Chapter 17

Mastering the Grid

The `Grid` is a powerful layout mechanism that organizes its children into rows and columns of cells. At first, the `Grid` seems to resemble the HTML `table`, but there is a very important distinction: The HTML `table` is designed for presentation purposes, while the `Grid` is solely for layout. There is no concept of a heading in a `Grid`, for example, and no built-in feature to draw boxes around the cells or to separate rows and columns with divider lines. The strengths of the `Grid` are in specifying cell dimensions with three options of height and width settings.

As you've seen, the `StackLayout` is ideal for one-dimensional collections of children. Although it's possible to nest a `StackLayout` within a `StackLayout` to accommodate a second dimension and mimic a table, often the result can exhibit alignment problems. The `Grid`, however, is designed specifically for two-dimensional arrays of children. As you'll see toward the end of this chapter, the `Grid` can also be very useful for managing layouts that adapt to both portrait and landscape modes.

The basic Grid

A `Grid` can be defined and filled with children in either code or XAML, but the XAML approach is easier and clearer, and hence by far the more common.

The Grid in XAML

When defined in XAML, a `Grid` almost always has a fixed number of rows and columns. The `Grid` definition generally begins with two important properties, named `RowDefinitions` (which is a collection of `RowDefinition` objects) and `ColumnDefinitions` (a collection of `ColumnDefinition` objects). These collections contain one `RowDefinition` for every row in the `Grid` and one `ColumnDefinition` for every column, and they define the row and column characteristics of the `Grid`.

A `Grid` can consist of a single row or single column (in which case it doesn't need one of the two `Definitions` collections), or even just a single cell.

`RowDefinition` has a `Height` property of type `GridLength`, and `ColumnDefinition` has a `Width` property, also of type `GridLength`. The `GridLength` structure specifies a row height or a column width in terms of the `GridUnitType` enumeration, which has three members:

- `Absolute`—the width or height is a value in device-independent units (a number in XAML)
- `Auto`—the width or height is autosized based on the cell contents ("Auto" in XAML)
- `Star`—leftover width or height is allocated proportionally (a number with "*" in XAML)

Here's the first half of the XAML file in the **SimpleGridDemo** project:

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

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

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="100" />
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        ...

    </Grid>
</ContentPage>
```

This `Grid` has four rows and two columns. The height of the first row is "Auto"—meaning that the height is calculated based on the maximum height of all the elements occupying that first row. The second row is 100 device-independent units in height.

The two `Height` settings using "*" (pronounced "star") require some additional explanation: This particular `Grid` has an overall height that is the height of the page minus the `Padding` setting on iOS. Internally, the `Grid` determines the height of the first row based on the contents of that row, and it knows that the height of the second row is 100. It subtracts those two heights from its own height and allocates the remaining height proportionally among the third and fourth rows based on the number in the star setting. The third row is twice the height of the fourth row.

The two `ColumnDefinition` objects both set the `Width` equal to "*," which is the same as "1*," which means that the width of the screen is divided equally between the two columns.

You'll recall from Chapter 14, "Absolute layout," that the `AbsoluteLayout` class defines two attached bindable properties and four static `Set` and `Get` methods that allow a program to specify the position and size of a child of the `AbsoluteLayout` in code or XAML.

The `Grid` is quite similar. The `Grid` class defines four attached bindable properties for specifying the cell or cells that a child of the `Grid` occupies:

- `Grid.RowProperty`—the zero-based row; default value is 0

- `Grid.ColumnProperty`—the zero-based column; default value is 0
- `Grid.RowSpanProperty`—the number of rows that the child spans; default value is 1
- `Grid.ColumnSpanProperty`—the number of columns that the child spans; default value is 1

All four properties are defined to be of type `int`.

For example, to specify in code that a `Grid` child named `view` resides in a particular row and column, you can call:

```
view.SetValue(Grid.RowProperty, 2);
view.SetValue(Grid.ColumnProperty, 1);
```

Those are zero-based row and column numbers, so the child is assigned to the third row and the second column.

The `Grid` class also defines eight static methods for streamlining the setting and getting of these properties in code:

- `Grid.SetRow` and `Grid.GetRow`
- `Grid.SetColumn` and `Grid.GetColumn`
- `Grid.SetRowSpan` and `Grid.GetRowSpan`
- `Grid.SetColumnSpan` and `Grid.GetColumnSpan`

Here's the equivalent of the two `SetValue` calls you just saw:

```
Grid.SetRow(view, 2);
Grid.SetColumn(view, 1);
```

As you learned in connection with `AbsoluteLayout`, such static `Set` and `Get` methods are implemented with `SetValue` and `GetValue` calls on the child of `Grid`. For example, here's how `SetRow` is very likely defined within the `Grid` class:

```
public static void SetRow(BindableObject bindable, int value)
{
    bindable.SetValue(Grid.RowProperty, value);
}
```

You cannot call these methods in XAML, so instead you use the following attributes for setting the attached bindable properties on a child of the `Grid`:

- `Grid.Row`
- `Grid.Column`
- `Grid.RowSpan`
- `Grid.ColumnSpan`

These XAML attributes are not actually defined by the `Grid` class, but the XAML parser knows that it must reference the associated attached bindable properties defined by `Grid`.

You don't need to set all these properties on every child of the `Grid`. If the child occupies just one cell, then don't set `Grid.RowSpan` or `Grid.ColumnSpan` because the default value is 1. The `Grid.Row` and `Grid.Column` properties have a default value of 0, so you don't need to set the values if the child occupies the first row or first column. However, for purposes of clarity, the code in this book will usually show the settings of these two properties. To save space, often these attributes will appear on the same line in the XAML listing.

Here's the complete XAML file for **SimpleGridDemo**:

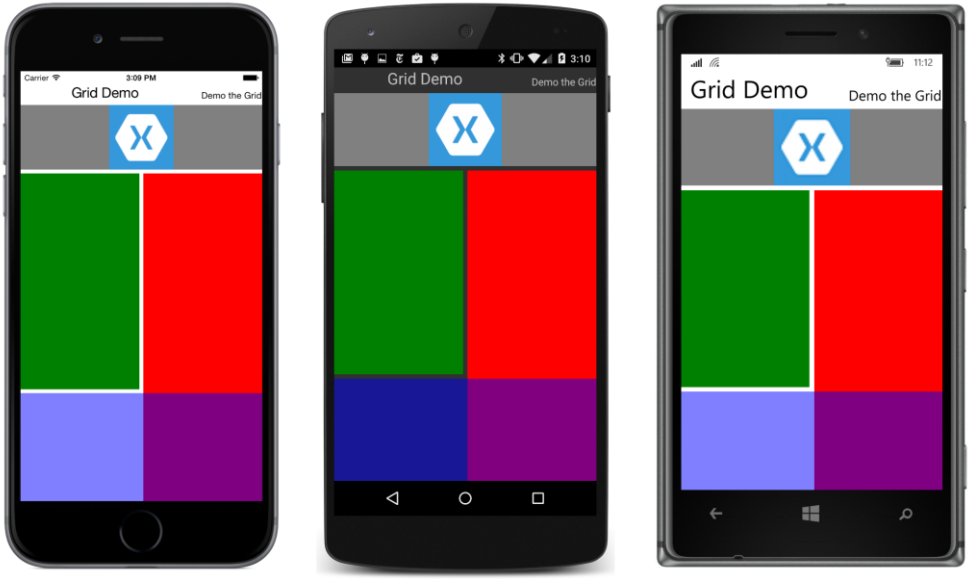
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SimpleGridDemo.SimpleGridDemoPage">

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

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="100" />
            <RowDefinition Height="2*" />
            <RowDefinition Height="1*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Label Text="Grid Demo"
               Grid.Row="0" Grid.Column="0"
               FontSize="Large"
               HorizontalOptions="End" />

        <Label Text="Demo the Grid"
               Grid.Row="0" Grid.Column="1"
               FontSize="Small"
               HorizontalOptions="End"
               VerticalOptions="End" />

        <Image BackgroundColor="Gray"
               Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2">
            <Image.Source>
                <OnPlatform x:TypeArguments="ImageSource"
                            iOS="Icon-60.png"
                            Android="icon.png"
                            WinPhone="Assets/StoreLogo.png" />
```

```
            </Image.Source>
        </Image>

        <BoxView Color="Green"
                 Grid.Row="2" Grid.Column="0" />

        <BoxView Color="Red"
                 Grid.Row="2" Grid.Column="1" Grid.RowSpan="2" />

        <BoxView Color="Blue"
                 Opacity="0.5"
                 Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2" />
    </Grid>
</ContentPage>
```

Two `Label` elements with different `FontSize` settings occupy the two columns of the first row. The height of that row is governed by the tallest element. Settings of `HorizontalOptions` and `VerticalOptions` can position a child within the cell.

The second row has a height of 100 device-independent units. That row is occupied by an `Image` element displaying an application icon with a gray background. The `Image` element spans both columns of that row.

The bottom two rows are occupied by three `BoxView` elements, one that spans two rows, and another that spans two columns, and these overlap in the bottom right cell:



The screenshots confirm that the first row is sized to the height of the large `Label`; the second row is 100 device-independent units tall; and the third and fourth rows occupy all the remaining space. The third row is twice as tall as the fourth. The two columns are equal in width and divide the entire `Grid`

in half. The red and blue `BoxView` elements overlap in the bottom right cell, but the blue `BoxView` is obviously sitting on top of the red one because it has an `Opacity` setting of 0.5 and the result is purple.

The left half of the blue semitransparent `BoxView` is lighter on the iPhone and Windows 10 Mobile device than on the Android phone because of the white background.

As you can see, children of the `Grid` can share cells. The order that the children appear in the XAML file is the order that the children are put into the `Grid`, with later children seemingly sitting on top of (and obscuring) earlier children.

You'll notice that a little gap seems to separate the rows and columns where the background peeks through. This is governed by two `Grid` properties:

- `RowSpacing`—default value of 6
- `ColumnSpacing`—default value of 6

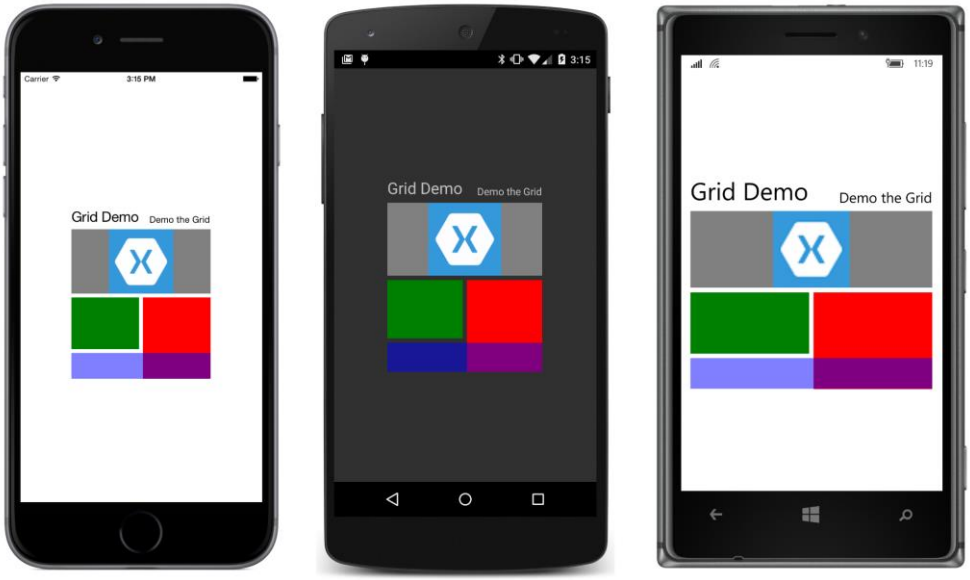
You can set these properties to 0 if you want to close up that space, and you can set the `BackgroundColor` property of the `Grid` if you want the color peeking through to be something different. You can also add space on the inside of the `Grid` around its perimeter with a `Padding` setting on the `Grid`.

You have now been introduced to all the public properties and methods defined by `Grid`.

Before moving on, let's perform a couple of experiments with **SimpleGridDemo**. First, comment out or delete the entire `RowDefinitions` and `ColumnDefinitions` section near the top of the `Grid`, and then redeploy the program. Here's what you'll see:

When you don't define your own `RowDefinition` and `ColumnDefinition` objects, the `Grid` generates them automatically as views are added to the `Children` collection. However, the default `RowDefinition` and `ColumnDefinition` is "*" (star), meaning that the four rows now equally divide the screen in quarters, and each cell is one-eighth of the total `Grid`.

Here's another experiment. Restore the `RowDefinitions` and `ColumnDefinitions` sections and set the `HorizontalOptions` and `VerticalOptions` properties on the `Grid` itself to `Center`. By default these two properties are `Fill`, which means that the `Grid` fills its container. Here's what happens with `Center`:



The third row is still twice the height of the bottom row, but now the bottom row's height is based on the default `HeightRequest` of `BoxView`, which is 40.

You'll see a similar effect when you put a `Grid` in a `StackLayout`. You can also put a `StackLayout` in a `Grid` cell, or another `Grid` in a `Grid` cell, but don't get carried away with this technique: The deeper you nest `Grid` and other layouts, the more the nested layouts will impact performance.

The Grid in code

It is also possible to define a `Grid` entirely in code, but usually without the clarity or orderliness of the XAML definition. The **GridCodeDemo** program demonstrates the code approach by reproducing the layout of **SimpleGridDemo**.

To specify the height of a `RowDefinition` and the width of the `ColumnDefinition`, you use values of the `GridLength` structure, often in combination with the `GridUnitType` enumeration. The row definitions toward the top of the `GridCodeDemoPage` class demonstrate the variations of

`GridLength`. The column definitions aren't included because they are the same as those generated by default:

```
public class GridCodeDemoPage : ContentPage
{
    public GridCodeDemoPage()
    {
        Grid grid = new Grid
        {
            RowDefinitions =
            {
                new RowDefinition { Height = GridLength.Auto },
                new RowDefinition { Height = new GridLength(100) },
                new RowDefinition { Height = new GridLength(2, GridUnitType.Star) },
                new RowDefinition { Height = new GridLength(1, GridUnitType.Star) }
            }
        };

        // First Label (row 0 and column 0).
        grid.Children.Add(new Label
        {
            Text = "Grid Demo",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.End
        });

        // Second Label.
        grid.Children.Add(new Label
            {
                Text = "Demo the Grid",
                FontSize = Device.GetNamedSize(NamedSize.Small, typeof(Label)),
                HorizontalOptions = LayoutOptions.End,
                VerticalOptions = LayoutOptions.End
            },
            1,          // left
            0);         // top

        // Image element.
        grid.Children.Add(new Image
            {
                BackgroundColor = Color.Gray,
                Source = Device.OnPlatform("Icon-60.png",
                                           "icon.png",
                                           "Assets/StoreLogo.png")
            },
            0,          // left
            2,          // right
            1,          // top
            2);         // bottom

        // Three BoxView elements.
        BoxView boxView1 = new BoxView { Color = Color.Green };
        Grid.SetRow(boxView1, 2);
        Grid.SetColumn(boxView1, 0);
```

```
        grid.Children.Add(boxView1);

        BoxView boxView2 = new BoxView { Color = Color.Red };
        Grid.SetRow(boxView2, 2);
        Grid.SetColumn(boxView2, 1);
        Grid.SetRowSpan(boxView2, 2);
        grid.Children.Add(boxView2);

        BoxView boxView3 = new BoxView
        {
            Color = Color.Blue,
            Opacity = 0.5
        };
        Grid.SetRow(boxView3, 3);
        Grid.SetColumn(boxView3, 0);
        Grid.SetColumnSpan(boxView3, 2);
        grid.Children.Add(boxView3);

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

The program shows several different ways to add children to the `Grid` and specify the cells in which they reside. The first `Label` is in row 0 and column 0, so it only needs to be added to the `Children` collection of the `Grid` to get default row and column settings:

```
grid.Children.Add(new Label
{
   …
});
```

The `Grid` redefines its `Children` collection to be of type `IGridList<View>`, which includes several additional `Add` methods. One of these `Add` methods lets you specify the row and column:

```
grid.Children.Add(new Label
    {
        …
    },
    1,          // left
    0);         // top
```

As the comments indicate, the arguments are actually named `left` and `top` rather than `column` and `row`. These names make more sense when you see the syntax for specifying row and column spans:

```
grid.Children.Add(new Image
    {
        …
    },
    0,          // left
    2,          // right
    1,          // top
    2);         // bottom
```

What this means is that the child element goes in the column starting at `left` but ending before `right`—in other words, columns 0 and 1. It occupies the row starting at `top` but ending before `bottom`, which is row 1. The `right` argument must always be greater than `left`, and the `bottom` argument must be greater than `top`. If not, the `Grid` throws an `ArgumentOutOfRangeException`.

The `IGridList<View>` interface also defines `AddHorizontal` and `AddVertical` methods to add children to a single row or single column `Grid`. The `Grid` expands in columns or rows as these calls are made, as well as automatically assigning `Grid.Column` or `Grid.Row` settings on the children. You'll see a use for this facility in the next section.

When adding children to a `Grid` in code, it's also possible to make explicit calls to `Grid.SetRow`, `Grid.SetColumn`, `Grid.SetRowSpan`, and `Grid.SetColumnSpan`. It doesn't matter whether you make these calls before or after you add the child to the `Children` collection of the `Grid`:

```
BoxView boxView1 = new BoxView { … };
Grid.SetRow(boxView1, 2);
Grid.SetColumn(boxView1, 0);
grid.Children.Add(boxView1);

BoxView boxView2 = new BoxView { … };
Grid.SetRow(boxView2, 2);
Grid.SetColumn(boxView2, 1);
Grid.SetRowSpan(boxView2, 2);
grid.Children.Add(boxView2);

BoxView boxView3 = new BoxView
{
    …
};
Grid.SetRow(boxView3, 3);
Grid.SetColumn(boxView3, 0);
Grid.SetColumnSpan(boxView3, 2);
grid.Children.Add(boxView3);
```

The Grid bar chart

The `AddVertical` and `AddHorizontal` methods defined by the `Children` collection of the `Grid` have the capability to add an entire collection of views to the `Grid` in one shot. By default, the new rows or columns get a height or width of "*" (star), so the resultant `Grid` consists of multiple rows or columns, each with the same size.

Let's use the `AddHorizontal` method to make a little bar chart that consists of 50 `BoxView` elements with random heights. The XAML file for the **GridBarChart** program defines an `AbsoluteLayout` that is parent to both a `Grid` and a `Frame`. This `Frame` serves as an overlay to display information about a particular bar in the bar chart. It has its `Opacity` set to 0, so it is initially invisible:

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

```
    <AbsoluteLayout>

        <!-- Grid occupying entire page. -->
        <Grid x:Name="grid"
              ColumnSpacing="1"
              AbsoluteLayout.LayoutBounds="0, 0, 1, 1"
              AbsoluteLayout.LayoutFlags="All" />

        <!-- Overlay in center of screen. -->
        <Frame x:Name="overlay"
               OutlineColor="Accent"
               BackgroundColor="#404040"
               Opacity="0"
               AbsoluteLayout.LayoutBounds="0.5, 0.5, AutoSize, AutoSize"
               AbsoluteLayout.LayoutFlags="PositionProportional">

            <Label x:Name="label"
                   TextColor="White"
                   FontSize="Large" />
        </Frame>
    </AbsoluteLayout>
</ContentPage>
```

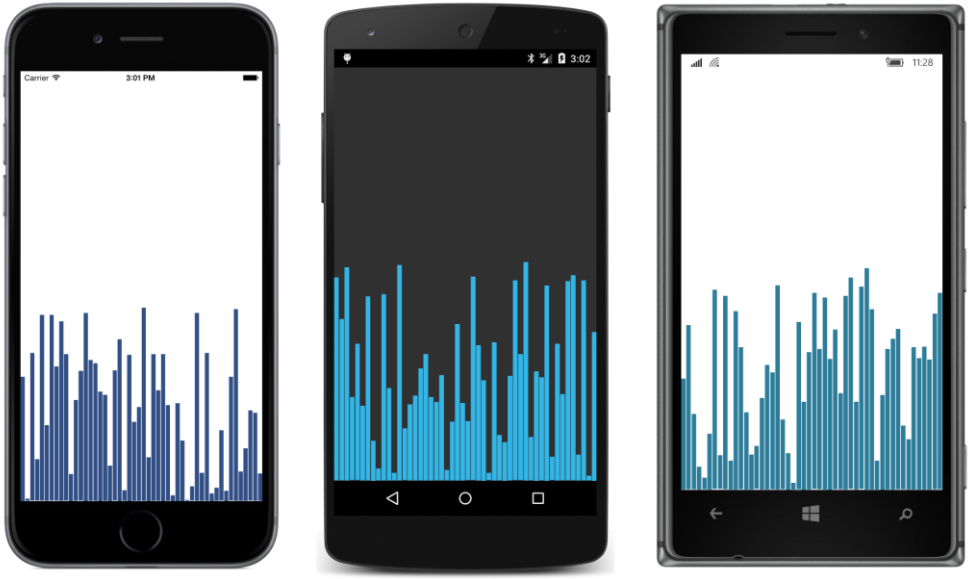
The code-behind file creates 50 `BoxView` elements with a random `HeightRequest` property between 0 and 300. In addition, the `StyleId` property of each `BoxView` is assigned a string that consists of alternated random consonants and vowels to resemble a name (perhaps of someone from another planet). All these `BoxView` elements are accumulated in a generic `List` collection and then added to the `Grid`. That job is the bulk of the code in the constructor:

```
public partial class GridBarChartPage : ContentPage
{
    const int COUNT = 50;
    Random random = new Random();

    public GridBarChartPage()
    {
        InitializeComponent();

        List<View> views = new List<View>();
        TapGestureRecognizer tapGesture = new TapGestureRecognizer();
        tapGesture.Tapped += OnBoxViewTapped;

        // Create BoxView elements and add to List.
        for (int i = 0; i < COUNT; i++)
        {
            BoxView boxView = new BoxView
            {
                Color = Color.Accent,
                HeightRequest = 300 * random.NextDouble(),
                VerticalOptions = LayoutOptions.End,
                StyleId = RandomNameGenerator()
            };
```

```
            boxView.GestureRecognizers.Add(tapGesture);
            views.Add(boxView);
        }

        // Add whole List of BoxView elements to Grid.
        grid.Children.AddHorizontal(views);

        // Start a timer at the frame rate.
        Device.StartTimer(TimeSpan.FromMilliseconds(15), OnTimerTick);
    }

    // Arrays for Random Name Generator.
    string[] vowels = { "a", "e", "i", "o", "u", "ai", "ei", "ie", "ou", "oo" };
    string[] consonants = { "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                            "n", "p", "q", "r", "s", "t", "v", "w", "x", "z" };

    string RandomNameGenerator()
    {
        int numPieces = 1 + 2 * random.Next(1, 4);
        StringBuilder name = new StringBuilder();

        for (int i = 0; i < numPieces; i++)
        {
            name.Append(i % 2 == 0 ?
                consonants[random.Next(consonants.Length)] :
                vowels[random.Next(vowels.Length)]);
        }
        name[0] = Char.ToUpper(name[0]);
        return name.ToString();
    }

    // Set text to overlay Label and make it visible.
    void OnBoxViewTapped(object sender, EventArgs args)
    {
        BoxView boxView = (BoxView)sender;
        label.Text = String.Format("The individual known as {0} " +
                                   "has a height of {1} centimeters.",
                                   boxView.StyleId, (int)boxView.HeightRequest);
        overlay.Opacity = 1;
    }

    // Decrease visibility of overlay.
    bool OnTimerTick()
    {
        overlay.Opacity = Math.Max(0, overlay.Opacity - 0.0025);
        return true;
    }
}
```

The `AddHorizontal` method of the `Children` collection adds the multiple `BoxView` elements to the `Grid` and gives them sequential `Grid.Column` settings. Each column by default has a width of "*" (star), so the width of each `BoxView` is the same while the height is governed by the `HeightRequest`

settings. The `Spacing` value of 1 set to the `Grid` in the XAML file provides a little separation between the bars of the bar chart:



The bars are more distinct when you turn the phone sideways to give them more width:

This program has another feature: When you tap on one of the bars, the overlay is made visible and displays information about that tapped bar—specifically, the interplanetary visitor's name from the

`StyleId` and the height of the bar. But a timer set in the constructor continuously decreases the `Opacity` value on the overlay, so this information gradually fades from view:

Even without a native graphics system, Xamarin.Forms is able to display something that looks quite a lot like graphics.

Alignment in the Grid

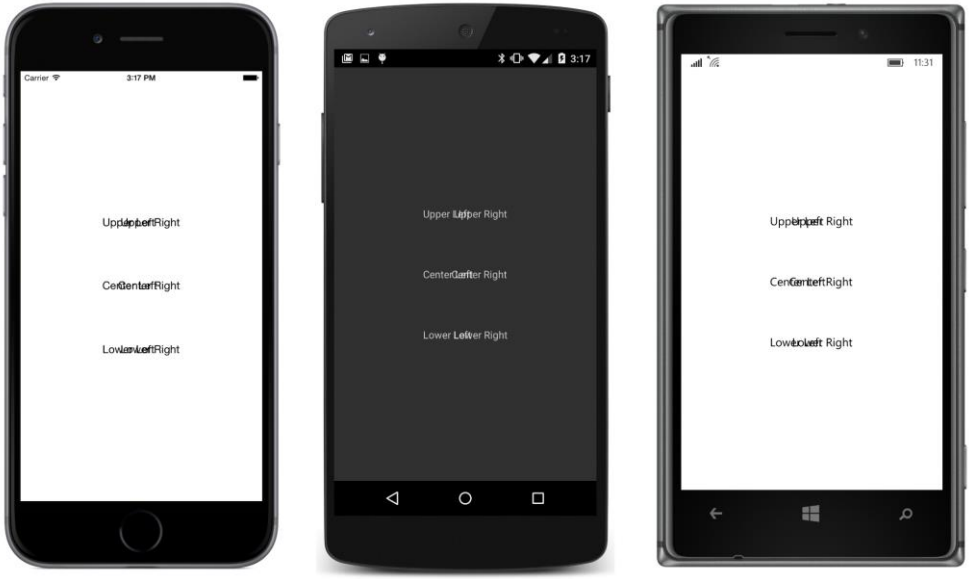
A `Grid` row with a `Height` property of `Auto` constrains the height of elements in that row in the same way as a vertical `StackLayout`. Similarly, a column with a `Width` of `Auto` works much like a horizontal `StackLayout`.

As you've seen earlier in this chapter, you can set the `HorizontalOptions` and `VerticalOptions` properties of children of the `Grid` to position them within the cell. Here's a program called **GridAlignment** that creates a `Grid` with nine equal-size cells and then puts six `Label` elements all in the center cell but with different alignment settings:

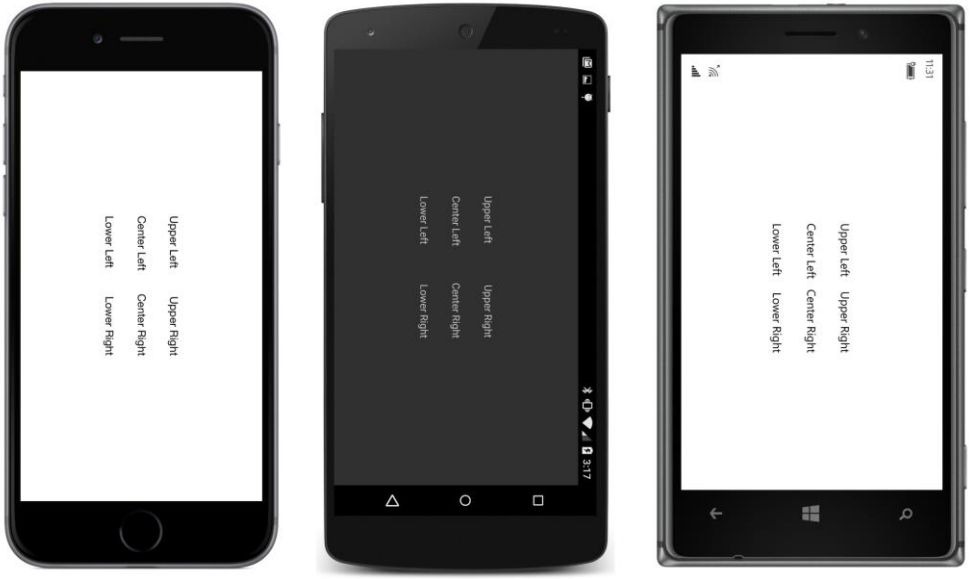
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="GridAlignment.GridAlignmentPage">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
```

```
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Label Text="Upper Left"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="Start"
               HorizontalOptions="Start" />

        <Label Text="Upper Right"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="Start"
               HorizontalOptions="End" />

        <Label Text="Center Left"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="Center"
               HorizontalOptions="Start" />

        <Label Text="Center Right"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="Center"
               HorizontalOptions="End" />

        <Label Text="Lower Left"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="End"
               HorizontalOptions="Start" />

        <Label Text="Lower Right"
               Grid.Row="1" Grid.Column="1"
               VerticalOptions="End"
               HorizontalOptions="End" />
    </Grid>
</ContentPage>
```

As you can see, some of the text overlaps:



But if you turn the phone sideways, the cells resize and the text doesn't overlap:



Although you can use `HorizontalOptions` and `VerticalOptions` on children of a `Grid` to set the child's alignment, you cannot use the `Expands` flag. Strictly speaking, you actually *can* use the `Expands` flag, but it has no effect on children of a `Grid`. The `Expands` flag only affects children of a `StackLayout`.

Often you've seen programs that use the `Expands` flag for children of a `StackLayout` to provide

extra space to surround elements within the layout. For example, if two `Label` children of a `StackLayout` both have their `VerticalOptions` properties set to `CenterAndExpand`, then all the extra space is divided equally between the two slots in the `StackLayout` allocated for these children.

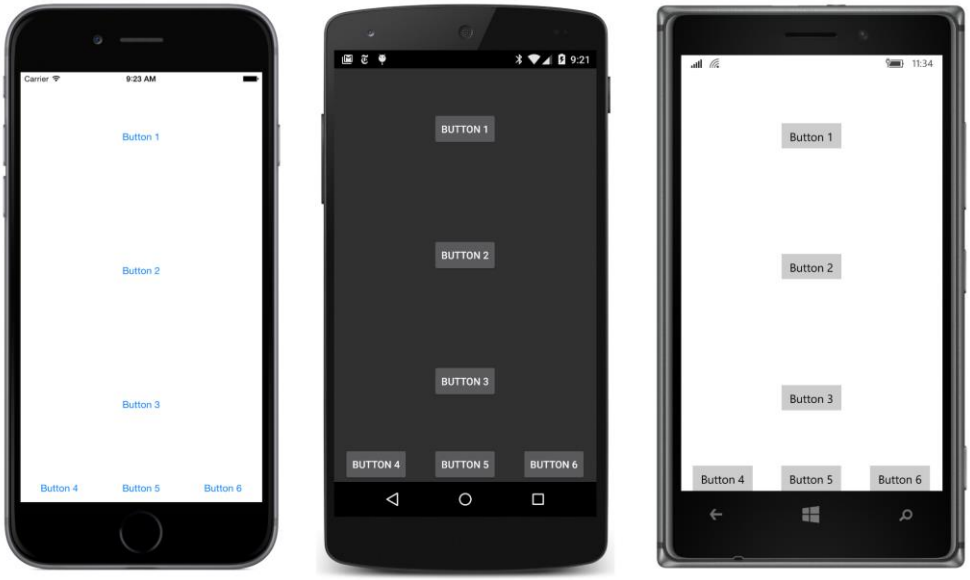
In a `Grid`, you can perform similar layout tricks by using cells sized with the "*" (star) specification together with `HorizontalOptions` and `VerticalOptions` settings on the children. You can even create empty rows or empty columns just for spacing purposes.

The **SpacingButtons** program equally spaces three vertical buttons and three horizontal buttons. The first three buttons occupy a three-row `Grid` that takes up much of the page, and the three horizontal buttons are in a three-column `Grid` down at the bottom of the page. The two grids are in a `StackLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SpacingButtons.SpacingButtonsPage">
    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>

            <Button Text="Button 1"
                    Grid.Row="0"
                    VerticalOptions="Center"
                    HorizontalOptions="Center" />

            <Button Text="Button 2"
                    Grid.Row="1"
                    VerticalOptions="Center"
                    HorizontalOptions="Center" />

            <Button Text="Button 3"
                    Grid.Row="2"
                    VerticalOptions="Center"
                    HorizontalOptions="Center" />
        </Grid>

        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Button Text="Button 4"
                    Grid.Column="0"
                    HorizontalOptions="Center" />

            <Button Text="Button 5"
```

```
                Grid.Column="1"
                HorizontalOptions="Center" />

        <Button Text="Button 6"
                Grid.Column="2"
                HorizontalOptions="Center" />
    </Grid>
  </StackLayout>
</ContentPage>
```

The second `Grid` has a default `VerticalOptions` value of `Fill`, while the first `Grid` has an explicit setting for `VerticalOptions` to `FillAndExpand`. This means that the first `Grid` will occupy all the area of the screen not occupied by the second `Grid`. The three `RowDefinition` objects of the first `Grid` divide that area into thirds. Within each cell, the `Button` is horizontal and vertically centered:



The second `Grid` divides its area into three equally spaced columns, and each `Button` is horizontally centered within that area.

Although the `Expands` flag of `LayoutOptions` can assist in equally spacing visual objects within a `StackLayout`, the technique breaks down when the visual objects are not a uniform size. The `Expands` option allocates leftover space equally among all the slots in the `StackLayout`, but the total size of each slot depends on the size of the individual visual objects. The `Grid`, however, allocates space equally to the cells, and then the visual objects are aligned within that space.

Cell dividers and borders

The `Grid` doesn't have any built-in cell dividers or borders. But if you'd like some, you can add them yourself. The **GridCellDividers** program defines a `GridLength` value in its `Resources` dictionary

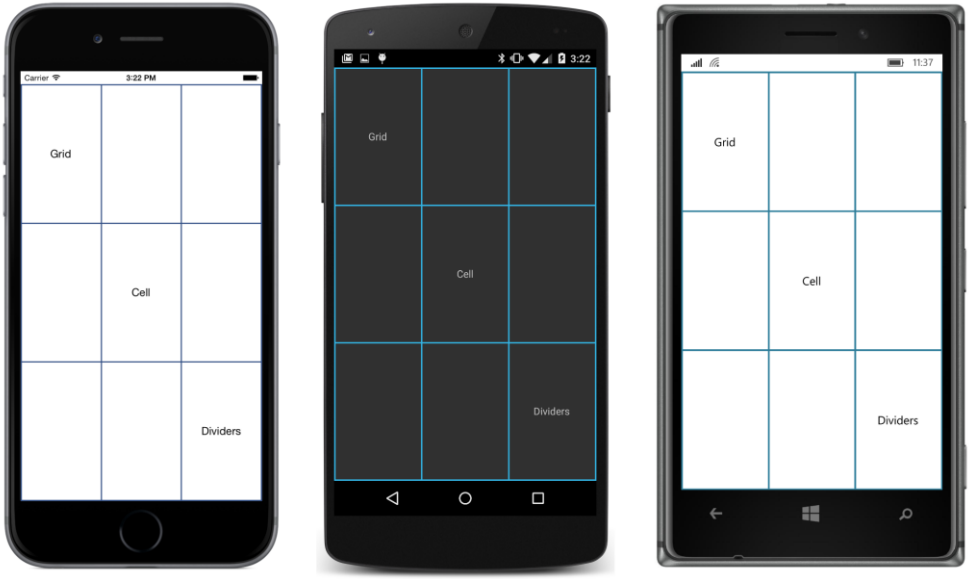
named `dividerThickness`. This is used for the height and width of every other row and column in the `Grid`. The idea here is that these rows and columns are for the dividers, while the other rows and columns are for regular content:

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

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

    <Grid>
        <Grid.Resources>
            <ResourceDictionary>
                <GridLength x:Key="dividerThickness">2</GridLength>

                <Style TargetType="BoxView">
                    <Setter Property="Color" Value="Accent" />
                </Style>

                <Style TargetType="Label">
                    <Setter Property="HorizontalOptions" Value="Center" />
                    <Setter Property="VerticalOptions" Value="Center" />
                </Style>
            </ResourceDictionary>
        </Grid.Resources>

        <Grid.RowDefinitions>
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
            <RowDefinition Height="*" />
            <RowDefinition Height="{StaticResource dividerThickness}" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="{StaticResource dividerThickness}" />
        </Grid.ColumnDefinitions>

        <BoxView Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="7" />
        <BoxView Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="7" />
        <BoxView Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="7" />
```

```
        <BoxView Grid.Row="6" Grid.Column="0" Grid.ColumnSpan="7" />

        <BoxView Grid.Row="0" Grid.Column="0" Grid.RowSpan="7" />
        <BoxView Grid.Row="0" Grid.Column="2" Grid.RowSpan="7" />
        <BoxView Grid.Row="0" Grid.Column="4" Grid.RowSpan="7" />
        <BoxView Grid.Row="0" Grid.Column="6" Grid.RowSpan="7" />

        <Label Text="Grid"
               Grid.Row="1" Grid.Column="1" />

        <Label Text="Cell"
               Grid.Row="3" Grid.Column="3" />

        <Label Text="Dividers"
               Grid.Row="5" Grid.Column="5" />
    </Grid>
</ContentPage>
```

Each row and column for the dividers is occupied by a `BoxView` colored with the `Accent` color from an implicit style. For the horizontal dividers, the height is set by the `RowDefinition` and the width is governed by the `Grid.ColumnSpan` attached bindable property; a similar approach is applied for the vertical dividers.

The `Grid` also contains three `Label` elements just to demonstrate how regular content fits in with these dividers:



It is not necessary to allocate entire rows and columns to these dividers. Keep in mind that visual objects can share cells, so it's possible to add a `BoxView` (or two or three or four) to a cell and set the horizontal and vertical options so that it hugs the wall of the cell and resembles a border.

Here's a similar program, called **GridCellBorders,** that displays content in the same three cells as **GridCellDividers**, but those three cells are also adorned with borders.

The `Resources` dictionary contains no fewer than seven styles that target `BoxView`! The base style sets the color, two more styles set the `HeightRequest` and `WidthRequest` for the horizontal and vertical borders, and then four more styles set the `VerticalOptions` to `Start` or `End` for the top and bottom borders and `HorizontalOptions` to `Start` and `End` for the left and right borders. The `borderThickness` dictionary entry is a `double` because it's used to set `WidthRequest` and `HeightRequest` properties of the `BoxView` elements:

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

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

    <Grid>
        <Grid.Resources>
            <ResourceDictionary>
                <x:Double x:Key="borderThickness">1</x:Double>

                <Style x:Key="baseBorderStyle" TargetType="BoxView">
                    <Setter Property="Color" Value="Accent" />
                </Style>

                <Style x:Key="horzBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource baseBorderStyle}">
                    <Setter Property="HeightRequest" Value="{StaticResource borderThickness}" />
                </Style>

                <Style x:Key="topBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource horzBorderStyle}">
                    <Setter Property="VerticalOptions" Value="Start" />
                </Style>

                <Style x:Key="bottomBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource horzBorderStyle}">
                    <Setter Property="VerticalOptions" Value="End" />
                </Style>

                <Style x:Key="vertBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource baseBorderStyle}">
                    <Setter Property="WidthRequest" Value="{StaticResource borderThickness}" />
                </Style>

                <Style x:Key="leftBorderStyle" TargetType="BoxView"
                       BasedOn="{StaticResource vertBorderStyle}">
```

```
            <Setter Property="HorizontalOptions" Value="Start" />
        </Style>

        <Style x:Key="rightBorderStyle" TargetType="BoxView"
               BasedOn="{StaticResource vertBorderStyle}">
            <Setter Property="HorizontalOptions" Value="End" />
        </Style>

        <Style TargetType="Label">
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="VerticalOptions" Value="Center" />
        </Style>
    </ResourceDictionary>
</Grid.Resources>

<Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<Label Text="Grid"
       Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource topBorderStyle}"
         Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource bottomBorderStyle}"
         Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource leftBorderStyle}"
         Grid.Row="0" Grid.Column="0" />

<BoxView Style="{StaticResource rightBorderStyle}"
         Grid.Row="0" Grid.Column="0" />

<Grid Grid.Row="1" Grid.Column="1">
    <Label Text="Cell" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />
    <BoxView Style="{StaticResource leftBorderStyle}" />
    <BoxView Style="{StaticResource rightBorderStyle}" />
</Grid>

<Grid Grid.Row="2" Grid.Column="2">
    <Label Text="Borders" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />
```

```
            <BoxView Style="{StaticResource leftBorderStyle}" />
            <BoxView Style="{StaticResource rightBorderStyle}" />
        </Grid>
    </Grid>
</ContentPage>
```

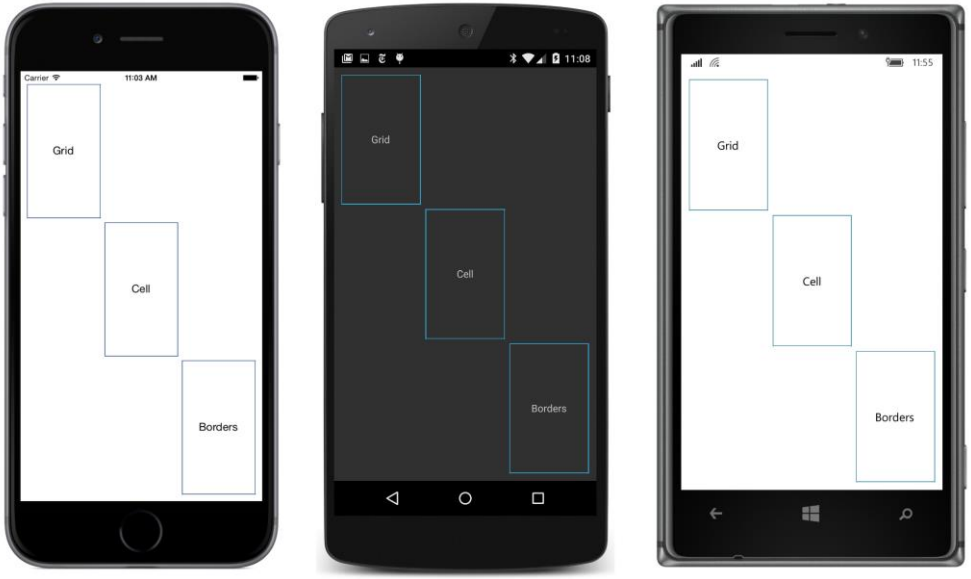
In the cell in the upper-left corner, the `Label` and four `BoxView` elements each gets its `Grid.Row` and `Grid.Column` attributes set to 0. However, for the middle `Grid` and the bottom-right `Grid`, a rather easier approach is taken: Another `Grid` with a single cell occupies the cell, and that single-cell `Grid` contains the `Label` and four `BoxView` elements. The simplicity results from setting `Grid.Row` and `Grid.Column` only on the single-cell `Grid`:

```
<Grid Grid.Row="1" Grid.Column="1">
    <Label Text="Cell" />
    <BoxView Style="{StaticResource topBorderStyle}" />
    <BoxView Style="{StaticResource bottomBorderStyle}" />
    <BoxView Style="{StaticResource leftBorderStyle}" />
    <BoxView Style="{StaticResource rightBorderStyle}" />
</Grid>
```

When nesting a `Grid` inside another `Grid`, the use of the `Grid.Row` and `Grid.Column` attributes can be confusing. This single-cell `Grid` occupies the second row and second column of its parent, which is the `Grid` that occupies the entire page.

Also, keep in mind that when a `Grid` is laying itself out, it looks only at the `Grid.Row` and `Grid.Column` settings of its children, and never its grandchildren or other descendants in the visual tree.

Here's the result:



It might be a little disconcerting that the corners of the borders don't meet, but that's due to the default row and column spacing of the `Grid`. Set the `RowSpacing` and `ColumnSpacing` attributes to 0, and the corners will meet although the lines will still seem somewhat discontinuous because the borders are in different cells. If this is unacceptable, use the technique shown in **GridCellDividers**.

If you want all the rows and columns shown with dividers as in **GridCellDividers**, another technique is to set the `BackgroundColor` property of the `Grid` and use the `RowSpacing` and `ColumnSpacing` properties to let that color peek through the spaces between the cells. But all the cells must contain content that has an opaque background for this technique to be visually convincing.

Almost real-life Grid examples

We are now ready to rewrite the **XamlKeypad** program from Chapter 8 to use a `Grid`. The new version is called **KeypadGrid**. The use of a `Grid` not only forces the `Button` elements that make up the keypad to be all the same size, but also allows components of the keypad to span cells.

The `Grid` that makes up the keypad is centered on the page with `HorizontalOptions` and `VerticalOptions` settings. It has five rows and three columns but the `RowDefinitions` and `ColumnDefinitions` collections don't need to be explicitly constructed because every cell has a "*" (star) height and width.

Moreover, the entire `Grid` is given a platform-specific `WidthRequest` and `HeightRequest`, where the width is three-fifths of the height. (The difference for Windows Phone is based on the somewhat larger size of the `Large` font size used for the `Button`.) This causes every cell in the `Grid` to be square:

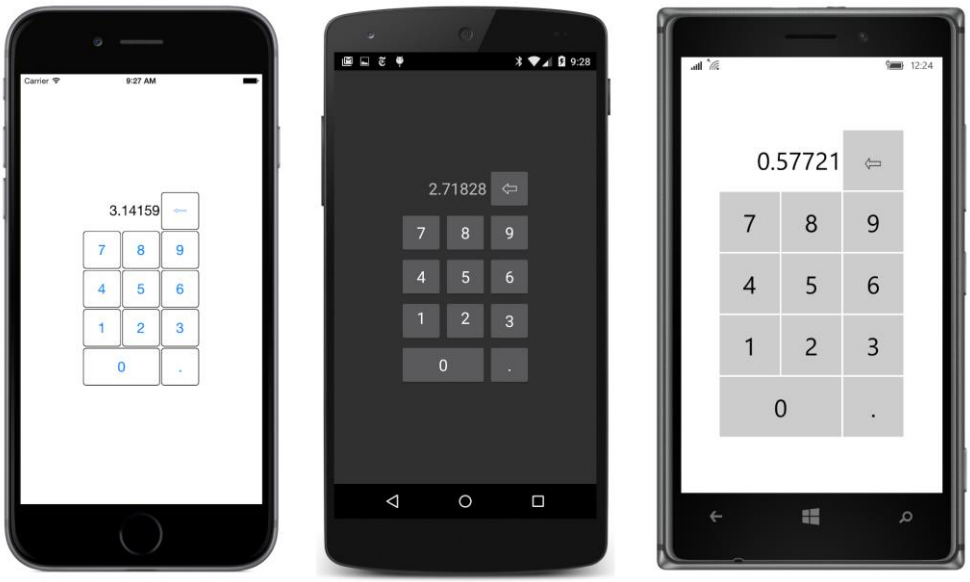
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="KeypadGrid.KeypadGridPage">

    <Grid RowSpacing="2"
          ColumnSpacing="2"
          VerticalOptions="Center"
          HorizontalOptions="Center">
        <Grid.WidthRequest>
            <OnPlatform x:TypeArguments="x:Double"
                        iOS="180"
                        Android="180"
                        WinPhone="240" />
        </Grid.WidthRequest>

        <Grid.HeightRequest>
            <OnPlatform x:TypeArguments="x:Double"
                        iOS="300"
                        Android="300"
                        WinPhone="400" />
        </Grid.HeightRequest>

        <Grid.Resources>
```

```
    <ResourceDictionary>
        <Style TargetType="Button">
            <Setter Property="FontSize" Value="Large" />
            <Setter Property="BorderWidth" Value="1" />
        </Style>
    </ResourceDictionary>
</Grid.Resources>

<Label x:Name="displayLabel"
       Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
       FontSize="Large"
       LineBreakMode="HeadTruncation"
       VerticalOptions="Center"
       HorizontalTextAlignment="End" />

<Button x:Name="backspaceButton"
        Text="&#x21E6;"
        Grid.Row="0" Grid.Column="2"
        IsEnabled="False"
        Clicked="OnBackspaceButtonClicked" />

<Button Text="7" StyleId="7"
        Grid.Row="1" Grid.Column="0"
        Clicked="OnDigitButtonClicked" />

<Button Text="8" StyleId="8"
        Grid.Row="1" Grid.Column="1"
        Clicked="OnDigitButtonClicked" />

<Button Text="9" StyleId="9"
        Grid.Row="1" Grid.Column="2"
        Clicked="OnDigitButtonClicked" />

<Button Text="4" StyleId="4"
        Grid.Row="2" Grid.Column="0"
        Clicked="OnDigitButtonClicked" />

<Button Text="5" StyleId="5"
        Grid.Row="2" Grid.Column="1"
        Clicked="OnDigitButtonClicked" />

<Button Text="6" StyleId="6"
        Grid.Row="2" Grid.Column="2"
        Clicked="OnDigitButtonClicked" />

<Button Text="1" StyleId="1"
        Grid.Row="3" Grid.Column="0"
        Clicked="OnDigitButtonClicked" />

<Button Text="2" StyleId="2"
        Grid.Row="3" Grid.Column="1"
        Clicked="OnDigitButtonClicked" />

<Button Text="3" StyleId="3"
```

```
                Grid.Row="3" Grid.Column="2"
                Clicked="OnDigitButtonClicked" />

        <Button Text="0" StyleId="0"
                Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2"
                Clicked="OnDigitButtonClicked" />

        <Button Text="." StyleId="."
                Grid.Row="4" Grid.Column="2"
                Clicked="OnDigitButtonClicked" />
    </Grid>
</ContentPage>
```

The `Label` and the backspace button occupy the top row, but the `Label` spans two columns and the backspace button is in the third column. Similarly, the bottom row of the `Grid` contains the zero button and the decimal-point button, but the zero button spans two columns as is typical on computer keypads.

The code-behind file is the same as the **XamlKeypad** program. In addition, the program saves entries when the program is put to sleep and then restores them when the program starts up again. A border has been added to the `Button` in an implicit style so that it looks more like a real keypad on iOS:



As you might recall, the `OnDigitButtonClicked` handler in the code-behind file uses the `StyleId` property to append a new character to the text string. But as you can see in the XAML file, for each of the buttons with this event handler, the `StyleId` is set to the same character as the `Text` property of the `Button`. Can't the event handler use that instead?

Yes, it can. But suppose you decide that the decimal point in the `Button` doesn't show up very well.

You might prefer to use a heavier and more central dot, such as \u00B7 (called Middle Dot) or \u22C5 (the mathematical Dot Operator) or even \u2022 (the Bullet). Perhaps you'd also like different styles of numbers for these other buttons, such as the set of encircled numbers that begin at \u2460 in the Unicode standard, or the Roman numerals that begin at \u2160. You can replace the `Text` property in the XAML file without touching the code-behind file:

The `StyleId` is one of the tools to keep the visuals and mechanics of the user interface restricted to markup and separated from your code. You'll see more tools to structure your program in the next chapter, which covers the Model-View-ViewModel application architecture. That chapter also presents a variation of the keypad program turned into an adding machine.

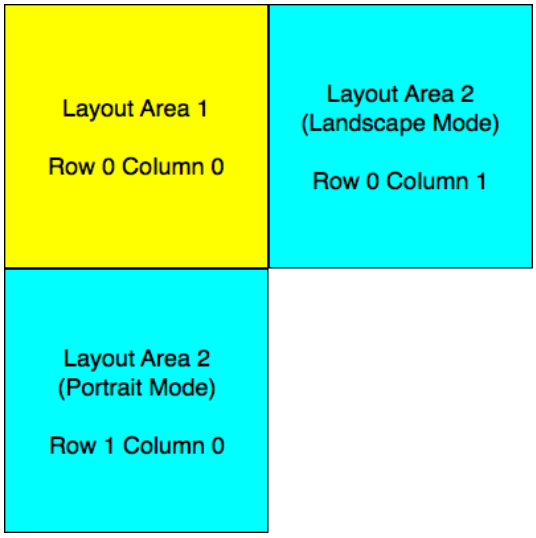
Responding to orientation changes

The layout of an application's page is usually tied fairly closely to a particular form factor and aspect ratio. Sometimes, an application will require that it be used only in portrait or landscape mode. But usually an application will attempt to move things around on the screen when the phone changes orientation.

A `Grid` can help an application accommodate itself to orientation changes. The `Grid` can be defined in XAML with certain allowances for both portrait and landscape modes, and then a little code can make the proper adjustments within a `SizeChanged` handler for the page.

This job is easiest if you can divide the entire layout of your application into two large areas that can be arranged vertically when the phone is oriented in portrait mode or horizontally for landscape mode. Put each of these areas in separate cells of a `Grid`. When the phone is in portrait mode, the `Grid` has two rows, and when it's in landscape mode, it has two columns. In the following diagram, the first area

is always at the top or the left. The second area can be in either the second row for portrait mode or the second column for landscape mode:



To keep things reasonably simple, you'll want to define the `Grid` in XAML with two rows and two columns, but in portrait mode, the second column has a width of zero, and in landscape mode the second row has a zero height.

The **GridRgbSliders** program demonstrates this technique. It is similar to the **RgbSliders** program from Chapter 15, "The interactive interface," except that the layout uses a combination of a `Grid` and a `StackLayout`, and the `Label` elements display the current values of the `Slider` elements by using data bindings with a value converter and a value converter parameter. (More on this later.) Setting the `Color` property of the `BoxView` based on the three `Slider` elements still requires code because the `R`, `G`, and `B` properties of the `Color` struct are not backed by bindable properties, and these properties cannot be individually changed anyway because they do not have public `set` accessors. (However, in the next chapter, on MVVM, you'll see a way to eliminate this logic in the code-behind file.)

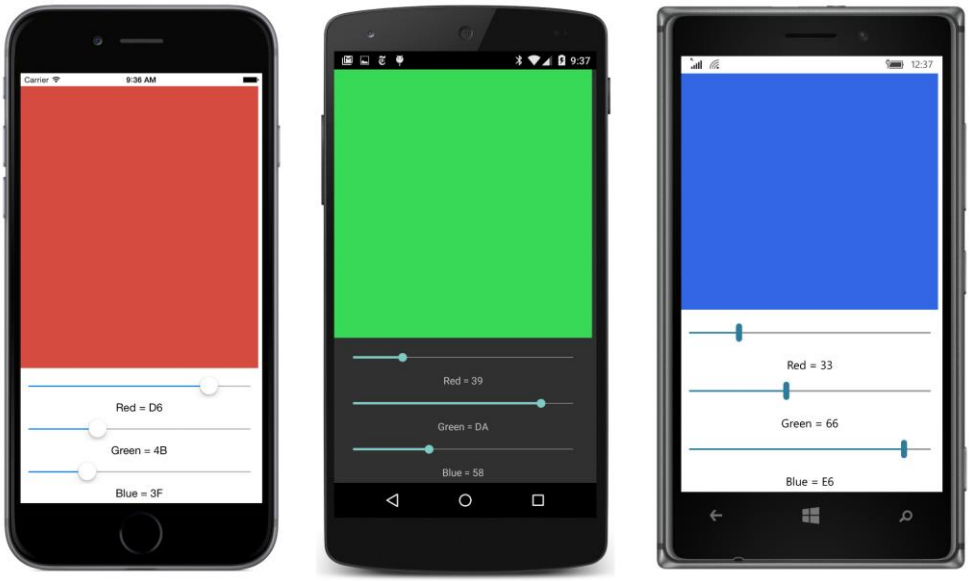
As you can see in the following listing, the `Grid` named `mainGrid` does indeed have two rows and two columns. However, it is initialized for portrait mode, so the second column has a width of zero. The top row of the `Grid` contains the `BoxView`, and that's made as large as possible with a "*" (star) setting, while the bottom row contains a `StackLayout` with all the interactive controls. This is given a height of `Auto`:

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

    <ContentPage.Padding>
```

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

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:DoubleToIntConverter x:Key="doubleToInt" />

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid x:Name="mainGrid">
        <!-- Initialized for portrait mode. -->
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="0" />
        </Grid.ColumnDefinitions>

        <BoxView x:Name="boxView"
                 Grid.Row="0" Grid.Column="0" />

        <StackLayout x:Name="controlPanelStack"
                     Grid.Row="1" Grid.Column="0"
                     Padding="10, 5">

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider x:Name="redSlider"
                        ValueChanged="OnSliderValueChanged" />

                <Label Text="{Binding Source={x:Reference redSlider},
                                      Path=Value,
                                      Converter={StaticResource doubleToInt},
                                      ConverterParameter=255,
                                      StringFormat='Red = {0:X2}'}" />
            </StackLayout>

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider x:Name="greenSlider"
                        ValueChanged="OnSliderValueChanged" />

                <Label Text="{Binding Source={x:Reference greenSlider},
                                      Path=Value,
                                      Converter={StaticResource doubleToInt},
                                      ConverterParameter=255,
                                      StringFormat='Green = {0:X2}'}" />
            </StackLayout>
```

```
            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider x:Name="blueSlider"
                        ValueChanged="OnSliderValueChanged" />

                <Label Text="{Binding Source={x:Reference blueSlider},
                                      Path=Value,
                                      Converter={StaticResource doubleToInt},
                                      ConverterParameter=255,
                                      StringFormat='Blue = {0:X2}'}" />
            </StackLayout>
        </StackLayout>
    </Grid>
</ContentPage>
```

And here's the portrait view:



The layout in the XAML file is prepared for landscape mode in a couple of ways. First, the `Grid` already has a second column. This means that to switch to landscape mode, the code-behind file needs to change the height of the second row to zero and the width of the second column to a nonzero value.

Secondly, the `StackLayout` containing all the `Slider` and `Label` elements is accessible from code because it has a name, specifically `controlPanelStack`. The code-behind file can then make `Grid.SetRow` and `Grid.SetColumn` calls on this `StackLayout` to move it from row 1 and column 0 to row 0 and column 1.

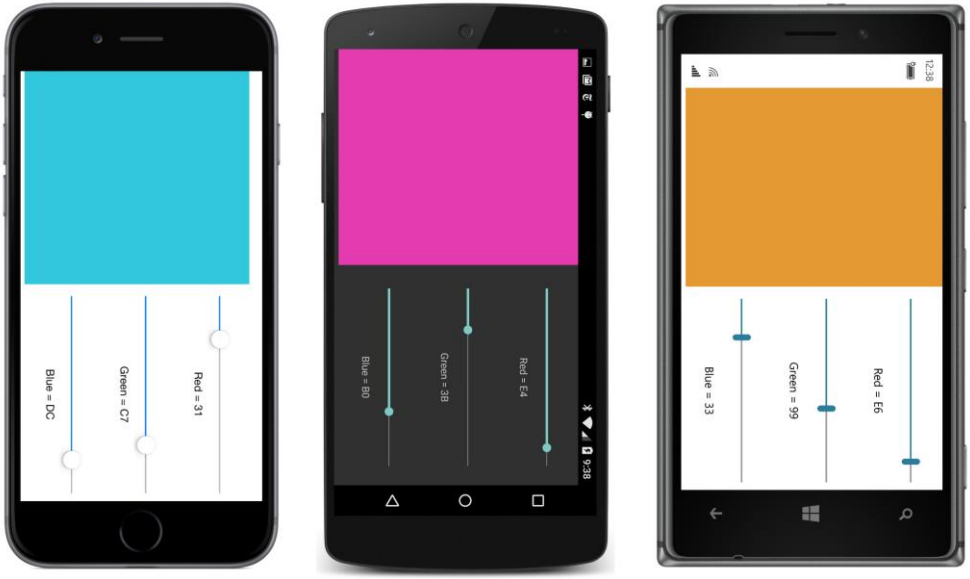
In portrait mode, the `BoxView` has a height of "*" (star) and the `StackLayout` has a height of `Auto`. Does this mean that the width of the `StackLayout` should be `Auto` in landscape mode? That wouldn't

be wise because it would shrink the widths of the `Slider` elements. A better solution for landscape mode is to give both the `BoxView` and the `StackLayout` a width of "*" (star) to divide the screen in half.

Here's the code-behind file showing the `SizeChanged` handler on the page responsible for switching between portrait and landscape mode, as well as the `ValueChanged` handler for the `Slider` elements that sets the `BoxView` color:

```
public partial class GridRgbSlidersPage : ContentPage
{
    public GridRgbSlidersPage()
    {
        // Ensure link to Toolkit library.
        new Xamarin.FormsBook.Toolkit.DoubleToIntConverter();

        InitializeComponent();
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // Portrait mode.
        if (Width < Height)
        {
            mainGrid.RowDefinitions[1].Height = GridLength.Auto;
            mainGrid.ColumnDefinitions[1].Width = new GridLength(0, GridUnitType.Absolute);

            Grid.SetRow(controlPanelStack, 1);
            Grid.SetColumn(controlPanelStack, 0);
        }
        // Landscape mode.
        else
        {
            mainGrid.RowDefinitions[1].Height = new GridLength(0, GridUnitType.Absolute);
            mainGrid.ColumnDefinitions[1].Width = new GridLength(1, GridUnitType.Star);

            Grid.SetRow(controlPanelStack, 0);
            Grid.SetColumn(controlPanelStack, 1);
        }
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        boxView.Color = new Color(redSlider.Value, greenSlider.Value, blueSlider.Value);
    }
}
```

And here's the landscape layout, displayed sideways as usual:



Notice, particularly on the iOS and Android displays, how each pair of `Slider` and `Label` elements is grouped together. This results from a third way that the XAML file is prepared to accommodate landscape mode. Each pair of `Slider` and `Label` elements is grouped in a nested `StackLayout`. This is given a `VerticalOptions` setting of `CenterAndExpand` to perform this spacing.

A little thought was given to arranging the `BoxView` and the control panel: In portrait mode, the fingers manipulating the `Slider` elements won't obscure the result in the `BoxView`, and in landscape mode, the fingers of right-handed users won't obscure the `BoxView` either. (Of course, left-handed users will probably insist on a program option to swap the locations!)

The screenshots show the `Slider` values displayed in hexadecimal. This is done with a data binding, and that would normally be a problem. The `Value` property of the `Slider` is of type `double`, and if you attempt to format a double with "X2" for hexadecimal, an exception will be raised. A type converter (named `DoubleToIntConverter`, for example) must convert the source `double` to an `int` for the string formatting. However, the `Slider` elements are set up for a range of 0 to 1, while integer values formatted as hexadecimal must range from 0 to 255.

A solution is to make use of the `ConverterParameter` property of `Binding`. Whatever is set to this property is passed as the third argument to the `Convert` and `ConvertBack` methods in the value converter. Here's the `DoubleToIntConverter` class in the **Xamarin.FormsBook.Toolkit** library:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class DoubleToIntConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
```

```
            string strParam = parameter as string;
            double multiplier = 1;

            if (!String.IsNullOrEmpty(strParam))
            {
                Double.TryParse(strParam, out multiplier);
            }

            return (int)Math.Round((double)value * multiplier);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            string strParam = parameter as string;
            double divider = 1;

            if (!String.IsNullOrEmpty(strParam))
            {
                Double.TryParse(strParam, out divider);
            }

            return (int)value / divider;
        }
    }
}
```

The `Convert` and `ConvertBack` methods assume that the `parameter` argument is a string and, if so, attempt to convert it to a `double`. This value is then multiplied by the `double` value being converted, and then the product is cast to an `int`.

The combination of the value converter, the converter parameter, and the string formatting converts values ranging from 0 to 1 coming from the `Slider` to integers in the range of 0 to 255 that are then formatted as two hexadecimal digits:

```
<Label Text="{Binding Source={x:Reference redSlider},
                      Path=Value,
                      Converter={StaticResource doubleToInt},
                      ConverterParameter=255,
                      StringFormat='Red = {0:X2}'}" />
```

Of course, if you were defining the `Binding` in code, you would probably set the `ConverterParameter` property to the numeric value of 255 rather than a string of "255", and the logic in the `DoubleToIntConverter` would fail. Simple value converters are usually simpler than they should be for complete bulletproofing.

Can a program like **GridRgbSliders** be entirely realized without the `Slider` event handlers in the code-behind file? Code will certainly still be required, but some of it will be moved away from the user-interface logic. That's the main objective of the Model-View-ViewModel architecture explored in the next chapter.