Chapter 14

Absolute layout

In Xamarin.Forms, the concept of layout encompasses all the ways that various views can be assembled on the screen. Here's the class hierarchy showing all the classes that derive from `Layout`:

```
System.Object
    BindableObject
        Element
            VisualElement
                View
                    Layout
                        ContentView
                            Frame
                        ScrollView
                        Layout<T>
                            AbsoluteLayout
                            Grid
                            RelativeLayout
                            StackLayout
```

You've already seen `ContentView`, `Frame`, and `ScrollView` (all of which have a `Content` property that you can set to one child), and you've seen `StackLayout`, which inherits a `Children` property from `Layout<T>` and displays its children in a vertical or horizontal stack. The `Grid` and `RelativeLayout` implement somewhat complex layout models and are explored in future chapters. `AbsoluteLayout` is the subject of this chapter.

At first, the `AbsoluteLayout` class seems to implement a rather primitive layout model—one that harks back to the not-so-good old days of graphical user interfaces when programmers were required to individually size and position every element on the screen. Yet, you'll discover that `AbsoluteLayout` also incorporates a proportional positioning and sizing feature that helps brings this ancient layout model into the modern age.

With `AbsoluteLayout`, many of the rules about layout that you've learned so far no longer apply: the `HorizontalOptions` and `VerticalOptions` properties that are so important when a `View` is the child of a `ContentPage` or `StackLayout` have absolutely no effect when a `View` is a child of an `AbsoluteLayout`. A program must instead assign to each child of an `AbsoluteLayout` a specific location in device-independent coordinates. The child can also be assigned a specific size or allowed to size itself.

You can use `AbsoluteLayout` either in code or in XAML. Either way, you'll encounter a feature you

haven't seen yet that is another part of the support provided by `BindableObject` and `BindableProperty`. This new feature is the *attached bindable property*. This is a special type of bindable property that is defined by one class (in this case the `AbsoluteLayout`) but which is set on other objects (the children of the `AbsoluteLayout`).

AbsoluteLayout in code

You can add a child view to the `Children` collection of an `AbsoluteLayout` the same way as with `StackLayout`:

```
absoluteLayout.Children.Add(child);
```

However, you also have other options. The `AbsoluteLayout` class redefines its `Children` property to be of type `AbsoluteLayout.IAbsoluteList<View>`, which includes two additional `Add` methods that allow you to specify the position of the child relative to the upper-left corner of the `AbsoluteLayout`. You can optionally specify the child's size.

To specify both the position and size, you use a `Rectangle` value. `Rectangle` is a structure, and you can create a `Rectangle` value with a constructor that accepts `Point` and `Size` values:

```
Point point = new Point(x, y);
Size size = new Size(width, height);
Rectangle rect = new Rectangle(point, size);
```

Or you can pass the `x`, `y`, `width`, and `height` arguments directly to a `Rectangle` constructor:

```
Rectangle rect = new Rectangle(x, y, width, height);
```

You can then use an alternative `Add` method to add a view to the `Children` collection of the `AbsoluteLayout`:

```
absoluteLayout.Children.Add(child, rect);
```

The `x` and `y` values indicate the position of the upper-left corner of the child view relative to the upper-left corner of the `AbsoluteLayout` parent in device-independent coordinates. If you prefer the child to size itself, you can use just a `Point` value with no `Size` value:

```
absoluteLayout.Children.Add(child, point);
```

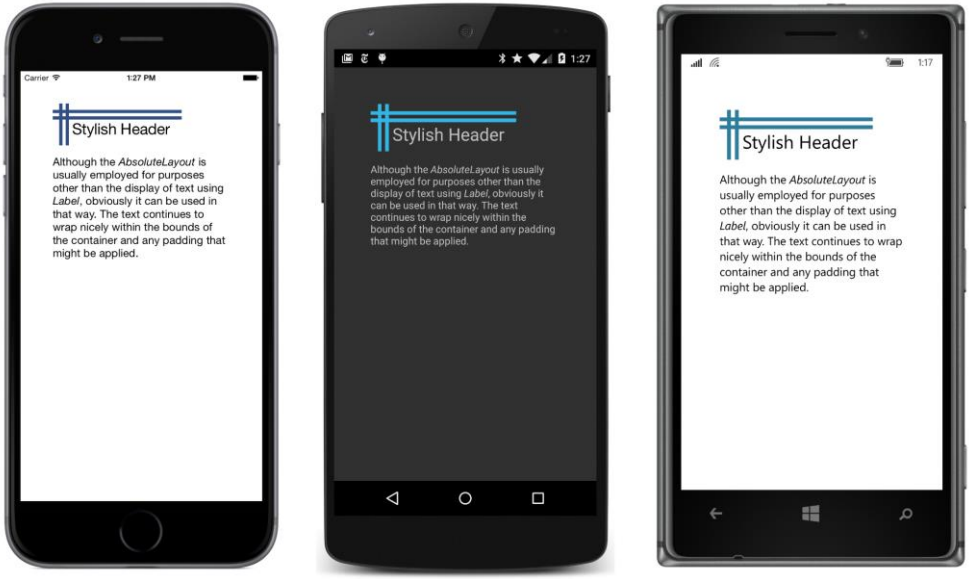
Here's a little demo in a program named **AbsoluteDemo**:

```
public class AbsoluteDemoPage : ContentPage
{
    public AbsoluteDemoPage()
    {
        AbsoluteLayout absoluteLayout = new AbsoluteLayout
        {
            Padding = new Thickness(50)
        };
```

```
absoluteLayout.Children.Add(
    new BoxView
    {
        Color = Color.Accent
    },
    new Rectangle(0, 10, 200, 5));

absoluteLayout.Children.Add(
    new BoxView
    {
        Color = Color.Accent
    },
    new Rectangle(0, 20, 200, 5));

absoluteLayout.Children.Add(
    new BoxView
    {
        Color = Color.Accent
    },
    new Rectangle(10, 0, 5, 65));

absoluteLayout.Children.Add(
    new BoxView
    {
        Color = Color.Accent
    },
    new Rectangle(20, 0, 5, 65));

absoluteLayout.Children.Add(
    new Label
    {
        Text = "Stylish Header",
        FontSize = 24
    },
    new Point(30, 25));

absoluteLayout.Children.Add(
    new Label
    {
        FormattedText = new FormattedString
        {
            Spans =
            {
                new Span
                {
                    Text = "Although the "
                },
                new Span
                {
                    Text = "AbsoluteLayout",
                    FontAttributes = FontAttributes.Italic
                },
                new Span
```

```
                        {
                            Text = " is usually employed for purposes other " +
                                   "than the display of text using "
                        },
                        new Span
                        {
                            Text = "Label",
                            FontAttributes = FontAttributes.Italic
                        },
                        new Span
                        {
                            Text = ", obviously it can be used in that way. " +
                                   "The text continues to wrap nicely " +
                                   "within the bounds of the container " +
                                   "and any padding that might be applied."
                        }
                    }
                }
            },
            new Point(0, 80));

        this.Content = absoluteLayout;
    }
}
```

Four `BoxView` elements form an overlapping crisscross pattern on the top to set off a header, and then a paragraph of text follows. The program positions and sizes all the `BoxView` elements, while it merely positions the two `Label` views because they size themselves:



A little trial and error was required to get the sizes of the four `BoxView` elements and the header

text to be approximately the same size. But notice that the `BoxView` elements overlap: `AbsoluteLayout` allows you to overlap views in a very freeform way that's simply impossible with `StackLayout` (or without using transforms, which are covered in a later chapter).

The big drawback of `AbsoluteLayout` is that you need to come up with the positioning coordinates yourself or calculate them at run time. Anything not explicitly sized—such as the two `Label` views—will calculate a size for itself when the page is laid out. But that size is not available until then. If you wanted to add another paragraph after the second `Label`, what coordinates would you use?

Actually, you can position multiple paragraphs of text by putting a `StackLayout` (or a `StackLayout` inside a `ScrollView`) in the `AbsoluteLayout` and then putting the `Label` views in that. Layouts can be nested.

As you can surmise, using `AbsoluteLayout` is more difficult than using `StackLayout`. In general it's much easier to let Xamarin.Forms and the other `Layout` classes handle much of the complexity of layout for you. But for some special uses, `AbsoluteLayout` is ideal.

Like all visual elements, `AbsoluteLayout` has its `HorizontalOptions` and `VerticalOptions` properties set to `Fill` by default, which means that `AbsoluteLayout` fills its container. With other settings of `HorizontalOptions` and `VerticalOptions`, an `AbsoluteLayout` sizes itself to the size of its contents, but there are some exceptions: Try giving the `AbsoluteLayout` in the **AbsoluteDemo** program a `BackgroundColor` so that you can see exactly the space it occupies on the screen. It normally fills the whole page, but if you set the `HorizontalOptions` and `VerticalOptions` properties of the `AbsoluteLayout` to `Center`, you'll see that the size that the `AbsoluteLayout` computes for itself includes the contents and padding but only one line of the paragraph of text.

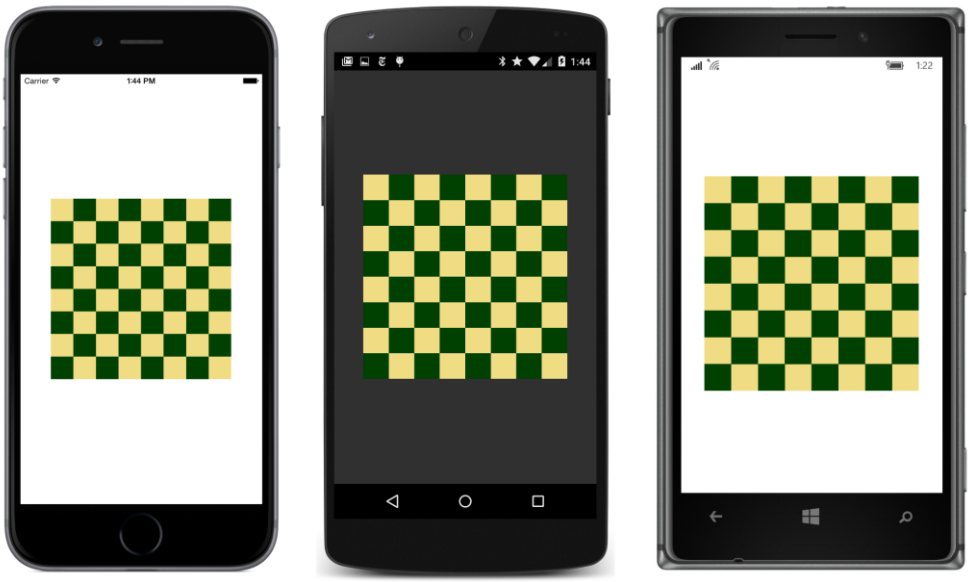
Figuring out sizes for visual elements in an `AbsoluteLayout` can be tricky. One simple approach is demonstrated by the **ChessboardFixed** program below. The program name has the suffix **Fixed** because the position and size of all the squares within the chessboard are set in the constructor. The constructor cannot anticipate the size of the screen, so it arbitrarily sets the size of each square to 35 units, as indicated by the `squareSize` constant at the top of the class. This value should be sufficiently small for the chessboard to fit on the screen of any device supported by Xamarin.Forms.

Notice that the `AbsoluteLayout` is centered so it will have a size that accommodates all its children. The board itself is given a color of buff, which is a pale yellow-brown, and then 32 dark-green `BoxView` elements are displayed in every other square position:

```
public class ChessboardFixedPage : ContentPage
{
    public ChessboardFixedPage()
    {
        const double squareSize = 35;

        AbsoluteLayout absoluteLayout = new AbsoluteLayout
        {
            BackgroundColor = Color.FromRgb(240, 220, 130),
            HorizontalOptions = LayoutOptions.Center,
```

```
            VerticalOptions = LayoutOptions.Center
        };

        for (int row = 0; row < 8; row++)
        {
            for (int col = 0; col < 8; col++)
            {
                // Skip every other square.
                if (((row ^ col) & 1) == 0)
                    continue;

                BoxView boxView = new BoxView
                    {
                        Color = Color.FromRgb(0, 64, 0)
                    };

                Rectangle rect = new Rectangle(col * squareSize,
                                               row * squareSize,
                                               squareSize, squareSize);

                absoluteLayout.Children.Add(boxView, rect);
            }
        }
        this.Content = absoluteLayout;
    }
}
```

The exclusive-or calculation on the `row` and `col` variables causes a `BoxView` to be created only when either the `row` or `col` variable is odd but both are not odd. Here's the result:



Attached bindable properties

If we wanted this chessboard to be as large as possible within the confines of the screen, we'd need to add the `BoxView` elements to the `AbsoluteLayout` during the `SizeChanged` handler for the page, or the `SizeChanged` handler would need to find some way to change the position and size of the `BoxView` elements already in the `Children` collection.

Both options are possible, but the second one is preferred because we can fill the `Children` collection of the `AbsoluteLayout` only once in the program's constructor and then adjust the sizes and position later.

At first encounter, the syntax that allows you to set the position and size of a child already in an `AbsoluteLayout` might seem somewhat odd. If `view` is an object of type `View` and `rect` is a `Rectangle` value, here's the statement that gives `view` a location and size of `rect`:

```
AbsoluteLayout.SetLayoutBounds(view, rect);
```

That's not an instance of `AbsoluteLayout` on which you're making a `SetLayoutBounds` call. No. That's a static method of the `AbsoluteLayout` class. You can call `AbsoluteLayout.SetLayoutBounds` either before or after you add the `view` child to the `AbsoluteLayout` children collection. Indeed, because it's a static method, you can call the method before the `AbsoluteLayout` has even been instantiated! A particular instance of `AbsoluteLayout` is not involved at all in this `SetLayoutBounds` method.

Let's look at some code that makes use of this mysterious `AbsoluteLayout.SetLayoutBounds` method and then examine how it works.

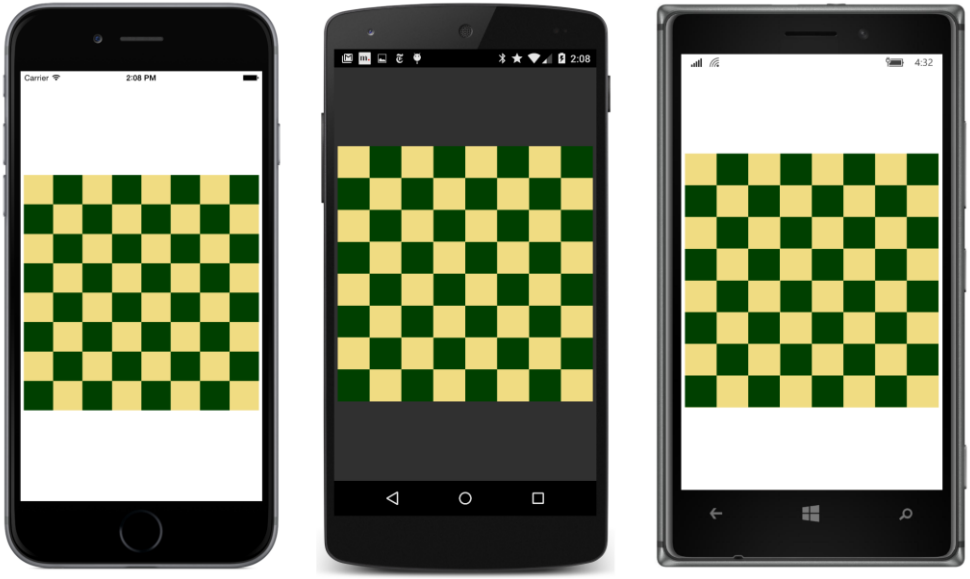
The **ChessboardDynamic** program page constructor uses the simple `Add` method without positioning or sizing to add 32 `BoxView` elements to the `AbsoluteLayout` in one `for` loop. To provide a little margin around the chessboard, the `AbsoluteLayout` is a child of a `ContentView` and padding is set on the page. This `ContentView` has a `SizeChanged` handler to position and size the `AbsoluteLayout` children based on the size of the container:

```
public class ChessboardDynamicPage : ContentPage
{
    AbsoluteLayout absoluteLayout;

    public ChessboardDynamicPage()
    {
        absoluteLayout = new AbsoluteLayout
        {
            BackgroundColor = Color.FromRgb(240, 220, 130),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        for (int i = 0; i < 32; i++)
        {
```

```
            BoxView boxView = new BoxView
                {
                    Color = Color.FromRgb(0, 64, 0)
                };
            absoluteLayout.Children.Add(boxView);
        }

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

        this.Padding = new Thickness(5, Device.OnPlatform(25, 5, 5), 5, 5);
        this.Content = contentView;
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
        double squareSize = Math.Min(contentView.Width, contentView.Height) / 8;
        int index = 0;

        for (int row = 0; row < 8; row++)
        {
            for (int col = 0; col < 8; col++)
            {
                // Skip every other square.
                if (((row ^ col) & 1) == 0)
                    continue;

                View view = absoluteLayout.Children[index];
                Rectangle rect = new Rectangle(col * squareSize,
                                               row * squareSize,
                                               squareSize, squareSize);

                AbsoluteLayout.SetLayoutBounds(view, rect);
                index++;
            }
        }
    }
}
```

The `SizeChanged` handler contains much the same logic as the constructor in **ChessboardFixed** except that the `BoxView` elements are already in the `Children` collection of the `AbsoluteLayout`. All that's necessary is to position and size each `BoxView` when the size of the container changes—for example, during phone orientation changes. The `for` loop concludes with a call to the static `AbsoluteLayout.SetLayoutBounds` method for each `BoxView` with a calculated `Rectangle` value.

Now the chessboard is sized to fit the screen with a little margin:



Obviously, the mysterious `AbsoluteLayout.SetLayoutBounds` method works, but how? What does it do? And how does it manage to do what it does without referencing a particular `AbsoluteLayout` object?

The `AbsoluteLayout.SetLayoutBounds` call that you've just seen looks like this:

```
AbsoluteLayout.SetLayoutBounds(view, rect);
```

That method call is exactly equivalent to the following call on the child view:

```
view.SetValue(AbsoluteLayout.LayoutBoundsProperty, rect);
```

This is a `SetValue` call on the child view. These two method calls are exactly equivalent because the second one is how `AbsoluteLayout` internally defines the `SetLayoutBounds` static method. `AbsoluteLayout.SetLayoutBounds` is merely a shortcut method, and the similar static `AbsoluteLayout.GetLayoutBounds` method is a shortcut for a `GetValue` call.

You'll recall that `SetValue` and `GetValue` are defined by `BindableObject` and used to implement bindable properties. Judging solely from the name, `AbsoluteLayout.LayoutBoundsProperty` certainly appears to be a `BindableProperty` object, and that is so. However, it is a very special type of bindable property called an *attached bindable property*.

Normal bindable properties can be set only on instances of the class that defines the property or on instances of a derived class. Attached bindable properties can break that rule: Attached bindable properties are defined by one class—in this case `AbsoluteLayout`—but set on another object, in this case a child of the `AbsoluteLayout`. The property is sometimes said to be *attached* to the child, hence the name.

The child of the `AbsoluteLayout` is ignorant of the purpose of the attached bindable property passed to its `SetValue` method, and the child makes no use of that value in its own internal logic. The `SetValue` method of the child simply saves the `Rectangle` value in a dictionary maintained by `BindableObject` within the child, in effect attaching this value to the child to be possibly used at some point by the parent—the `AbsoluteLayout` object.

When the `AbsoluteLayout` is laying out its children, it can interrogate the value of this property on each child by calling the `AbsoluteLayout.GetLayoutBounds` static method on the child, which in turn calls `GetValue` on the child with the `AbsoluteLayout.LayoutBoundsProperty` attached bindable property. The call to `GetValue` fetches the `Rectangle` value from the dictionary stored within the child.

You might wonder: Why is such a roundabout process required to set positioning and sizing information on a child of the `AbsoluteLayout`? Wouldn't it have been easier for `View` to define simple `X`, `Y`, `Width`, and `Height` properties that an application could set?

Maybe, but those properties would be suitable only for `AbsoluteLayout`. When using the `Grid`, an application needs to specify `Row` and `Column` values on the children of the `Grid`, and when using a layout class of your own devising, perhaps some other properties are required. Attached bindable properties can handle all these cases and more.

Attached bindable properties are a general-purpose mechanism that allows properties defined by one class to be stored in instances of another class. You can define your own attached bindable properties by using static creation methods of `BindableObject` named `CreateAttached` and `CreateAttachedReadOnly`. (You'll see an example in Chapter 27, "Custom renderers.")

Attached properties are mostly used with layout classes. As you'll see, `Grid` defines attached bindable properties to specify the row and column of each child, and `RelativeLayout` also defines attached bindable properties.

Earlier you saw additional `Add` methods defined by the `Children` collection of `AbsoluteLayout`. These are actually implemented using these attached bindable properties. The call

```
absoluteLayout.Children.Add(view, rect);
```

is implemented like this:

```
AbsoluteLayout.SetLayoutBounds(view, rect);
absoluteLayout.Children.Add(view);
```

The `Add` call with only a `Point` argument merely sets the child's position and lets the child size itself:

```
absoluteLayout.Children.Add(view, new Point(x, y));
```

This is implemented with the same static `AbsoluteLayout.SetLayoutBounds` calls but using a special constant for the view's width and height:

```
AbsoluteLayout.SetLayoutBounds(view,
        new Rectangle(x, y, AbsoluteLayout.AutoSize, AbsoluteLayout.AutoSize));
```

```
absoluteLayout.Children.Add(view);
```

You can use that `AbsoluteLayout.AutoSize` constant in your own code.

Proportional sizing and positioning

As you saw, the **ChessboardDynamic** program repositions and resizes the `BoxView` children with calculations based on the size of the `AbsoluteLayout` itself. In other words, the size and position of each child is proportional to the size of the container. Interestingly, this is often the case with an `AbsoluteLayout`, and it might be nice if `AbsoluteLayout` accommodated such situations automatically.

It does!

`AbsoluteLayout` defines a second attached bindable property, named `LayoutFlagsProperty`, and two more static methods, named `SetLayoutFlags` and `GetLayoutFlags`. Setting this attached bindable property allows you to specify child position coordinates or sizes (or both) that are proportional to the size of the `AbsoluteLayout`. When laying out its children, `AbsoluteLayout` scales those coordinates and sizes appropriately.

You select how this feature works with one or more members of the `AbsoluteLayoutFlags` enumeration:

- `None` (equal to 0)
- `XProportional` (1)
- `YProportional` (2)
- `PositionProportional` (3)
- `WidthProportional` (4)
- `HeightProportional` (8)
- `SizeProportional` (12)
- `All` (\xFFFFFFFF)

You can set a proportional position and size on a child of `AbsoluteLayout` using the two static methods:

```
AbsoluteLayout.SetLayoutBounds(view, rect);
AbsoluteLayout.SetLayoutFlags(view, AbsoluteLayoutFlags.All);
```

Or you can use a version of the `Add` method on the `Children` collection that accepts an `AbsoluteLayoutFlags` enumeration member:

```
absoluteLayout.Children.Add(view, rect, AbsoluteLayoutFlags.All);
```

For example, if you use the `SizeProportional` flag and set the width of the child to 0.25 and the height to 0.10, the child will be one-quarter of the width of the `AbsoluteLayout` and one-tenth the height. Easy enough.

The `PositionProportional` flag is similar, but it takes the size of the child into account: a position of (0, 0) puts the child in the upper-left corner, a position of (1, 1) puts the child in the lower-right corner, and a position of (0.5, 0.5) centers the child within the `AbsoluteLayout`. Taking the size of the child into account is great for some tasks—such as centering a child in an `AbsoluteLayout` or displaying it against the right or bottom edge—but a bit awkward for other tasks.

Here's **ChessboardProportional**. The bulk of the job of positioning and sizing has been moved back to the constructor. The `SizeChanged` handler now merely maintains the overall aspect ratio by setting the `WidthRequest` and `HeightRequest` properties of the `AbsoluteLayout` to the minimum of the width and height of the `ContentView`. Remove that `SizeChanged` handling and the chessboard expands to the size of the page less the padding.

```
public class ChessboardProportionalPage : ContentPage
{
    AbsoluteLayout absoluteLayout;

    public ChessboardProportionalPage()
    {
        absoluteLayout = new AbsoluteLayout
        {
            BackgroundColor = Color.FromRgb(240, 220, 130),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };

        for (int row = 0; row < 8; row++)
        {
            for (int col = 0; col < 8; col++)
            {
                // Skip every other square.
                if (((row ^ col) & 1) == 0)
                    continue;

                BoxView boxView = new BoxView
                {
                    Color = Color.FromRgb(0, 64, 0)
                };

                Rectangle rect = new Rectangle(col / 7.0,    // x
                                               row / 7.0,    // y
                                               1 / 8.0,      // width
                                               1 / 8.0);     // height

                absoluteLayout.Children.Add(boxView, rect, AbsoluteLayoutFlags.All);
            }
        }
```

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

        this.Padding = new Thickness(5, Device.OnPlatform(25, 5, 5), 5, 5);
        this.Content = contentView;
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
        double boardSize = Math.Min(contentView.Width, contentView.Height);
        absoluteLayout.WidthRequest = boardSize;
        absoluteLayout.HeightRequest = boardSize;
    }
}
```

The screen looks the same as the **ChessboardDynamic** program.

Each `BoxView` is added to the `AbsoluteLayout` with the following code. All the denominators are floating-point values, so the results of the divisions are converted to `double`:

```
Rectangle rect = new Rectangle(col / 7.0,   // x
                               row / 7.0,   // y
                               1 / 8.0,     // width
                               1 / 8.0);    // height

absoluteLayout.Children.Add(boxView, rect, AbsoluteLayoutFlags.All);
```

The width and height are always equal to one-eighth the width and height of the `AbsoluteLayout`. That much is clear. But the `row` and `col` variables are divided by 7 (rather than 8) for the relative `x` and `y` coordinates. The `row` and `col` variables in the `for` loops range from 0 through 7. The `row` and `col` values of 0 correspond to left or top, but `row` and `col` values of 7 must map to `x` and `y` coordinates of 1 to position the child against the right or bottom edge.

If you think you might need some solid rules to derive proportional coordinates, read on.

Working with proportional coordinates

Working with proportional positioning in an `AbsoluteLayout` can be tricky. Sometimes you need to compensate for the internal calculation that takes the size into account. For example, you might prefer to specify coordinates so that an X value of 1 means that the left edge of the child is positioned at the right edge of the `AbsoluteLayout`, and you'll need to convert that to a coordinate that `AbsoluteLayout` understands.

In the discussion that follows, a coordinate that does *not* take size into account—a coordinate in

which 1 means that the child is positioned just outside the right or bottom edge of the `AbsoluteLayout`—is referred to as a *fractional* coordinate. The goal of this section is to develop rules for converting a fractional coordinate to a proportional coordinate that you can use with `AbsoluteLayout`. This conversion requires that you know the size of the child view.

Suppose you're putting a view named `child` in an `AbsoluteLayout` named `absoluteLayout`, with a layout bounds rectangle for the child named `layoutBounds`. Let's restrict this analysis to horizontal coordinates and sizes. The process is the same for vertical coordinates and sizes.

This child must first get a width in some way. The child might calculate its own width, or a width in device-independent units might be assigned to it via the `LayoutBounds` attached property. But let's assume that the `AbsoluteLayoutFlags.WidthProportional` flag is set, which means that the width is calculated based on the `Width` field of the layout bounds and the width of the `AbsoluteLayout`:

$$child.Width = layoutBounds.Width * absoluteLayout.Width$$

If the `AbsoluteLayoutFlags.XProportional` flag is also set, then internally the `AbsoluteLayout` calculates a coordinate for the child relative to itself by taking the size of the child into account:

$$relativeChildCoordinate.X = (absoluteLayout.Width - child.Width) * layoutBounds.X$$

For example, if the `AbsoluteLayout` has a width of 400, and the child has a width of 100, and `layoutBounds.X` is 0.5, then `relativeChildCoordinate.X` is calculated as 150. This means that the left edge of the child is 150 pixels from the left edge of the parent. That causes the child to be horizontally centered within the `AbsoluteLayout`.

It's also possible to calculate a fractional child coordinate:

$$fractionalChildCoordinate.X = \frac{relativeChildCoordinate.X}{absoluteLayout.Width}$$

This is not the same as the proportional coordinate because a fractional child coordinate of 1 means that the child's left edge is just outside the right edge of the `AbsoluteLayout`, and hence the child is outside the surface of the `AbsoluteLayout`. To continue the example, the fractional child coordinate is 150 divided by 400 or 0.375. The left of the child view is positioned at (0.375 * 400) or 150 units from the left edge of the `AbsoluteLayout`.

Let's rearrange the terms of the formula that calculates the relative child coordinate to solve for `layoutBounds.X`:

$$layoutBounds.X = \frac{relativeChildCoordinate.X}{(absoluteLayout.Width - child.Width)}$$

And let's divide both the top and bottom of that ratio by the width of the `AbsoluteLayout`:

$$layoutBounds.X = \frac{fractionalChildCoordinate.X}{\left(1 - \frac{child.Width}{absoluteLayout.Width}\right)}$$

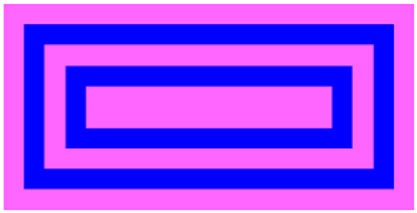
If you're also using proportional width, then that ratio in the denominator is `layoutBounds.Width`:

$$layoutBounds.X = \frac{fractionalChildCoordinate.X}{(1 - \ layoutBounds.Width)}$$

And that is often a very handy formula, for it allows you to convert from a fractional child coordinate to a proportional coordinate for use in the layout bounds rectangle.

In the **ChessboardProportional** example, when `col` equals 7, the `fractionalChildCoordinate.X` is 7 divided by the number of columns (8), or 7/8. The denominator is 1 minus 1/8 (the proportional width of the square), or 7/8 again. The ratio is 1.

Let's look at an example where the formula is applied in code to fractional coordinates. The **ProportionalCoordinateCalc** program attempts to reproduce this simple figure using eight blue `BoxView` elements on a pink `AbsoluteLayout`:



The whole figure has a 2:1 aspect. You can think of the figure as comprising four horizontal rectangles and four vertical rectangles. The pairs of horizontal blue rectangles at the top and bottom have a height of 0.1 fractional units (relative to the height of the `AbsoluteLayout`) and are spaced 0.1 units from the top and bottom and between each other. The vertical blue rectangles appear to be spaced and sized similarly, but because the aspect ratio is 2:1, the vertical rectangles have a width of 0.05 units and are spaced with 0.05 units from the left and right and between each other.

The `AbsoluteLayout` is defined and centered in a XAML file and colored pink:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ProportionalCoordinateCalc.ProportionalCoordinateCalcPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="5, 25, 5, 5"
                    Android="5"
                    WinPhone="5" />
    </ContentPage.Padding>

    <ContentView SizeChanged="OnContentViewSizeChanged">
        <AbsoluteLayout x:Name="absoluteLayout"
                        BackgroundColor="Pink"
                        HorizontalOptions="Center"
                        VerticalOptions="Center" />
    </ContentView>
```

```
</ContentPage>
```

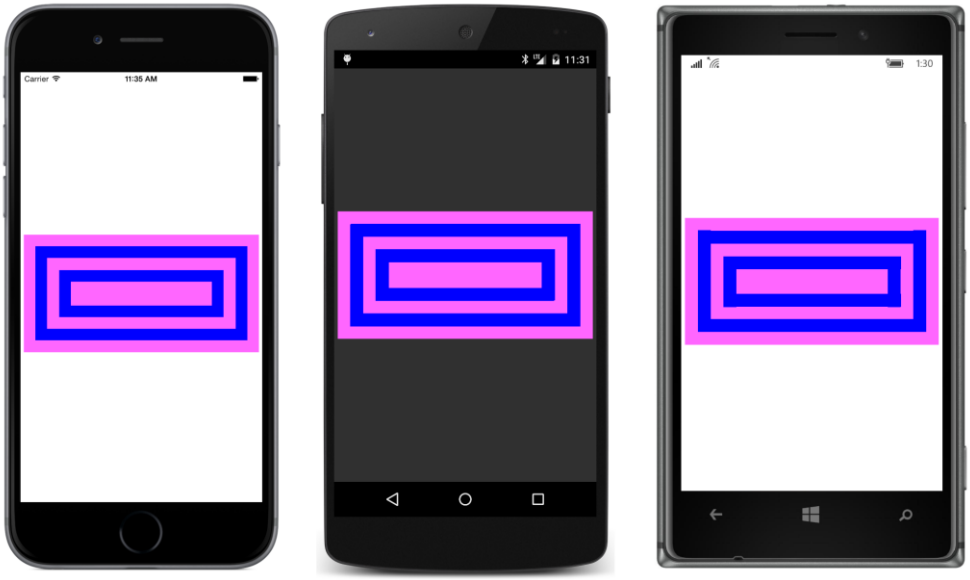
The code-behind file defines an array of `Rectangle` structures with the fractional coordinates for each of the eight `BoxView` elements. In a `foreach` loop, the program applies a slight variation of the final formula shown above. Rather than a denominator equal to 1 minus the value of `layoutBounds.Width` (or `layoutBounds.Height`), it uses the `Width` (or `Height`) of the fractional bounds, which is the same value.

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

        Rectangle[] fractionalRects =
        {
            new Rectangle(0.05, 0.1, 0.90, 0.1),    // outer top
            new Rectangle(0.05, 0.8, 0.90, 0.1),    // outer bottom
            new Rectangle(0.05, 0.1, 0.05, 0.8),    // outer left
            new Rectangle(0.90, 0.1, 0.05, 0.8),    // outer right

            new Rectangle(0.15, 0.3, 0.70, 0.1),    // inner top
            new Rectangle(0.15, 0.6, 0.70, 0.1),    // inner bottom
            new Rectangle(0.15, 0.3, 0.05, 0.4),    // inner left
            new Rectangle(0.80, 0.3, 0.05, 0.4),    // inner right
        };

        foreach (Rectangle fractionalRect in fractionalRects)
        {
            Rectangle layoutBounds = new Rectangle
            {
                // Proportional coordinate calculations.
                X = fractionalRect.X / (1 - fractionalRect.Width),
                Y = fractionalRect.Y / (1 - fractionalRect.Height),

                Width = fractionalRect.Width,
                Height = fractionalRect.Height
            };

            absoluteLayout.Children.Add(
                new BoxView
                {
                    Color = Color.Blue
                },
                layoutBounds,
                AbsoluteLayoutFlags.All);
        }
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
```

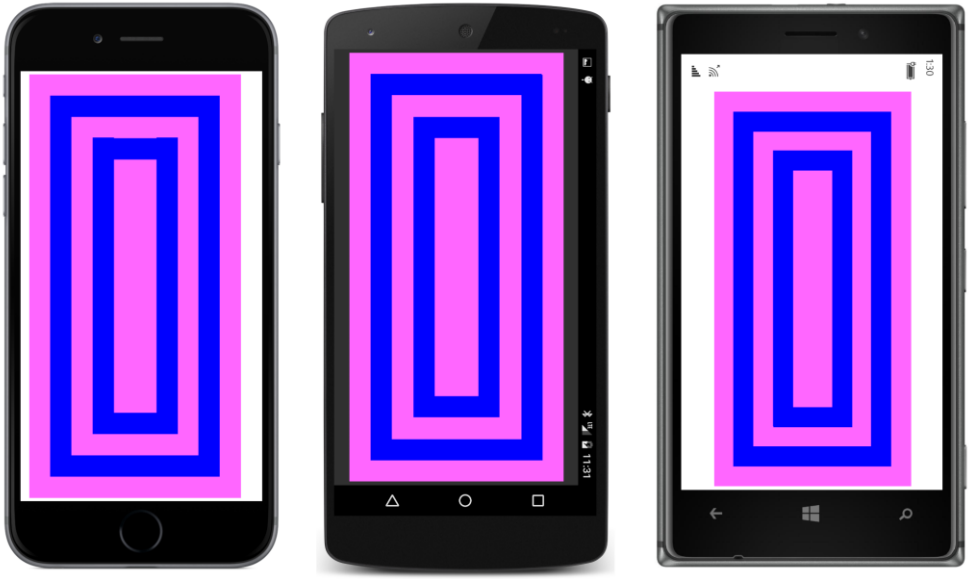
```
        // Figure has an aspect ratio of 2:1.
        double height = Math.Min(contentView.Width / 2, contentView.Height);
        absoluteLayout.WidthRequest = 2 * height;
        absoluteLayout.HeightRequest = height;
    }
}
```

The `SizeChanged` handler simply fixes the aspect ratio.

Here's the result:



And, of course, you can turn the phone sideways and see a larger figure in landscape mode, which you'll have to view by turning this book sideways:



AbsoluteLayout and XAML

As you've seen, you can position and size a child of an `AbsoluteLayout` in code by using one of the `Add` methods available on the `Children` collection or by setting an attached property through a static method call.

But how on earth do you set the position and size of `AbsoluteLayout` children in XAML?

A very special syntax is involved. This syntax is illustrated by this XAML version of the earlier **AbsoluteDemo** program, called **AbsoluteXamlDemo**:

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

    <AbsoluteLayout Padding="50">
        <BoxView Color="Accent"
                 AbsoluteLayout.LayoutBounds="0, 10, 200, 5" />

        <BoxView Color="Accent"
                 AbsoluteLayout.LayoutBounds="0, 20, 200, 5" />

        <BoxView Color="Accent"
                 AbsoluteLayout.LayoutBounds="10, 0, 5, 65" />

        <BoxView Color="Accent"
                 AbsoluteLayout.LayoutBounds="20, 0, 5, 65" />
```

```
        <Label Text="Stylish Header"
               FontSize="24"
               AbsoluteLayout.LayoutBounds="30, 25, AutoSize, AutoSize" />

        <Label AbsoluteLayout.LayoutBounds="0, 80, AutoSize, AutoSize">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Although " />
                    <Span Text="AbsoluteLayout"
                          FontAttributes="Italic" />
                    <Span Text=
" is usually employed for purposes other
than the display of text using " />
                    <Span Text="Label"
                          FontAttributes="Italic" />
                    <Span Text=
", obviously it can be used in that way.
The text continues to wrap nicely
within the bounds of the container
and any padding that might be applied." />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </AbsoluteLayout>
</ContentPage>
```

The code-behind file contains only an `InitializeComponent` call.

Here's the first `BoxView`:

```
<BoxView Color="Accent"
         AbsoluteLayout.LayoutBounds="0, 10, 200, 5" />
```

In XAML, an attached bindable property is expressed as an attribute that consists of a class name (`AbsoluteLayout`) and a property name (`LayoutBounds`) separated by a period. Whenever you see such an attribute, it's always an attached bindable property. That's the only application of this attribute syntax.

In summary, combinations of class names and property names only appear in XAML in three specific contexts: If they appear as elements, they are property elements. If they appear as attributes, they are attached bindable properties. And the only other context for a class name and property name is an argument to an `x:Static` markup extension.

The `AbsoluteLayout.LayoutBounds` attribute is commonly set to four numbers separated by commas. You can also express `AbsoluteLayout.LayoutBounds` as a property element:

```
<BoxView Color="Accent">
    <AbsoluteLayout.LayoutBounds>
        0, 10, 200, 5
    </AbsoluteLayout.LayoutBounds>
</BoxView>
```

Those four numbers are parsed by the `BoundsTypeConverter` and not the `RectangleTypeConverter` because the `BoundsTypeConverter` allows the use of `AutoSize` for the width and height parts. You can see the `AutoSize` arguments later in the **AbsoluteXamlDemo** XAML file:

```
<Label Text="Stylish Header"
       FontSize="24"
       AbsoluteLayout.LayoutBounds="30, 25, AutoSize, AutoSize" />
```

Or you can leave them out:

```
<Label Text="Stylish Header"
       FontSize="24"
       AbsoluteLayout.LayoutBounds="30, 25" />
```

The odd thing about attached bindable properties that you specify in XAML is that they don't really exist! There is no field, property, or method in `AbsoluteLayout` called `LayoutBounds`. There is certainly a public static read-only field of type `BindableProperty` named `LayoutBoundsProperty`, and there are public static methods named `SetLayoutBounds` and `GetLayoutBounds`, but there is nothing named `LayoutBounds`. The XAML parser recognizes the syntax as referring to an attached bindable property and then looks for `LayoutBoundsProperty` in the `AbsoluteLayout` class. From there it can call `SetValue` on the target view with that `BindableProperty` object together with the value from the `BoundsTypeConverter`.

The **Chessboard** series of programs seems an unlikely candidate for duplicating in XAML because the file would need 32 instances of `BoxView` without the benefit of loops. However, the **ChessboardXaml** program shows how to specify two properties of `BoxView` in an implicit style, including the `AbsoluteLayout.LayoutFlags` attached bindable property:

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

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="5, 25, 5, 5"
                    Android="5"
                    WinPhone="5" />
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="Color" Value="#004000" />
                <Setter Property="AbsoluteLayout.LayoutFlags" Value="All" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ContentView SizeChanged="OnContentViewSizeChanged">
        <AbsoluteLayout x:Name="absoluteLayout"
                        BackgroundColor="#F0DC82"
```

```
                    VerticalOptions="Center"
                    HorizontalOptions="Center">

            <BoxView AbsoluteLayout.LayoutBounds="0.00, 0.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.29, 0.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.57, 0.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.86, 0.00, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.14, 0.14, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.43, 0.14, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.71, 0.14, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="1.00, 0.14, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.00, 0.29, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.29, 0.29, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.57, 0.29, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.86, 0.29, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.14, 0.43, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.43, 0.43, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.71, 0.43, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="1.00, 0.43, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.00, 0.57, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.29, 0.57, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.57, 0.57, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.86, 0.57, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.14, 0.71, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.43, 0.71, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.71, 0.71, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="1.00, 0.71, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.00, 0.86, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.29, 0.86, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.57, 0.86, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.86, 0.86, 0.125, 0.125" />

            <BoxView AbsoluteLayout.LayoutBounds="0.14, 1.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.43, 1.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="0.71, 1.00, 0.125, 0.125" />
            <BoxView AbsoluteLayout.LayoutBounds="1.00, 1.00, 0.125, 0.125" />
        </AbsoluteLayout>
    </ContentView>
</ContentPage>
```

Yes, it's a lot of individual `BoxView` elements, but you can't argue with the cleanliness of the file. The code-behind file simply adjusts the aspect ratio:

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

```
    }

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        ContentView contentView = (ContentView)sender;
        double boardSize = Math.Min(contentView.Width, contentView.Height);
        absoluteLayout.WidthRequest = boardSize;
        absoluteLayout.HeightRequest = boardSize;
    }
}
```

Overlays

The ability to overlap children in the `AbsoluteLayout` has some interesting and useful applications, among them being the ability to cover up your entire user interface with something sometimes called an *overlay*. Perhaps your page is carrying out a lengthy job and you don't want the user interacting with the page until the job is completed. You can place a semitransparent overlay over the page and perhaps display an `ActivityIndicator` or a `ProgressBar`.

Here's a program called **SimpleOverlay** that demonstrates this technique. The XAML file begins with an `AbsoluteLayout` filling the entire page. The first child of that `AbsoluteLayout` is a `StackLayout`, which you want to fill the page as well. However, the default `HorizontalOptions` and `VerticalOptions` settings of `Fill` on the `StackLayout` don't work for children of an `AbsoluteLayout`. Instead, the `StackLayout` fills the `AbsoluteLayout` through the use of the `AbsoluteLayout.LayoutBounds` and `AbsoluteLayout.LayoutFlags` attached bindable properties:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SimpleOverlay.SimpleOverlayPage">
    <AbsoluteLayout>
        <StackLayout AbsoluteLayout.LayoutBounds="0, 0, 1, 1"
                     AbsoluteLayout.LayoutFlags="All">
            <Label Text=
"This might be a page full of user-interface objects except
that the only functional user-interface object on the page
is a Button."
                   FontSize="Medium"
                   VerticalOptions="CenterAndExpand"
                   HorizontalTextAlignment="Center" />

            <Button Text="Run 5-Second Job"
                    FontSize="Large"
                    VerticalOptions="CenterAndExpand"
                    HorizontalOptions="Center"
                    Clicked="OnButtonClicked" />

            <Button Text="A Do-Nothing Button"
                    FontSize="Large"
                    VerticalOptions="CenterAndExpand"
```

```
                    HorizontalOptions="Center" />

            <Label Text=
"This continues the page full of user-interface objects except
that the only functional user-interface object on the page
is the Button."
                   FontSize="Medium"
                   VerticalOptions="CenterAndExpand"
                   HorizontalTextAlignment="Center" />
        </StackLayout>

        <!-- Overlay -->
        <ContentView x:Name="overlay"
                     AbsoluteLayout.LayoutBounds="0, 0, 1, 1"
                     AbsoluteLayout.LayoutFlags="All"
                     IsVisible="False"
                     BackgroundColor="#C0808080"
                     Padding="10, 0">

            <ProgressBar x:Name="progressBar"
                         VerticalOptions="Center" />

        </ContentView>
    </AbsoluteLayout>
</ContentPage>
```

The second child of the `AbsoluteLayout` is a `ContentView`, which also fills the `AbsoluteLayout` and basically sits on top of the `StackLayout`. However, notice that the `IsVisible` property is set to `False`, which means that this `ContentView` and its children do not participate in the layout. The `ContentView` is still a child of the `AbsoluteLayout`, but it's simply skipped when the layout system is sizing and rendering all the elements of the page.

This `ContentView` is the overlay. When `IsVisible` is set to `True`, it blocks user input to the views below it. The `BackgroundColor` is set to a semitransparent gray, and a `ProgressBar` is vertically centered within it.

A `ProgressBar` resembles a `Slider` without a thumb. A `ProgressBar` is always horizontally oriented. Do not set the `HorizontalOptions` property of a `ProgressBar` to `Start`, `Center`, or `End` unless you also set its `WidthRequest` property.

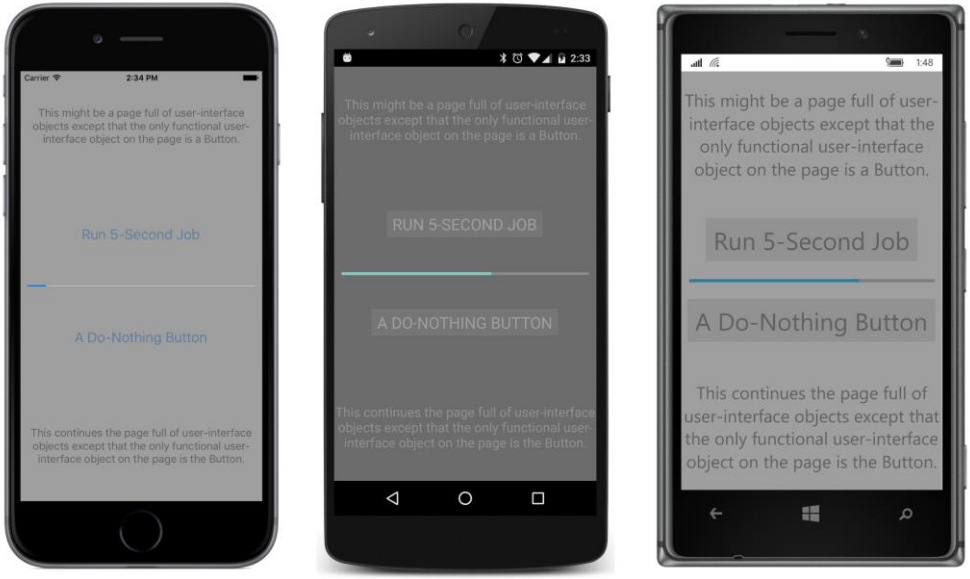
A program can indicate progress by setting the `Progress` property of the `ProgressBar` to a value between 0 and 1. This is demonstrated in the `Clicked` handler for the only functional `Button` in the program. This handler simulates a lengthy job being performed in code with a timer that determines when five seconds have elapsed:

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

```
    void OnButtonClicked(object sender, EventArgs args)
    {
        // Show overlay with ProgressBar.
        overlay.IsVisible = true;

        TimeSpan duration = TimeSpan.FromSeconds(5);
        DateTime startTime = DateTime.Now;

        // Start timer.
        Device.StartTimer(TimeSpan.FromSeconds(0.1), () =>
            {
                double progress = (DateTime.Now - startTime).TotalMilliseconds /
                                  duration.TotalMilliseconds;
                progressBar.Progress = progress;
                bool continueTimer = progress < 1;

                if (!continueTimer)
                {
                    // Hide overlay.
                    overlay.IsVisible = false;
                }
                return continueTimer;
            });
    }
}
```

The `Clicked` handler begins by setting the `IsVisible` property of the overlay to `true`, which reveals the overlay and its child `ProgressBar` and prevents further interaction with the user interface underneath. The timer is set for one-tenth second and calculates a new `Progress` property for the `ProgressBar` based on the elapsed time. When the five seconds are up, the overlay is again hidden and the timer callback returns `false`.

Here's what it looks like with the overlay covering the page and the lengthy job in progress:



An overlay need not be restricted to a `ProgressBar` or an `ActivityIndicator`. You can include a **Cancel** button or other views.

Some fun

As you can probably see by now, the `AbsoluteLayout` is often used for some special purposes that wouldn't be easy otherwise. Some of these might actually be classified as "fun."

DotMatrixClock displays the digits of the current time using a simulated 5 × 7 dot matrix display. Each dot is a `BoxView`, individually sized and positioned on the screen and colored either red or light-gray depending on whether the dot is on or off. Conceivably, the dots of this clock could be organized in nested `StackLayout` elements or a `Grid`, but each `BoxView` needs to be given a size anyway. The sheer quantity and regularity of these views suggests that the programmer knows better than a layout class how to arrange them on the screen, because `StackLayout` and `Grid` need to perform the location calculations in a more generalized manner. For that reason, this is an ideal job for `AbsoluteLayout`.

A XAML file sets a little padding on the page and prepares an `AbsoluteLayout` for filling by code:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DotMatrixClock.DotMatrixClockPage"
             Padding="10"
             SizeChanged="OnPageSizeChanged">

    <AbsoluteLayout x:Name="absoluteLayout"
```

```
                 VerticalOptions="Center" />

</ContentPage>
```

The code-behind file contains several fields, including two arrays, named `numberPatterns` and `colonPattern`, that define the dot matrix patterns for the 10 digits and a colon separator:

```
public partial class DotMatrixClockPage : ContentPage
{
    // Total dots horizontally and vertically.
    const int horzDots = 41;
    const int vertDots = 7;

    // 5 x 7 dot matrix patterns for 0 through 9.
    static readonly int[,,] numberPatterns = new int[10,7,5]
    {
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 1, 1}, { 1, 0, 1, 0, 1},
            { 1, 1, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 0, 0}, { 0, 1, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0},
            { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0},
            { 0, 0, 1, 0, 0}, { 0, 1, 0, 0, 0}, { 1, 1, 1, 1, 1}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 0, 1, 0},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 0, 1, 0}, { 0, 0, 1, 1, 0}, { 0, 1, 0, 1, 0}, { 1, 0, 0, 1, 0},
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 0, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0}, { 0, 0, 0, 0, 1},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 1, 0}, { 0, 1, 0, 0, 0}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0},
            { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 1},
```

```
            { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 1, 1, 0, 0}
        },
    };

    // Dot matrix pattern for a colon.
    static readonly int[,] colonPattern = new int[7, 2]
    {
            { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }
    };

    // BoxView colors for on and off.
    static readonly Color colorOn = Color.Red;
    static readonly Color colorOff = new Color(0.5, 0.5, 0.5, 0.25);

    // Box views for 6 digits, 7 rows, 5 columns.
    BoxView[,,] digitBoxViews = new BoxView[6, 7, 5];
    …
}
```

Fields are also defined for an array of `BoxView` objects for the six digits of the time—two digits each for hour, minutes, and seconds. The total number of dots horizontally (set as `horzDots`) includes five dots for each of the six digits, four dots for the colon between the hour and minutes, four for the colon between the minutes and seconds, and a one dot width between the digits otherwise.

The program's constructor (shown below) creates a total of 238 `BoxView` objects and adds them to an `AbsoluteLayout`, but it also saves the `BoxView` objects for the digits in the `digitBoxViews` array. (In theory, the `BoxView` objects can be referenced later by indexing the `Children` collection of the `AbsoluteLayout`. But in that collection, they appear simply as a linear list. Storing them also in a multidimensional array allows them to be more easily identified and referenced.) All the positioning and sizing is proportional based on an `AbsoluteLayout` that is assumed to have an aspect ratio of 41 to 7, which encompasses the 41 `BoxView` widths and 7 `BoxView` heights.

```
public partial class DotMatrixClockPage : ContentPage
{
    …
    public DotMatrixClockPage()
    {
        InitializeComponent();

        // BoxView dot dimensions.
        double height = 0.85 / vertDots;
        double width = 0.85 / horzDots;

        // Create and assemble the BoxViews.
        double xIncrement = 1.0 / (horzDots - 1);
        double yIncrement = 1.0 / (vertDots - 1);
        double x = 0;

        for (int digit = 0; digit < 6; digit++)
        {
            for (int col = 0; col < 5; col++)
            {
```

```
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the digit BoxView and add to layout.
                    BoxView boxView = new BoxView();
                    digitBoxViews[digit, row, col] = boxView;
                    absoluteLayout.Children.Add(boxView,
                                                new Rectangle(x, y, width, height),
                                                AbsoluteLayoutFlags.All);
                    y += yIncrement;
                }
                x += xIncrement;
            }
            x += xIncrement;

            // Colons between the hour, minutes, and seconds.
            if (digit == 1 || digit == 3)
            {
                int colon = digit / 2;

                for (int col = 0; col < 2; col++)
                {
                    double y = 0;

                    for (int row = 0; row < 7; row++)
                    {
                        // Create the BoxView and set the color.
                        BoxView boxView = new BoxView
                            {
                                Color = colonPattern[row, col] == 1 ?
                                            colorOn : colorOff
                            };
                        absoluteLayout.Children.Add(boxView,
                                                    new Rectangle(x, y, width, height),
                                                    AbsoluteLayoutFlags.All);
                        y += yIncrement;
                    }
                    x += xIncrement;
                }
                x += xIncrement;
            }
        }

        // Set the timer and initialize with a manual call.
        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimer);
        OnTimer();
    }
    …
}
```

As you'll recall, the `horzDots` and `vertDots` constants are set to 41 and 7, respectively. To fill up the `AbsoluteLayout`, each `BoxView` needs to occupy a fraction of the width equal to 1 / `horzDots`

and a fraction of the height equal to 1 / `vertDots`. The height and width set to each `BoxView` is 85 percent of that value to separate the dots enough so that they don't run into each other:

```
double height = 0.85 / vertDots;
double width = 0.85 / horzDots;
```

To position each `BoxView`, the constructor calculates proportional `xIncrement` and `yIncrement` values like so:

```
double xIncrement = 1.0 / (horzDots - 1);
double yIncrement = 1.0 / (vertDots - 1);
```

The denominators here are 40 and 6 so that the final X and Y positional coordinates are values of 1.

The `BoxView` objects for the time digits are not colored at all in the constructor, but those for the two colons are given a `Color` property based on the `colonPattern` array. The `DotMatrixClockPage` constructor concludes by a one-second timer.

The `SizeChanged` handler for the page is set from the XAML file. The `AbsoluteLayout` is automatically stretched horizontally to fill the width of the page (minus the padding), so the `HeightRequest` really just sets the aspect ratio:

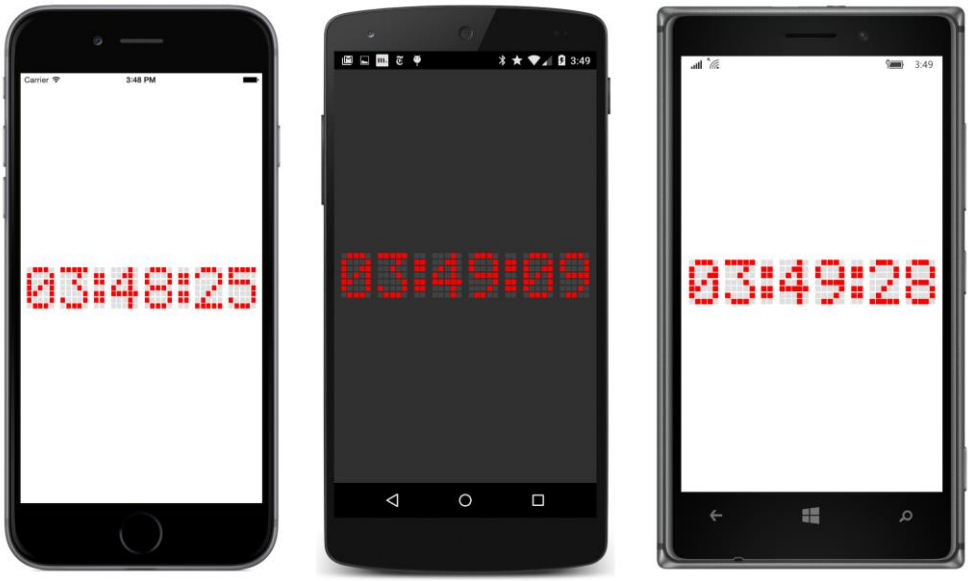
```
public partial class DotMatrixClockPage : ContentPage
{
    …
    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // No chance a display will have an aspect ratio > 41:7
        absoluteLayout.HeightRequest = vertDots * Width / horzDots;
    }
    …
}
```

It seems that the `Device.StartTimer` event handler should be rather complex because it is responsible for setting the `Color` property of each `BoxView` based on the digits of the current time. However, the similarity between the definitions of the `numberPatterns` array and the `digitBoxViews` array makes it surprisingly straightforward:

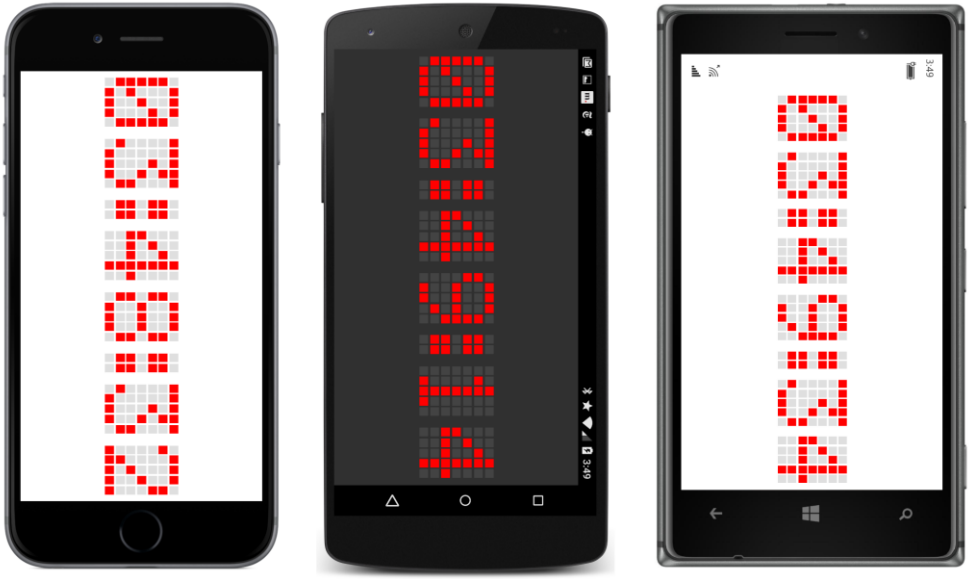
```
public partial class DotMatrixClockPage : ContentPage
{
    …

    bool OnTimer()
    {
        DateTime dateTime = DateTime.Now;

        // Convert 24-hour clock to 12-hour clock.
        int hour = (dateTime.Hour + 11) % 12 + 1;

        // Set the dot colors for each digit separately.
        SetDotMatrix(0, hour / 10);
        SetDotMatrix(1, hour % 10);
```

```
        SetDotMatrix(2, dateTime.Minute / 10);
        SetDotMatrix(3, dateTime.Minute % 10);
        SetDotMatrix(4, dateTime.Second / 10);
        SetDotMatrix(5, dateTime.Second % 10);
        return true;
    }

    void SetDotMatrix(int index, int digit)
    {
        for (int row = 0; row < 7; row++)
            for (int col = 0; col < 5; col++)
            {
                bool isOn = numberPatterns[digit, row, col] == 1;
                Color color = isOn ? colorOn : colorOff;
                digitBoxViews[index, row, col].Color = color;
            }
    }
}
```

And here's the result:



Of course, bigger is better, so you'll probably want to turn the phone (or the book) sideways for something large enough to read from across the room:



Another special type of application suitable for `AbsoluteLayout` is animation. The **BouncingText** program use its XAML file to instantiate two `Label` elements:

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

    <AbsoluteLayout>
        <Label x:Name="label1"
               Text="BOUNCE"
               FontSize="Large"
               AbsoluteLayout.LayoutFlags="PositionProportional" />

        <Label x:Name="label2"
               Text="BOUNCE"
               FontSize="Large"
               AbsoluteLayout.LayoutFlags="PositionProportional" />

    </AbsoluteLayout>
</ContentPage>
```

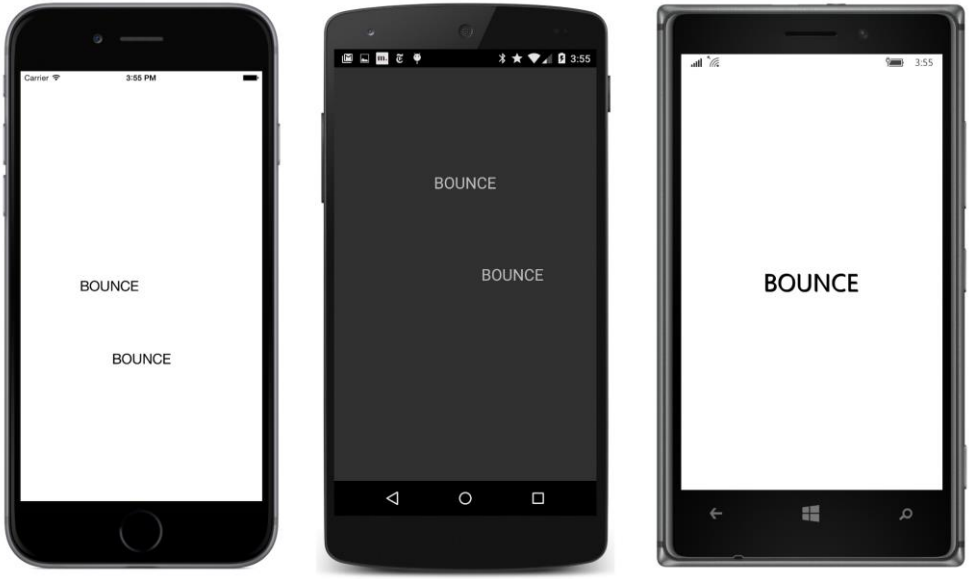
Notice that the `AbsoluteLayout.LayoutFlags` attributes are set to `PositionProportional`. The `Label` calculates its own size, but the positioning is proportional. Values between 0 and 1 can position the two `Label` elements anywhere within the page.

The code-behind file starts a timer going with a 15-millisecond duration. This is equivalent to approximately 60 ticks per second, which is generally the refresh rate of video displays. A 15-millisecond timer duration is ideal for performing animations:

```
public partial class BouncingTextPage : ContentPage
```

```
{
    const double period = 2000;                    // in milliseconds
    readonly DateTime startTime = DateTime.Now;

    public BouncingTextPage()
    {
        InitializeComponent();
        Device.StartTimer(TimeSpan.FromMilliseconds(15), OnTimerTick);
    }

    bool OnTimerTick()
    {
        TimeSpan elapsed = DateTime.Now - startTime;
        double t = (elapsed.TotalMilliseconds % period) / period;   // 0 to 1
        t = 2 * (t < 0.5 ? t : 1 - t);                              // 0 to 1 to 0

        AbsoluteLayout.SetLayoutBounds(label1,
            new Rectangle(t, 0.5, AbsoluteLayout.AutoSize, AbsoluteLayout.AutoSize));

        AbsoluteLayout.SetLayoutBounds(label2,
            new Rectangle(0.5, 1 - t, AbsoluteLayout.AutoSize, AbsoluteLayout.AutoSize));

        return true;
    }
}
```

The `OnTimerTick` handler computes an elapsed time since the program started and converts that to a value `t` (for time) that goes from 0 to 1 every two seconds. The second calculation of `t` makes it increase from 0 to 1 and then decrease back down to 0 every two seconds. This value is passed directly to the `Rectangle` constructor in the two `AbsoluteLayout.SetLayoutBounds` calls. The result is that the first `Label` moves horizontally across the center of the screen and seems to bounce off the left and right sides. The second `Label` moves vertically up and down the center of the screen and seems to bounce off the top and bottom:



The two `Label` views meet briefly in the center every second, as the Windows 10 Mobile screenshot confirms.

From here on out, the pages of our Xamarin.Forms applications will become more active and animated and dynamic. In the next chapter, you'll see how the interactive views of Xamarin.Forms establish a means of communication between the user and the app.