margin.

As you might have noticed, Xamarin.Forms has no concept of a margin, which traditionally is similar to padding except that padding is inside a view and a part of the view, while a margin is outside the view and actually part of the parent's view. A `ContentView` lets us simulate this. If you find a need to set a margin on a view, put the view in a `ContentView` and set the `Padding` property on the `ContentView`. `ContentView` inherits a `Padding` property from `Layout`.

The **EstimatedFontSize** program uses `ContentView` in a slightly different manner: It sets the customary padding on the page to avoid the iOS status bar, but then it sets a `ContentView` as the content of that page. Hence, this `ContentView` is the same size as the page, but excluding the iOS status bar. It is on this `ContentView` that the `SizeChanged` event is attached, and it is the size of this `ContentView` that is used to calculate the text font size.

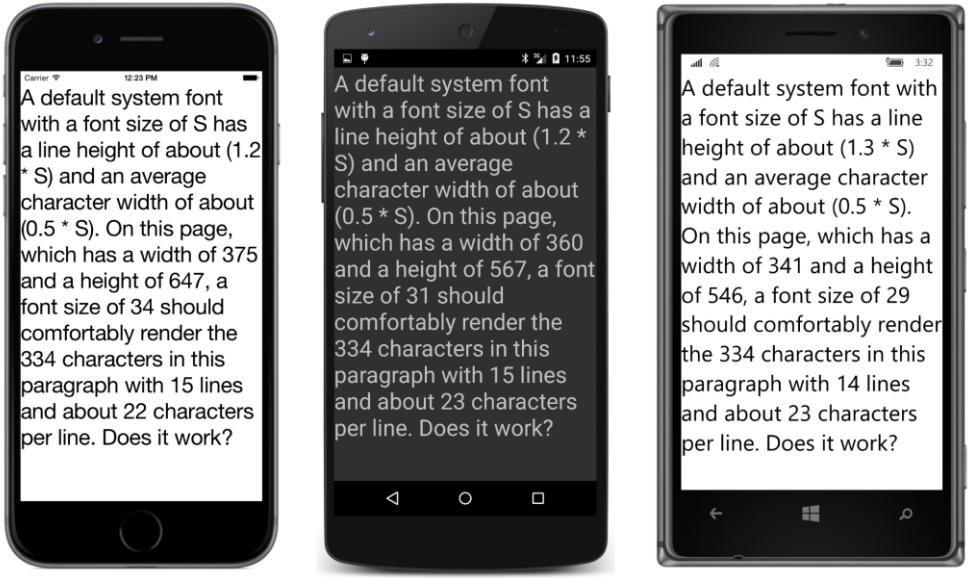
The `SizeChanged` handler uses the first argument to obtain the object firing the event (in this case the `ContentView`), which is the object in which the `Label` must fit. The calculation is described in comments:

```
public class EstimatedFontSizePage : ContentPage
{
    Label label;

    public EstimatedFontSizePage()
    {
        label = new Label();
        Padding = new Thickness(0, Device.OnPlatform(20, 0, 0), 0, 0);
        ContentView contentView = new ContentView
        {
            Content = label
        };
        contentView.SizeChanged += OnContentViewSizeChanged;
        Content = contentView;
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        string text =
            "A default system font with a font size of S " +
```

```
            "has a line height of about ({0:F1} * S) and an " +
            "average character width of about ({1:F1} * S). " +
            "On this page, which has a width of {2:F0} and a " +
            "height of {3:F0}, a font size of ?1 should " +
            "comfortably render the ??2 characters in this " +
            "paragraph with ?3 lines and about ?4 characters " +
            "per line. Does it work?";

        // Get View whose size is changing.
        View view = (View)sender;

        // Define two values as multiples of font size.
        double lineHeight = Device.OnPlatform(1.2, 1.2, 1.3);
        double charWidth = 0.5;

        // Format the text and get its character length.
        text = String.Format(text, lineHeight, charWidth, view.Width, view.Height);
        int charCount = text.Length;

        // Because:
        //   lineCount = view.Height / (lineHeight * fontSize)
        //   charsPerLine = view.Width / (charWidth * fontSize)
        //   charCount = lineCount * charsPerLine
        // Hence, solving for fontSize:
        int fontSize = (int)Math.Sqrt(view.Width * view.Height /
                       (charCount * lineHeight * charWidth));

        // Now these values can be calculated.
        int lineCount = (int)(view.Height / (lineHeight * fontSize));
        int charsPerLine = (int)(view.Width / (charWidth * fontSize));

        // Replace the placeholders with the values.
        text = text.Replace("?1", fontSize.ToString());
        text = text.Replace("??2", charCount.ToString());
        text = text.Replace("?3", lineCount.ToString());
        text = text.Replace("?4", charsPerLine.ToString());

        // Set the Label properties.
        label.Text = text;
        label.FontSize = fontSize;
    }
}
```

The text placeholders named “?1”, “??2”, “?3”, and “?4” were chosen to be unique but also to be the same number of characters as the numbers that replace them.

If the goal is to make the text as large as possible without the text spilling off the page, the results validate the approach:



Not bad. Not bad at all. The text actually displays in one less line that indicated on all three platforms, but the technique seems sound. It's not always the case that the same `FontSize` is calculated for landscape mode, but it happens sometimes:

A fit-to-size clock

The `Device` class includes a static `StartTimer` method that lets you set a timer that fires a periodic event. The availability of a timer event means that a clock application is possible, even if it displays the time only in text.

The first argument to `Device.StartTimer` is an interval expressed as a `TimeSpan` value. The timer fires an event periodically based on that interval. (You can go down as low as 15 or 16 milliseconds, which is about the period of the frame rate of 60 frames per second common on video displays.) The event handler has no arguments but must return `true` to keep the timer going.

The **FitToSizeClock** program creates a `Label` for displaying the time and then sets two events: the `SizeChanged` event on the page for changing the font size, and the `Device.StartTimer` event for one-second intervals to change the `Text` property.

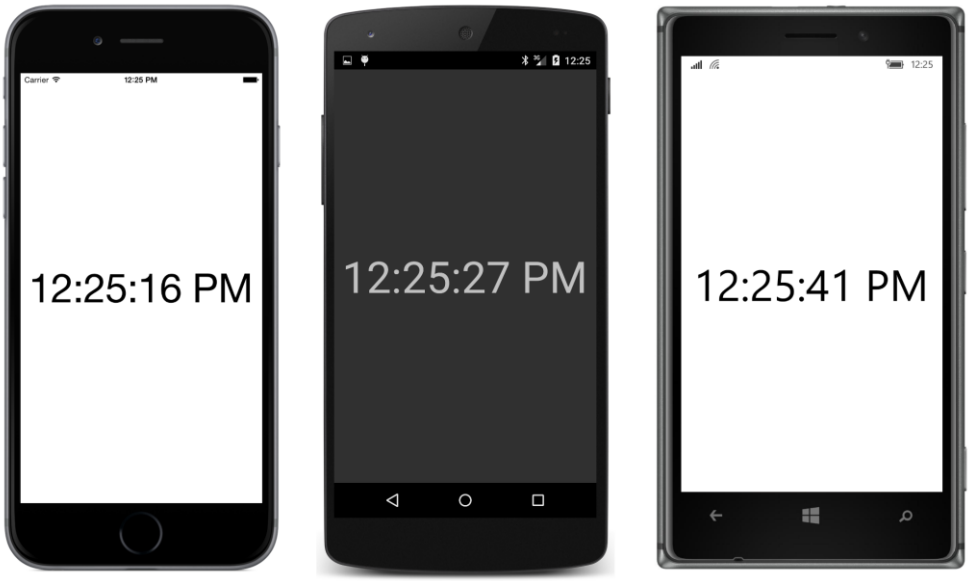
Many C# programmers these days like to define small event handlers as anonymous lambda functions. This allows the event-handling code to be very close to the instantiation and initialization of the object firing the event instead of somewhere else in the file. It also allows referencing objects within the event handler without storing those objects as fields.

In this program, both event handlers simply change a property of the `Label`, and they are both expressed as lambda functions so that they can access the `Label` without it being stored as a field:

```
public class FitToSizeClockPage : ContentPage
{
    public FitToSizeClockPage()
    {
        Label clockLabel = new Label
        {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        Content = clockLabel;

        // Handle the SizeChanged event for the page.
        SizeChanged += (object sender, EventArgs args) =>
            {
                // Scale the font size to the page width
                //      (based on 11 characters in the displayed string).
                if (this.Width > 0)
                    clockLabel.FontSize = this.Width / 6;
            };

        // Start the timer going.
        Device.StartTimer(TimeSpan.FromSeconds(1), () =>
            {
                // Set the Text property of the Label.
                clockLabel.Text = DateTime.Now.ToString("h:mm:ss tt");
```

```
                return true;
            });
    }
}
```

The `StartTimer` handler specifies a custom formatting string for `DateTime` that results in 10 or 11 characters, but two of those are capital letters, and those are wider than average characters. The `SizeChanged` handler implicitly assumes that 12 characters are displayed by setting the font size to one-sixth of the page width:



Of course, the text is much larger in landscape mode:

This one-second timer doesn't tick exactly at the beginning of every second, so the displayed time might not precisely agree with other time displays on the same device. You can make it more accurate by setting a more frequent timer tick. Performance won't be impacted much because the display still changes only once per second and won't require a new layout cycle until then.

Accessibility issues

The **EstimatedFontSize** program and the **FitToSizeClock** program both have a subtle flaw, but the problem might not be so subtle if you're one of the many people who can't comfortably read text on a mobile device and uses the device's accessibility features to make the text larger.

On iOS, run the **Settings** app, and choose **General**, and **Accessibility**, and **Larger Text**. You can then use a slider to make text on the screen larger or smaller. The page indicates that text will only be adjusted in iOS applications that support the **Dynamic Type** feature.

On Android, run the **Settings** app, and choose **Display** and then **Font size**. You are presented with four radio buttons for selecting **Small**, **Normal** (the default), **Large**, or **Huge**.

On a Windows 10 Mobile device, run the **Settings** app, and choose **Ease of Access** and then **More options**. You can then move a slider labeled **Text scaling** from 100% to 200%.

Here's what you will discover:

The iOS setting has no effect on Xamarin.Forms applications.

The Android setting affects the values returned from `Device.GetNamedSize`. If you select something other than **Normal** and run the **FontSizes** program again, you'll see that for the `NamedSize.Default` argument, `Device.GetNamedSize` returns 14 when the setting is **Normal** (as the earlier screenshot shows), but returns 12 for a setting of **Small**, 16 for **Large**, and 18 1/3 for **Huge.**

Also, *all* the text displayed on the Android screen is a different size—either smaller or larger depending on what setting you selected—even for constant `FontSize` values.

On Windows 10 Mobile, the values returned from `Device.GetNamedSize` do not depend on the accessibility setting, but all the text is displayed larger.

This means that the **EstimatedFontSize** or **FitToSizeClock** programs do not run correctly on Android or Windows 10 Mobile with the accessibility setting for larger text. Part of the text is truncated.

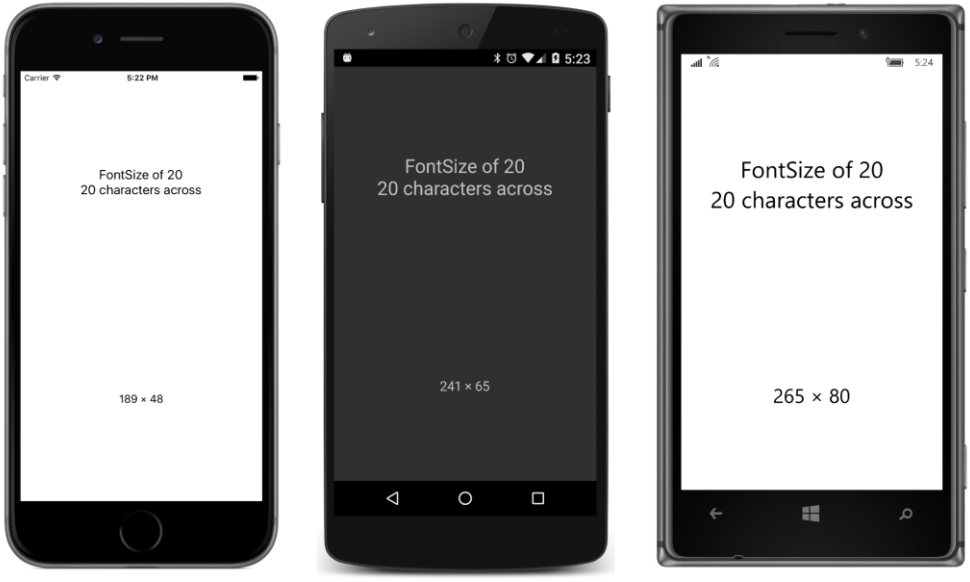
Let's explore this a little more. The **AccessibilityTest** program displays two `Label` elements on its page. The first has a constant `FontSize` of 20, and the second merely displays the size of the first `Label` when its size changes:

```
public class AccessibilityTestPage : ContentPage
{
    public AccessibilityTestPage()
    {
        Label testLabel = new Label
        {
            Text = "FontSize of 20" + Environment.NewLine + "20 characters across",
            FontSize = 20,
            HorizontalTextAlignment = TextAlignment.Center,
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions =  LayoutOptions.CenterAndExpand
        };

        Label displayLabel = new Label
        {
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        testLabel.SizeChanged += (sender, args) =>
        {
            displayLabel.Text = String.Format("{0:F0} \u00D7 {1:F0}", testLabel.Width,
                                                                      testLabel.Height);
        };

        Content = new StackLayout
        {
            Children =
            {
                testLabel,
                displayLabel
            }
        };
    }
```

```
}
```

Normally, the second `Label` displays a size that is roughly consistent with the assumptions described earlier:

But now go into the accessibility settings and crank them all the way up. Both Android and Windows 10 Mobile display larger text:



The character size assumptions described earlier are no longer valid, and that's why the programs fail to fit the text.

But there is an alternative approach to sizing text to a rectangular area.

Empirically fitting text

Another approach to fitting text within a rectangle of a particular size involves empirically determining the size of the rendered text based on a particular font size and then adjusting that font size up or down. This approach has the advantage of working on all devices regardless of the accessibility settings.

But the process can be tricky: The first problem is that there is not a clean linear relationship between the font size and the height of the rendered text. As text gets larger relative to the width of its container, more line breaks result, with more wasted space. A calculation to find the optimum font size often involves a loop that narrows in on the value.

A second problem involves the actual mechanism of obtaining the size of a `Label` rendered with a particular font size. You can set a `SizeChanged` handler on the `Label`, but within that handler you don't want to make any changes (such as setting a new `FontSize` property) that will cause recursive calls to that handler.

A better approach is calling the `GetSizeRequest` method defined by `VisualElement` and inherited by `Label` and all other views. `GetSizeRequest` requires two arguments—a width constraint and a height constraint. These values indicate the size of the rectangle in which you want to fit the element, and one or the other can be infinity. When using `GetSizeRequest` with a `Label`, generally you set the width constraint argument to the width of the container and the height constraint to `Double.PositiveInfinity`.

The `GetSizeRequest` method returns a value of type `SizeRequest`, a structure with two properties, named `Request` and `Minimum`, both of type `Size`. The `Request` property indicates the size of the rendered text. (More information on this and related methods can be found in Chapter 26.)

The **EmpiricalFontSize** project demonstrates this technique. For convenience, it defines a small structure named `FontCalc` whose constructor makes the call to `GetSizeRequest` for a particular `Label` (already initialized with text), a trial font size, and a text width:

```
struct FontCalc
{
    public FontCalc(Label label, double fontSize, double containerWidth)
        : this()
    {
        // Save the font size.
        FontSize = fontSize;

        // Recalculate the Label height.
```

```
        label.FontSize = fontSize;
        SizeRequest sizeRequest =
            label.GetSizeRequest(containerWidth, Double.PositiveInfinity);

        // Save that height.
        TextHeight = sizeRequest.Request.Height;
    }

    public double FontSize { private set; get; }

    public double TextHeight { private set; get; }
}
```

The resultant height of the rendered `Label` is saved in the `TextHeight` property.

When you make a call to `GetSizeRequest` on a page or a layout, the page or layout needs to obtain the sizes of all its children down through the visual tree. This has a performance penalty, of course, so you should avoid making calls like that unless necessary. But a `Label` has no children, so calling `GetSizeRequest` on a `Label` is not nearly as bad. However, you should still try to optimize the calls. Avoid looping through a sequential series of font size values to determine the maximum value that doesn't result in text exceeding the container height. A process that algorithmically narrows in on an optimum value is better.

`GetSizeRequest` requires that the element be part of a visual tree and that the layout process has at least partially begun. Don't call `GetSizeRequest` in the constructor of your page class. You won't get information from it. The first reasonable opportunity is in an override of the page's `OnAppearing` method. Of course, you might not have sufficient information at this time to pass arguments to the `GetSizeRequest` method.

However, calling `GetSizeRequest` doesn't have any side effects. It doesn't cause a new size to be set on the element, which means that it doesn't cause a `SizeChanged` event to be fired, which means that it's safe to call in a `SizeChanged` handler.

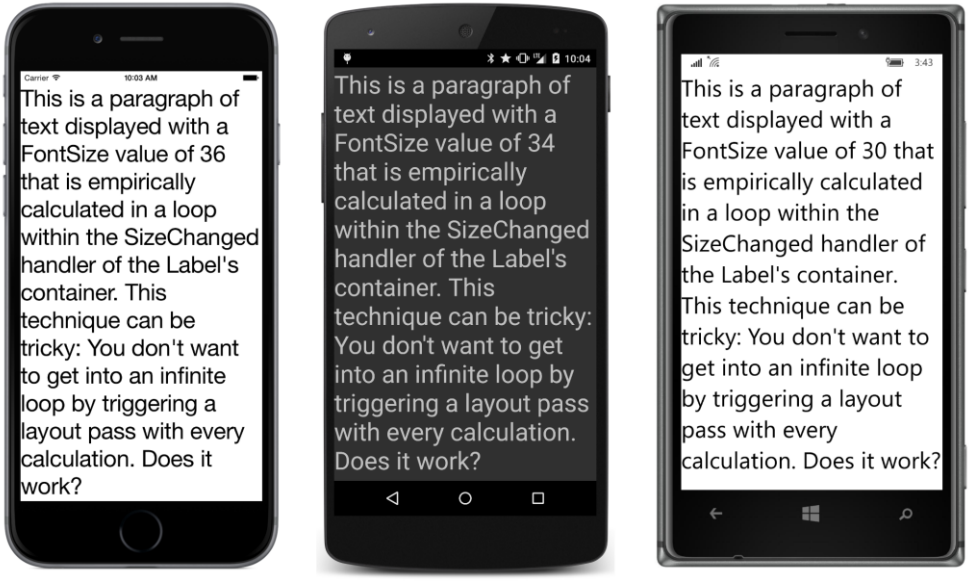
The `EmpiricalFontSizePage` class instantiates `FontCalc` values in the `SizeChanged` handler of the `ContentView` that hosts the `Label`. The constructor of each `FontCalc` value makes `GetSizeRequest` calls on the `Label` and saves the resultant `TextHeight`. The `SizeChanged` handler begins with trial font sizes of 10 and 100 under the assumption that the optimum value is somewhere between these two and that these represent lower and upper bounds. Hence the variable names `lowerFontCalc` and `upperFontCalc`:

```
public class EmpiricalFontSizePage : ContentPage
{
    Label label;

    public EmpiricalFontSizePage()
    {
        label = new Label();

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

```
        ContentView contentView = new ContentView
        {
            Content = label
        };
        contentView.SizeChanged += OnContentViewSizeChanged;
        Content = contentView;
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        // Get View whose size is changing.
        View view = (View)sender;

        if (view.Width <= 0 || view.Height <= 0)
            return;

        label.Text =
            "This is a paragraph of text displayed with " +
            "a FontSize value of ?? that is empirically " +
            "calculated in a loop within the SizeChanged " +
            "handler of the Label's container. This technique " +
            "can be tricky: You don't want to get into " +
            "an infinite loop by triggering a layout pass " +
            "with every calculation. Does it work?";

        // Calculate the height of the rendered text.
        FontCalc lowerFontCalc = new FontCalc(label, 10, view.Width);
        FontCalc upperFontCalc = new FontCalc(label, 100, view.Width);

        while (upperFontCalc.FontSize - lowerFontCalc.FontSize > 1)
        {
            // Get the average font size of the upper and lower bounds.
            double fontSize = (lowerFontCalc.FontSize + upperFontCalc.FontSize) / 2;

            // Check the new text height against the container height.
            FontCalc newFontCalc = new FontCalc(label, fontSize, view.Width);

            if (newFontCalc.TextHeight > view.Height)
            {
                upperFontCalc = newFontCalc;
            }
            else
            {
                lowerFontCalc = newFontCalc;
            }
        }

        // Set the final font size and the text with the embedded value.
        label.FontSize = lowerFontCalc.FontSize;
        label.Text = label.Text.Replace("??", label.FontSize.ToString("F0"));
    }
}
```

In each iteration of the `while` loop, the `FontSize` properties of those two `FontCalc` values are averaged and a new `FontCalc` is obtained. This becomes the new `lowerFontCalc` or `upperFontCalc` value depending on the height of the rendered text. The loop ends when the calculated font size is within one unit of the optimum value.

About seven iterations of the loop are sufficient to get a value that is clearly better than the estimated value calculated in the earlier program:



Turning the phone sideways triggers another recalculation that results in a similar (though not necessarily the same) font size:

It might seem that the algorithm could be improved beyond simply averaging the `FontSize` properties from the lower and upper `FontCalc` values. But the relationship between the font size and rendered text height is rather complex, and sometimes the easiest approach is just as good.