Chapter 15
The interactive interface

Interactivity is the defining feature of modern computing. The many interactive views that Xamarin.Forms implements respond to touch gestures such as tapping and dragging, and a few even read keystrokes from the phone’s virtual keyboard.

These interactive views incorporate paradigms that are familiar to users, and even have names that are familiar to programmers: users can trigger commands with Button, specify a number from a range of values with Slider and Stepper, enter text from the phone’s keyboard using Entry and Editor, and select items from a collection with Picker, ListView, and TableView.

This chapter is devoted to demonstrating many of these interactive views.

View overview

Xamarin.Forms defines 20 instantiable classes that derive from View but not from Layout. You’ve already seen six of these classes in previous chapters: Label, BoxView, Button, Image, ActivityIndicator, and Progress.Bar.

This chapter focuses on eight views that allow the user to select or interact with basic .NET data types:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>Slider, Stepper</td>
</tr>
<tr>
<td>Boolean</td>
<td>Switch</td>
</tr>
<tr>
<td>String</td>
<td>Entry, Editor, SearchBar</td>
</tr>
<tr>
<td>DateTime</td>
<td>DatePicker, TimePicker</td>
</tr>
</tbody>
</table>

These views are often the visual representations of underlying data items. In the next chapter, you’ll begin to explore data binding, which is a feature of Xamarin.Forms that links properties of views with properties of other classes so that these views and underlying data can be structured in correspondences.

Four of the remaining six views are discussed in later chapters. In Chapter 16, “Data binding,” you’ll see:

- WebView, to display webpages or HTML.

Chapter 19, "Collection views" covers these three views:

- Picker, selectable strings for program options.
- ListView, a scrollable list of data items of the same type.
Table View, a list of items separated into categories, which is flexible enough to be used for data, forms, menus, or settings.

Two views are not covered in this edition of this book:

- Map, an interactive map display.
- OpenGLView, which allows a program to display 2-D and 3-D graphics by using the Open Graphics Library.

**Slider and Stepper**

Both Slider and Stepper let the user select a numeric value from a range. They have nearly identical programming interfaces but incorporate very different visual and interactive paradigms.

**Slider basics**

The Xamarin.Forms Slider is a horizontal bar that represents a range of values between a minimum at the left and a maximum at the right. (The Xamarin.Forms Slider does not support a vertical orientation.) The user selects a value on the Slider a little differently on the three platforms: On iOS devices, the user drags a round “thumb” along the horizontal bar. The Android and Windows 10 Mobile Slider views also have thumbs, but they are too small for a touch target, and the user can simply tap on the horizontal bar, or drag a finger to a specific location.

The Slider defines three public properties of type double, named Minimum, Maximum, and Value. Whenever the Value property changes, the Slider fires a ValueChanged event indicating the new value.

When displaying a Slider you’ll want a little padding at the left and right to prevent the Slider from extending to the edges of the screen. The XAML file in the SliderDemo program applies the Padding to the StackLayout, which is parent to both a Slider and a Label that is intended to display the current value of the Slider:

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

  <StackLayout Padding="10, 0">
    <Slider VerticalOptions="CenterAndExpand"
            ValueChanged="OnSliderValueChanged" />

    <Label x:Name="label"
           FontSize="Large"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />
  </StackLayout>
</ContentPage>
```
When the program starts up, the Label displays nothing, and the Slider thumb is positioned at the far left:

Do not set HorizontalOptions on the Slider to Start, Center, or End without also setting WidthRequest to an explicit value, or the Slider will collapse into a very small or even unusable width.

The Slider notifies code of changes to the Value property by firing the ValueChanged event. The event is fired if Value is changed programatically or by user manipulation. Here’s the SliderDemo code-behind file with the event handler:

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

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        label.Text = string.Format("Slider = {0}\n", args.NewValue);
    }
}
```

As usual, the first argument to the event handler is the object firing the event, in this case the Slider, and the second argument provides more information about this event. The handler for ValueChanged is of type EventHandler<ValueChangedEventArgs>, which means that the second argument to the handler is a ValueChangedEventArgs object. ValueChangedEventArgs defines two properties
of type `double` named `OldValue` and `NewValue`. This particular handler simply uses `NewValue` in a string that it sets to the `Text` property of the `Label`:

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    label.Text = String.Format("Slider = {0:F2}" , args.NewValue);
}
```

A little experimentation reveals that the default `Minimum` and `Maximum` settings for `Slider` are 0 and 1. At the time this chapter is being written, the `Slider` on the Windows platforms has a default increment of 0.1. For other settings of `Minimum` and `Maximum`, the `Slider` is restricted to 10 increments or steps of 1, whichever is less. (A more flexible `Slider` is presented in Chapter 27, “Custom renderers.”)

If you’re not happy with the excessive number of decimal points displayed on the iOS screen, you can reduce the number of decimal places with a formatting specification in `String.Format`:

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    label.Text = String.Format("Slider = {0:F2}" , args.NewValue);
}
```

This is not the only way to write the `ValueChanged` handler. An alternative implementation involves casting the first argument to a `Slider` object and then accessing the `Value` property directly:

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    Slider slider = (Slider)sender;
    label.Text = String.Format("Slider = {0}" , slider.Value);
}
```

Using the `sender` argument is a good approach if you’re sharing the event handler among multiple `Slider` views. By the time the `ValueChanged` event handler is called, the `Value` property already has its new value.
You can set the **Minimum** and **Maximum** properties of the **Slider** to any negative or positive value, with the stipulation that **Maximum** is always greater than **Minimum**. For example, try this:

```xml
<Slider ValueChanged="OnSliderValueChanged"
Maximunm="100"
VerticalOptions="CenterAndExpand" />
```

Now the **Slider** value ranges from 0 to 100.

### Common pitfalls

Suppose you want the **Slider** value to range from 1 to 100. You can set both **Minimum** and **Maximum** like this:

```xml
<Slider ValueChanged="OnSliderValueChanged"
Minimum="1"
Maximunm="100"
VerticalOptions="CenterAndExpand" />
```

However, when you run the new version of the program, an **ArgumentException** is raised with the text explanation “Value was an invalid value for Minimum.” What does that mean?

When the XAML parser encounters the **Slider** tag, a **Slider** is instantiated, and then the properties and events are set in the order in which they appear in the **Slider** tag. But when the **Minimum** property is set to 1, the **Maximum** value now equals the **Minimum** value. That can’t be. The **Maximum** property must be greater than the **Minimum**. The **Slider** signals this problem by raising an exception.

Internal to the **Slider** class, the **Minimum** and **Maximum** values are compared in a callback method set to the **validateValue** argument to the **BindableProperty.Create** method calls that create the **Minimum** and **Maximum** bindable properties. The **validateValue** callback returns true if **Minimum** is less than **Maximum**, indicating that the values are valid. A return value of false from this callback triggers the exception. This is the standard way that bindable properties implement validity checks.

This isn’t a problem specific to XAML. It also happens if you instantiate and initialize the **Slider** properties in this order in code. The solution is to reverse the order that **Minimum** and **Maximum** are set. First set the **Maximum** property to 100. That’s legal because now the range is between 0 and 100. Then set the **Minimum** property to 1:

```xml
<Slider ValueChanged="OnSliderValueChanged"
Maximunm="100"
Minimum="1"
VerticalOptions="CenterAndExpand" />
```

However, this results in another run-time error. Now it’s a **NullReferenceException** in the **ValueChanged** handler. Why is that?

The **Value** property of the **Slider** must be within the range of **Minimum** and **Maximum** values, so when the **Minimum** property is set to 1, the **Slider** automatically adjust its **Value** property to 1.
Internally, **Value** is adjusted in a callback method set to the **coerceValue** argument of the **BindableProperty.Create** method calls for the **Minimum**, **Maximum**, and **Value** properties. The callback method returns an adjusted value of the property being set after being subjected to this coercion. In this example, when **Minimum** is set to 1, the **coerceValue** method sets the slider’s **Value** property to 1, and the **coerceValue** callback returns the new value of **Minimum**, which remains at the value 1.

However, as a result of the coercion, the **Value** property has changed, and this causes the **ValueChanged** event to fire. The **ValueChanged** handler in the code-behind file attempts to set the **Text** property of the **Label**, but the XAML parser has not yet instantiated the **Label** element. The **Label** field is **null**.

There are a couple of solutions to this problem. The safest and most general solution is to check for a **null** value for **Label** right in the event handler:

```csharp
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    if (label != null)
    {
        label.Text = String.Format("Slider = {0}", args.NewValue);
    }
}
```

However, you can also fix the problem by moving the assignment of the **ValueChanged** event in the tag to after the **Maximum** and **Minimum** properties have been set:

```xml
<Slider Maximum="100"
    Minimum="1"
    ValueChanged="OnSliderValueChanged"
    VerticalOptions="CenterAndExpand" />
```

The **Value** property is still coerced to 1 after the **Minimum** property is set, but the **ValueChanged** event handler has not yet been assigned, so no event is fired.

Let’s assume that the **Slider** has the default range of 0 to 1. You might want the **Label** to display the initial value of the **Slider** when the program first starts up. You could initialize the **Text** property of the **Label** to “**Slider = 0**” in the XAML file, but if you ever wanted to change the text to something a little different, you’d need to change it in two places.

You might try giving the **Slider** a name of **slider** in the XAML file and then add some code to the constructor:

```csharp
public SliderDemoPage()
{
    InitializeComponent();

    slider.Value = 0;
}
```

All the elements in the XAML file have been created and initialized when **InitializeComponent** returns, so if this code causes the **Slider** to fire a **ValueChanged** event, that shouldn’t be a problem.
But it won’t work. The value of the Slider is already 0, so setting it to 0 again does nothing. You could try this:

```csharp
public SliderDemoPage()
{
    InitializeComponent();

    slider.Value = 1;
    slider.Value = 0;
}
```

That will work. But you might want to add a comment to the code so that another programmer doesn’t later remove the statement that sets `Value` to 1 because it appears to be unnecessary.

Or you could simulate an event by calling the handler directly. The two arguments to the `ValueChangedEventArgs` constructor are the old value and the new value (in that order), but the `OnSliderValueChanged` handler uses only the `NewValue` property, so it doesn’t matter what the other argument is or whether they’re equal:

```csharp
public partial class SliderDemoPage : ContentPage
{
    public SliderDemoPage()
    {
        InitializeComponent();

        OnSliderValueChanged(null, new ValueChangedEventArgs(0, 0));
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        label.Text = String.Format("Slider = {0}", args.NewValue);
    }
}
```

That works as well. But remember to set the arguments to the call to `OnSliderValueChanged` so that they agree with what the handler expects. If you replaced the handler body with code that casts the `sender` argument to the Slider object, you then need a valid first argument in the `OnSliderValueChanged` call.

The problems involving the event handler disappear when you connect the Label with the Slider by using data bindings, which you’ll learn about in the next chapter. You’ll still need to set the properties of the Slider in the correct order, but you’ll experience none of the problems with the event handler because the event handler will be gone.

### Slider color selection

Here’s a program named `RgbSliders` that contains three Slider elements for selecting red, green, and blue components of a Color. An implicit style for Slider sets the `Maximum` value to 255:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
The Slider elements alternate with three Label elements to display their values, and the StackLayout concludes with a BoxView to show the resultant color.

The constructor of the code-behind file initializes the Slider settings to 128 for a medium gray.

The shared ValueChanged handler checks to see which Slider has changed, and hence which Label needs to be updated, and then computes a new color for the BoxView:

```csharp
public partial class RgbSlidersPage : ContentPage
{
    public RgbSlidersPage()
```
{  
  InitializeComponent();

  redSlider.Value = 128;
  greenSlider.Value = 128;
  blueSlider.Value = 128;
}

void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
  if (sender == redSlider)
  {
    redLabel.Text = String.Format("Red = {0:X2}", (int)redSlider.Value);
  }
  else if (sender == greenSlider)
  {
    greenLabel.Text = String.Format("Green = {0:X2}", (int)greenSlider.Value);
  }
  else if (sender == blueSlider)
  {
    blueLabel.Text = String.Format("Blue = {0:X2}", (int)blueSlider.Value);
  }

  boxView.Color = Color.FromRgb((int)redSlider.Value,
    (int)greenSlider.Value,
    (int)blueSlider.Value);
}

Strictly speaking, the if and else statements here are not required. The code can simply set all three labels regardless of which slider is changing. The event handler accesses all three sliders anyway for setting a new color:
You can turn the phone sideways, but the BoxView becomes much shorter, particularly on the Windows 10 Mobile device, where the Slider seems to have a vertical height beyond what’s required. Once the Grid is introduced in Chapter 18, you’ll see how it becomes easier for applications to respond to orientation changes.

The following TextFade program uses a single Slider to control the Opacity and horizontal position of two Label elements in an AbsoluteLayout. In the initial layout, both Label elements are positioned at the left center of the AbsoluteLayout, but the second one has its Opacity set to 0:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TextFade.TextFadePage"
             Padding="10, 0, 10, 20">
  <StackLayout>
    <AbsoluteLayout VerticalOptions="CenterAndExpand">
      <Label x:Name="label1"
             Text="TEXT"
             FontSize="Large"
             AbsoluteLayout.LayoutBounds="0, 0.5"
             AbsoluteLayout.LayoutFlags="PositionProportional" />
      <Label x:Name="label2"
             Text="FADE"
             FontSize="Large"
             Opacity="0"
             AbsoluteLayout.LayoutBounds="0, 0.5"
             AbsoluteLayout.LayoutFlags="PositionProportional" />
    </AbsoluteLayout>
  </StackLayout>
</ContentPage>
```
The Slider event handler moves both Label elements from left to right across the screen. The proportional positioning helps a lot here because the Slider values range from 0 to 1, which results in the Label elements being positioned progressively from the far left to the far right of the screen:

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

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        AbsoluteLayout.SetLayoutBounds(label1,
            new Rectangle(args.NewValue, 0.5, AbsoluteLayout.AutoSize,
                AbsoluteLayout.AutoSize));

        AbsoluteLayout.SetLayoutBounds(label2,
            new Rectangle(args.NewValue, 0.5, AbsoluteLayout.AutoSize,
                AbsoluteLayout.AutoSize));

        label1.Opacity = 1 - args.NewValue;
        label2.Opacity = args.NewValue;
    }
}
```

At the same time, the Opacity values are set so that one Label seems to fade into the other as both labels move across the screen:
The Stepper difference

The Stepper view has very nearly the same programming interface as the Slider: It has Minimum, Maximum, and Value properties of type double and fires a ValueChanged event handler.

However, the Maximum property of Stepper has a default value of 100, and Stepper also adds an Increment property with a default value of 1. The Stepper visuals consist solely of two buttons labeled with minus and plus signs. Presses of those two buttons change the value incrementally between Minimum to Maximum based on the Increment property.

Although Value and other properties of Stepper are of type double, Stepper is often used for the selection of integral values. You probably don’t want the value of ((Maximum – Minimum) ÷ Increment) to be as high as 100, as the default values suggest. If you press and hold your finger on one of the buttons, you’ll trigger a typematic repeat on iOS, but not on Android or Windows 10 Mobile. Unless your program provides another way for the user to change the Stepper value (perhaps with a text Entry view), you don’t want to force the user to press a button 100 times to get from Minimum to Maximum.

The StepperDemo program sets the Maximum property of the Stepper to 10 and uses the Stepper as a rudimentary design aid in determining an optimum border width for a Button border. The Button at the top of the StackLayout is solely for display purposes and has the necessary property settings of BackgroundColor andBorderColor to enable the border display on Android and Windows 10 Mobile.

The Stepper is the last child in the following StackLayout. Between the Button and Stepper are a pair of Label elements for displaying the current Stepper value:
The Label displaying the Stepper value is initialized from the constructor of the code-behind file. With each change in the Value property of the Stepper, the event handler displays the new value and sets the Button border width:

```csharp
public partial class StepperDemoPage : ContentPage {
    public StepperDemoPage()
    {
```
InitializeComponent();

    // Initialize display.
    OnStepperValueChanged(stepper, null);
}

void OnStepperValueChanged(object sender, ValueChangedEventArgs args)
{
    Stepper stepper = (Stepper)sender;
    button.BorderWidth = stepper.Value;
    label.Text = stepper.Value.ToString("F0");
}

Switch and CheckBox

Application programs often need Boolean input from the user, which requires some way for the user to toggle a program option to On or Off, Yes or No, True or False, or however you want to think of it. In Xamarin.Forms, this is a view called the Switch.

Switch basics

Switch defines just one property on its own, named IsToggled of type bool, and it fires the Toggled event to indicate a change in this property. In code, you might be inclined to give a Switch a name of switch, but that's a C# keyword, so you'll want to pick something else. In XAML, however, you can set the x:Name attribute to switch, and the XAML parser will smartly create a field named
@switch, which is how C# allows you to define a variable name using a C# keyword.

The **SwitchDemo** program creates two Switch elements with two identifying labels: “Italic” and “Boldface”. Each Switch has its own event handler, which formats the larger Label at the bottom of the StackLayout:

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

    <StackLayout Padding="10, 0">
        <StackLayout HorizontalOptions="Center"
            VerticalOptions="CenterAndExpand">
            <StackLayout Orientation="Horizontal"
                HorizontalOptions="End">
                <Label Text="Italic: "
                    VerticalOptions="Center" />
                <Switch Toggled="OnItalicSwitchToggled"
                    VerticalOptions="Center" />
            </StackLayout>
        </StackLayout>
        <StackLayout Orientation="Horizontal"
            HorizontalOptions="End">
            <Label Text="Boldface: "
                VerticalOptions="Center" />
            <Switch Toggled="OnBoldSwitchToggled"
                VerticalOptions="Center" />
        </StackLayout>
    </StackLayout>

    <Label x:Name="label"
        Text="Just a little passage of some sample text that can be formatted in italic or boldface by toggling the two Switch elements."
        FontSize="Large"
        HorizontalTextAlignment="Center"
        VerticalOptions="CenterAndExpand" />

</StackLayout>
</ContentPage>
```

The Toggled event handler has a second argument of ToggledEventArgs, which has a Value property of type bool that indicates the new state of the IsToggled property. The event handlers in **SwitchDemo** use this value to set or clear the particular FontAttributes flag in the FontAttributes property of the long Label:

```csharp
public partial class SwitchDemoPage : ContentPage
{
    public SwitchDemoPage()
    {
        InitializeComponent();
    }
```
void OnItalicSwitchToggled(object sender, ToggledEventArgs args)
{
    if (args.Value)
    {
        label.FontAttributes |= FontAttributes.Italic;
    }
    else
    {
        label.FontAttributes &= ~FontAttributes.Italic;
    }
}

void OnBoldSwitchToggled(object sender, ToggledEventArgs args)
{
    if (args.Value)
    {
        label.FontAttributes |= FontAttributes.Bold;
    }
    else
    {
        label.FontAttributes &= ~FontAttributes.Bold;
    }
}

The Switch has a different appearance on the three platforms:

Notice that the program aligns the two Switch views, which gives it a more attractive look, but which also means that the text labels are necessarily somewhat misaligned. To accomplish this formatting, the XAML file puts each of the pair of Label and Switch elements in a horizontal StackLayout.
Each horizontal StackLayout has its HorizontalOptions set to End, which aligns each StackLayout at the right, and a parent StackLayout centers the collection of labels and switches on the screen with a HorizontalOptions setting of Center. Within the horizontal StackLayout, both views have their VerticalOptions properties set to Center. If the Switch is taller than the Label, then the Label is vertically centered relative to the Switch. But if the Label is taller than the Switch, the Switch is also vertically centered relative to the Label.

A traditional CheckBox

In more traditional graphical environments, the user-interface object that allows users to choose a Boolean value is called a CheckBox, usually featuring some text with a box that can be empty or filled with an X or a check mark. One advantage of the CheckBox over the Switch is that the text identifier is part of the visual and doesn’t need to be added with a separate Label.

One way to create custom views in Xamarin.Forms is by writing special classes called renderers that are specific to each platform and that reference views in each platform. That is demonstrated in Chapter 27.

However, it’s also possible to create custom views right in Xamarin.Forms by assembling a view from other views. You first derive a class from ContentView, set its Content property to a StackLayout (for example), and then add one or more views on that. (You saw an example of this technique in the ColorView class in Chapter 8.) You’ll probably also need to define one or more properties, and possibly some events, but you’ll want to take advantage of the bindable infrastructure established by the BindableObject and BindableProperty classes. That allows your properties to be styled and to be targets of data bindings.

A CheckBox consists of just two Label elements on a ContentView: one Label displays the text associated with the CheckBox, while the other displays a box. A TapGestureRecognizer detects when the CheckBox is tapped.

A CheckBox class has already been added to the Xamarin.FormsBook.Toolkit library that is included in the downloadable code for this book. Here’s how you would do it on your own:

In Visual Studio, you can select Forms Xaml Page from the Add New Item dialog box. However, this creates a class that derives from ContentPage when you really want a class that derives from ContentView. Simply change the root element of the XAML file from ContentPage to ContentView, and change the base class in the code-behind file from ContentPage to ContentView.

In Xamarin Studio, however, you can simply choose Forms ContentView Xaml from the New File dialog.

Here’s the CheckBox.xaml file:

```xml
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Xamarin.FormsBook.Toolkit.CheckBox">
```
That Unicode character \u2610 is called the Ballot Box character, and it's just an empty square. Character \u2611 is a Ballot Box with Check, while \u2612 is a Ballot Box with X. To indicate a checked state, this CheckBox code-behind file sets the Text property of boxLabel to \u2611 (as you'll see shortly).

The code-behind file of CheckBox defines three properties:

- Text
- FontSize
- IsChecked

CheckBox also defines an event named IsCheckedChanged.

Should CheckBox also define FontAttributes and FontFamily properties like Label and Button do? Perhaps, but these additional properties are not quite as crucial for views devoted to user interaction.

All three of the properties that CheckBox defines are backed by bindable properties. The code-behind file creates all three BindableProperty objects, and the property-changed handlers are defined as lambda functions within these methods.

Keep in mind that the property-changed handlers are static, so they need to cast the first argument to a CheckBox object to reference the instance properties and events in the class. The property-changed handler for IsChecked is responsible for changing the character representing the checked and unchecked state and firing the IsCheckedChanged event:
	namespace Xamarin.FormsBook.Toolkit
{
    public partial class CheckBox : ContentView
    {
        public static readonly BindableProperty TextProperty =
            BindableProperty.Create("Text",
            typeof(string),
            typeof(CheckBox),
            null, propertyChanged: (Bindable, oldValue, newValue) =>
            {
                ((CheckBox)Bindable).TextLabel.Text = (string)newValue;
            });
    }
}
public static readonly BindableProperty FontSizeProperty = BindableProperty.Create(
    "FontSize",
typeof(double),
typeof(CheckBox),
Device.GetNamedSize(NamedSize.Default, typeof(Label)),
propertyChanged: (bindable, oldValue, newValue) =>
{
    CheckBox checkbox = (CheckBox)bindable;
    checkbox.boxLabel.FontSize = (double)newValue;
    checkbox.textLabel.FontSize = (double)newValue;
});

public static readonly BindableProperty IsCheckedProperty = BindableProperty.Create(
    "IsChecked",
typeof(bool),
typeof(CheckBox),
false,
propertyChanged: (bindable, oldValue, newValue) =>
{
    // Set the graphic.
    CheckBox checkbox = (CheckBox)bindable;
    checkbox.boxLabel.Text = (bool)newValue ? "\u2611" : "\u2610";

    // Fire the event.
    EventHandler<bool> eventHandler = checkbox.CheckedChanged;
    if (eventHandler != null)
    {
        eventHandler(checkbox, (bool)newValue);
    }
});

public event EventHandler<bool> CheckedChanged;

public CheckBox()
{
    InitializeComponent();
}

public string Text
{
    set { SetValue(TextProperty, value); }
    get { return (string)GetValue(TextProperty); }
}

>TypeConverter(typeof(FontSizeConverter))
public double FontSize
{
    set { SetValue(FontSizeProperty, value); }
    get { return (double)GetValue(FontSizeProperty); }
}
public bool IsChecked
{
    set { SetValue(IsCheckedProperty, value); }
    get { return (bool)GetValue(IsCheckedProperty); }
}

// TapGestureRecognizer handler.
void OnCheckBoxTapped(object sender, EventArgs args)
{
    IsChecked = !IsChecked;
}

Notice the TypeConverter on the FontSize property. That allows the property to be set in XAML with attribute values such as “Small” and “Large”.

The Tapped handler for the TapGestureRecognizer is at the bottom of the class and simply toggles the IsChecked property by using the C# logical negation operator. An even shorter statement to toggle a Boolean variable uses the exclusive-OR assignment operator:

IsChecked ^= true;

The CheckBoxDemo program is very similar to the SwitchDemo program except that the markup is considerably simplified because the CheckBox includes its own Text property:

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

<StackLayout Padding="10, 0">
    <StackLayout HorizontalOptions="Center"
                   VerticalOptions="CenterAndExpand">
        <toolkit:CheckBox Text="Italic"
                           FontSize="Large"
                           CheckedChanged="OnItalicCheckBoxChanged" />
        <toolkit:CheckBox Text="Boldface"
                           FontSize="Large"
                           CheckedChanged="OnBoldCheckBoxChanged" />
    </StackLayout>

    <Label x:Name="label"
           Text="Just a little passage of some sample text that can be formatted in italic or boldface by toggling the two custom CheckBox views."
           FontSize="Large"
           HorizontalTextAlignment="Center"
           VerticalOptions="CenterAndExpand" />
</StackLayout>

</ContentPage>
The code-behind file is also very similar to the earlier program:

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

    void OnItalicCheckBoxChanged(object sender, bool isChecked)
    {
        if (isChecked)
        {
            label.FontAttributes |= FontAttributes.Italic;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Italic;
        }
    }

    void OnBoldCheckBoxChanged(object sender, bool ischecked)
    {
        if (ischecked)
        {
            label.FontAttributes |= FontAttributes.Bold;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Bold;
        }
    }
}
```

Interestingly, the character for the checked box shows up in color on the Android and Windows platforms:
Typing text

Xamarin.Forms defines three views for obtaining text input from the user:

- Entry for a single line of text.
- Editor for multiple lines of text.
- SearchBar for a single line of text specifically for search operations.

Both Entry and Editor derive from InputView, which derives from View. SearchBar derives directly from View.

Both Entry and SearchBar implement horizontal scrolling if the entered text exceeds the width of the view. The Editor implements word wrapping and is capable of vertical scrolling for text that exceeds its height.

Keyboard and focus

Entry, Editor, and SearchBar are different from all the other views in that they make use of the phone’s onscreen keyboard, sometimes called the virtual keyboard. From the user’s perspective, tapping the Entry, Editor, or SearchBar view invokes the onscreen keyboard, which slides in from the bottom. Tapping anywhere else on the screen (except another Entry, Editor, or SearchBar view) often makes the keyboard go away, and sometimes the keyboard can be dismissed in other ways.
From the program’s perspective, the presence of the keyboard is closely related to *input focus*, a concept that originated in desktop graphical user interface environments. On both desktop environments and mobile devices, input from the keyboard can be directed to only one user-interface object at a time, and that object must be clearly selectable and identifiable by the user. The object that receives keyboard input is known as the object with *keyboard input focus*, or more simply, just *input focus* or *focus*.

The **VisualElement** class defines several methods, properties, and events related to input focus:

- The **Focus** method attempts to set input focus to a visual element and returns `true` if successful.
- The **Unfocus** method removes input focus from a visual element.
- The **IsFocused** get-only property is `true` if a visual element currently has input focus.
- The **Focused** event is fired when a visual element acquires input focus.
- The **Unfocused** event is fired when a visual element loses input focus.

As you know, mobile environments make far less use of the keyboard than desktop environments do, and most mobile views (such as the Slider, Stepper, and Switch that you’ve already seen) don’t make use of the keyboard at all. Although these five focus-related members of the **VisualElement** class appear to implement a generalized system for passing input focus between visual elements, they really only pertain to **Entry**, **Editor**, and **SearchBar**.

These views signal that they have input focus with a flashing caret showing the text input point, and they trigger the keyboard to slide up. When the view loses input focus, the keyboard slides back down.

A view must have its **IsEnabled** property set to `true` (the default state) to acquire input focus, and of course the **IsVisible** property must also be `true` or the view won’t be on the screen at all.

**Choosing the keyboard**

**Entry** and **Editor** are different from **SearchBar** in that they both derive from **InputView**. Interestingly, although **Entry** and **Editor** define similar properties and events, **InputView** defines just one property: **Keyboard**. This property allows a program to select the type of keyboard that is displayed. For example, a keyboard for typing a URL should be different from a keyboard for entering a phone number. All three platforms have various styles of virtual keyboards appropriate for different types of text input. A program cannot select the keyboard used for **SearchBar**.

This **Keyboard** property is of type **Keyboard**, a class that defines seven static read-only properties of type **Keyboard** appropriate for different keyboard uses:

- Default
- Text
- Chat
- Url
- Email
- Telephone
- Numeric

On all three platforms, the Numeric keyboard allows typing decimal points but does not allow typing a negative sign, so it’s limited to positive numbers.

The following program creates seven Entry views that let you see how these keyboards are implemented in the three platforms. The particular keyboard attached to each Entry is identified by a property defined by Entry named Placeholder. This is the text that appears in the Entry prior to anything the user types as a hint for the nature of the text the program is expecting. Placeholder text is commonly a short phrase such as “First Name” or “Email Address”:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="EntryKeyboards.EntryKeyboardsPage">
  <ContentPage.Padding>
    <OnPlatform x:TypeArguments="Thickness">
      <iOS>"10, 20, 10, 0"
          Android="10, 0"
          WinPhone="10, 0" />
    </OnPlatform>
  </ContentPage.Padding>
  <ScrollView>
    <StackLayout>
      <StackLayout.Resources>
        <ResourceDictionary>
          <Style TargetType="Entry">
            <Setter Property="VerticalOptions" Value="CenterAndExpand" />
          </Style>
        </ResourceDictionary>
      </StackLayout.Resources>
      <Entry Placeholder="Default"
             Keyboard="Default" />
      <Entry Placeholder="Text"
             Keyboard="Text" />
      <Entry Placeholder="Chat"
             Keyboard="Chat" />
      <Entry Placeholder="Url"
             Keyboard="Url" />
      <Entry Placeholder="Email"
```
Keyboard="Email" />

<Entry Placeholder="Telephone"
   Keyboard="Telephone" />

<Entry Placeholder="Numeric"
   Keyboard="Numeric" />
</StackLayout>
</ScrollView>
</ContentPage>

The placeholders appear as gray text. Here's how the display looks when the program first begins to run:

![Screen shot of the Entry component](image)

Just as with the Slider, you don’t want to set HorizontalOptions on an Entry to Left, Center, or Right unless you also set the WidthRequest property. If you do so, the Entry collapses to a very small width. It can still be used—the Entry automatically provides horizontal scrolling for text longer than the Entry can display—but you should really try to provide an adequate size. In this program each Entry is as wide as the screen minus a 10-unit padding on the left and right.

You can estimate an adequate WidthRequest through experimentation with different text lengths. The next program in this chapter sets the Entry width to a value equivalent to one inch.

The EntryKeyboards program evenly spaces the seven Entry views vertically using a VerticalOptions value of CenterAndExpand set through an implicit style. Clearly there is enough vertical room for all seven Entry views, so you might be puzzled about the use of the ScrollView in the XAML file.

The ScrollView is specifically for iOS. If you tap an Entry close to the bottom of the Android or
Windows 10 Mobile screen, the operating system will automatically move up the contents of the page when the keyboard pops up, so the Entry is still visible while you are typing. But iOS doesn’t do that unless a ScrollView is provided.

Here’s how each screen looks when text is being typed in one of the Entry views toward the bottom of the screen:

**Entry properties and events**

Besides inheriting the Keyboard property from InputView, Entry defines four more properties, only one of which you saw in the previous program:

- **Text** — the string that appears in the Entry
- **TextColor** — a Color value
- **IsPassword** — a Boolean that causes characters to be masked right after they’re typed
- **Placeholder** — light-colored text that appears in the Entry but disappears as soon as the user begins typing.

Generally, a program obtains what the user typed by accessing the Text property, but the program can also initialize the Text property. Perhaps the program wishes to suggest some text input.

The Entry also defines two events:

- **TextChanged**
The TextChanged event is fired for every change in the Text property, which generally corresponds to every keystroke (except shift and some special keys). A program can monitor this event to perform validity checks. For example, you might check for valid numbers or valid email addresses to enable a Calculate or Send button.

The Completed event is fired when the user presses a particular key on the keyboard to indicate that the text is completed. This key is platform specific:

- iOS: The key is labeled return, which is not on the Telephone or Numeric keyboard.
- Android: The key is a green check mark in the lower-right corner of the keyboard.
- Windows Phone: The key is an enter (or return) symbol (↵) on most keyboards but is a go symbol (→) on the Url keyboard. Such a key is not present on the Telephone and Numeric keyboards.

On iOS and Android, the completed key dismisses the keyboard in addition to generating the Completed event. On Windows 10 Mobile it does not.

Android and Windows users can also dismiss the keyboard by using the phone’s Back button at the bottom left of the portrait screen. This causes the Entry to lose input focus but does not cause the Completed event to fire.

Let’s write a program named QuadraticEquations that solves quadratic equations, which are equations of the form:

\[ ax^2 + bx + c = 0 \]

For any three constants \(a\), \(b\), and \(c\), the program uses the quadratic equation to solve for \(x\):

\[ x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

You enter \(a\), \(b\), and \(c\) in three Entry views and then press a Button labeled Solve for \(x\).

Here’s the XAML file. Unfortunately, the Numeric keyboard is not suitable for this program because on all three platforms it does not allow entering negative numbers. For that reason, no particular keyboard is specified:

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

  <ContentPage.Resources>
    <ResourceDictionary>
      <Style TargetType="Label">
        <Setter Property="FontSize" Value="Medium" />
        <Setter Property="VerticalOptions" Value="Center" />
      </Style>
    </ResourceDictionary>
  </ContentPage.Resources>
</ContentPage>
```
<Style TargetType="Entry">
  <Setter Property="WidthRequest" Value="180" />
</Style>
</ResourceDictionary>
</ContentPage.Resources>

<StackLayout>
  <!-- Entry section -->
  <StackLayout Padding="20, 0, 0, 0"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center">
    <StackLayout Orientation="Horizontal">
      <Entry x:Name="entryA"
             TextChanged="OnEntryTextChanged"
             Completed="OnEntryCompleted" />
      <Label Text="x\^2 + " />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
      <Entry x:Name="entryB"
             TextChanged="OnEntryTextChanged"
             Completed="OnEntryCompleted" />
      <Label Text="x + " />
    </StackLayout>

    <StackLayout Orientation="Horizontal">
      <Entry x:Name="entryC"
             TextChanged="OnEntryTextChanged"
             Completed="OnEntryCompleted" />
      <Label Text=" = 0" />
    </StackLayout>
  </StackLayout>

  <!-- Button -->
  <Button x:Name="solveButton"
          Text="Solve for x"
          FontSize="Large"
          IsEnabled="False"
          VerticalOptions="CenterAndExpand"
          HorizontalOptions="Center"
          Clicked="OnSolveButtonClicked" />

  <!-- Results section -->
  <StackLayout VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center">
    <Label x:Name="solution1Label"
           HorizontalTextAlignment="Center" />
  </StackLayout>
</ContentPage>
The Label, Entry, and Button views are divided into three sections: data input at the top, the Button in the middle, and the results at the bottom. Notice the platform-specific WidthRequest setting in the implicit Style for the Entry. This gives each Entry a one-inch width.

The program provides two ways to trigger a calculation: by pressing the completion key on the keyboard, or by pressing the Button in the middle of the page. Another option in a program such as this would be to perform the calculation for every keystroke (or to be more accurate, every TextChanged event). That would work here because the recalculation is very quick. However, in the present design the results are near the bottom of the screen and are covered when the virtual keyboard is active, so the page would have to be reorganized for such a scheme to make sense.

The QuadraticEquations program uses the TextChanged event but solely to determine the validity of the text typed into each Entry. The text is passed to Double.TryParse, and if the method returns false, the Entry text is displayed in red. (On Windows 10 Mobile, the red text coloring shows up only when the Entry loses input focus.) Also, the Button is enabled only if all three Entry views contain valid double values. Here’s the first half of the code-behind file that shows all the program interaction:

```csharp
public partial class QuadraticEquationsPage : ContentPage
{
    public QuadraticEquationsPage()
    {
        InitializeComponent();

        // Initialize Entry views.
        entryA.Text = "1";
        entryB.Text = "-1";
        entryC.Text = "-1";
    }

    void OnEntryTextChanged(object sender, TextChangedEventArgs args)
    {
        // Clear out solutions.
        solution1Label.Text = " ";
        solution2Label.Text = " ";

        // Color current entry text based on validity.
        Entry entry = (Entry)sender;
        double result;

        // Enable the button based on validity.
        solveButton.IsEnabled = Double.TryParse(entryA.Text, out result) &&
                                Double.TryParse(entryB.Text, out result) &&
                                Double.TryParse(entryC.Text, out result);
    }

    void OnEntryCompleted(object sender, EventArgs args)
    {
        if (solveButton.IsEnabled)
        {
            Solve();
        }
    }
```
The Completed handler for the Entry calls the Solve method only when the Button is enabled, which (as you’ve seen) indicates that all three Entry views contain valid values. Therefore, the Solve method can safely assume that all three Entry views contain valid numbers that won’t cause Double.Parse to raise an exception.

The Solve method is necessarily complicated because the quadratic equation might have one or two solutions, and each solution might have an imaginary part as well as a real part. The method initializes the real part of the second solution to Double.NaN ("not a number") and displays the second result only if that’s no longer the case. The imaginary parts are displayed only if they’re nonzero, and either a plus sign or an en dash (Unicode \u2013) connects the real and imaginary parts:

```csharp
public partial class QuadraticEquationsPage : ContentPage
{
    ...png
    void Solve()
    {
        double a = Double.Parse(entryA.Text);
        double b = Double.Parse(entryB.Text);
        double c = Double.Parse(entryC.Text);
        double solution1Real = 0;
        double solution1Imag = 0;
        double solution2Real = Double.NaN;
        double solution2Imag = 0;
        string str1 = " ";
        string str2 = " ";

        if (a == 0 && b == 0 && c == 0)
        {
            str1 = "x = anything";
        }
        else if (a == 0 && b == 0)
        {
            str1 = "x = nothing";
        }
        else
        {
            if (a == 0)
            {
                solution1Real = -c / b;
            }
            else
            {
```
double discriminant = b * b - 4 * a * c;

if (discriminant == 0)
{
    solution1Real = -b / (2 * a);
}
else if (discriminant > 0)
{
    solution1Real = (-b + Math.Sqrt(discriminant)) / (2 * a);
    solution2Real = (-b - Math.Sqrt(discriminant)) / (2 * a);
}
else
{
    solution1Real = -b / (2 * a);
    solution2Real = solution1Real;

    solution1Imag = Math.Sqrt(-discriminant) / (2 * a);
    solution2Imag = -solution1Imag;
}

str1 = Format(solution1Real, solution1Imag);
str2 = Format(solution2Real, solution2Imag);

solution1Label.Text = str1;
solution2Label.Text = str2;

string Format(double real, double imag)
{
    string str = " ";

    if (!Double.IsNaN(real))
    {
        str = String.Format("x = {0:F5}\", real);

        if (imag != 0)
        {
            str += String.Format(" {0} {1:F5} i",
                                Math.Sign(imag) == 1 ? "+" : "\u2013",
                                Math.Abs(imag));
        }
    }

    return str;
}

Here are a couple of solutions:
The Editor difference

You might assume that the Editor has a more extensive API than the Entry because it can handle multiple lines and even paragraphs of text. But in Xamarin.Forms, the API for Editor is actually somewhat simpler. Besides inheriting the Keyboard property from InputView, Editor defines just one property on its own: the essential Text property. Editor also defines the same two events as Entry:

- TextChanged
- Completed

However, the Completed event is of necessity a little different. While a return or enter key can signal completion on an Entry, these same keys used with the Editor instead mark the end of a paragraph.

The Completed event for Editor works a little differently on the three platforms: For iOS, Xamarin.Forms displays a special Done button above the keyboard that dismisses the keyboard and causes a Completed event to fire. On Android and Windows 10 Mobile, the system Back button—the button at the lower-left corner of the phone in portrait mode—dismisses the keyboard and fires the Completed event. This Back button does not fire the Completed event for an Entry view, but it does dismiss the keyboard.

It is likely that what users type into an Editor is not telephone numbers and URLs but actual words, sentences, and paragraphs. In most cases, you’ll want to use the Text keyboard for Editor because it provides spelling checks, suggestions, and automatic capitalization of the first word of sentences. If you don’t want these features, the Keyboard class provides an alternative means of specifying a keyboard by using a static Create method and the following members of the KeyboardFlags enumeration:
Chapter 15  The interactive interface

- CapitalizeSentence (equal to 1)
- Spellcheck (2)
- Suggestions (4)
- All (0xFFFFFFFF)

The Text keyboard is equivalent to creating the keyboard with KeyboardFlags.All. The Default keyboard is equivalent to creating the keyboard with (KeyboardFlags)0. You can’t create a keyboard in XAML using these flags. It must be done in code.

The JustNotes program is intended as a freeform note-taking program that automatically saves and restores the contents of an Editor view by using the Properties collection of the Application class. The page basically consists of a large Editor, but to give the user some clue about what the program does, the name of the program is displayed at the top. On iOS and Android, such text can be set by the Title property of the page, but to display that property, the ContentPage must be wrapped in an ApplicationPage (as you discovered with the ToolbarDemo program in Chapter 13). That’s done in the constructor of the App class:

```csharp
public class App : Application
{
    public App()
    {
        MainPage = new NavigationPage(new JustNotesPage());
    }

    protected override void OnStart()
    {
        // Handle when your app starts
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
    }
}
```

The OnSleep method in App calls a method also named OnSleep defined in the JustNotesPage code-behind file. This is how the contents of the Editor are saved in application memory.

The root element of the XAML page sets the Title property. The remainder of the page is occupied by an AbsoluteLayout filled with the Editor:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms""
So why does the program use an **AbsoluteLayout** to host the **Editor**?

The **JustNotes** program is a work in progress. It doesn’t quite work right for iOS. As you’ll recall, when an **Entry** view is positioned toward the bottom of the screen, you want to put it in a **ScrollView** so that it scrolls up when the iOS virtual keyboard is displayed. However, because **Editor** implements its own scrolling, you can’t put it in a **ScrollView**.

For that reason, the code-behind file sets the height of the **Editor** to one-half the height of the **AbsoluteLayout** when the **Editor** gets input focus so that the keyboard doesn’t overlap it, and it restores the **Editor** height when it loses input focus:

```csharp
public partial class JustNotesPage : ContentPage
{
    public JustNotesPage()
    {
        InitializeComponent();

        // Retrieve last saved Editor text.
        IDictionary<string, object> properties = Application.Current.Properties;

        if (properties.ContainsKey("text"))
        {
            editor.Text = (string)properties["text"];
        }
    }

    void OnEditorFocused(object sender, FocusEventArgs args)
    {
        if (Device.OS == TargetPlatform.iOS)
        {
            AbsoluteLayout.SetLayoutBounds(editor, new Rectangle(0, 0, 1, 0.5));
        }
    }

    void OnEditorUnfocused(object sender, FocusEventArgs args)
    {
        if (Device.OS == TargetPlatform.iOS)
        {
            AbsoluteLayout.SetLayoutBounds(editor, new Rectangle(0, 0, 1, 1));
        }
    }
}
```
{  
    AbsoluteLayout.SetLayoutBounds(editor, new Rectangle(0, 0, 1, 1));
}

public void OnSleep()
{
    // Save Editor text.
}

That adjustment is only approximate, of course. It varies by device, and it varies by portrait and landscape mode, but sufficient information is not currently available in Xamarin.Forms to do it more accurately. For now, you should probably restrict your use of Editor views to the top area of the page.

The code for saving and restoring the Editor contents is rather prosaic in comparison with the Editor manipulation. The OnSleep method (called from the App class) saves the text in the Properties dictionary with a key of “text” and the constructor restores it.

Here’s the program running on all three platforms with the Text keyboard in view with word suggestions. On the Windows 10 Mobile screen, a word has been selected and might be copied to the clipboard for a later paste operation:

The SearchBar

The SearchBar doesn’t derive from InputView like Entry and Editor, and it doesn’t have a Key-
board property. The keyboard that SearchBar displays when it acquires input focus is platform specific and appropriate for a search command. The SearchBar itself is similar to an Entry view, but depending on the platform, it might be adorned with some other graphics and contain a button that erases the typed text.

SearchBar defines two events:

- TextChanged
- SearchButtonPressed

The TextChanged event allows your program to access a text entry in progress. Perhaps your program can actually begin a search or offer context-specific suggestions before the user completes typing. The SearchButtonPressed event is equivalent to the Completed event fired by Entry. It is triggered by a particular button on the keyboard in the same location as the completed button for Entry but possibly labeled differently.

SearchBar defines five properties:

- Text — the text entered by the user
- Placeholder — hint text displayed before the user begins typing
- CancelButtonColor — of type Color
- SearchCommand — for use with data binding
- SearchCommandParameter — for use with data binding

The SearchBarDemo program uses only Text and Placeholder, but the XAML file attaches handlers for both events:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SearchBarDemo.SearchBarDemoPage">
  <ContentPage.Padding>
    <OnPlatform x:TypeArguments="Thickness">
      <iOS>10, 20, 10, 0</iOS>
      <Android>10, 0</Android>
      <WinPhone>10, 0</WinPhone>
    </OnPlatform>
  </ContentPage.Padding>

  <StackLayout>
    <SearchBar x:Name="searchBar"
               Placeholder="Search text"
               TextChanged="OnSearchBarTextChanged"
               SearchButtonPressed="OnSearchBarButtonPressed" />

    <ScrollView x:Name="resultsScroll"
                 VerticalOptions="FillAndExpand">
      <StackLayout x:Name="resultsStack" />
    </ScrollView>
  </StackLayout>
</ContentPage>
```
The program uses the scrollable StackLayout named resultsStack to display the results of the search.

Here’s the SearchBar and keyboard for the three platforms. Notice the search icon and a delete button on all three platforms, and the special search keys on the iOS and Android keyboards:

You might guess from the entries in the three SearchBar views that the program allows searching through the text of Herman Melville’s *Moby-Dick*. That is true! The entire novel is stored as an embedded resource in the Texts folder of the Portable Class Library project with the name MobyDick.txt. The file is a plain-text, one-line-per-paragraph format that originated with a file on the Gutenberg.net website.

The constructor of the code-behind file reads that whole file into a string field named bookText. The TextChanged handler clears the resultsStack of any previous search results so that there’s no discrepancy between the text being typed into the SearchBar and this list. The SearchButton-Pressed event initiates the search:

```csharp
public partial class SearchBarDemoPage : ContentPage
{
    const double MaxMatches = 100;
    string bookText;

    public SearchBarDemoPage()
    {
        InitializeComponent();
```
// Load embedded resource bitmap.
string resourceID = "SearchBarDemo.Texts.MobyDick.txt";
Assembly assembly = GetType().GetTypeInfo().Assembly;

using (Stream stream = assembly.GetManifestResourceStream(resourceID))
{
    using (StreamReader reader = new StreamReader(stream))
    {
        bookText = reader.ReadToEnd();
    }
}

void OnSearchBarTextChanged(object sender, TextChangedEventArgs args)
{
    resultsStack.Children.Clear();
}

void OnSearchBarButtonPressed(object sender, EventArgs args)
{
    // Detach resultsStack from layout.
    resultsScroll.Content = null;
    resultsStack.Children.Clear();
    SearchBookForText(searchBar.Text);
    // Reattach resultsStack to layout.
    resultsScroll.Content = resultsStack;
}

void SearchBookForText(string searchText)
{
    int count = 0;
    bool isTruncated = false;

    using (StringReader reader = new StringReader(bookText))
    {
        int lineNumber = 0;
        string line;

        while (null != (line = reader.ReadLine()))
        {
            lineNumber++;
            int index = 0;

            while (-1 != (index = (line.IndexOf(searchText, index,
                StringComparison.OrdinalIgnoreCase))))
            {
                if (count == MaxMatches)
                {
                    isTruncated = true;
                    break;
                }
                index += 1;
            }
The SearchBookForText method uses the search text with the IndexOf method applied to each line of the book for case-insensitive comparison and adds a Label to resultsStack for each match. However, this process has performance problems because each Label that is added to the StackLayout potentially triggers a new layout calculation. That’s unnecessary. For this reason, before beginning the search, the program detaches the StackLayout from the visual tree by setting the Content property of its parent (the ScrollView) to null:

```csharp
resultsScroll.Content = null;
```

After all the Label views have been added to the StackLayout, the StackLayout is added back to the visual tree:

```csharp
resultsScroll.Content = resultsStack;
```

But even that’s not a sufficient performance improvement for some searches, and that is why the program limits itself to the first 100 matches. (Notice the MaxMatches constant defined at the top of the class.) Here’s the program showing the results of the searches you saw entered earlier:
You’ll need to reference the actual file to see what those matches are.

Would running the search in a second thread of execution speed things up? No. The actual text search is very fast. The performance issues involve the user interface. If the `SearchBookForText` method were run in a secondary thread, then it would need to use `Device.BeginInvokeOnMainThread` to add each `Label` to the `StackLayout`. If that `StackLayout` is attached to the visual tree, this would make the program operate more dynamically—the individual items would appear on the screen following each item added to the list—but the switching back and forth between threads would slow down the overall operation.

### Date and time selection

A Xamarin.Forms application that needs a date or time from the user can use the `DatePicker` or `TimePicker` view.

These are very similar: The two views simply display a date or time in a box similar to an `Entry` view. Tapping the view invokes the platform-specific date or time selector. The user then selects (or dials in) a new date or time and signals completion.

#### The `DatePicker`

`DatePicker` has three properties of type `DateTime`:

- `MinimumDate`, initialized to January 1, 1900
MaximumDate, initialized to December 31, 2100

Date, initialized to DateTime.Today

A program can set these properties to whatever it wants as long as MinimumDate is prior to MaximumDate. The Date property reflects the user’s selection.

If you’d like to set those properties in XAML, you can do so using the x:DateTime element. Use a format that is acceptable to the DateTime.Parse method with a second argument of CultureInfo.InvariantCulture. Probably the easiest is the short-date format, which is a two-digit month, a two-digit day, and a four-digit year, separated by slashes:

```xml
<DatePicker … >
    <DatePicker.MinimumDate>03/01/2016</DatePicker.MinimumDate>
    <DatePicker.MaximumDate>10/31/2016</DatePicker.MaximumDate>
    <DatePicker.Date>04/24/2016</DatePicker.Date>
</DatePicker>
```

The DatePicker displays the selected date by using the normal ToString method, but you can set the Format property of the view to a custom .NET formatting string. The initial value is “d”—the short-date format.

Here’s the XAML file from a program called DaysBetweenDates that lets you select two dates and then calculates the number of days between them. It contains two DatePicker views labeled To and From:

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

    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style TargetType="DatePicker">
                    <Setter Property="Format" Value="D" />
                    <Setter Property="VerticalOptions" Value="Center" />
                    <Setter Property="HorizontalOptions" Value="FillAndExpand" />
                </Style>
            </ResourceDictionary>
        </StackLayout.Resources>
    </StackLayout>
</ContentPage>
```
An implicit style sets the Format property of the two DatePicker views to “D”, which is the long-date format, to include the text day of the week and month name. The XAML file uses two horizontal StackLayout objects for displaying a Label and DatePicker side by side.

Watch out: If you use the long-date format, you’ll want to avoid setting the HorizontalOptions property of the DatePicker to Start, Center, or End. If you put the DatePicker in a horizontal StackLayout (as in this program), set the HorizontalOptions to FillAndExpand. Otherwise, if the user selects a date with a longer text string than the original date, the result is not formatted well. The DaysBetweenDates program uses an implicit style to give the DatePicker a HorizontalOptions value of FillAndExpand so that it occupies the entire width of the horizontal StackLayout except for what’s occupied by the Label.
When you tap one of the DatePicker fields, a platform-specific panel comes up. On iOS, it occupies just the bottom part of the screen, but on Android and Windows 10 Mobile, it pretty much takes over the screen:

Notice the Done button on iOS, the OK button on Android, and the check-mark toolbar button on Windows Phone. All three of these buttons dismiss the date-picking panel and return to the program with a firing of the DateSelected event. The event handler in the DaysBetweenDates code-behind file accesses both DatePicker views and calculates the number of days between the two dates:

```csharp
public partial class DaysBetweenDatesPage : ContentPage {
    public DaysBetweenDatesPage()
    {
        InitializeComponent();

        // Initialize.
        OnDateSelected(null, null);
    }

    void OnDateSelected(object sender, DateChangedEventArgs args)
    {
        int days = (toDatePicker.Date - fromDatePicker.Date).Days;
        resultLabel.Text = String.Format("{0} day{1} between dates",
                                         days, days == 1 ? "" : "s");
    }
}
```

Here's the result:
The TimePicker (or is it a TimeSpanPicker?)

The TimePicker is somewhat simpler than DatePicker. It defines only Time and Format properties, and it doesn’t include an event to indicate a new selected Time value. If you need to be notified, you can install a handler for the PropertyChanged event.

Although TimePicker displays the selected time by using the ToString method of DateTime, the Time property is actually of type TimeSpan, indicating a duration of time since midnight.

The SetTimer program includes a TimePicker. The program assumes that the time picked from the TimePicker is within the next 24 hours and then notifies you when that time has come. The XAML file puts a TimePicker, a Switch, and an Entry on the page.

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SetTimer.SetTimerPage"
             Padding="50">

  <StackLayout Spacing="20"
               VerticalOptions="Center">
    <TimePicker x:Name="timePicker"
                PropertyChanged="OnTimePickerPropertyChanged" />
    <Switch x:Name="switch"
            HorizontalOptions="End"
            Toggled="OnSwitchToggled" />
    <Entry x:Name="entry"
           Text="Sample Timer" />
  </StackLayout>

```

Although TimePicker displays the selected time by using the ToString method of DateTime, the Time property is actually of type TimeSpan, indicating a duration of time since midnight.

The SetTimer program includes a TimePicker. The program assumes that the time picked from the TimePicker is within the next 24 hours and then notifies you when that time has come. The XAML file puts a TimePicker, a Switch, and an Entry on the page.
The **TimePicker** has a `PropertyChanged` event handler attached. The `Entry` lets you remind yourself what the timer is supposed to remind you of.

When you tap the **TimePicker**, a platform-specific panel pops up. As with the **DatePicker**, the Android and Windows 10 Mobile panels obscure much of the screen underneath, but you can see the **SetTimer** user interface in the center of the iPhone screen:

![TimePicker Screenshots](image)

In a real timer program—a timer program that is actually useful and not just a demonstration of the **TimePicker** view—the code-behind file would access the platform-specific notification interfaces so that the user would be notified even if the program were no longer active.

**SetTimer** doesn’t do that. **SetTimer** instead uses a platform-specific alert box that a program can invoke by calling the `DisplayAlert` method that is defined by `Page` and inherited by `ContentPage`.

The **SetTriggerTime** method at the bottom of the code-behind file (shown below) calculates the timer time based on `DateTime.Today`—a property that returns a `DateTime` indicating the current date, but with a time of midnight—and the `TimeSpan` returned from the **TimePicker**. If that time has already passed today, then it’s assumed to be tomorrow.

The timer, however, is set for one second. Every second the timer handler checks whether the **Switch** is on and whether the current time is greater than or equal to the timer time:

```csharp
public partial class SetTimerPage : ContentPage
{
    DateTime triggerTime;
```
public SetTimerPage()
{
    InitializeComponent();

    Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerTick);
}

bool OnTimerTick()
{
    if (@switch.IsToggled && DateTime.Now >= triggerTime)
    {
        @switch.IsToggled = false;
        DisplayAlert("Timer Alert",
                      "The '" + entry.Text + '" timer has elapsed",
                      "OK");
    }
    return true;
}

void OnTimePickerPropertyChanged(object obj, PropertyChangedEventArgs args)
{
    if (args.PropertyName == "Time")
    {
        SetTriggerTime();
    }
}

void OnSwitchToggled(object obj, ToggledEventArgs args)
{
    SetTriggerTime();
}

void SetTriggerTime()
{
    if (@switch.IsToggled)
    {
        triggerTime = DateTime.Today + timePicker.Time;

        if (triggerTime < DateTime.Now)
        {
            triggerTime += TimeSpan.FromDays(1);
        }
    }
}

When the timer time has come, the program uses DisplayAlert to signal a reminder to the user. Here’s how this alert appears on the three platforms:
Throughout this chapter, you’ve seen interactive views that define events, and you’ve seen application programs that implement event handlers. Often these event handlers access a property of the view and set a property of another view.

In the next chapter, you’ll see how these event handlers can be eliminated and how properties of different views can be linked, either in code or markup. This is the exciting feature of data binding.