Chapter 5

Dealing with sizes

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

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

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

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

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

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

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

Pixels, points, dps, DIPs, and DIUs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This information is summarized in the following table:

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

* Includes 115 percent downsampling.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

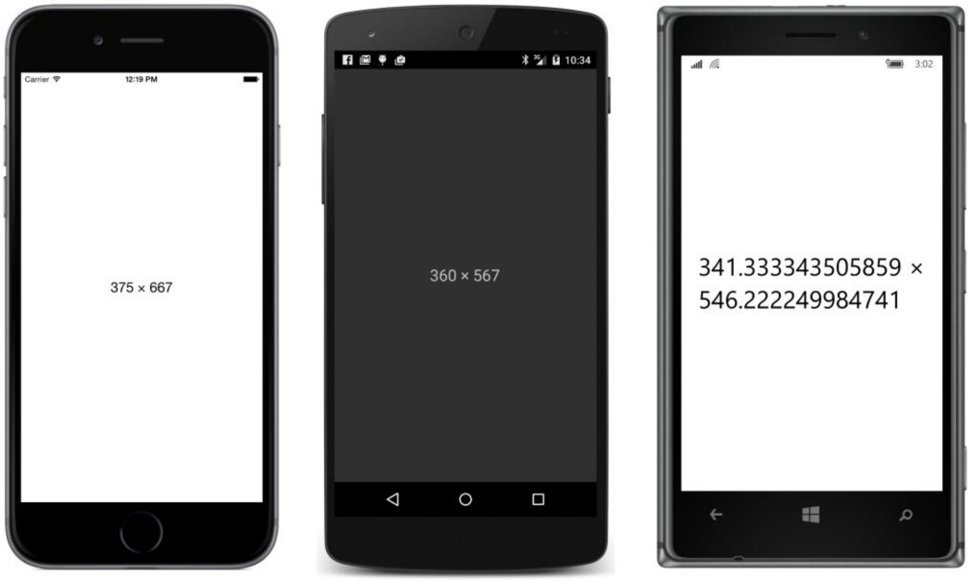
```
public class WhatSizePage : ContentPage
{
    Label label;

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

        Content = label;

        SizeChanged += OnPageSizeChanged;
    }

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        label.Text = String.Format("{0} \u00D7 {1}", Width, Height);
    }
}
```

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

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

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

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



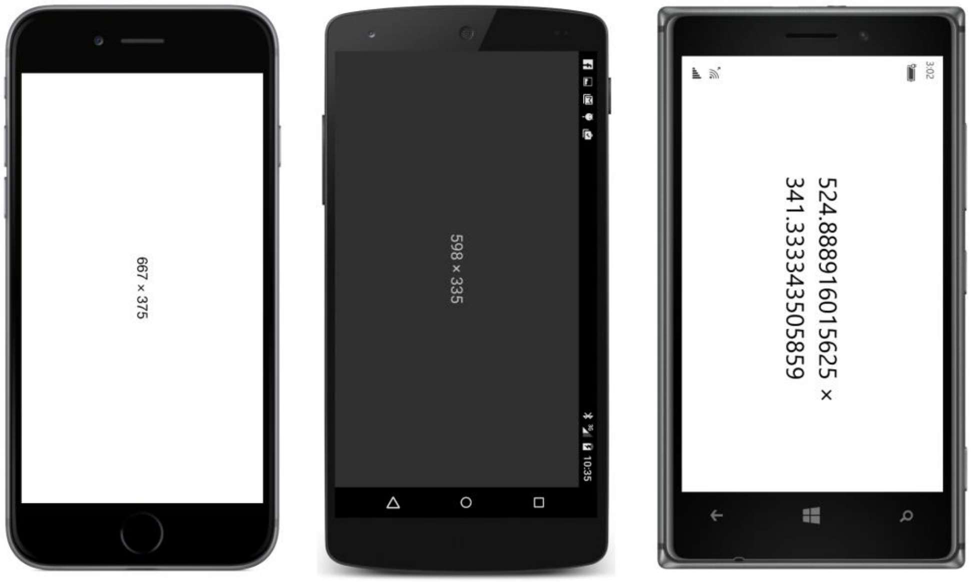
For the record, these are the sources of the screens in these three images:

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

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

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

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



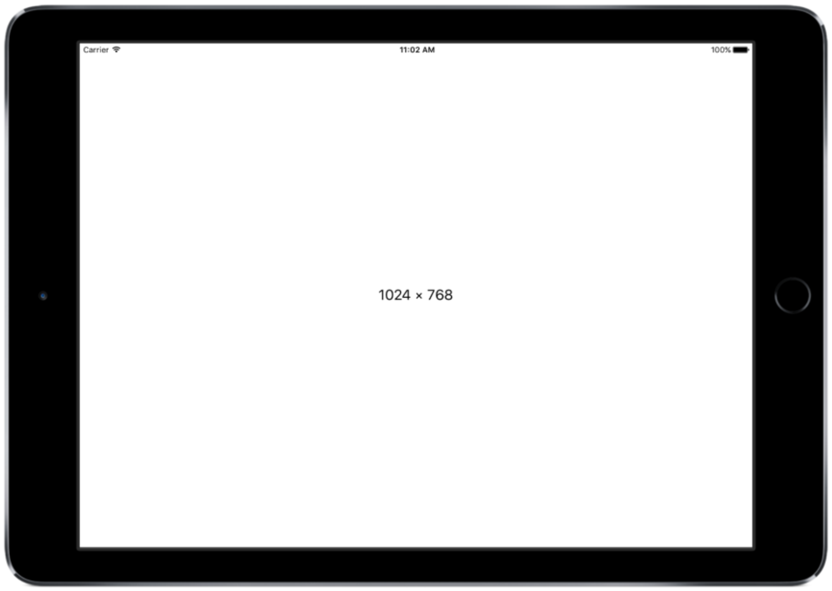
667 × 375

598 × 335

524.888916015625 × 341.33334350585 9

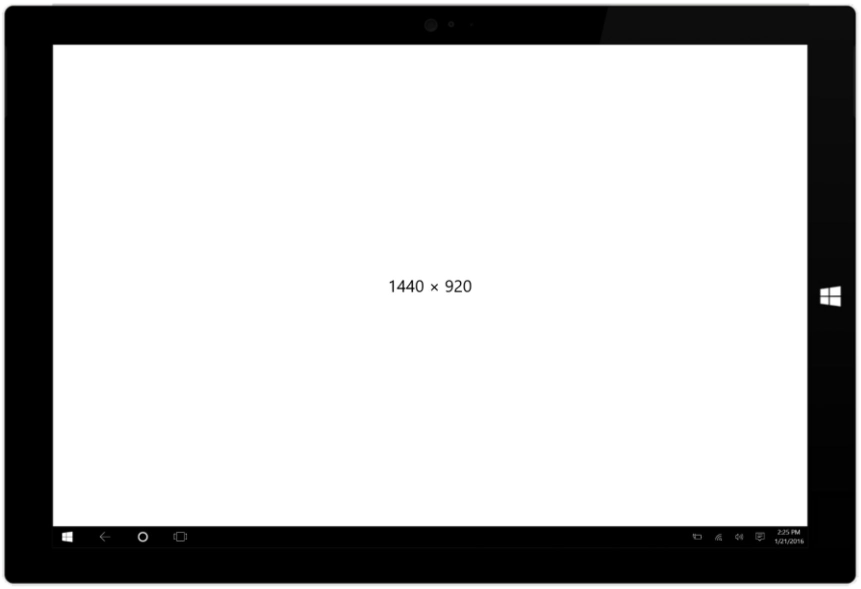
The screenshots for this book are designed only for portrait mode, so you'll need to turn this book sideways to see what the program looks like in landscape. The 598-pixel width on the Android excludes the area for the buttons; the 335-pixel height excludes the status bar, which always appears above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

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



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

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



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

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

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

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

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

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

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

```
ScreenOrientation = ScreenOrientation.Landscape
```

or

```
ScreenOrientation = ScreenOrientation.Portrait
```

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

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

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

or:

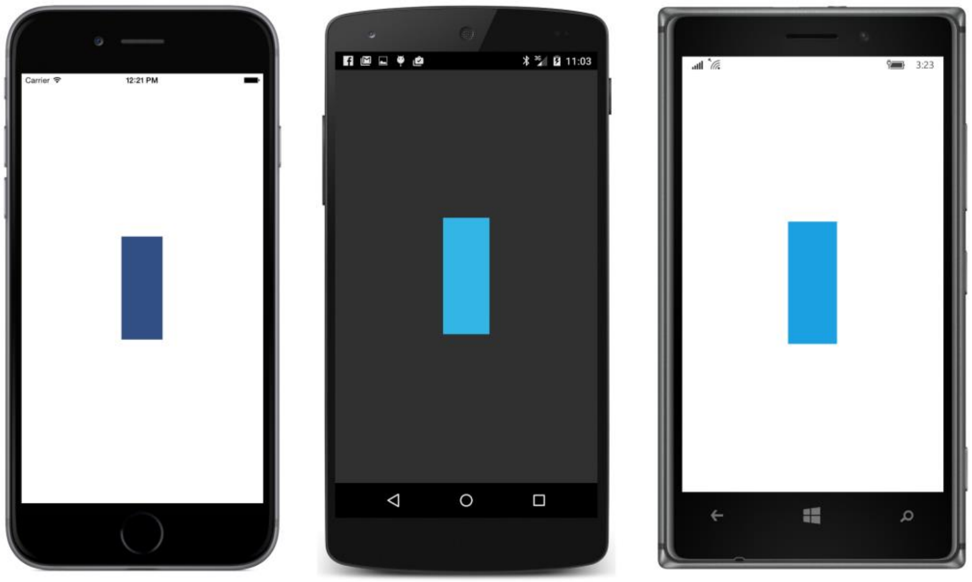
```
DisplayInformation.AutoRotationPreferences = DisplayOrientations.Portrait;
```

Metrical sizes

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

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

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



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

Estimated font sizes

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

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

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

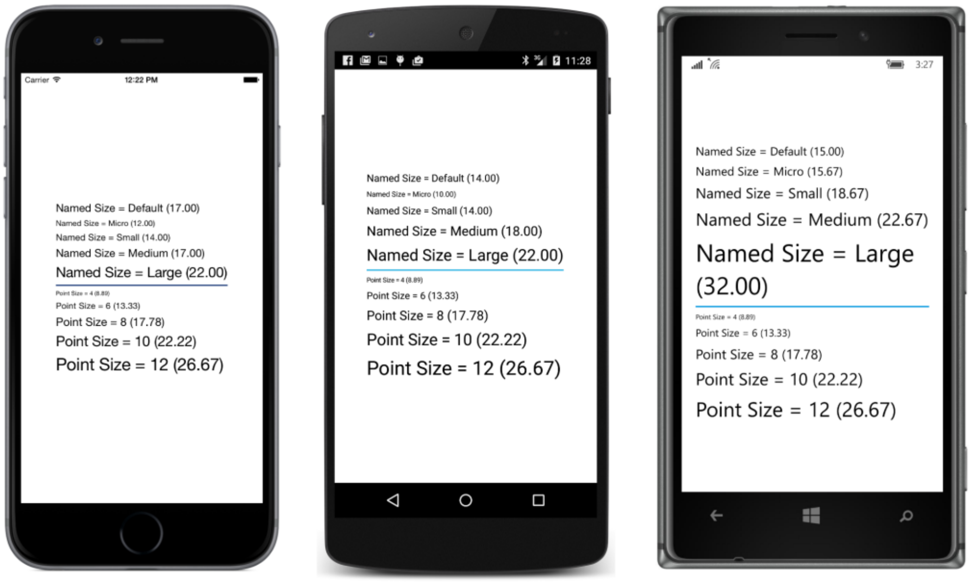
There are 72 points to the inch, so a 12-point font is one-sixth of an inch. Multiply by the DPI resolution of 160 and that's about 27 device-independent units.

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

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

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

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

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

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

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

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

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

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

        Content = stackLayout;
    }
}
```

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

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



Named Size = Default (17.00)
Named Size = Micro (12.00)
Named Size = Small (14.00)
Named Size = Medium (17.00)
Named Size = Large (22.00)
Point Size = 4 (8.89)
Point Size = 6 (13.33)
Point Size = 8 (17.78)
Point Size = 10 (22.22)
Point Size = 12 (26.67)

Named Size = Default (14.00)
Named Size = Micro (10.00)
Named Size = Small (14.00)
Named Size = Medium (18.00)
Named Size = Large (22.00)
Point Size = 4 (8.89)
Point Size = 6 (13.33)
Point Size = 8 (17.78)
Point Size = 10 (22.22)
Point Size = 12 (26.67)

Named Size = Default (15.00)
Named Size = Micro (15.67)
Named Size = Small (18.67)
Named Size = Medium (22.67)
Named Size = Large (32.00)
Point Size = 4 (8.89)
Point Size = 6 (13.33)
Point Size = 8 (17.78)
Point Size = 10 (22.22)
Point Size = 12 (26.67)

Fitting text to available size

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

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

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

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

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

For example, suppose you want to fit a text string containing 80 characters in a width of 320 units, and you'd like the font size to be as large as possible. Divide the width (320) by half the number of characters (40), and you get a font size of 8, which you can set to the `FontSize` property of `Label`. For text that's somewhat indeterminate and can't be tested beforehand, you might want to make this calculation a little more conservative to avoid surprises.

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

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

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

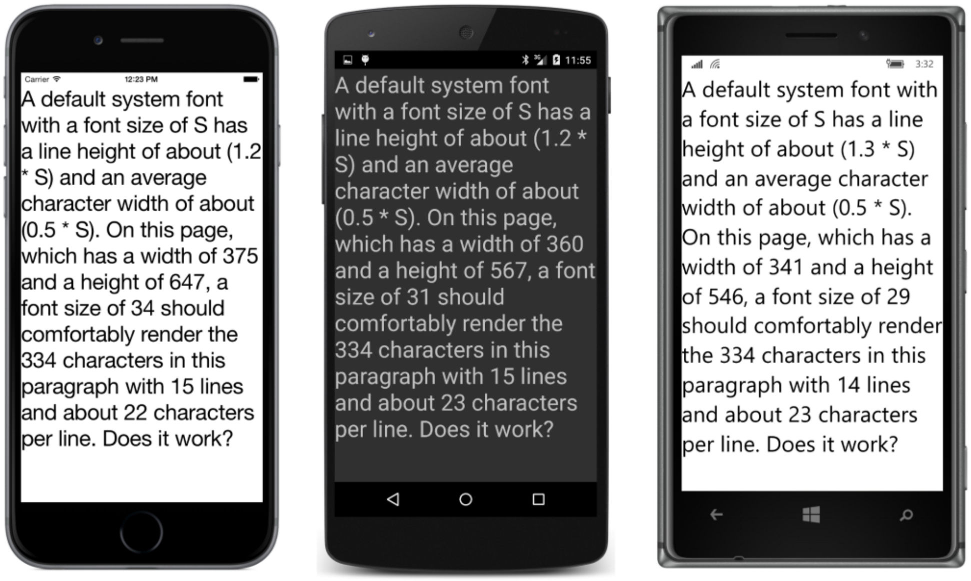
The **EstimatedFontSize** program uses `ContentView` in a slightly different manner: It sets the customary padding on the page to avoid the iOS status bar, but then it sets a `ContentView` as the content of that page. Hence, this `ContentView` is the same size as the page, but excluding the iOS status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

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

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

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

    void OnContentViewSizeChanged(object sender, EventArgs args)
    {
        string text =
            "A default system font with a font size of S " +
            "has a line height of about ({0:F1} * S) and an " +
            "average character width of about ({1:F1} * S). " +
            "On this page, which has a width of {2:F0} and a " +
            "height of {3:F0}, a font size of ?1 should " +
            "comfortably render the ??2 characters in this " +
            "paragraph with ?3 lines and about ?4 characters " +
            "per line. Does it work?";

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

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

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

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

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

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

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

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

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

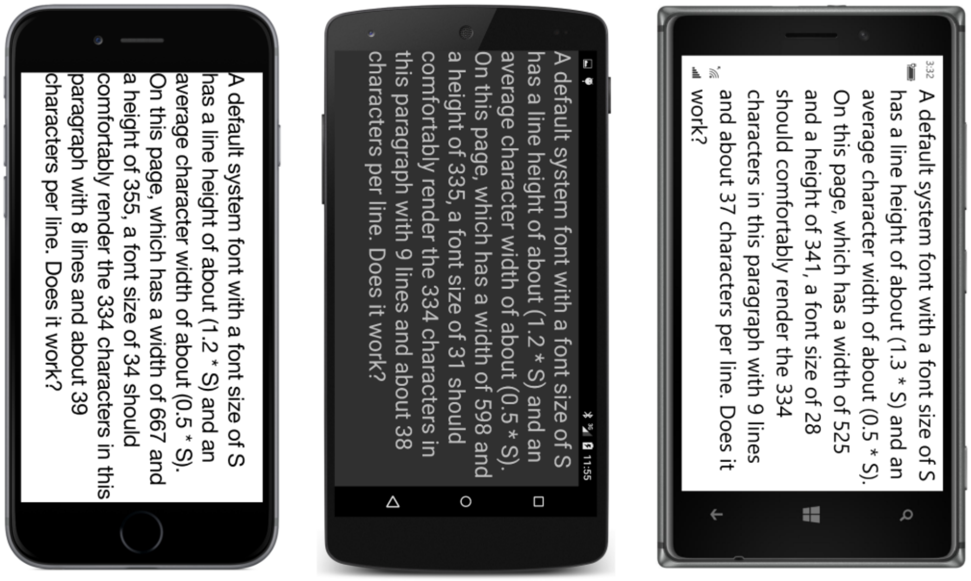


A default system font
with a font size of S has
a line height of about (1.2
* S) and an average
character width of about
(0.5 * S). On this page,
which has a width of 375
and a height of 647, a
font size of 34 should
comfortably render the
334 characters in this
paragraph with 15 lines
and about 22 characters
per line. Does it work?

A default system font
with a font size of S has a
line height of about (1.2 *
S) and an average
character width of about
(0.5 * S). On this page,
which has a width of 360
and a height of 567, a font
size of 31 should
comfortably render the
334 characters in this
paragraph with 15 lines
and about 23 characters
per line. Does it work?

A default system font with
a font size of S has a line
height of about (1.3 * S)
and an average character
width of about (0.5 * S).
On this page, which has a
width of 341 and a height
of 546, a font size of 29
should comfortably render
the 334 characters in this
paragraph with 14 lines
and about 23 characters
per line. Does it work?

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



A default system font with a font size of S
has a line height of about (1.2 * S) and an
average character width of about (0.5 * S).
On this page, which has a width of 667 and
a height of 355, a font size of 34 should
comfortably render the 334 characters in this
paragraph with 8 lines and about 39
characters per line. Does it work?

A default system font with a font size of S
has a line height of about (1.2 * S) and an
average character width of about (0.5 * S).
On this page, which has a width of 598 and
a height of 335, a font size of 31 should
comfortably render the 334 characters in
this paragraph with 9 lines and about 38
characters per line. Does it work?

A default system font with a font size of S
has a line height of about (1.3 * S) and an
average character width of about (0.5 * S).
On this page, which has a width of 525
and a height of 341, a font size of 28
should comfortably render the 334
characters in this paragraph with 9 lines
and about 37 characters per line. Does it
work?

A fit-to-size clock

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

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

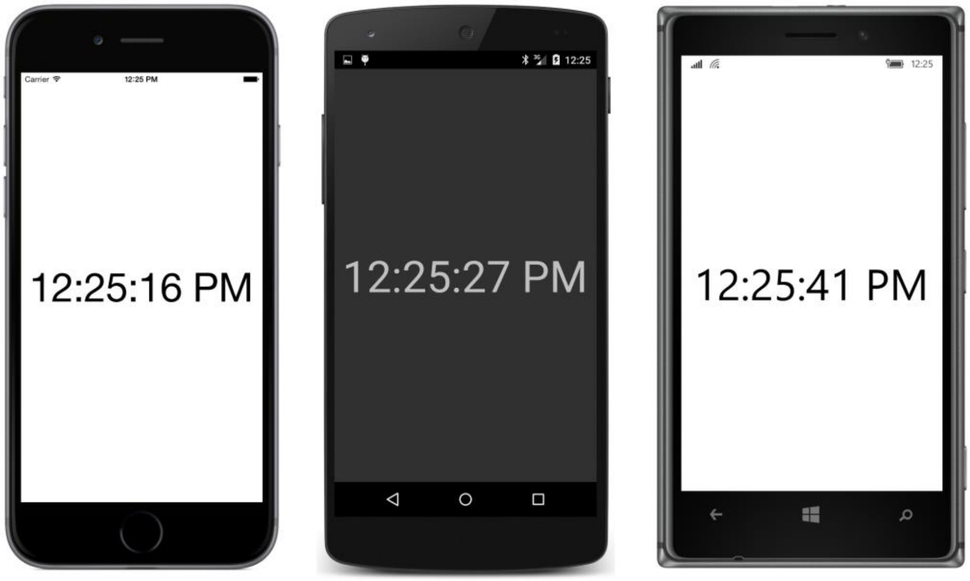
The **FitToSizeClock** program creates a `Label` for displaying the time and then sets two events: the `SizeChanged` event on the page for changing the font size, and the `Device.StartTimer` event for one-second intervals to change the `Text` property.

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

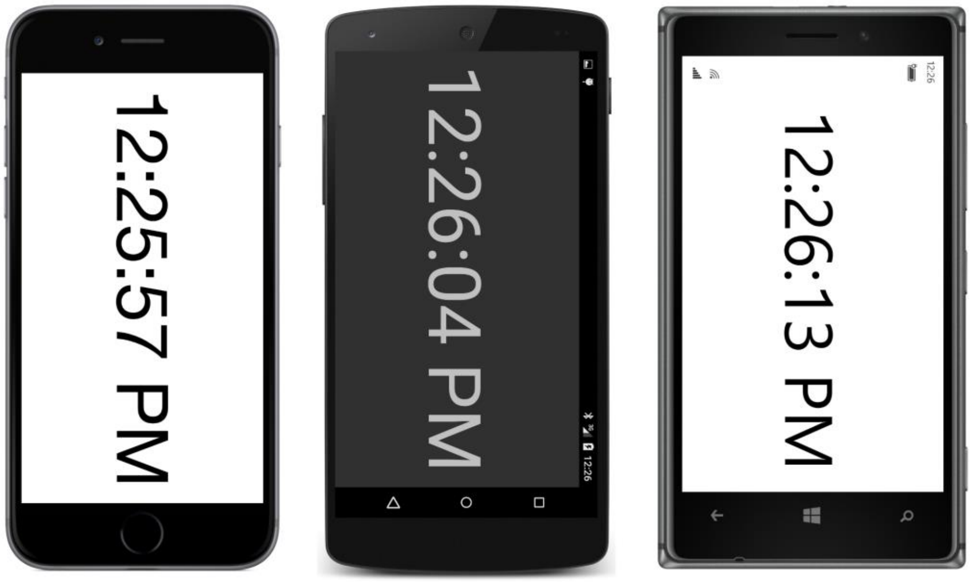
In this program, both event handlers simply change a property of the `Label`, and they are both expressed as lambda functions so that they can access the `Label` without it being stored as a field:

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

        Content = clockLabel;

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

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

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



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



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

Accessibility issues

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

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

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

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

Here's what you will discover:

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

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

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

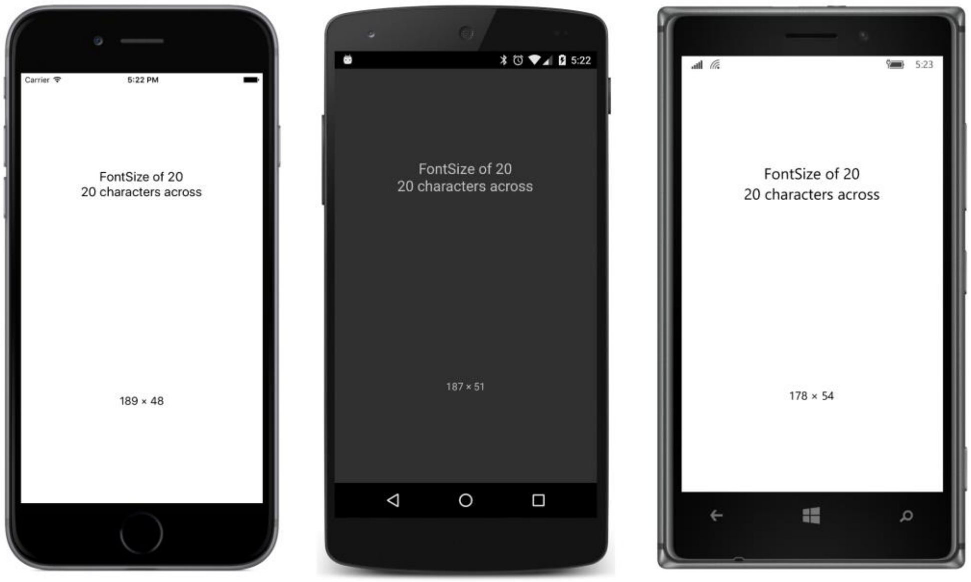
On Windows 10 Mobile, the values returned from `Device.GetNamedSize` do not depend on the accessibility setting, but all the text is displayed larger.

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

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

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

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

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

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

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

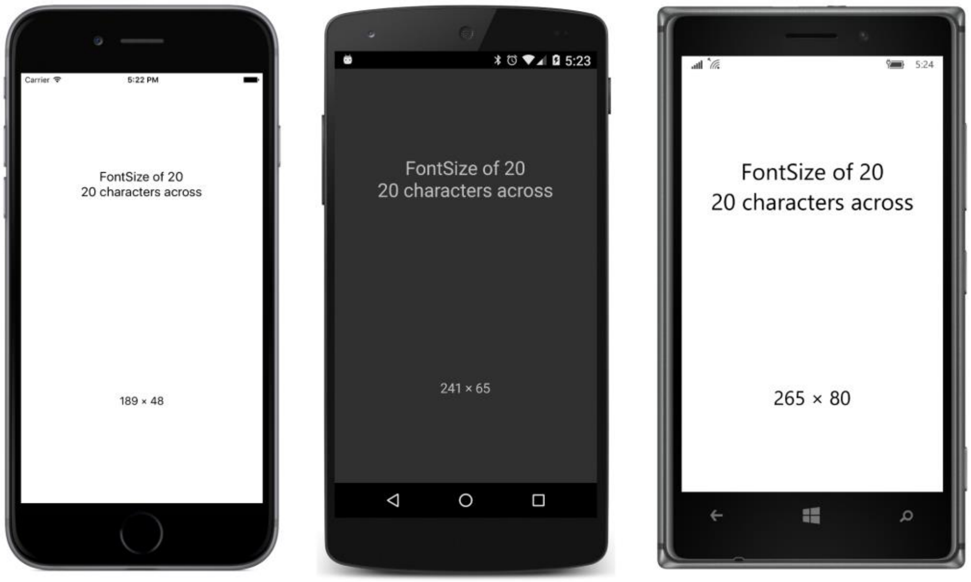


FontSize of 20
20 characters across
189 × 48

FontSize of 20
20 characters across
187 × 51

FontSize of 20
20 characters across
178 × 54

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



FontSize of 20
20 characters across
189 × 48

FontSize of 20
20 characters across
241 × 65

FontSize of 20
20 characters across
265 × 80

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

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

Empirically fitting text

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

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

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

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

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

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

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

        // Recalculate the Label height.
        label.FontSize = fontSize;
        SizeRequest sizeRequest =
            label.GetSizeRequest(containerWidth, Double.PositiveInfinity);

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

    public double FontSize { private set; get; }

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

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

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

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

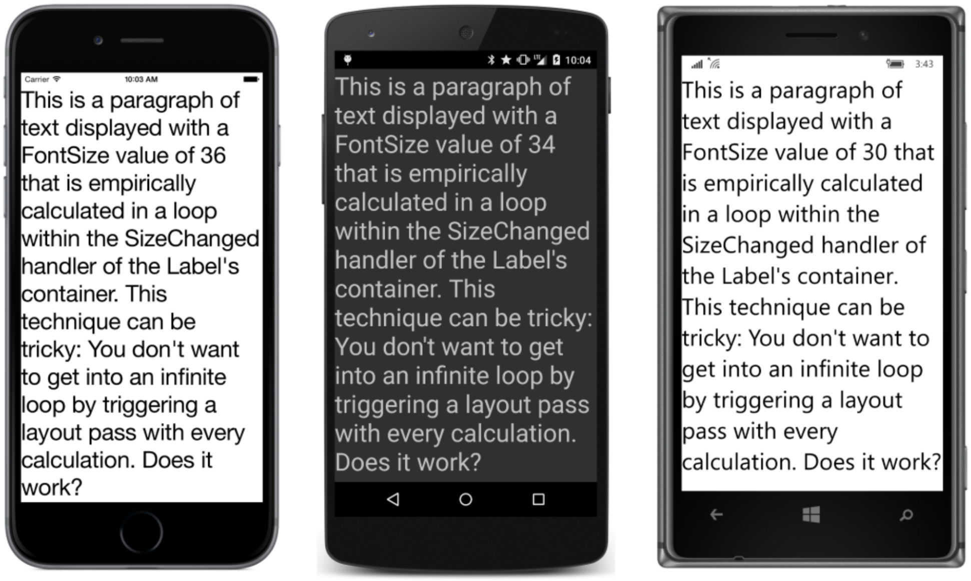
However, calling `GetSizeRequest` doesn't have any side effects. It doesn't cause a new size to be set on the element, which means that it doesn't cause a `SizeChanged` event to be fired, which means that it's safe to call in a `SizeChanged` handler.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

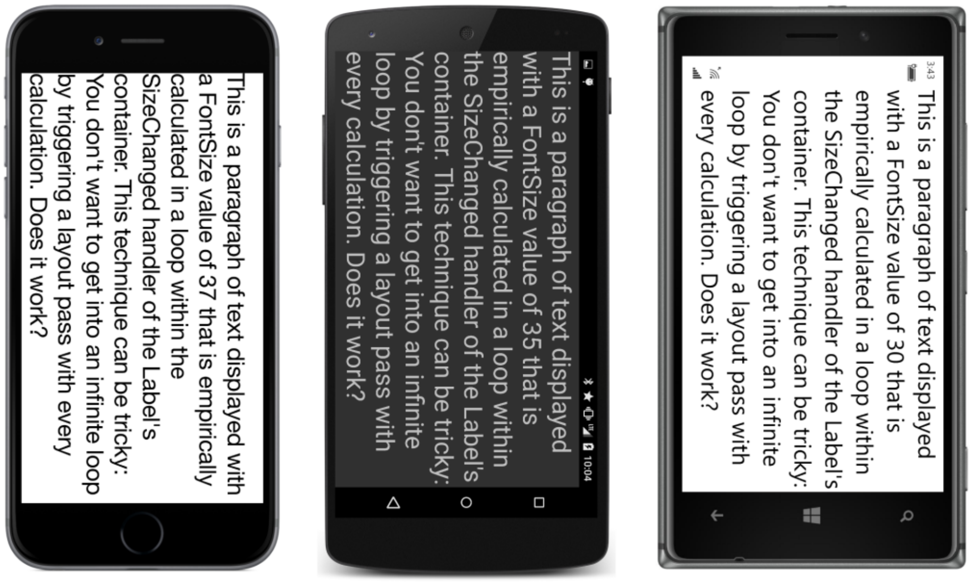


This is a paragraph of text displayed with a FontSize value of 36 that is empirically calculated in a loop within the SizeChanged handler of the Label's container. This technique can be tricky: You don't want to get into an infinite loop by triggering a layout pass with every calculation. Does it work?

This is a paragraph of text displayed with a FontSize value of 34 that is empirically calculated in a loop within the SizeChanged handler of the Label's container. This technique can be tricky: You don't want to get into an infinite loop by triggering a layout pass with every calculation. Does it work?

This is a paragraph of text displayed with a FontSize value of 30 that is empirically calculated in a loop within the SizeChanged handler of the Label's container. This technique can be tricky: You don't want to get into an infinite loop by triggering a layout pass with every calculation. Does it work?

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


This is a paragraph of text displayed with
a FontSize value of 37 that is empirically
calculated in a loop within the
SizeChanged handler of the Label's
container. This technique can be tricky:
You don't want to get into an infinite loop
by triggering a layout pass with every
calculation. Does it work?

This is a paragraph of text displayed
with a FontSize value of 35 that is
empirically calculated in a loop within
the SizeChanged handler of the Label's
container. This technique can be tricky:
You don't want to get into an infinite
loop by triggering a layout pass with
every calculation. Does it work?

This is a paragraph of text displayed
with a FontSize value of 30 that is
empirically calculated in a loop within
the SizeChanged handler of the Label's
container. This technique can be tricky:
You don't want to get into an infinite
loop by triggering a layout pass with
every calculation. Does it work?

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