Chapter 24

Page navigation

Different types of computing environments tend to develop different metaphors for presenting information to the user. Sometimes a metaphor developed within one environment is so good that it influences other environments.

Such is the case with the page and navigation metaphor that evolved on the World Wide Web. Prior to that, desktop computer applications simply were not organized around the concept of navigable pages. But the web demonstrated the power and convenience of the page metaphor, and now mobile and desktop operating systems generally support a page-based architecture, and many applications have taken advantage of that.

A page architecture is particularly popular in mobile applications, and for that reason such an architecture is supported by Xamarin.Forms. A Xamarin.Forms application can contain multiple classes that derive from `ContentPage`, and the user can navigate between these pages. (In the next chapter, you'll see several alternatives to `ContentPage`. Those other page types can also participate in navigation.)

Generally, a page will include a `Button` (or perhaps a `Label` or an `Image` with a `TapGestureRecognizer`) that the user taps to navigate to another page. Sometimes, that second page will allow further navigation to other pages.

But there also must be a way for the user to return to the previous page, and here's where platform differences begin manifesting themselves: Android and Windows Phone devices incorporate a standard **Back** button (symbolized as a left-pointing arrow or triangle) at the bottom of the screen; iOS devices do not, and neither does Windows running on the desktop or a tablet.

Also, as you'll see, standard software **Back** buttons are provided at the top of some (but not all) navigable pages as part of the standard user interface by iOS and Android, and also by the Windows Runtime when running on desktop computers or tablets.

From the programmer's perspective, page navigation is implemented with the familiar concept of a stack. When one page navigates to another, the new page is pushed on the stack and becomes the active page. When the second page returns back to the first page, a page is popped from the stack, and the new topmost page then becomes active. The application has access to the navigation stack that Xamarin.Forms maintains for the application and supports methods to manipulate the stack by inserting pages or removing them.

An application that is structured around multiple pages always has one page that is special because it's the starting point of the application. This is often called the *main* page, or the *home* page, or the *start* page.

All the other pages in the application are intrinsically different from that start page, however, because they fall into two different categories: modal pages and modeless pages.

Modal pages and modeless pages

In user interface design, "modal" refers to something that requires user interaction before the application can continue. Computer applications on the desktop sometimes display modal windows or modal dialogs. When one of these modal objects is displayed, the user can't simply use the mouse to switch to the application's main window. The modal object demands more attention from the user before it goes away.

A window or dialog that is not modal is often called *modeless* when it's necessary to distinguish between the two types.

The Xamarin.Forms page-navigation system likewise implements modal and modeless pages by defining two different methods that a page can call to navigate to another page:

```
Task PushAsync(Page page)

Task PushModalAsync(Page page)
```

The page to navigate to is passed as the argument. As the name of the second method implies, it navigates to a modal page. The simple `PushAsync` method navigates to a modeless page, which in real-life programming is the more common page type.

Two other methods are defined to go back to the previous page:

```
Task<Page> PopAsync()

Task<Page> PopModalAsync()
```

In many cases an application does not need to call `PopAsync` directly if it relies on the back navigation provided by the phone or operating system.

The `Task` return value and the `Async` suffix on these `Push` and `Pop` method names indicate that they are asynchronous, but this does not mean that a navigated page runs in a different thread of execution! What the completion of the task indicates is discussed later in this chapter.

These four methods—as well as other navigation methods and properties—are defined in the `INavigation` interface. The object that implements this interface is internal to Xamarin.Forms, but `VisualElement` defines a read-only property named `Navigation` of type `INavigation`, and this gives you access to the navigation methods and properties.

This means that you can use these navigation methods from an instance of any class that derives from `VisualElement`. Generally, however, you'll use the `Navigation` property of the page object, so the code to navigate to a new page often looks like this:

```
await Navigation.PushAsync(new MyNewPage());
```

or this:

```
await Navigation.PushModalAsync(new MyNewModalPage());
```

The difference between modal and modeless pages mostly involves the user interface that the operating system provides on the page to return back to the previous page. This difference varies by platform. A greater difference in the user interface between modeless and modal pages exists on iOS and the Windows desktop or tablets; somewhat less difference is found on Android and the Windows phone platforms.

Generally, you'll use modal pages when your application needs some information from the user and you don't want the user to return to the previous page until that information is provided. To work across all platforms, a modal page must provide its own user-interface for navigating back to the previous page.

Let's begin by exploring the difference between modeless and modal pages in more detail. The **ModelessAndModal** program contains three pages with the class names `MainPage`, `ModalPage`, and `ModelessPage`. The pages themselves are rather simple, so to keep the file bulk to a minimum, these are code-only pages. In a real application, pages can be implemented with XAML or—at the other extreme—generated dynamically by code. (You'll see examples of both options later in this chapter.)

`MainPage` creates two `Button` elements, one that navigates to a modeless page and the other that navigates to a modal page. Notice the `Title` property set at the top of the constructor. This `Title` property has no effect in a single-page application but plays an important role in multipage applications:

```
public class MainPage : ContentPage
{
    public MainPage()
    {
        Title = "Main Page";

        Button gotoModelessButton = new Button
        {
            Text = "Go to Modeless Page",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };
        gotoModelessButton.Clicked += async (sender, args) =>
        {
            await Navigation.PushAsync(new ModelessPage());
        };

        Button gotoModalButton = new Button
        {
            Text = "Go to Modal Page",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };
```

```
        gotoModalButton.Clicked += async (sender, args) =>
        {
            await Navigation.PushModalAsync(new ModalPage());
        };

        Content = new StackLayout
        {
            Children =
            {
                gotoModelessButton,
                gotoModalButton
            }
        };
    }
}
```

The `Clicked` handler for the first `Button` calls `PushAsync` with a new instance of `ModelessPage`, and the second calls `PushModalAsync` with a new instance of `ModalPage`. The `Clicked` handlers are flagged with the `async` keyword and call the `Push` methods with `await`.

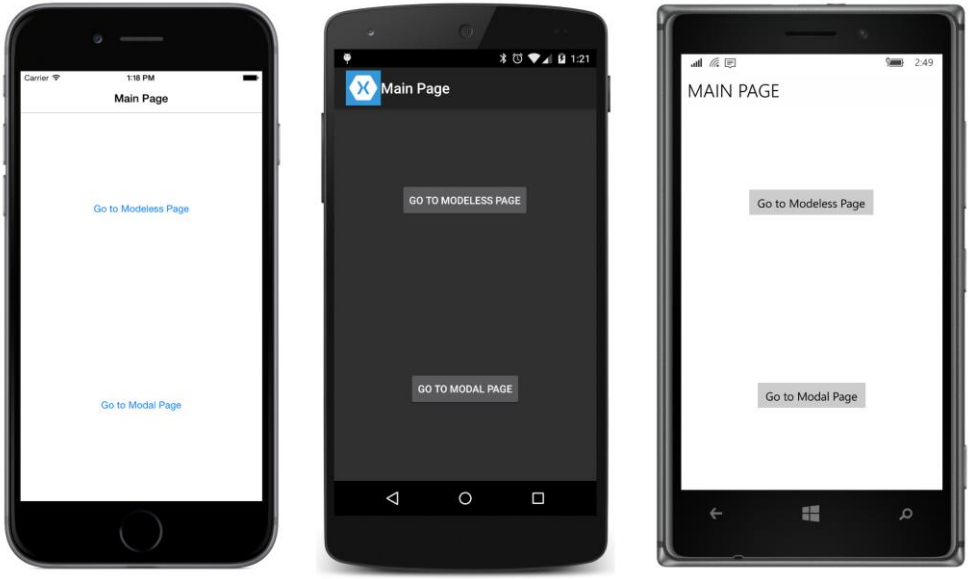
A program that makes calls to `PushAsync` or `PushModalAsync` must have slightly different startup code in the constructor of the `App` class. Rather than setting the `MainPage` property of `App` to an instance of the application's sole page, an instance of the application's startup page is generally passed to the `NavigationPage` constructor, and this is set to the `MainPage` property.

Here's how the constructor of your `App` class usually looks when the application incorporates page navigation:

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

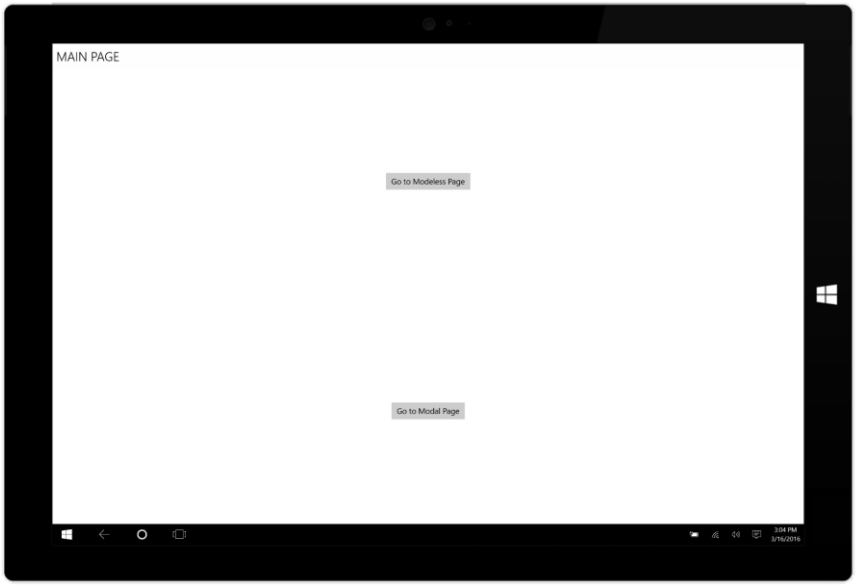
Most of the `App` classes in all the programs in this chapter contain similar code. As an alternative you can instantiate `NavigationPage` by using its parameterless constructor and then call the `PushAsync` method of the `NavigationPage` to go to the home page.

The use of `NavigationPage` results in a visible difference in the page. The `Title` property is displayed at the top of `MainPage`, and it is accompanied by the application icon on the Android screen:

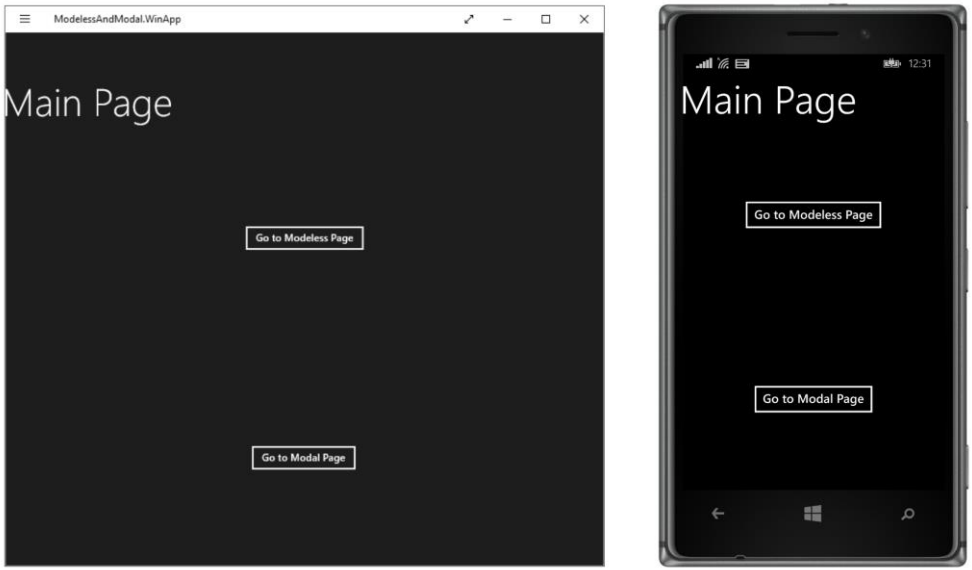


Another big difference is that you no longer need to set `Padding` on the iOS page to avoid overwriting the status bar at the top of the screen.

The title is also displayed at the top of the Windows 10 program running in tablet mode:



A rather larger title is displayed on the Windows 8.1 and Windows Phone 8.1 platforms:



Clicking the **Go to Modeless Page** button causes the following code to execute:

```
await Navigation.PushAsync(new ModelessPage());
```

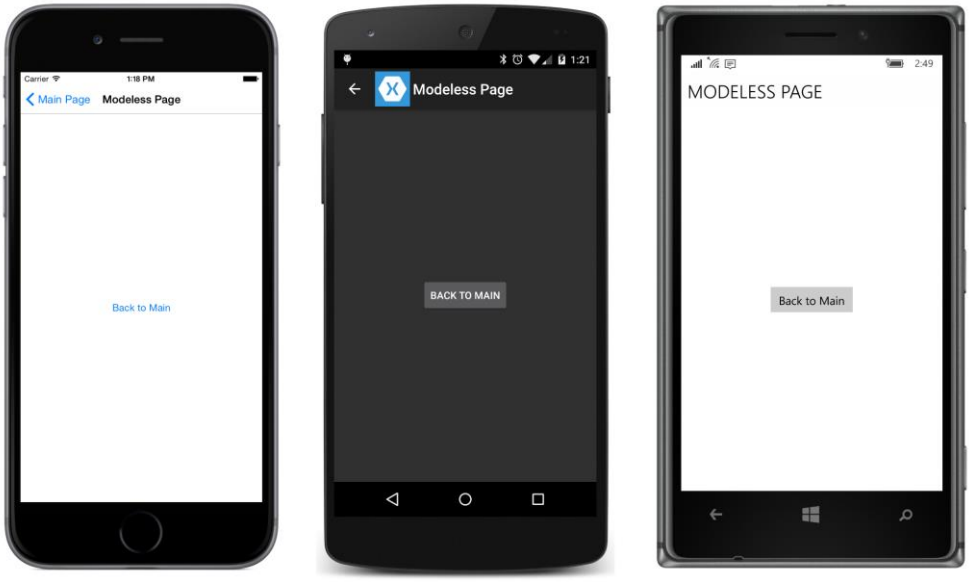
This code instantiates a new `ModelessPage` and navigates to that page.

The `ModelessPage` class defines a `Title` property with the text "Modeless Page" and a `Button` element labeled **Back to Main** with a `Clicked` handler that calls `PopAsync`:

```
public class ModelessPage : ContentPage
{
    public ModelessPage()
    {
        Title = "Modeless Page";

        Button goBackButton = new Button
        {
            Text = "Back to Main",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
        goBackButton.Clicked += async (sender, args) =>
        {
            await Navigation.PopAsync();
        };

        Content = goBackButton;
    }
}
```

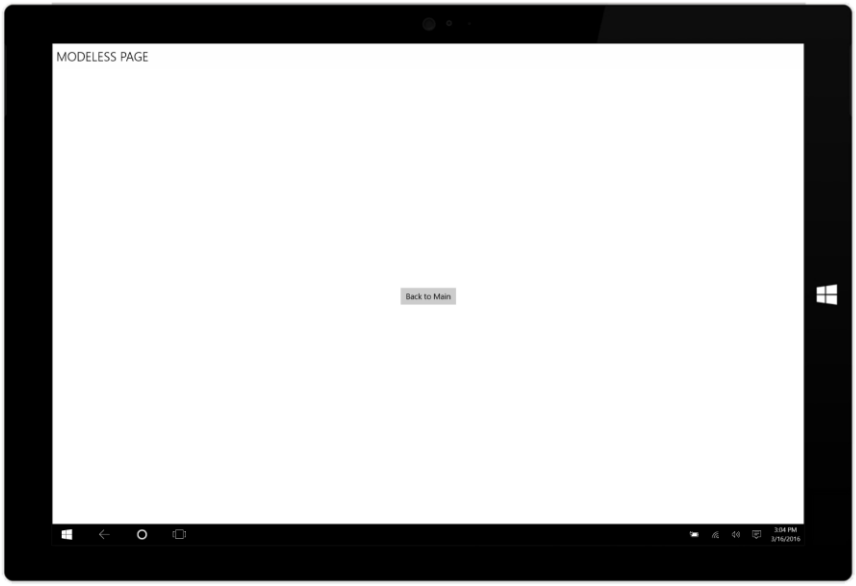
You don't actually need the **Back to Main** button on the iOS and Android pages because a left-

pointing arrow at the top of the page performs that same function. The Windows Phone doesn't need that `Button` either because it has a **Back** button at the bottom of the screen, as does the Android device:

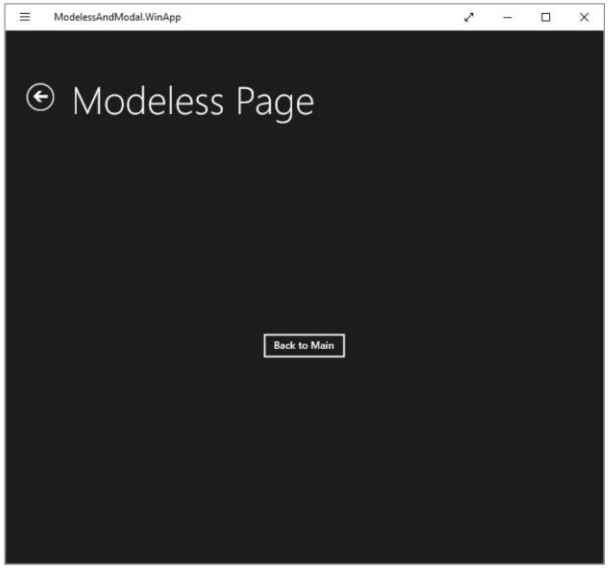


That top area on the iOS and Android pages is called the *navigation bar*. In that navigation bar, both the iOS and Android pages show the `Title` property of the current page, and the iOS page also displays the `Title` property of the previous page in another color.

A program running in tablet mode under Windows 10 contains a **Back** button in the lower-left corner, directly to the right of the Windows logo:



In contrast, the Windows 8.1 program displays a button in the form of a circled arrow to navigate back to the previous page. The Windows Phone 8.1 screen doesn't need that button because it has a **Back** button on the bottom of the screen:



In summary, you don't need to include your own **Back to Main** button (or its equivalent) on a modeless page. Either the navigation interface or the device itself provides a **Back** button.

Let's go back to `MainPage`. When you click the **Go to Modal Page** button on the main page, the

`Clicked` handler executes the following code:

```
await Navigation.PushModalAsync(new ModalPage(), true);
```

The `ModalPage` class is nearly identical to the `ModelessPage` except for the different `Title` setting and the call to `PopModalAsync` in the `Clicked` handler:

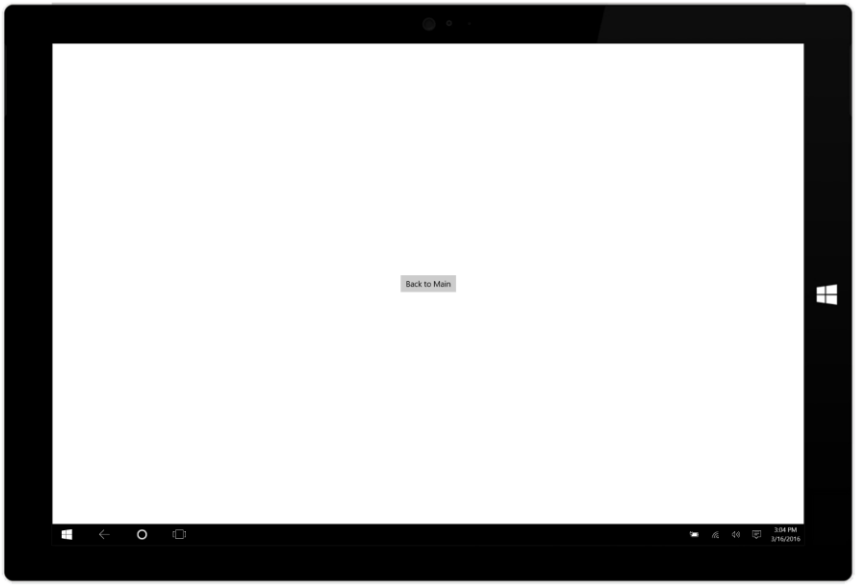
```
public class ModalPage : ContentPage
{
    public ModalPage()
    {
        Title = "Modal Page";

        Button goBackButton = new Button
        {
            Text = "Back to Main",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
        goBackButton.Clicked += async (sender, args) =>
        {
            await Navigation.PopModalAsync();
        };

        Content = goBackButton;
    }
}
```

Despite the `Title` property setting in the class, none of the three platforms displays the `Title` or any other page-navigation interface:

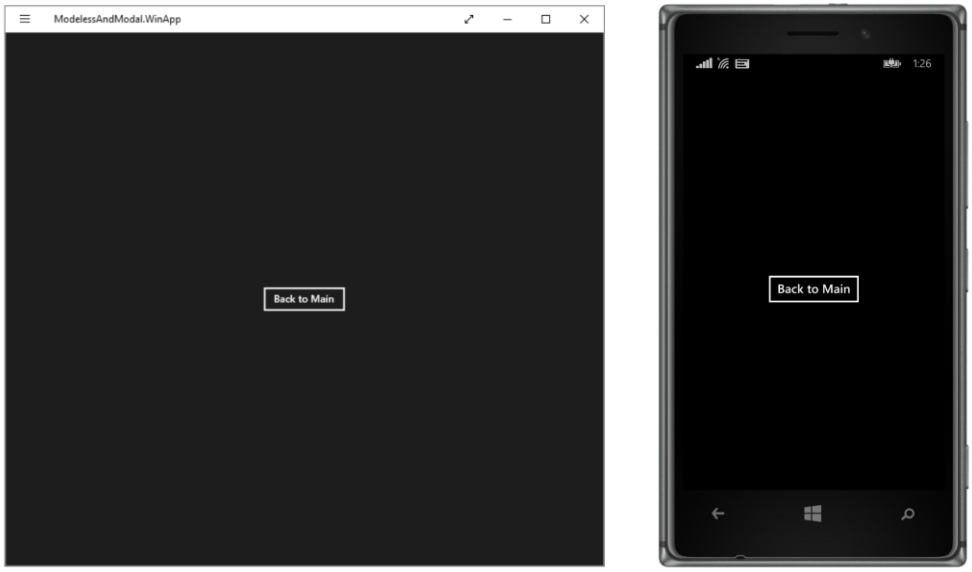
These screens now look like typical single-page applications. Although it's not quite obvious, you'll once again need to be careful to avoid overwriting the status bar on the iOS page.

You don't need the **Back to Main** button on the Android and Windows Phone pages because you can use the **Back** button on the bottom of the phone, but you definitely need it on the iOS page: That **Back to Main** button on the iPhone is the only path back to `MainPage`.

The UWP application running under Windows 10 in tablet mode doesn't display a title, but the **Back** button in the lower-left corner still works to navigate back to `MainPage`.



However, the Windows 8.1 page needs the **Back to Main** button, while the Windows Phone 8.1 page does not because it has a **Back** button on the bottom of the phone:



Nothing internal to the page definition distinguishes a modeless page and a modal page. It depends on how the page is invoked—whether through `PushAsync` or `PushModalAsync`. However, a particular page must know how it was invoked so that it can call either `PopAsync` or `PopModalAsync` to navigate back.

Throughout the time this program is running, there is only one instance of `MainPage`. It continues to remain in existence when `ModelessPage` and `ModalPage` are active. This is always the case in a multipage application. A page that calls `PushAsync` or `PushModalAsync` does not cease to exist when the next page is active.

However, in this program, each time you navigate to `ModelessPage` or `ModalPage`, a new instance of that page is created. When that page returns back to `MainPage`, there are no further references to that instance of `ModelessPage` or `ModalPage`, and that object becomes eligible for garbage collection. This is *not* the only way to manage navigable pages, and you'll see alternatives later in this chapter, but in general it is best to instantiate a page right before navigating to it.

A page always occupies the full screen. Sometimes it's desirable for a modal page to occupy only part of the screen, and for the previous page to be visible (but disabled) underneath that popup. You can't do this with Xamarin.Forms pages. If you want something like that, look at the **SimpleOverlay** program in Chapter 14, "Absolute layout."

Animated page transitions

All four of the methods you've seen are also available with an overload that has an additional argument of type `bool`:

```
Task PushAsync(Page page, bool animated)
```

```
Task PushModalAsync(Page page, bool animated)

Task<Page> PopAsync(bool animated)

Task<Page> PopModalAsync(bool animated)
```

Setting this argument to `true` enables a page-transition animation if such an animation is supported by the underlying platform. However, the simpler `Push` and `Pop` methods enable this animation by default, so you'll only need these four overloads if you want to *suppress* the animation, in which case you set the Boolean argument to `false`.

Toward the end of this chapter, you'll see some code that saves and restores the entire page navigation stack when a multipage application is terminated. To restore the navigation stack, these pages must be created and navigated to during program startup. In this case, the animations should be suppressed, and these overloads are handy for that.

You'll also want to suppress the animation if you provide one of your own page-entrance animations, such as demonstrated in Chapter 22, "Animation."

In general, however, you'll want to use the simpler forms of the `Push` and `Pop` methods.

Visual and functional variations

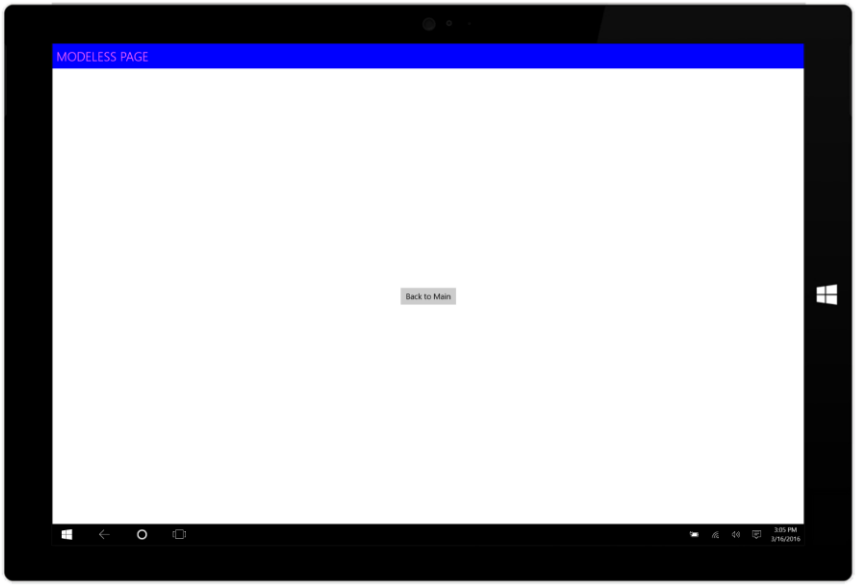
`NavigationPage` defines several properties—and several attached bindable properties—that have the power to change the appearance of the navigation bar and even to eliminate it altogether.

You can set the `BarBackgroundColor` and `BarTextColor` properties when you instantiate the `NavigationPage` in the `App` class. Try this in the **ModalAndModeless** program:

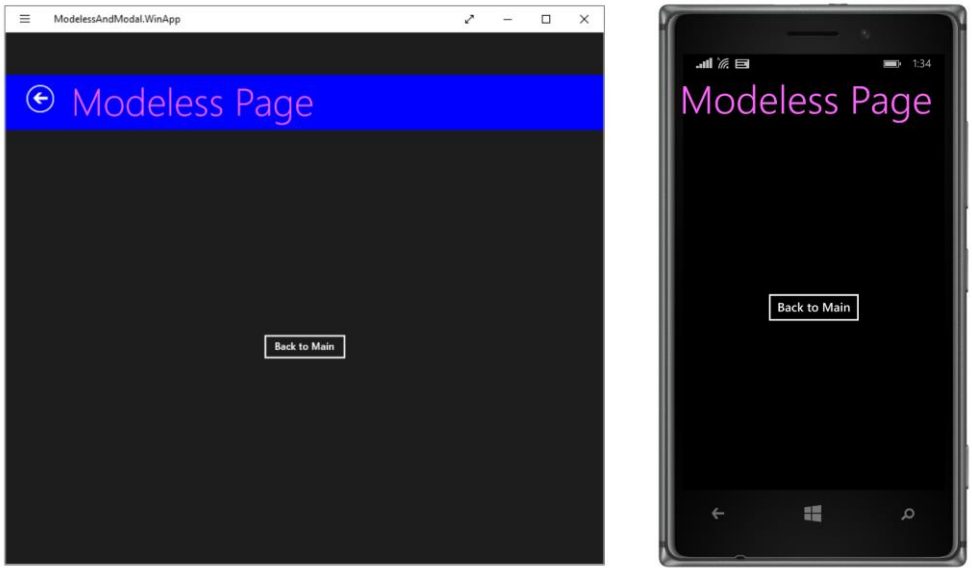
```
public class App : Application
{
    public App()
    {
        MainPage = new NavigationPage(new MainPage())
        {
            BarBackgroundColor = Color.Blue,
            BarTextColor = Color.Pink
        };
    }
```

The various platforms use these colors in different ways. The iOS navigation bar is affected by both colors, but on the Android screen, only the background color appears. All these screenshots show `ModelessPage`, but the top area of `MainPage` is colored in the same way:

The Windows 10 application in tablet mode looks quite similar to the Windows 10 Mobile screen:



The other two Windows Runtime platforms also make use of the `BarTextColor`, and the Windows 8.1 page uses `BarBackgroundColor` as well:



The `NavigationPage` class also defines a `Tint` property, but that property is deprecated and should be considered obsolete.

`NavigationPage` also defines four attached bindable properties that affect the particular `Page` class on which they are set. For example, suppose you don't want the **Back** button to appear on a modeless page. Here's how you set the `NavigationPage.HasBackButton` attached bindable property in code in the `ModelessPage` constructor:

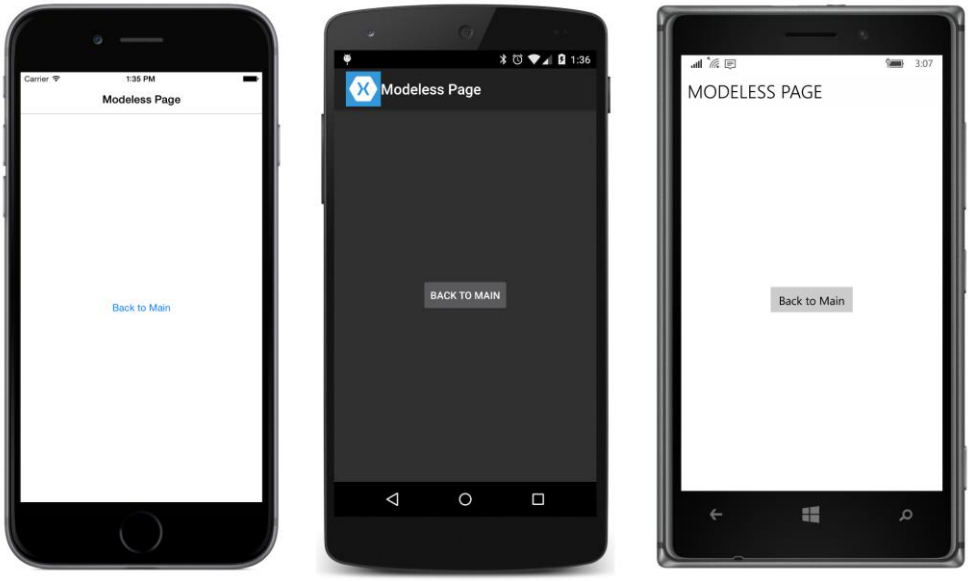
```
public class ModelessPage : ContentPage
{
    public ModelessPage()
    {
        Title = "Modeless Page";

        NavigationPage.SetHasBackButton(this, false);
        …
    }
}
```

In XAML, you would do it like so:

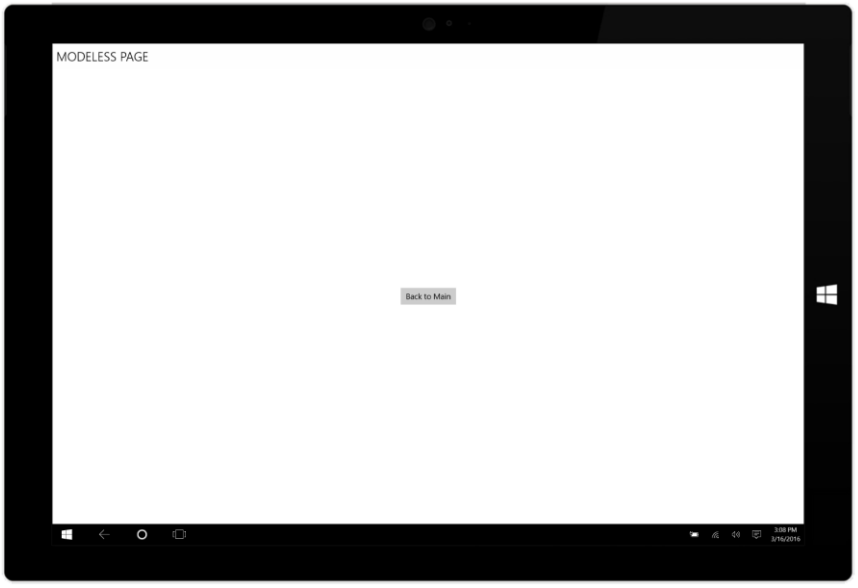
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ModelessAndModal.ModelessPage"
             Title="Modeless Page"
             NavigationPage.HasBackButton="False">
    …
</ContentPage>
```

And sure enough, when you navigate to `ModelessPage`, the **Back** button in the navigation bar is

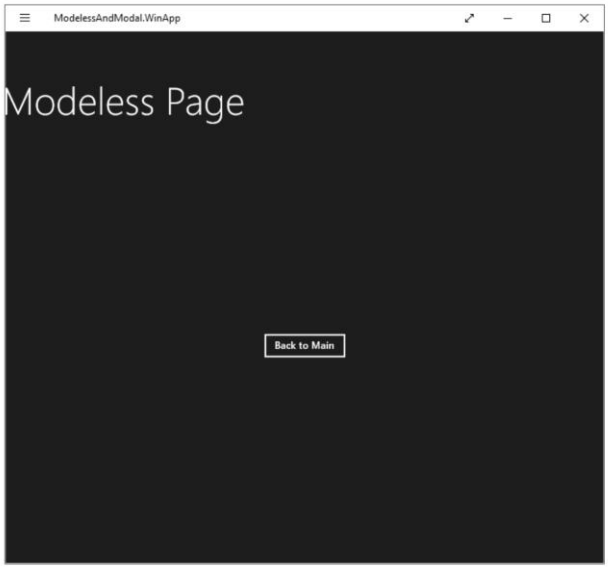
gone:



However, a functional **Back** button continues to exist on Windows 10:



The **Back** button is also gone from the Windows 8.1 screen:



A more extreme attached bindable property of `NavigationPage` eliminates the navigation bar entirely and renders the page visually indistinguishable from a modal page:

```
public class ModelessPage : ContentPage
{
    public ModelessPage()
    {
        Title = "Modeless Page";

        NavigationPage.SetHasNavigationBar(this, false);
        …
    }
}
```

Two more attached bindable properties affect the text and the icon in the navigation bar. As you've seen, all the platforms display the `Title` property at the top of the main page and a modeless page. However, on a modeless page, the iOS screen also displays the `Title` property of the *previous* page—the page from which you navigated to the modeless page. The `NavigationPage.BackButtonTitle` attached bindable property can change that text on the iOS page. You need to set it on the page from which you navigate to the modeless page. In the **ModelessAndModal** program, you can set the property on `MainPage` like so:

```
public class MainPage : ContentPage
{
    public MainPage()
    {
        Title = "Main Page";

        NavigationPage.SetBackButtonTitle(this, "go back");
        …
```

```
    }
}
```

This does not affect the title on `MainPage` itself, but only the text that accompanies the **Back** button on the navigation bar on `ModelessPage`, and then only on iOS. You'll see a screenshot shortly.

The second attached bindable property is `NavigationPage.TitleIcon`, which replaces the application icon on the Android navigation bar and replaces the title with an icon on the iOS page. The property is of type `FileImageSource`, which refers to a bitmap file in the platform project. In use, it's similar to the `Icon` property of `MenuItem` and `ToolbarItem`.

To let you experiment with this, some appropriate icons have been added to the iOS and Android projects in the **ModelessAndModal** solution. These icons come from the Android **Action Bar Icon Pack** discussed in Chapter 13, "Bitmaps." (In Chapter 13, look for the section "Platform-specific bitmaps," and then "Toolbars and their icons," and finally "Icons for Android.")

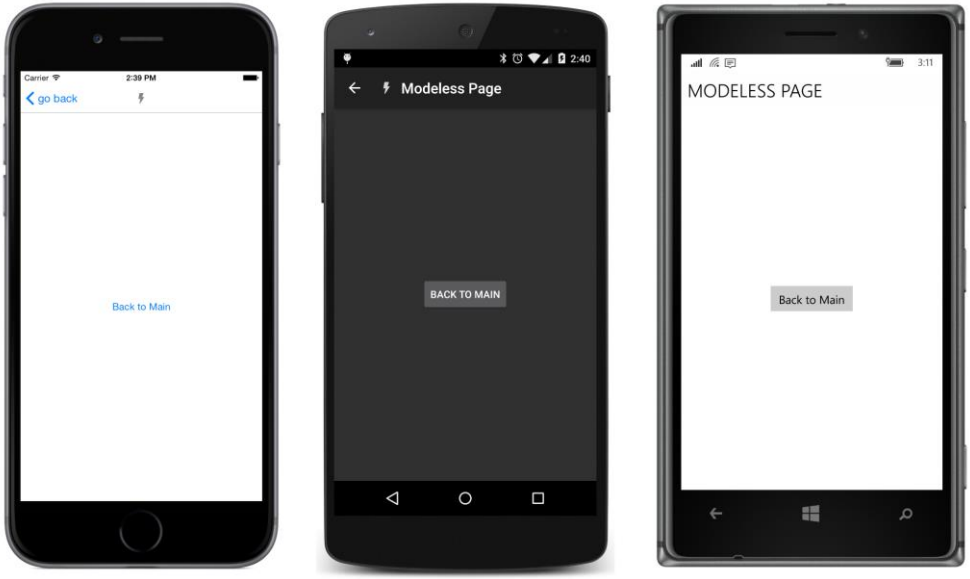
For iOS, the icons are from the **ActionBarIcons/holo_light/08_camera_flash_on** directory. These icons display a lightning bolt. The images in the **mdpi**, **xdpi**, and **xxdpi** directories are 32, 64, and 96 pixels square, respectively. Within the **Resources** folder of the iOS project, the 32-pixel square bitmap has the original name of ic_action_flash_on.png, and the two larger files were renamed with @2 and @3 suffixes, respectively.

For Android, the icons are from the **ActionBarIcons/holo_dark/08_camera_flash_on** directory; these are white foregrounds on transparent backgrounds. The files in the **mdpi**, **hdpi**, **xdpi**, and **xxdpi** directories were added to the Android project.

You can display these icons on the modeless page by adding the following code to the `ModelessPage` constructor:

```
public class ModelessPage : ContentPage
{
    public ModelessPage()
    {
        Title = "Modeless Page";

        if (Device.OS == TargetPlatform.iOS || Device.OS == TargetPlatform.Android)
            NavigationPage.SetTitleIcon(this, "ic_action_flash_on.png");
        …
    }
}
```

Here is `ModelessPage` with both the alternate **Back** button text of "go back" set on `MainPage` and the icons set on `ModelessPage`:



As you can see, the icon replaces the normal `Title` text on the iOS page.

Neither the `NavigationPage.BackButtonTitle` nor the `NavigationPage.TitleIcon` attached bindable property affect any of the Windows or Windows Phone platforms.

Programmers familiar with Android architecture are sometimes curious how Xamarin.Forms page navigation integrates with the aspect of Android application architecture known as the *activity*. A Xamarin.Forms application running on an Android device comprises only one activity, and the page navigation is built on top of that. A `ContentPage` is a Xamarin.Forms object; it is not an Android activity, or a fragment of an activity.

Exploring the mechanics

As you've seen, the `Push` and `Pop` methods return `Task` objects. Generally you'll use `await` when calling those methods. Here's the call to `PushAsync` in the `MainPage` class of **ModelessAndModal**:

```
await Navigation.PushAsync(new ModelessPage());
```

Suppose you have some code following this statement. When does that code get executed? We know it executes when the `PushAsync` task completes, but when is that? Is it after the user has tapped a **Back** button on `ModelessPage` to return back to `MainPage`?

No, that is not the case. The `PushAsync` task completes rather quickly. The completion of this task doesn't indicate that the process of page navigation has completed, but it *does* indicate when it is safe to obtain the current status of the page-navigation stack.

Following a `PushAsync` or `PushModalAsync` call, the following events occur. However, the precise order in which these events occur is platform dependent:

- The page calling `PushAsync` or `PushModalAsync` generally gets a call to its `OnDisappearing` override.
- The page being navigated to gets a call to its `OnAppearing` override.
- The `PushAsync` or `PushModalAsync` task completes.

To repeat: The order in which these events occur is dependent on the platform and also on whether navigation is to a modeless page or a modal page.

Following a `PopAsync` or `PopModalAsync` call, the following events occur, again in an order that is platform dependent:

- The page calling `PopAsync` or `PopModalAsync` gets a call to its `OnDisappearing` override.
- The page being returned to generally gets a call to its `OnAppearing` override.
- The `PopAsync` or `PopModalAsync` task returns.

You'll notice two uses of the word "generally" in those descriptions. This word refers to an exception to these rules when an Android device navigates to a modal page. The page that calls `PushModalAsync` does not get a call to its `OnDisappearing` override, and that same page does not get a call to its `OnAppearing` override when the modal page calls `PopModalAsync`.

Also, calls to the `OnDisappearing` and `OnAppearing` overrides do not necessarily indicate page navigation. On iOS, the `OnDisappearing` override is called on the active page when the program terminates. On the Windows Phone Silverlight platform (which is no longer supported by Xamarin.Forms), a page received `OnDisappearing` calls when the user invokes a `Picker`, `DatePicker`, or `TimePicker` on the page. For these reasons, the `OnDisappearing` and `OnAppearing` overrides cannot be treated as guaranteed indications of page navigation, although there are times when they must be used for that purpose.

The `INavigation` interface that defines these `Push` and `Pop` calls also defines two properties that provide access to the actual navigation stack:

- `NavigationStack`, which contains the modeless pages
- `ModalStack`, which contains the modal pages

The `set` accessors of these two properties are not public, and the properties themselves are of type `IReadOnlyList<Page>`, so you cannot directly modify them. (As you'll see, methods are available to modify the page stack in a more structured manner.) Although these properties are not implemented with `Stack<T>` classes, they function like a stack anyway. The item in the `IReadOnlyList` with an index of zero is the oldest page, and the last item is the most recent page.

The existence of these two collections for modeless and modal pages suggests that modeless and modal page navigation cannot be intermixed, and this is true: A modeless page can navigate to a modal page, but a modal page *cannot* navigate to a modeless page.

Some experimentation reveals that the `Navigation` property of different page instances retains different collections of the navigation stack. (In particular, after navigation to a modal page, the `NavigationStack` associated with that modal page is empty.) The most foolproof approach is to work with the instances of these collections maintained by the `Navigation` property of the `NavigationPage` instance set to the `MainPage` property of the `App` class.

With each call to `PushAsync` or `PopAsync`, the contents of the `NavigationStack` change—either a new page is added to the collection or a page is removed from the collection. Similarly, with each call to `PushModalAsync` or `PopModalAsync`, the contents of the `ModalStack` change.

Experimentation reveals that it is not safe to use the contents of the `NavigationStack` or `ModalStack` during calls to the `OnAppearing` or `OnDisappearing` overrides while the page navigation is in progress. The only approach that works for all the platforms is to wait until the `PushAsync`, `PushModalAsync`, `PopAsync`, or `PopModalAsync` task completes. That's your indication that these stack collections are stable and accurate.

The `NavigationPage` class also defines a get-only property named `CurrentPage`. This page instance is the same as the last item in the `NavigationStack` collections available from `NavigationPage`. However, when a modal page is active, `CurrentPage` continues to indicate the last *modeless* page that was active before navigation to a modal page.

Let's explore the details and mechanics of page navigation with a program called **SinglePageNavigation**, so called because the program contains only one page class, named `SinglePageNavigationPage`. The program navigates between various instances of this one class.

One of the purposes of the **SinglePageNavigation** program is to prepare you for writing an application that saves the navigation stack when the application is suspended or terminated, and to restore the stack when the application is restarted. Doing this depends on your application's ability to extract trustworthy information from the `NavigationStack` and `ModalStack` properties.

Here's the XAML file for the `SinglePageNavigationPage` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SinglePageNavigation.SinglePageNavigationPage"
             x:Name="page">

    <StackLayout>
        <StackLayout.Resources>
            <ResourceDictionary>
                <Style x:Key="baseStyle" TargetType="View">
                    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                </Style>

                <Style TargetType="Button" BasedOn="{StaticResource baseStyle}">
                    <Setter Property="HorizontalOptions" Value="Center" />
                </Style>

                <Style TargetType="Label" BasedOn="{StaticResource baseStyle}">
```

```
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </StackLayout.Resources>

    <Label Text="{Binding Source={x:Reference page}, Path=Title}" />

    <Button x:Name="modelessGoToButton"
            Text="Go to Modeless Page"
            Clicked="OnGoToModelessClicked" />

    <Button x:Name="modelessBackButton"
            Text="Back from Modeless Page"
            Clicked="OnGoBackModelessClicked" />

    <Button x:Name="modalGoToButton"
            Text="Go to Modal Page"
            Clicked="OnGoToModalClicked" />

    <Button x:Name="modalBackButton"
            Text="Back from Modal Page"
            Clicked="OnGoBackModalClicked" />

    <Label x:Name="currentPageLabel"
           Text=" " />

    <Label x:Name="modelessStackLabel"
           Text=" " />

    <Label x:Name="modalStackLabel"
           Text=" " />
  </StackLayout>
</ContentPage>
```

The XAML file instantiates four `Button` and four `Label` elements. The first `Label` has a data binding to display the page's `Title` property so that the title is visible regardless of the platform and whether the page is modal or modeless. The four buttons are for navigating to and from modeless or modal pages. The remaining three labels display other information set from code.

Here's roughly the first half of the code-behind file. Notice the constructor code that sets the `Title` property to the text "Page #," where the hash sign indicates a number starting at zero for the first instantiated page. Each time this class is instantiated, that number is increased:

```
public partial class SinglePageNavigationPage : ContentPage
{
    static int count = 0;
    static bool firstPageAppeared = false;
    static readonly string separator = new string('-', 20);

    public SinglePageNavigationPage()
    {
        InitializeComponent();
```

```
        // Set Title to zero-based instance of this class.
        Title = "Page " + count++;
    }

    async void OnGoToModelessClicked(object sender, EventArgs args)
    {
        SinglePageNavigationPage newPage = new SinglePageNavigationPage();
        Debug.WriteLine(separator);
        Debug.WriteLine("Calling PushAsync from {0} to {1}", this, newPage);
        await Navigation.PushAsync(newPage);
        Debug.WriteLine("PushAsync completed");

        // Display the page stack information on this page.
        newPage.DisplayInfo();
    }

    async void OnGoToModalClicked(object sender, EventArgs args)
    {
        SinglePageNavigationPage newPage = new SinglePageNavigationPage();
        Debug.WriteLine(separator);
        Debug.WriteLine("Calling PushModalAsync from {0} to {1}", this, newPage);
        await Navigation.PushModalAsync(newPage);
        Debug.WriteLine("PushModalAsync completed");

        // Display the page stack information on this page.
        newPage.DisplayInfo();
    }

    async void OnGoBackModelessClicked(object sender, EventArgs args)
    {
        Debug.WriteLine(separator);
        Debug.WriteLine("Calling PopAsync from {0}", this);
        Page page = await Navigation.PopAsync();
        Debug.WriteLine("PopAsync completed and returned {0}", page);

        // Display the page stack information on the page being returned to.
        NavigationPage navPage = (NavigationPage)App.Current.MainPage;
        ((SinglePageNavigationPage)navPage.CurrentPage).DisplayInfo();
    }

    async void OnGoBackModalClicked(object sender, EventArgs args)
    {
        Debug.WriteLine(separator);
        Debug.WriteLine("Calling PopModalAsync from {0}", this);
        Page page = await Navigation.PopModalAsync();
        Debug.WriteLine("PopModalAsync completed and returned {0}", page);

        // Display the page stack information on the page being returned to.
        NavigationPage navPage = (NavigationPage)App.Current.MainPage;
        ((SinglePageNavigationPage)navPage.CurrentPage).DisplayInfo();
    }

    protected override void OnAppearing()
    {
```

```
        base.OnAppearing();
        Debug.WriteLine("{0} OnAppearing", Title);

        if (!firstPageAppeared)
        {
            DisplayInfo();
            firstPageAppeared = true;
        }
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
        Debug.WriteLine("{0} OnDisappearing", Title);
    }

    // Identify each instance by its Title.
    public override string ToString()
    {
        return Title;
    }
    …
}
```

Each of the `Clicked` handlers for the four buttons displays some information using `Debug.WriteLine`. When you run the program under the debugger in Visual Studio or Xamarin Studio, this text appears in the **Output** window.

The code-behind file also overrides the `OnAppearing` and `OnDisappearing` methods. These are important, for they generally tell you when the page is being navigated to (`OnAppearing`) or navigated from (`OnDisappearing`).

But, as mentioned earlier, Android is a little different: An Android page that calls `PushModalAsync` does not get a call to its `OnDisappearing` method, and when the modal page returns to that page, the original page does not get a corresponding call to its `OnAppearing` method. It's as if the page stays in the background while the modal page is displayed, and that's very much the case: If you go back to **ModelessAndModal** and set the `BackgroundColor` of the modal page to `Color.FromRgba(0, 0, 0, 0.5)`, you can see the previous page behind the modal page. But this is only the case for Android.

All the `Clicked` handlers in `SinglePageNavigationPage` make a call to a method named `DisplayInfo`. This method is shown below and displays information about the `NavigationStack` and `ModalStack`—including the pages in the stacks—and the `CurrentPage` property maintained by the `NavigationPage` object.

However, these `Clicked` handlers do *not* call the `DisplayInfo` method in the current instance of the page because the `Clicked` handlers are effecting a transition to another page. The `Clicked` handlers must call the `DisplayInfo` method in the page instance to which they are navigating.

The calls to `DisplayInfo` in the `Clicked` handlers that call `PushAsync` and `PushModalAsync` are

easy because each `Clicked` handler already has the new page instance being navigated to. The calls to `DisplayInfo` in the `Clicked` handlers that call `PopAsync` and `PopModalAsync` are a little more difficult because they need to obtain the page being returned to. This is not the `Page` instance returned from the `PopAsync` and `PopModalAsync` tasks. That `Page` instance turns out to be the same page that calls these methods.

Instead, the `Clicked` handlers that call `PopAsync` and `PopModalAsync` obtain the page being returned to from the `CurrentPage` property of `NavigationPage`:

```
NavigationPage navPage = (NavigationPage)App.Current.MainPage;
((SinglePageNavigationPage)navPage.CurrentPage).DisplayInfo();
```

What's crucial is that the code to obtain this new `CurrentPage` property and the calls to `DisplayInfo` all occur *after* the asynchronous `Push` or `Pop` task has completed. That's when this information becomes valid.

However, the `DisplayInfo` method must also be called when the program first starts up. As you'll see, `DisplayInfo` makes use of the `MainPage` property of the `App` class to obtain the `NavigationPage` instantiated in the `App` constructor. However, that `MainPage` property has not yet been set in the `App` constructor when the `SinglePageNavigationPage` constructor executes, so the page constructor cannot call `DisplayInfo`. Instead, the `OnAppearing` override makes that call, but only for the first page instance:

```
if (!firstPageAppeared)
{
    DisplayInfo();
    firstPageAppeared = true;
}
```

Besides displaying the value of `CurrentPage` and the `NavigationStack` and `ModalStack` collections, the `DisplayInfo` method also enables and disables the four `Button` elements on the page so that it's always legal to press an enabled `Button`.

Here's `DisplayInfo` and the two methods it uses to display the stack collections:

```
public partial class SinglePageNavigationPage : ContentPage
{
    …
    public void DisplayInfo()
    {
        // Get the NavigationPage and display its CurrentPage property.
        NavigationPage navPage = (NavigationPage)App.Current.MainPage;

        currentPageLabel.Text = String.Format("NavigationPage.CurrentPage = {0}",
                                              navPage.CurrentPage);

        // Get the navigation stacks from the NavigationPage.
        IReadOnlyList<Page> navStack = navPage.Navigation.NavigationStack;
        IReadOnlyList<Page> modStack = navPage.Navigation.ModalStack;

        // Display the counts and contents of these stacks.
```

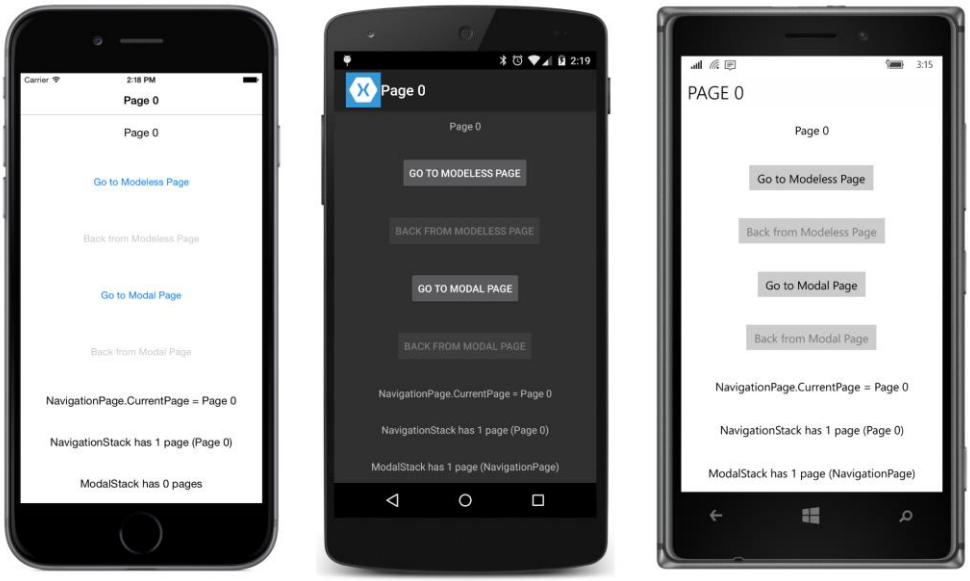
```
        int modelessCount = navStack.Count;
        int modalCount = modStack.Count;

        modelessStackLabel.Text = String.Format("NavigationStack has {0} page{1}{2}",
                                                modelessCount,
                                                modelessCount == 1 ? "" : "s",
                                                ShowStack(navStack));

        modalStackLabel.Text = String.Format("ModalStack has {0} page{1}{2}",
                                             modalCount,
                                             modalCount == 1 ? "" : "s",
                                             ShowStack(modStack));

        // Enable and disable buttons based on the counts.
        bool noModals = modalCount == 0 || (modalCount == 1 && modStack[0] is NavigationPage);

        modelessGoToButton.IsEnabled = noModals;
        modelessBackButton.IsEnabled = modelessCount > 1 && noModals;
        modalBackButton.IsEnabled = !noModals;
    }

    string ShowStack(IReadOnlyList<Page> pageStack)
    {
        if (pageStack.Count == 0)
            return "";

        StringBuilder builder = new StringBuilder();

        foreach (Page page in pageStack)
        {
            builder.Append(builder.Length == 0 ? " (" : ", ");
            builder.Append(StripNamespace(page));
        }

        builder.Append(")");
        return builder.ToString();
    }

    string StripNamespace(Page page)
    {
        string pageString = page.ToString();

        if (pageString.Contains("."))
            pageString = pageString.Substring(pageString.LastIndexOf('.') + 1);

        return pageString;
    }
}
```

Some of the logic involving the enabling and disabling of the `Button` elements will become apparent when you see some of the screens that the program displays. You can always comment out that enabling and disabling code to explore what happens when you press an invalid `Button`.

The general rules are these:

- A modeless page can navigate to another modeless page or a modal page.
- A modal page can navigate only to another modal page.

When you first run the program, you'll see the following. The XAML includes a display of the `Title` property at the top, so it's visible on all the pages:



The three `Label` elements on the bottom of the page display the `CurrentPage` property of the `NavigationPage` object and the `NavigationStack` and `ModalStack`, both obtained from the `Navigation` property of the `NavigationPage`.

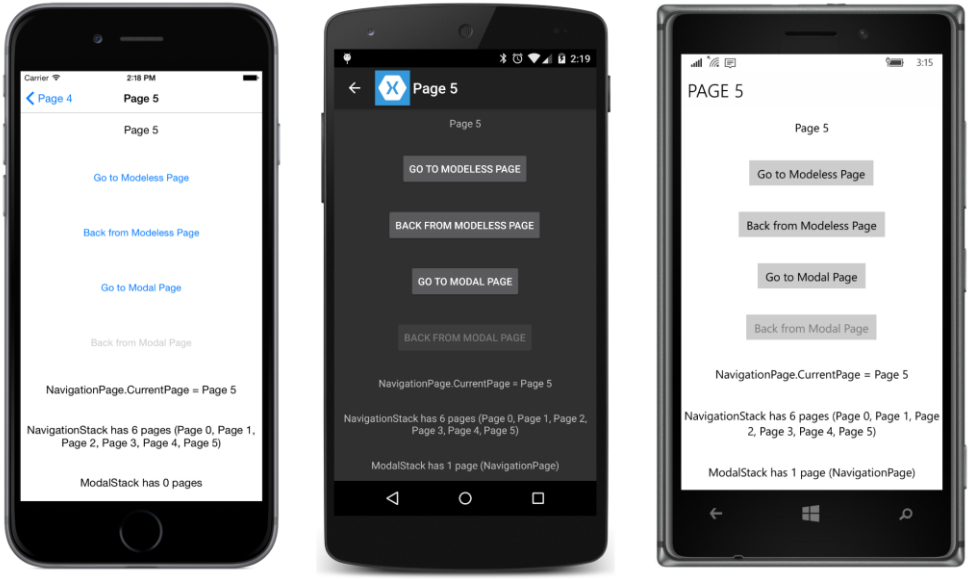
On all three platforms, the `NavigationStack` contains one item, which is the home page. The contents of the `ModalStack`, however, vary by platform. On the Android and Windows Runtime platforms, the modal stack contains one item (the `NavigationPage` object), but the modal stack is empty on iOS.

This is why the `DisplayInfo` method sets the `noModals` Boolean variable to `true` if either the modal stack has a count of zero or if it contains one item but that item is `NavigationPage`:

```
bool noModals = modalCount == 0 || (modalCount == 1 && modStack[0] is NavigationPage);
```

Notice that the `CurrentPage` property and the item in `NavigationStack` are not instances of `NavigationPage`, but instead are instances of `SinglePageNavigationPage`, which derives from `ContentPage`. It is `SinglePageNavigationPage` that defines its `ToString` method to display the page title.

Now press the **Go to Modeless Page** button five times and here's what you'll see. The screens are consistent aside from the modal stack on the iOS screen:



As soon as you press the **Go to Modeless Page** button once, the **Back from Modeless Page** button is enabled. The logic is this:

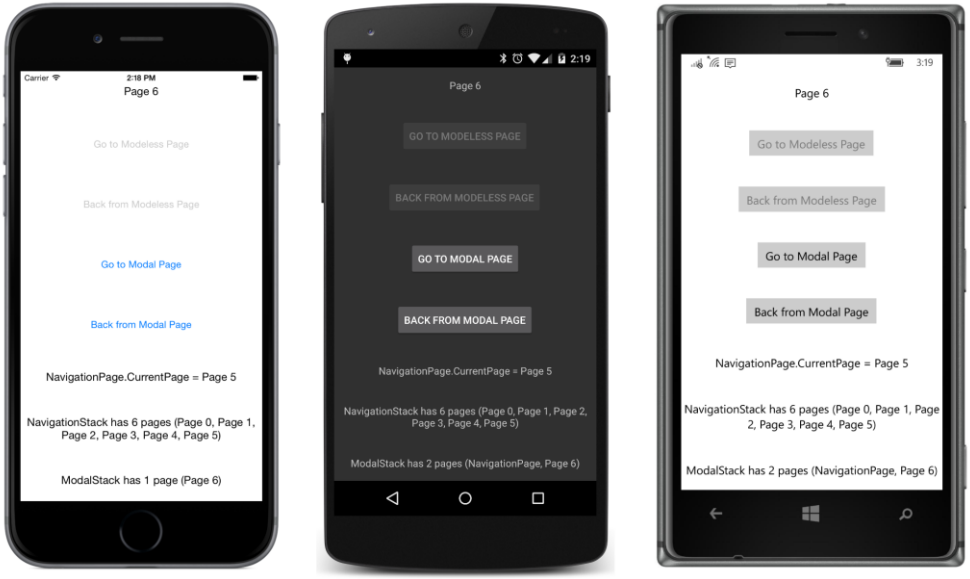
```
modelessBackButton.IsEnabled = modelessCount > 1 && noModals;
```

In plain English, the **Back from Modeless Page** button should be enabled if there are at least two items in the modeless stack (the original page and the current page) and if the current page is not a modal page.

If you press that **Back from ModelessPage** button at this point, you'll see the `NavigationStack` shrink in size until you get back to **Page 0**. Throughout, the `CurrentPage` property continues to indicate the last item in `NavigationStack`.

If you then press **Go to Modeless Page** again, you will see more items added to the `NavigationStack` with ever-increasing page numbers because new `SinglePageNavigationPage` objects are being instantiated.

Instead, try pressing the **Go to Modal Page** button:



Now `ModalStack` contains that new page, but `CurrentPage` still refers to the last modeless page. The iOS modal stack is still missing that initial `NavigationPage` object present in the other platforms.

If you then press **Back from Modal Page**, the modal stack is properly restored to its initial state.

Multipage applications usually try to save the navigation stack when they are suspended or terminated, and then restore that stack when they start up again. Toward the end of this chapter, you'll see code that uses `NavigationStack` and `ModalStack` to do that job.

Enforcing modality

In general, your applications will probably use modeless pages except for special circumstances when the application needs to obtain crucial information from the user. The application can then display a modal page that the user cannot dismiss until this crucial information has been entered.

One little problem, however, is that an Android or Windows Phone user can always return to the previous page by pressing the standard **Back** button on the device. To enforce modality—to make sure that the user enters the desired information before leaving the page—the application must disable that button.

This technique is demonstrated in the **ModalEnforcement** program. The home page consists solely of a `Button`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ModalEnforcement.ModalEnforcementHomePage"
             Title="Main Page">
```

```
    <Button Text="Go to Modal Page"
            HorizontalOptions="Center"
            VerticalOptions="Center"
            Clicked="OnGoToButtonClicked" />

</ContentPage>
```

The code-behind file handles the `Clicked` event of the button by navigating to a modal page:

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

    async void OnGoToButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PushModalAsync(new ModalEnforcementModalPage());
    }
}
```

The XAML file for the `ModalEnforcementModalPage` contains two `Entry` elements, a `Picker` element, and a `Button` labeled **Done**. The markup is more extensive than you might anticipate because it contains a `MultiTrigger` to set the `IsEnabled` property of the button to `True` only if something has been typed into the two `Entry` elements and something has also been entered into the `Picker`. This `MultiTrigger` requires three hidden `Switch` elements, using a technique discussed in Chapter 23, "Triggers and behaviors":

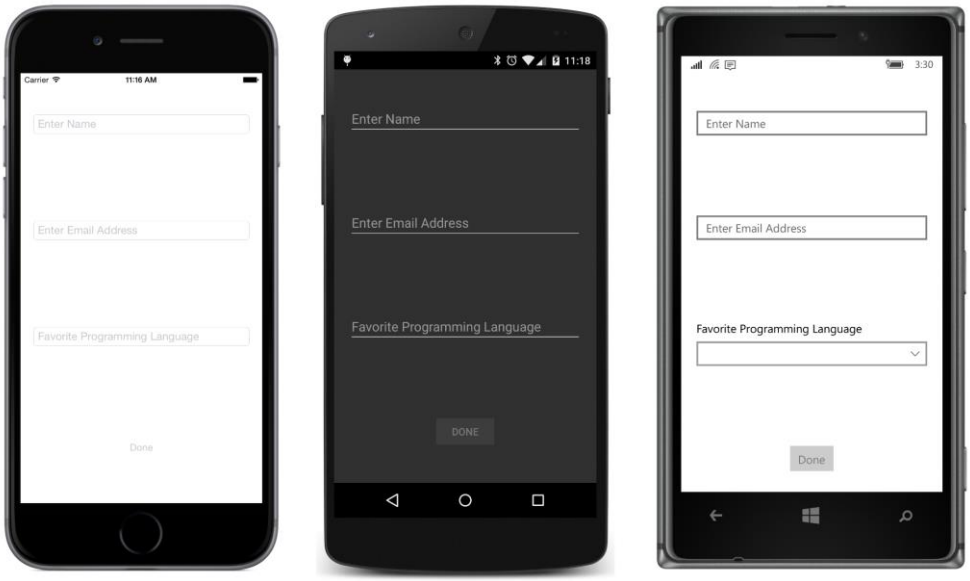
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ModalEnforcement.ModalEnforcementModalPage"
             Title="Modal Page">

    <StackLayout Padding="20, 0">
        <Entry x:Name="entry1"
               Text=""
               Placeholder="Enter Name"
               VerticalOptions="CenterAndExpand" />

        <!-- Invisible Switch to help with MultiTrigger logic -->
        <Switch x:Name="switch1" IsVisible="False">
            <Switch.Triggers>
                <DataTrigger TargetType="Switch"
                             Binding="{Binding Source={x:Reference entry1}, Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsToggled" Value="True" />
                </DataTrigger>
            </Switch.Triggers>
        </Switch>

        <Entry x:Name="entry2"
               Text=""
```

```
            Placeholder="Enter Email Address"
            VerticalOptions="CenterAndExpand" />

        <!-- Invisible Switch to help with MultiTrigger logic -->
        <Switch x:Name="switch2" IsVisible="False">
            <Switch.Triggers>
                <DataTrigger TargetType="Switch"
                             Binding="{Binding Source={x:Reference entry2}, Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsToggled" Value="True" />
                </DataTrigger>
            </Switch.Triggers>
        </Switch>

        <Picker x:Name="picker"
                Title="Favorite Programming Language"
                VerticalOptions="CenterAndExpand">
            <Picker.Items>
                <x:String>C#</x:String>
                <x:String>F#</x:String>
                <x:String>Objective C</x:String>
                <x:String>Swift</x:String>
                <x:String>Java</x:String>
            </Picker.Items>
        </Picker>

        <!-- Invisible Switch to help with MultiTrigger logic -->
        <Switch x:Name="switch3" IsVisible="False">
            <Switch.Triggers>
                <DataTrigger TargetType="Switch"
                             Binding="{Binding Source={x:Reference picker}, Path=SelectedIndex}"
                             Value="-1">
                    <Setter Property="IsToggled" Value="True" />
                </DataTrigger>
            </Switch.Triggers>
        </Switch>

        <Button x:Name="doneButton"
                Text="Done"
                IsEnabled="False"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Clicked="OnDoneButtonClicked">
            <Button.Triggers>
                <MultiTrigger TargetType="Button">
                    <MultiTrigger.Conditions>
                        <BindingCondition Binding="{Binding Source={x:Reference switch1},
                                                            Path=IsToggled}"
                                          Value="False" />

                        <BindingCondition Binding="{Binding Source={x:Reference switch2},
                                                            Path=IsToggled}"
                                          Value="False" />
```

```
                    <BindingCondition Binding="{Binding Source={x:Reference switch3},
                                                        Path=IsToggled}"
                                      Value="False" />
                </MultiTrigger.Conditions>

                <Setter Property="IsEnabled" Value="True" />
            </MultiTrigger>
        </Button.Triggers>
    </Button>
</StackLayout>
</ContentPage>
```

In a real-life program, there would probably also be a check that the email address is valid. The simple logic in the XAML file simply checks for the presence of at least one character.

Here's the modal page as it first appears, when nothing has yet been entered. Notice that the **Done** button is disabled:



Normally the user can still press the standard **Back** button at the bottom left of the Android and Windows Phone screens to return back to the main page. To inhibit the normal behavior of the **Back** button, the modal page must override the virtual `OnBackButtonPressed` method. You can supply your own **Back** button processing in this override and return `true`. To entirely disable the **Back** button, simply return `true` without doing anything else. To allow the default **Back** button processing to occur, call the base class implementation. Here's how the code-behind file of `ModalEnforcementModalPage` does it:

```
public partial class ModalEnforcementModalPage : ContentPage
{
    public ModalEnforcementModalPage()
```

```
    {
        InitializeComponent();
    }

    protected override bool OnBackButtonPressed()
    {
        if (doneButton.IsEnabled)
        {
            return base.OnBackButtonPressed();
        }
        return true;
    }

    async void OnDoneButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PopModalAsync();
    }
}
```

Only if the **Done** button in the XAML file is enabled will the `OnBackButtonPressed` override call the base class implementation of the method and return the value that is returned from that implementation. This causes the modal page to return to the page that invoked it. If the **Done** button is disabled, then the override returns `true` indicating that it is finished performing all the handling that it desires for the **Back** button.

The `Clicked` handler for the **Done** button simply calls `PopModalAsync`, as usual.

The `Page` class also defines a `SendBackButtonPressed` that causes the `OnBackButtonPressed` method to be called. It should be possible to implement the `Clicked` handler for the **Done** button by calling this method:

```
void OnDoneButtonClicked(object sender, EventArgs args)
{
    SendBackButtonPressed();
}
```

Although this works on iOS and Android, it currently does not work on the Windows Runtime platforms.

In real-world programming, it's more likely that you'll be using a ViewModel to accumulate the information that the user enters into the modal page. In that case, the ViewModel itself can contain a property that indicates whether all the information entered is valid.

The **MvvmEnforcement** program uses this technique, and includes a little ViewModel—appropriately named `LittleViewModel`:

```
namespace MvvmEnforcement
{
    public class LittleViewModel : INotifyPropertyChanged
    {
        string name, email;
        string[] languages = {  "C#", "F#", "Objective C", "Swift", "Java" };
```

```
int languageIndex = -1;
bool isValid;

public event PropertyChangedEventHandler PropertyChanged;

public string Name
{
    set
    {
        if (name != value)
        {
            name = value;
            OnPropertyChanged("Name");
            TestIfValid();
        }
    }
    get { return name; }
}

public string Email
{
    set
    {
        if (email != value)
        {
            email = value;
            OnPropertyChanged("Email");
            TestIfValid();
        }
    }
    get { return email; }
}

public IEnumerable<string> Languages
{
    get { return languages; }
}

public int LanguageIndex
{
    set
    {
        if (languageIndex != value)
        {
            languageIndex = value;
            OnPropertyChanged("LanguageIndex");

            if (languageIndex >= 0 && languageIndex < languages.Length)
            {
                Language = languages[languageIndex];
                OnPropertyChanged("Language");
            }
            TestIfValid();
        }
```

```
            }
            get { return languageIndex; }
        }

        public string Language { private set; get; }

        public bool IsValid
        {
            private set
            {
                if (isValid != value)
                {
                    isValid = value;
                    OnPropertyChanged("IsValid");
                }
            }
            get { return isValid; }
        }

        void TestIfValid()
        {
            IsValid = !String.IsNullOrWhiteSpace(Name) &&
                      !String.IsNullOrWhiteSpace(Email) &&
                      !String.IsNullOrWhiteSpace(Language);
        }

        void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;

            if (handler != null)
                handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The `Name` and `Email` properties are of type `string` for the purpose of binding with the `Text` properties of an `Entry` element. The `LanguageIndex` property is intended to be bound to the `SelectedIndex` property of the `Picker`. But the `set` accessor for `LanguageIndex` uses that value to set the `Language` property of type `string` from an array of strings in the `Languages` collection.

Whenever the `Name`, `Email`, or `LanguageIndex` property changes, the `TestIfValid` method is called to set the `IsValid` property. This property can be bound to the `IsEnabled` property of the `Button`.

The home page in **MvvmEnforcement** is the same as the one in **ModalEnforcement,** but of course the XAML file for the modal page is quite a bit simpler and implements all the data bindings:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MvvmEnforcement.MvvmEnforcementModalPage"
             Title="Modal Page">
```

```
    <StackLayout Padding="20, 0">
        <Entry Text="{Binding Name}"
               Placeholder="Enter Name"
               VerticalOptions="CenterAndExpand" />

        <Entry Text="{Binding Email}"
               Placeholder="Enter Email Address"
               VerticalOptions="CenterAndExpand" />

        <Picker x:Name="picker"
                Title="Favorite Programming Language"
                SelectedIndex="{Binding LanguageIndex}"
                VerticalOptions="CenterAndExpand" />

        <Button Text="Done"
                IsEnabled="{Binding IsValid}"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Clicked="OnDoneButtonClicked" />
    </StackLayout>
</ContentPage>
```

The markup contains four bindings to properties in the ViewModel.

The modal page's code-behind file is responsible for instantiating `LittleViewModel` and setting the object to the `BindingContext` property of the page, which it does in the constructor. The constructor also accesses the `Languages` collection of the ViewModel to set the `Items` property of the `Picker`. (Unfortunately the `Items` property is not backed by a bindable property and hence is not bindable.)

The remainder of the file is fairly similar to the modal page in **ModalEnforcement** except that the `OnBackButtonPressed` override accesses the `IsValid` property of `LittleViewModel` to determine whether to call the base class implementation or return `true`:

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

        LittleViewModel viewModel = new LittleViewModel();
        BindingContext = viewModel;

        // Populate Picker Items list.
        foreach (string language in viewModel.Languages)
        {
            picker.Items.Add(language);
        }
    }

    protected override bool OnBackButtonPressed()
    {
        LittleViewModel viewModel = (LittleViewModel)BindingContext;
```

```
        return viewModel.IsValid ? base.OnBackButtonPressed() : true;
    }

    async void OnDoneButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PopModalAsync();
    }
}
```

Navigation variations

As you've experimented with the **ModalEnforcement** and **MvvmEnforcement** programs, you might have felt disconcerted by the failure of the modal pages to retain any information. We've all encountered programs and websites that navigate to a page used to enter information, but when you leave that page and then later return, all the information you entered is gone! Such pages can be very annoying.

Even in simple demonstration samples like **ModalEnforcement** and **MvvmEnforcement**, it's possible to fix that problem very easily by creating only a single instance of the modal page—perhaps when the program starts up—and then using that single instance throughout.

Despite the apparent ease of this solution, it is not a good generalized approach to the problem of retaining page information. This technique should probably be avoided except for the simplest of cases. Keeping a lot of pages active could result in memory issues, and you must be careful to avoid having the same page instance in the navigation stack more than once.

Nevertheless, here's how you can modify the `ModalEnforcementHomePage` code-behind file for this technique:

```
public partial class ModalEnforcementHomePage : ContentPage
{
    ModalEnforcementModalPage modalPage = new ModalEnforcementModalPage();

    public ModalEnforcementHomePage()
    {
        InitializeComponent();
    }

    async void OnGoToButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PushModalAsync(modalPage);
    }
}
```

The `ModalEnforcementHomePage` saves an instance of `ModalEnforcementModalPage` as a field and then always passes that single instance to `PushModalAsync`.

In less-simple applications, this technique can easily go wrong: Sometimes a particular type of page

in an application can be navigated to from several different pages, and that might result in two separate, inconsistent instances of `ModalPage`.

This technique collapses entirely if you need to save the state of the program when it terminates and restore it when it executes again. You can't save and restore the page instances themselves. It's generally the data associated with the page that must be saved.

In real-life programming, ViewModels often form the backbone of page types in a multipage application, and the best way that an application can retain page data is through the ViewModel rather than the page.

A much better way to maintain page state when a modal page is invoked several times in succession can be demonstrated using **MvvmEnforcement**. First, add a property to the `App` page for `LittleViewModel` and instantiate that class in the `App` constructor:

```
namespace MvvmEnforcement
{
    public class App : Application
    {
        public App()
        {
            ModalPageViewModel = new LittleViewModel();

            MainPage = new NavigationPage(new MvvmEnforcementHomePage());
        }

        public LittleViewModel ModalPageViewModel { private set; get; }
        …
    }
}
```

Because the `LittleViewModel` is instantiated just once, it maintains the information for the duration of the application. Each new instance of `MvvmEnforcementModalPage` can then simply access this property and set the `ViewModel` object to its `BindingContext`:

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

        LittleViewModel viewModel = ((App)Application.Current).ModalPageViewModel;
        BindingContext = viewModel;

        // Populate Picker Items list.
        foreach (string language in viewModel.Languages)
        {
            picker.Items.Add(language);
        }
    }
    …
}
```

Of course, once the program terminates, the information is lost, but the `App` class can also save that information in the `Properties` property of `Application`—a technique first demonstrated in the **PersistentKeypad** program toward the end of Chapter 6, "Button clicks"—and then retrieve it when the application starts up again.

The problems of retaining data—and passing data between pages—will occupy much of the focus of the later sections of this chapter.

Making a navigation menu

If your application consists of a variety of different but architecturally identical pages, all of which are navigable from the home page, you might be interested in constructing what is sometimes called a *navigation menu*. This is a menu in which each entry is a particular page type.

The **ViewGalleryType** program is intended to demonstrate all the `View` classes in Xamarin.Forms. It contains a home page and one page for every instantiable class in Xamarin.Forms that derives from `View`—but not `Layout`—with the exception of `Map` and `OpenGLView`. That's 18 classes and 18 `ContentPage` derivatives, plus the home page. (The reason for the **Type** suffix on the project name will become apparent shortly.)

These 18 page classes are all stored in a folder named **ViewPages** in the Portable Class Library. Here is one example: SliderPage.xaml. It's just a `Slider` with a `Label` bound to the `Value` property:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ViewGalleryType.SliderPage"
             Title="Slider">

    <StackLayout Padding="10, 0">
        <Slider x:Name="slider"
                VerticalOptions="CenterAndExpand" />

        <Label Text="{Binding Source={x:Reference slider},
                              Path=Value,
                              StringFormat='The Slider value is {0}'}"
               VerticalOptions="CenterAndExpand"
               HorizontalAlignment="Center" />
    </StackLayout>
</ContentPage>
```

The other 17 are similar. Some of the pages have a little code in the code-behind file, but most of them simply have a call to `InitializeComponent`.

In addition, the **ViewGalleryType** project has a folder named **Images** that contains 18 bitmaps with the name of each `View` derivative stretched out to nearly fill the bitmap's surface. These bitmaps were generated by a Windows Presentation Foundation program and are flagged as **EmbeddedResource** in the project. The project also contains an `ImageResourceExtension` class described in Chapter 13, in the section "Embedded resources," to reference the bitmaps from the XAML file.

The home page is named `ViewGalleryTypePage`. It assembles 18 `ImageCell` elements in six different sections of a `Table`:

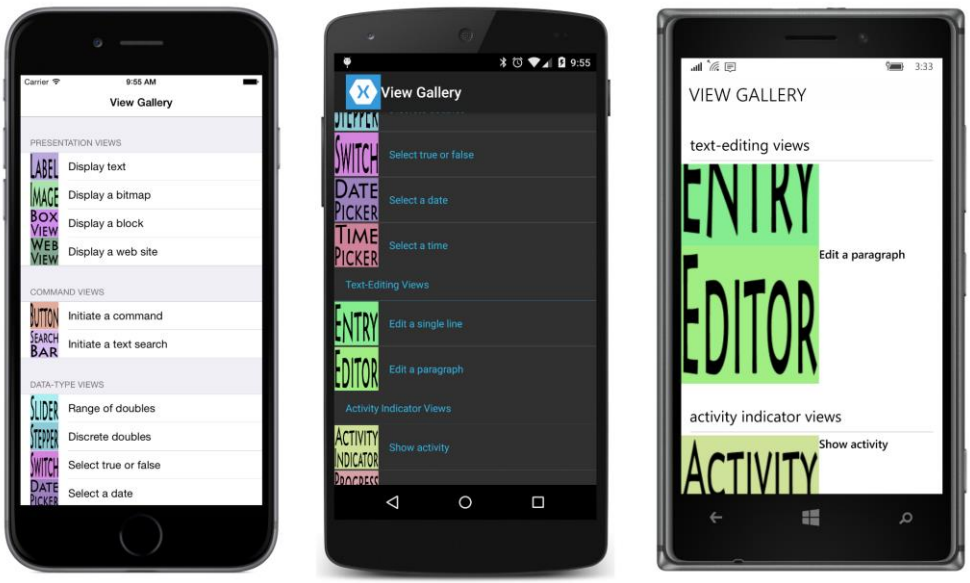
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ViewGalleryType;assembly=ViewGalleryType"
             x:Class="ViewGalleryType.ViewGalleryTypePage"
             Title="View Gallery">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection Title="Presentation Views">
                <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Label.png}"
                           Text="Display text"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:LabelPage}" />

                <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Image.png}"
                           Text="Display a bitmap"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:ImagePage}" />

                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryType.Images.BoxView.png}"
                           Text="Display a block"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:BoxViewPage}" />

                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryType.Images.WebView.png}"
                           Text="Display a web site"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:WebViewPage}" />
            </TableSection>

            <TableSection Title="Command Views">
                <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Button.png}"
                           Text="Initiate a command"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:ButtonPage}" />

                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryType.Images.SearchBar.png}"
                           Text="Initiate a text search"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:SearchBarPage}" />
            </TableSection>

            <TableSection Title="Data-Type Views">
                <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Slider.png}"
                           Text="Range of doubles"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:SliderPage}" />

                <ImageCell ImageSource=
```

```
                    "{local:ImageResource ViewGalleryType.Images.Stepper.png}"
                    Text="Discrete doubles"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:StepperPage}" />

        <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Switch.png}"
                    Text="Select true or false"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:SwitchPage}" />

        <ImageCell ImageSource=
                    "{local:ImageResource ViewGalleryType.Images.DatePicker.png}"
                    Text="Select a date"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:DatePickerPage}" />

        <ImageCell ImageSource=
                    "{local:ImageResource ViewGalleryType.Images.TimePicker.png}"
                    Text="Select a time"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:TimePickerPage}" />
    </TableSection>

    <TableSection Title="Text-Editing Views">
        <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Entry.png}"
                    Text="Edit a single line"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:EntryPage}" />

        <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Editor.png}"
                    Text="Edit a paragraph"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:EditorPage}" />
    </TableSection>

    <TableSection Title="Activity Indicator Views">
        <ImageCell ImageSource=
                    "{local:ImageResource ViewGalleryType.Images.ActivityIndicator.png}"
                    Text="Show activity"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ActivityIndicatorPage}" />

        <ImageCell ImageSource=
                    "{local:ImageResource ViewGalleryType.Images.ProgressBar.png}"
                    Text="Show progress"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ProgressBarPage}" />
    </TableSection>

    <TableSection Title="Collection Views">
        <ImageCell ImageSource="{local:ImageResource ViewGalleryType.Images.Picker.png}"
                    Text="Pick item from list"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:PickerPage}" />
```

```
                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryType.Images.ListView.png}"
                           Text="Show a collection"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:ListViewPage}" />

                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryType.Images.TableView.png}"
                           Text="Show form or menu"
                           Command="{Binding NavigateCommand}"
                           CommandParameter="{x:Type local:TableViewPage}" />
            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>
```

Each `ImageCell` has a reference to a bitmap indicating the view's name and a `Text` property that briefly describes the view. The `Command` property of `ImageCell` is bound to an `ICommand` object that is implemented in the code-behind file, and the `CommandParameter` is an `x:Type` markup extension that references one of the page classes. As you'll recall, the `x:Type` markup extension is the XAML equivalent of the C# `typeof` operator and results in each `CommandParameter` being of type `Type`.

Here's what the home page looks like on the three platforms:

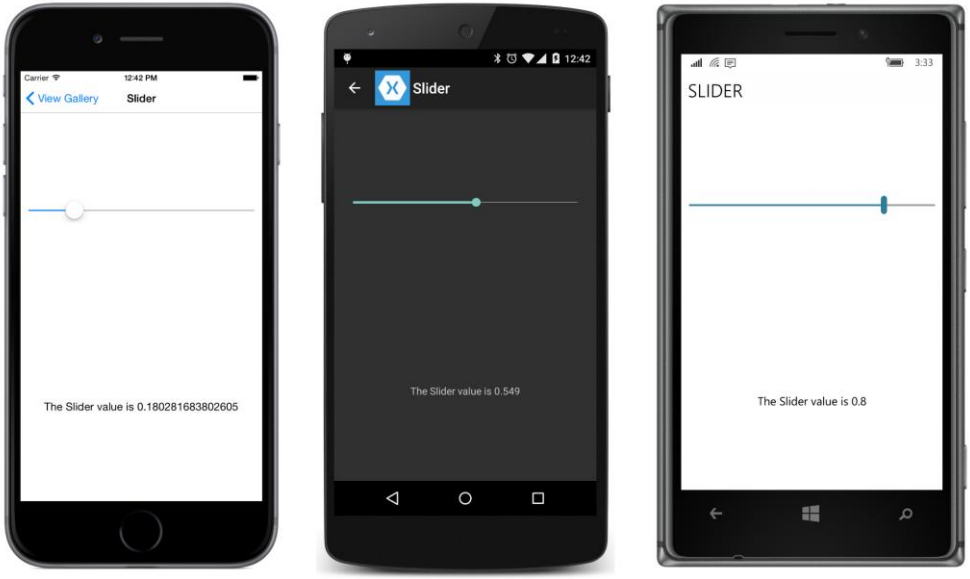


The code-behind file defines the `NavigateCommand` property that each `ImageCell` references in a binding. The `Execute` method is implemented as a lambda function: It passes the `Type` argument (set from the `CommandParameter` in the XAML file) to `Activator.CreateInstance` to instantiate the page and then navigates to that page:

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

        NavigateCommand = new Command<Type>(async (Type pageType) =>
        {
            Page page = (Page)Activator.CreateInstance(pageType);
            await Navigation.PushAsync(page);
        });

        BindingContext = this;
    }

    public ICommand NavigateCommand { private set; get; }
}
```

The constructor concludes by setting its `BindingContext` property to itself, so each `ImageCell` in the XAML file can reference the `NavigateCommand` property with a simple `Binding`.

Taping the `Slider` entry (for example) navigates to `SliderPage`:



Returning to the home page requires using the navigation bar on the iOS and Android screens or the **Back** button on the Android and Windows 10 Mobile screens.

A new instance of each page is created each time you navigate to that page, so of course these different instances of `SliderPage` won't retain the value of the `Slider` you might have previously set.

Is it possible to create just a single instance of each of these 18 pages? Yes, and that is demonstrated in **ViewGalleryInst**. The **Inst** suffix stands for "instance" to distinguish the program from the use of a page type in **ViewGalleryType**.

The 18 page classes for each view are the same, as are the bitmaps. The home page, however, now expresses the `CommandParameter` property of each `ImageCell` as a property element to instantiate each page class. Here is an excerpt:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ViewGalleryInst;assembly=ViewGalleryInst"
             x:Class="ViewGalleryInst.ViewGalleryInstPage"
             Title="View Gallery">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection Title="Presentation Views">
                <ImageCell ImageSource="{local:ImageResource ViewGalleryInst.Images.Label.png}"
                           Text="Display text"
                           Command="{Binding NavigateCommand}">
                    <ImageCell.CommandParameter>
                        <local:LabelPage />
                    </ImageCell.CommandParameter>
                </ImageCell>
                …
                <ImageCell ImageSource=
                           "{local:ImageResource ViewGalleryInst.Images.TableView.png}"
                           Text="Show form or menu"
                           Command="{Binding NavigateCommand}">
                    <ImageCell.CommandParameter>
                        <local:TableViewPage />
                    </ImageCell.CommandParameter>
            </ImageCell>
            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>
```

Now when you manipulate the `Slider` on the `SliderPage`, and then return back to home and navigate to the `SliderPage` again, the `Slider` will be the same because it's the same page instance.

Keep in mind that with this configuration, a total of 19 page classes are instantiated when your program starts up, and that means 19 XAML files are being parsed, and that might affect the startup performance, and occupy a lot of memory as well.

Moreover, any errors in the XAML files that are found during this run-time parsing will also manifest themselves at program startup. It might be difficult to discover exactly which XAML file has the problem! When building a program that instantiates many page classes in one shot, you'll want to add new classes incrementally to make sure everything works well before proceeding.

Better yet, avoid this technique entirely. Instantiate each page as you need it, and retain data associated with the page by using a ViewModel.

Manipulating the navigation stack

Sometimes it's necessary to alter the normal stack-oriented flow of navigation. For example, suppose a page needs some information from the user, but first it navigates to a page that provides some instructions or a disclaimer, and then from there navigates to the page that actually obtains the information. When the user is finished and goes back, you'll want to skip that page with the instructions or disclaimer. That page should be removed from the navigation stack.

Here's a similar example: Suppose the user is interacting with a page that obtains some information and then wants to go back to the previous page. However, the program detects that something is wrong with this information that requires an extended discussion on a separate page. The program could insert a new page into the navigation stack to provide that discussion.

Or a certain sequence of pages might end with a `Button` labeled **Go to Home**, and all the pages in between can simply be skipped when navigating back to the home page.

The `INavigation` interface defines methods for all three of these cases. They are named `RemovePage`, `InsertPageBefore`, and `PopToRootAsync`.

The **StackManipulation** program demonstrates these three methods, but in a very abstract manner. The program consists of five code-only pages, named `PageA`, `PageB`, `PageBAlternative`, `PageC`, and `PageD`. Each page sets its `Title` property to identify itself.

`PageA` has a `Button` to navigate to `PageB`:

```
public class PageA : ContentPage
{
    public PageA()
    {
        Button button = new Button
        {
            Text = "Go to Page B",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.Center
        };
        button.Clicked += async (sender, args) =>
        {
            await Navigation.PushAsync(new PageB());
        };

        Title = "Page A";
        Content = new button;
    }
}
```

`PageB` is similar, except that it navigates to `PageC`. `PageBAlternative` is the same as `PageB` except that it identifies itself as "Page B Alt". `PageC` has a `Button` to navigate to `PageD`, and `PageD` has two buttons:

```
public class PageD : ContentPage
```

```
{
    public PageD()
    {
        // Create Button to go directly to PageA.
        Button homeButton = new Button
        {
            Text = "Go Directly to Home",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        homeButton.Clicked += async (sender, args) =>
        {
            await Navigation.PopToRootAsync();
        };

        // Create Button to swap pages.
        Button swapButton = new Button
        {
            Text = "Swap B and Alt B",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Button)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        swapButton.Clicked += (sender, args) =>
        {
            IReadOnlyList<Page> navStack = Navigation.NavigationStack;
            Page pageC = navStack[navStack.Count - 2];
            Page existingPageB = navStack[navStack.Count - 3];
            bool isOriginal = existingPageB is PageB;
            Page newPageB = isOriginal ? (Page)new PageBAlternative() : new PageB();

            // Swap the pages.
            Navigation.RemovePage(existingPageB);
            Navigation.InsertPageBefore(newPageB, pageC);

            // Finished: Disable the Button.
            swapButton.IsEnabled = false;
        };

        Title = "Page D";
        Content = new StackLayout
        {
            Children =
            {
                homeButton,
                swapButton
            }
        };
    }
}
```

The button labeled **Go Directly to Home** has a `Clicked` handler that calls `PopToRootAsync`. This causes the program to jump back to `PageA` and effectively clears the navigation stack of all intermediary pages.

The button labeled **Swap B and Alt B** is a little more complex. The `Clicked` handler for this button replaces `PageB` with `PageBAlternative` in the navigation stack (or vice versa), so when you go back through the pages, you'll encounter a different page B. Here's how the `Clicked` handler does it:

At the time the `Button` is clicked, the `NavigationStack` has four items with indices 0, 1, 2, and 3. These four indices correspond to objects in the stack of type `PageA`, `PageB` (or `PageBAlternative`), `PageC`, and `PageD`. The handler accesses the `NavigationStack` to obtain these actual instances:

```
IReadOnlyList<Page> navStack = Navigation.NavigationStack;
Page pageC = navStack[navStack.Count - 2];
Page existingPageB = navStack[navStack.Count - 3];
```

That `existingPageB` object might be of type `PageB` or `PageBAlternative`, so a `newPageB` object is created of the other type:

```
bool isOriginal = existingPageB is PageB;
Page newPageB = isOriginal ? (Page)new PageBAlternative() : new PageB();
```

The next two statements remove the `existingPageB` object from the navigation stack and then insert the `newPageB` object in the slot before `pageC`, effectively swapping the pages:

```
// Swap the pages.
Navigation.RemovePage(existingPageB);
Navigation.InsertPageBefore(newPageB, pageC);
```

Obviously, the first time you click this button, `existingPageB` will be a `PageB` object and `newPageB` will be a `PageBAlternative` object, but you can then go back to `PageC` or `PageBAlternative`, and navigate forward again to `PageD`. Clicking the button again will replace the `PageBAlternative` object with a `PageB` object.

Dynamic page generation

The **BuildAPage** program is a multipage application, but the **BuildAPage** project contains only a single page class named `BuildAPageHomePage`. As the name suggests, the program constructs a new page from code and then navigates to it.

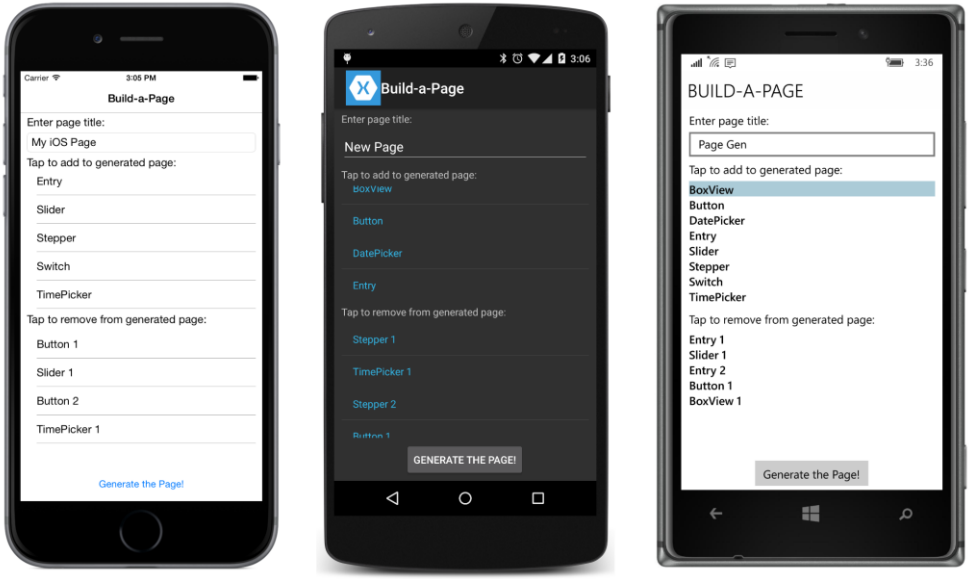
The XAML file lets you specify what you want on this constructed page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="BuildAPage.BuildAPageHomePage"
             Title="Build-a-Page"
             Padding="10, 5">

    <StackLayout>
        <Label Text="Enter page title:" />
```

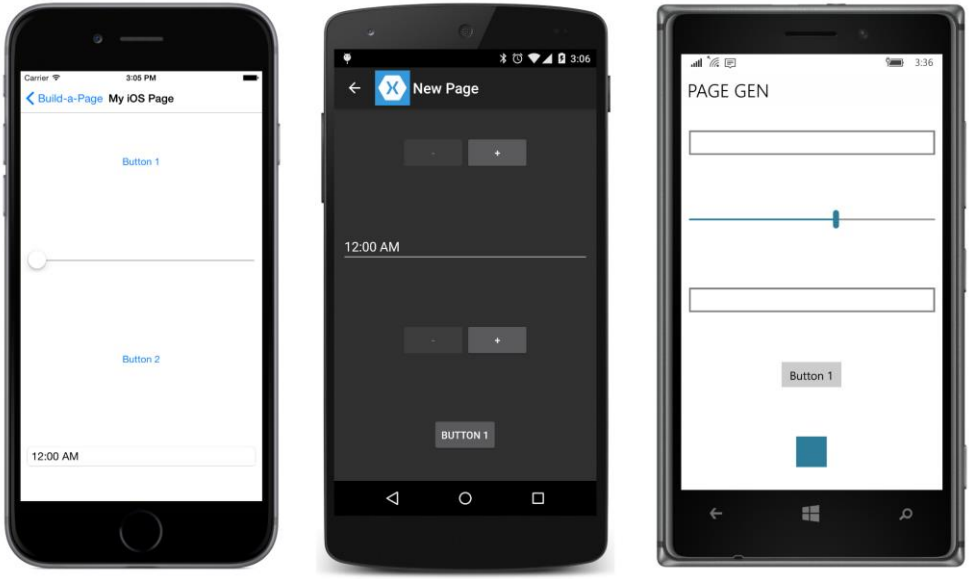
```
        <Entry x:Name="titleEntry"
               Placeholder="page title" />

        <Grid VerticalOptions="FillAndExpand">
            <ContentView Grid.Row="0">
                <StackLayout>
                <Label Text="Tap to add to generated page:" />
                    <ListView x:Name="viewList"
                              ItemSelected="OnViewListItemSelected">
                        <ListView.ItemsSource>
                            <x:Array Type="{x:Type x:String}">
                                <x:String>BoxView</x:String>
                                <x:String>Button</x:String>
                                <x:String>DatePicker</x:String>
                                <x:String>Entry</x:String>
                                <x:String>Slider</x:String>
                                <x:String>Stepper</x:String>
                                <x:String>Switch</x:String>
                                <x:String>TimePicker</x:String>
                            </x:Array>
                        </ListView.ItemsSource>
                    </ListView>
                </StackLayout>
            </ContentView>

            <ContentView Grid.Row="1">
                <StackLayout>
                    <Label Text="Tap to remove from generated page:" />
                    <ListView x:Name="pageList"
                              ItemSelected="OnPageListItemSelected" />
                </StackLayout>
            </ContentView>
        </Grid>

        <Button x:Name="generateButton"
                Text="Generate the Page!"
                IsEnabled="False"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="OnGenerateButtonClicked" />
    </StackLayout>
</ContentPage>
```

Use the `Entry` at the top of the page to specify a `Title` property for the constructed page. A `ListView` then lists eight common views that you might want on your page. As you select these views, they are transferred to the second `ListView`. If you want to delete one of them from the page, just tap it in this second `ListView`.

Here's how it might look after you've selected a few elements for the page. Notice that you can select multiple elements of the same type and each is given a unique number:



When you're all finished "designing" your page, simply tap the **Generate the Page!** button on the bottom, and the program builds that page and navigates to it:



The code-behind file has `ItemSelected` handlers for the two `ListView` elements to add items and remove items from the second `ListView`, but the more interesting processing occurs in the `Clicked` handler for the `Button`:

```
public partial class BuildAPageHomePage : ContentPage
```

```
{
    ObservableCollection<string> viewCollection = new ObservableCollection<string>();
    Assembly xamarinForms = typeof(Label).GetTypeInfo().Assembly;

    public BuildAPageHomePage()
    {
        InitializeComponent();

        pageList.ItemsSource = viewCollection;
    }

    void OnViewListItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            viewList.SelectedItem = null;
            int number = 1;
            string item = null;

            while (-1 != viewCollection.IndexOf(
                item = ((string)args.SelectedItem) + ' ' + number))
            {
                number++;
            }

            viewCollection.Add(item);
            generateButton.IsEnabled = true;
        }
    }

    void OnPageListItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            pageList.SelectedItem = null;
            viewCollection.Remove((string)args.SelectedItem);
            generateButton.IsEnabled = viewCollection.Count > 0;
        }
    }

    async void OnGenerateButtonClicked(object sender, EventArgs args)
    {
        ContentPage contentPage = new ContentPage
        {
            Title = titleEntry.Text,
            Padding = new Thickness(10, 0)
        };
        StackLayout stackLayout = new StackLayout();
        contentPage.Content = stackLayout;

        foreach (string item in viewCollection)
        {
            string viewString = item.Substring(0, item.IndexOf(' '));
            Type viewType = xamarinForms.GetType("Xamarin.Forms." + viewString);
```

```
            View view = (View)Activator.CreateInstance(viewType);
            view.VerticalOptions = LayoutOptions.CenterAndExpand;

            switch (viewString)
            {
                case "BoxView":
                    ((BoxView)view).Color = Color.Accent;
                    goto case "Stepper";

                case "Button":
                    ((Button)view).Text = item;
                    goto case "Stepper";

                case "Stepper":
                case "Switch":
                    view.HorizontalOptions = LayoutOptions.Center;
                    break;
            }
            stackLayout.Children.Add(view);
        }
        await Navigation.PushAsync(contentPage);
    }
}
```

This `Clicked` handler creates a `ContentPage` and a `StackLayout` and then simply loops through the strings in the second `ListView`, finding a corresponding `Type` object by using the `GetType` method defined by the `Assembly` class. (Notice the `Assembly` object named `xamarinForms` defined as a field.) A call to `Activator.CreateInstance` creates the actual element, which can then be tailored slightly for the final layout.

Creating a `ContentPage` object in code and adding elements to it isn't new. In fact, the standard Xamarin.Forms project template includes an `App` class with a constructor that instantiates `ContentPage` and adds a `Label` to it, so you were introduced to this technique way back in Chapter 2. But that approach was quickly abandoned in favor of the more flexible technique of deriving a class from `ContentPage`. Deriving a class is more powerful because the derived class has access to protected methods such as `OnAppearing` and `OnDisappearing`.

Sometimes, however, it's helpful to go back to basics.

Patterns of data transfer

It's often necessary for pages within a multipage application to share data, and particularly for one page to pass information to another page. Sometimes this process resembles a function call: When `HomePage` displays a list of items and navigates to `DetailPage` to display a detailed view of one of these items, `HomePage` must pass that particular item to `DetailPage`. Or when the user enters information into `FillOutFormPage`, that information must be returned back to the page that invoked `FillOutFormPage`.

Several techniques are available to transfer data between pages. Which one you use depends on the particular application. Keep in mind throughout this discussion that you'll probably also need to save the contents of the page when the application terminates, and restore the contents when the program starts up again. Some of the data-sharing techniques are more conducive to saving and restoring page state than others. This issue is explored in more detail later in this chapter.

Constructor arguments

When one page navigates to another page and needs to pass data to that page, one obvious way to pass that data is through the second page's constructor.

The **SchoolAndStudents** program illustrates this technique. The program makes use of the **SchoolOfFineArt** library introduced in Chapter 19, "Collection views." The program consists of two pages named `SchoolPage` and `StudentPage`. The `SchoolPage` class uses a `ListView` to display a scrollable list of all the students in the school. When the user selects one, the program navigates to a `StudentPage` that displays details about the individual student. The program is similar to the **SelectedStudentDetail** program in Chapter 19, except that the list and detail have been separated into two pages.

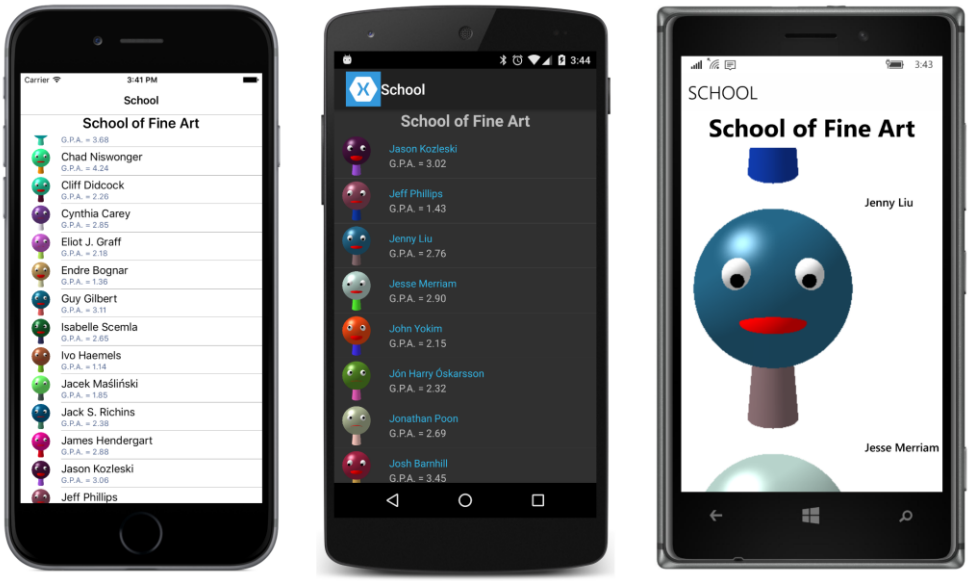
Here's `SchoolPage`. To keep things as simple as possible, the `ListView` uses an `ImageCell` to display each student in the school:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SchoolAndStudents.SchoolPage"
             Title="School">

    <StackLayout BindingContext="{Binding StudentBody}">
        <Label Text="{Binding School}"
               FontSize="Large"
               FontAttributes="Bold"
               HorizontalTextAlignment="Center" />

        <ListView x:Name="listView"
                  ItemsSource="{Binding Students}"
                  ItemSelected="OnListViewItemSelected">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ImageCell ImageSource="{Binding PhotoFilename}"
                               Text="{Binding FullName}"
                               Detail="{Binding GradePointAverage,
                                                StringFormat='G.P.A. = {0:F2}'}" />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

The data bindings in this XAML file assume that the `BindingContext` for the page is set to an object of type `SchoolViewModel` defined in the **SchoolOfFineArt** Library. The `SchoolViewModel` has a

property of type `StudentBody`, which is set to the `BindingContext` of the `StackLayout`. The `Label` is bound to the `School` property of `StudentBody`, and the `ItemsSource` of the `ListView` is bound to the `Students` collection property of `StudentBody`. This means that each item in the `ListView` has a `BindingContext` of type `Student`. The `ImageCell` references the `PhotoFilename`, `FullName`, and `GradePointAverage` properties of that `Student` object.

Here's that `ListView` running on iOS, Android, and Windows 10 Mobile:



The constructor in the code-behind file is responsible for setting the `BindingContext` of the page from an instance of `SchoolViewModel`. The code-behind file also contains a handler for the `ItemSelected` event of the `ListView`. This event is fired when the user taps one of the students:

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

        // Set BindingContext.
        BindingContext = new SchoolViewModel();
    }

    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        // The selected item is null or of type Student.
        Student student = args.SelectedItem as Student;

        // Make sure that an item is actually selected.
        if (student != null)
        {
```

```
            // Deselect the item.
            listView.SelectedItem = null;

            // Navigate to StudentPage with Student argument.
            await Navigation.PushAsync(new StudentPage(student));
        }
    }
}
```

The `SelectedItem` property of the event arguments is the `Student` object that is tapped, and the handler uses that as an argument to the `StudentPage` class in the `PushAsync` call.

Notice also that the handler sets the `SelectedItem` property of the `ListView` to `null`. This deselects the item so that it won't still be selected when the user returns to the `SchoolPage`, and the user can tap it again. But setting that `SelectedItem` property to `null` also causes another call to the `ItemSelected` event handler. Fortunately, the handler ignores the event if the `SelectedItem` is `null`.

The code-behind file for `StudentPage` simply uses that constructor argument to set the `BindingContext` of the page:

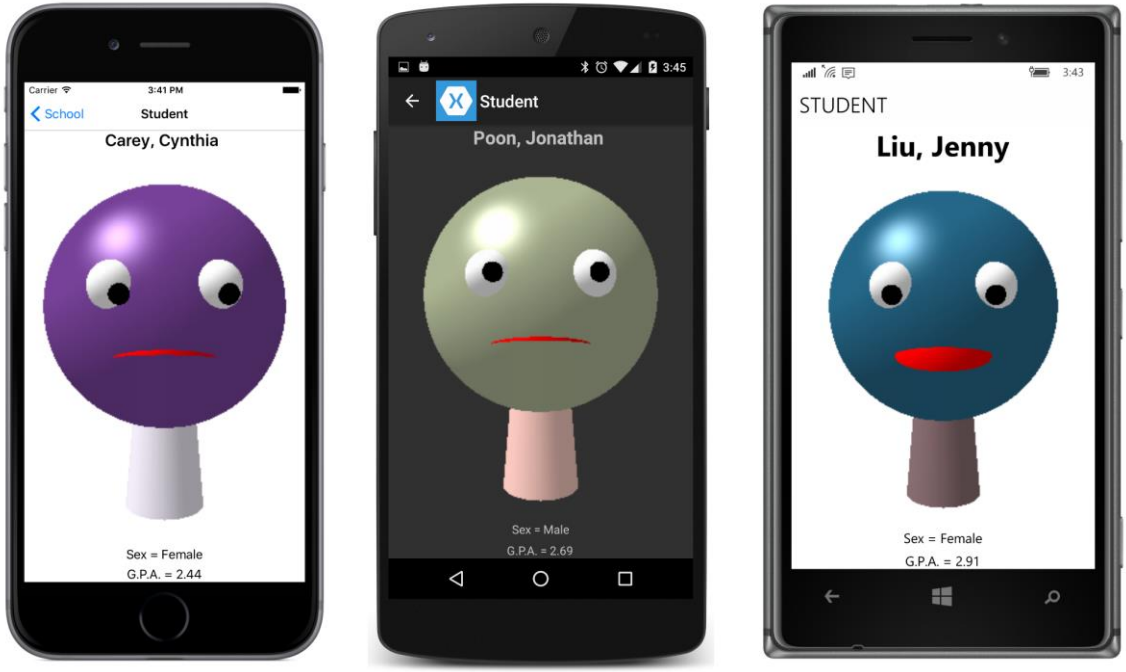
```
public partial class StudentPage : ContentPage
{
    public StudentPage(Student student)
    {
        InitializeComponent();
        BindingContext = student;
    }
}
```

The XAML file for the `StudentPage` class contains bindings to various properties of the `Student` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SchoolAndStudents.StudentPage"
             Title="Student">

    <StackLayout>
        <!-- Name -->
        <StackLayout Orientation="Horizontal"
                     HorizontalOptions="Center"
                     Spacing="0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style TargetType="Label">
                        <Setter Property="FontSize" Value="Large" />
                        <Setter Property="FontAttributes" Value="Bold" />
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>

            <Label Text="{Binding LastName}" />
```

```
            <Label Text="{Binding FirstName, StringFormat=', {0}'}" />
            <Label Text="{Binding MiddleName, StringFormat=' {0}'}" />
        </StackLayout>

        <!-- Photo -->
        <Image Source="{Binding PhotoFilename}"
               VerticalOptions="FillAndExpand" />

        <!-- Sex -->
        <Label Text="{Binding Sex, StringFormat='Sex = {0}'}"
               HorizontalOptions="Center" />

        <!-- GPA -->
        <Label Text="{Binding GradePointAverage, StringFormat='G.P.A. = {0:F2}'}"
               HorizontalOptions="Center" />
    </StackLayout>
</ContentPage>
```

The XAML file doesn't require a button or any other user-interface object to return back to `SchoolPage` because that's provided automatically, either as part of the standard navigation user interface for the platform or as part of the phone itself:



Passing information to the navigated page through the constructor is versatile, but for this particular example, it's unnecessary. `StudentPage` could have a parameterless constructor, and the `SchoolPage` could set the `BindingContext` of the newly created `StudentPage` right in the `PushAsync` call:

```
await Navigation.PushAsync(new StudentPage { BindingContext = student });
```

One of the problems with either approach is preserving the application state when the program is suspended. If you want `StudentPage` to save the current student when the program terminates, it needs to save all the properties of the `Student` object. But when that `Student` object is re-created when the program starts up again, it's a different object from the particular `Student` object for the same student in the `Students` collection even though all the properties are the same.

If the `Students` collection is known to be constant, it makes more sense for `StudentPage` to save only an index into the `Students` collection that references this particular `Student` object. But in this example, `StudentPage` does not have access to that index or to the `Students` collection.

Properties and method calls

A page calling `PushAsync` or `PushModalAsync` obviously has direct access to the class that it's navigating to, so it can set properties or call methods in that page object to pass information to it. A page calling `PopAsync` or `PopModalAsync`, however, has some more work to do to determine the page that it's returning to. In the general case, a page can't always be expected to be familiar with the page type of the page that navigated to it.

You'll need to exercise caution when setting properties or calling methods from one page to another. You can't make any assumptions about the sequence of calls to the `OnAppearing` and `OnDisappearing` overrides and the completion of the `PushAsync`, `PopAsync`, `PushModalAsync`, and `PopModalAsync` tasks.

Let's assume you have pages named `HomePage` and `InfoPage`. As the names suggest, `HomePage` uses `PushAsync` to navigate to `InfoPage` to obtain some information from the user, and somehow `InfoPage` must transfer that information to `HomePage`.

Here are some ways that `HomePage` and `InfoPage` can interact (or not interact):

`HomePage` can access a property in `InfoPage` or call a method in `InfoPage` after instantiating `InfoPage` or after the `PushAsync` task completes. This is straightforward, and you already saw an example in the **SinglePageNavigation** program.

`InfoPage` can access a property in `HomePage` or call a method in `HomePage` at any time during its existence. Most conveniently, `InfoPage` can perform these operations during its `OnAppearing` override (for initialization) or the `OnDisappearing` override (to prepare final values). For the duration of its existence, `InfoPage` can obtain the `HomePage` instance from the `NavigationStack` collection. However, depending on the order of the `OnAppearing` and `OnDisappearing` calls relative to the completion of the `PushAsync` or `PopAsync` tasks, `HomePage` might be the last item in the `NavigationStack`, or `InfoPage` might be the last item in the `NavigationStack`, in which case `HomePage` is the next to last item.

`HomePage` can be informed that `InfoPage` has returned control back to `HomePage` by overriding its `OnAppearing` method. (But keep in mind that this method is not called on Android devices when a

modal page has returned back to the page that invoked it.) However, during the `OnAppearing` override of `HomePage`, `HomePage` cannot be entirely certain that the instance of `InfoPage` is still in the `NavigationStack` collection, or even that it exists at all. `HomePage` can save the instance of `InfoPage` when it navigates to `InfoPage`, but that creates problems if the application needs to save the page state when it terminates.

Let's examine a program named **DataTransfer1** that uses a second page to obtain information from the user and then adds that information as an item to a `ListView`. The user can add multiple items to the `ListView` or edit an existing item by tapping it. To focus entirely on the mechanism of interpage communication, the program uses no data bindings, and the class that stores the information does not implement `INotifyPropertyChanged`:

```
public class Information
{
    public string Name { set; get; }

    public string Email { set; get; }

    public string Language { set; get; }

    public DateTime Date { set; get; }

    public override string ToString()
    {
        return String.Format("{0} / {1} / {2} / {3:d}",
                             String.IsNullOrWhiteSpace(Name) ? "???" : Name,
                             String.IsNullOrWhiteSpace(Email) ? "???" : Email,
                             String.IsNullOrWhiteSpace(Language) ? "???" : Language,
                             Date);
    }
}
```

The `ToString` method allows the `ListView` to display the items with minimum fuss.

The **DataTransfer1** program has two pages, named `DataTransfer1HomePage` and `DataTransfer1InfoPage`, that communicate to each other by calling public methods. The `DataTransfer1HomePage` has a XAML file with a `Button` for invoking a page to obtain information and a `ListView` for displaying each item and allowing an item to be edited:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataTransfer1.DataTransfer1HomePage"
             Title="Home Page">
    <Grid>
        <Button Text="Add New Item"
                Grid.Row="0"
                FontSize="Large"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="OnGetInfoButtonClicked" />
```

```
        <ListView x:Name="listView"
                  Grid.Row="1"
                  ItemSelected="OnListViewItemSelected" />
    </Grid>
</ContentPage>
```

Let's bounce back and forth between the two classes to examine the transfer of data. Here's the portion of the code-behind file showing the initialization of the `ListView` with an `ObservableCollection` so that the `ListView` updates its display whenever the collection changes:

```
public partial class DataTransfer1HomePage : ContentPage
{
    ObservableCollection<Information> list = new ObservableCollection<Information>();

    public DataTransfer1HomePage()
    {
        InitializeComponent();

        // Set collection to ListView.
        listView.ItemsSource = list;
    }

    // Button Clicked handler.
    async void OnGetInfoButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PushAsync(new DataTransfer1InfoPage());
    }
    …
}
```

This code also implements the `Clicked` handler for the `Button` simply by instantiating the `DataTransfer1InfoPage` and navigating to it.

The XAML file of `DataTransfer1InfoPage` has two `Entry` elements, a `Picker`, and a `DatePicker` corresponding to the properties of `Information`. This page relies on each platform's standard user interface for returning to the home page:

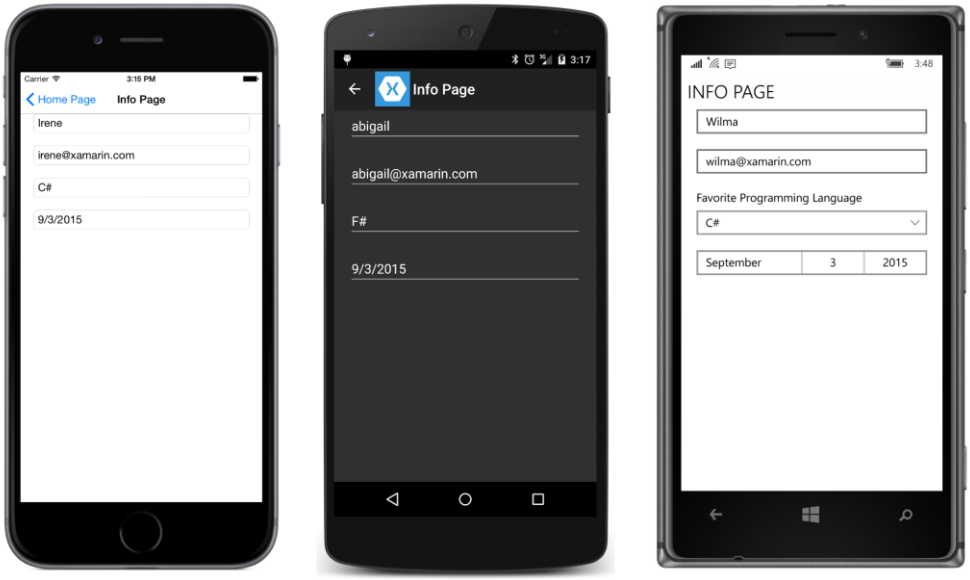
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataTransfer1.DataTransfer1InfoPage"
             Title="Info Page">

    <StackLayout Padding="20, 0"
                 Spacing="20">
        <Entry x:Name="nameEntry"
               Placeholder="Enter Name" />

        <Entry x:Name="emailEntry"
               Placeholder="Enter Email Address" />

        <Picker x:Name="languagePicker"
                Title="Favorite Programming Language">
            <Picker.Items>
```

```
                <x:String>C#</x:String>
                <x:String>F#</x:String>
                <x:String>Objective C</x:String>
                <x:String>Swift</x:String>
                <x:String>Java</x:String>
            </Picker.Items>
        </Picker>

        <DatePicker x:Name="datePicker" />
    </StackLayout>
</ContentPage>
```

The code-behind file of the info page instantiates an `Information` object that is associated with this page instance:

```
public partial class DataTransfer1InfoPage : ContentPage
{
    // Instantiate an Information object for this page instance.
    Information info = new Information();

    public DataTransfer1InfoPage()
    {
        InitializeComponent();
    }
    …
}
```

The user interacts with the elements on the page by entering some information:



Nothing else happens in the class until `DataTransfer1InfoPage` gets a call to its `OnDisappearing` override. This usually indicates that the user has pressed the **Back** button that is either part of the

navigation bar (on iOS and Android) or below the screen (on Android and Windows Phone).

However, you might be aware that in a platform no longer supported by Xamarin.Forms (Windows Phone Silverlight), `OnDisappearing` was called when the user invoked the `Picker` or `DatePicker`, and you might be nervous about it being called in other circumstances on the current platforms. This implies that nothing should be done in the `OnDisappearing` override that can't be undone when `OnDisappearing` is called as part of the normal navigation back to the home page. This is why `DataTransfer1InfoPage` instantiates its `Information` object when the page is first created and not during the `OnDisappearing` override.

The `OnDisappearing` override sets the properties of the `Information` object from the four views and then obtains the instance of `DataTransfer1HomePage` that invoked it from the `NavigationStack` collection. It then calls a method named `InformationReady` in that home page:

```
public partial class DataTransfer1InfoPage : ContentPage
{
    …
    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        // Set properties of Information object.
        info.Name = nameEntry.Text;
        info.Email = emailEntry.Text;

        int index = languagePicker.SelectedIndex;
        info.Language = index == -1 ? null : languagePicker.Items[index];

        info.Date = datePicker.Date;

        // Get the DataTransfer1HomePage that invoked this page.
        NavigationPage navPage = (NavigationPage)Application.Current.MainPage;
        IReadOnlyList<Page> navStack = navPage.Navigation.NavigationStack;
        int lastIndex = navStack.Count - 1;
        DataTransfer1HomePage homePage = navStack[lastIndex] as DataTransfer1HomePage;

        if (homePage == null)
        {
            homePage = navStack[lastIndex - 1] as DataTransfer1HomePage;
        }
        // Transfer Information object to DataTransfer1HomePage.
        homePage.InformationReady(info);
    }
}
```

The `InformationReady` method in `DataTransfer1HomePage` checks whether the `Information` object is already in the `ObservableCollection` set to the `ListView`, and if so, it replaces it. Otherwise, it adds the object to that collection:

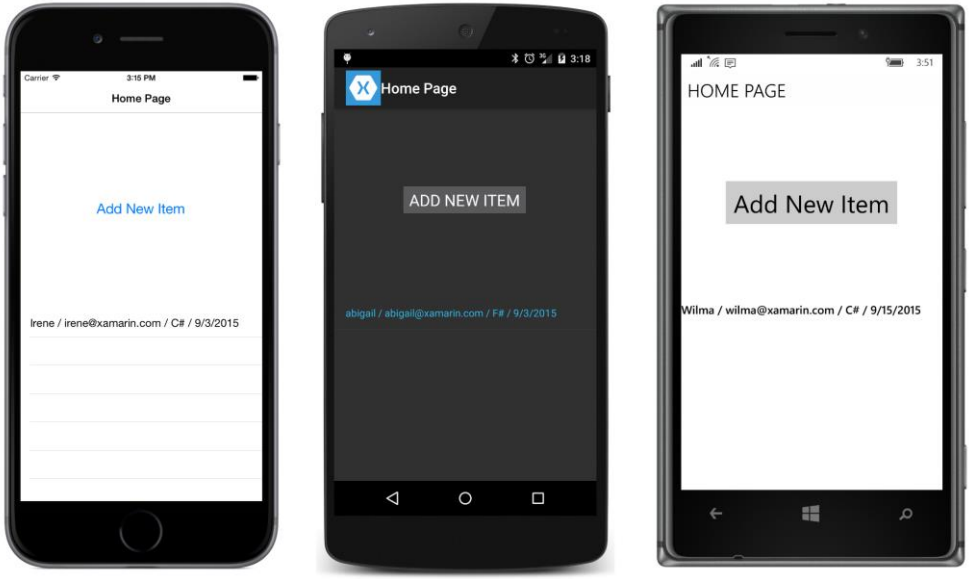
```
public partial class DataTransfer1HomePage : ContentPage
{
    …
```

```
    // Called from InfoPage.
    public void InformationReady(Information info)
    {
        // If the object has already been added, replace it.
        int index = list.IndexOf(info);

        if (index != -1)
        {
            list[index] = info;
        }
        // Otherwise, add it.
        else
        {
            list.Add(info);
        }
    }
}
```

There are two reasons for checking whether the `Information` object is already in the `ListView` collection. It might be there already if the info page received an earlier call to its `OnDisappearing` override, which then results in a call to `InformationReady` in the home page. Also—as you'll see—existing items in the `ListView` can be edited.

The code that replaces the `Information` object with itself in the `ObservableCollection` might seem superfluous. However, the act of replacing the item causes the `ObservableCollection` to fire a `CollectionChanged` event, and the `ListView` redraws itself. Another solution would be for `Information` to implement `INotifyPropertyChanged`, in which case the change in the values of a property would cause the `ListView` to update the display of that item.

At this point, we're back on the home page, and the `ListView` displays the newly added item:



You can now tap the `Button` again to create a new item, or you can tap an existing item in the `ListView`. The `ItemSelected` handler for the `ListView` also navigates to `DataTransfer1InfoPage`:

```
public partial class DataTransfer1HomePage : ContentPage
{
    …
    // ListView ItemSelected handler.
    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            listView.SelectedItem = null;

            DataTransfer1InfoPage infoPage = new DataTransfer1InfoPage();
            await Navigation.PushAsync(infoPage);
            infoPage.InitializeInfo((Information)args.SelectedItem);
        }
    }
    …
}
```

However, after the `PushAsync` task completes, the handler calls a method in `DataTransfer1InfoPage` named `InitializeInfo` with the selected item.

The `InitializeInfo` method in `DataTransfer1InfoPage` replaces the `Information` object it originally created as a field with this existing instance and initializes the views on the page with the properties of the object:

```
public partial class DataTransfer1InfoPage : ContentPage
```

```
{
    …
    public void InitializeInfo(Information info)
    {
        // Replace the instance.
        this.info = info;

        // Initialize the views.
        nameEntry.Text = info.Name ?? "";
        emailEntry.Text = info.Email ?? "";

        if (!String.IsNullOrWhiteSpace(info.Language))
        {
            languagePicker.SelectedIndex = languagePicker.Items.IndexOf(info.Language);
        }
        datePicker.Date = info.Date;
    }
    …
}
```

Now the user is editing an existing item instead of a new instance.

Generally, a program that allows editing of existing items will also give the user an opportunity to abandon any changes already made to the item. To allow that, the `DataTransfer1InfoPage` would need to differentiate between returning back to the home page with changes and cancelling the edit operation. At least one `Button` or `ToolbarItem` is required, and that should probably be a **Cancel** button so that the standard **Back** button saves the changes.

Such a program should also have a facility to delete items. Later on in this chapter, you'll see such a program.

The messaging center

You might not like the idea of the two page classes making method calls directly to each other. It seems to work well for a small sample, but for a larger program with lots of interclass communication, you might prefer something a little more flexible that doesn't require actual page instances.

Such a facility is the Xamarin.Forms `MessagingCenter` class. This is a static class with three methods, named `Subscribe`, `Unsubscribe`, and `Send`. Messages are identified with a text string and can be accompanied by any object. The `Send` method broadcasts a message that is received by any subscriber to that message.

The **DataTransfer2** program has the same `Information` class and the same XAML files as **DataTransfer1,** but it uses the `MessagingCenter` class rather than direct method calls.

The constructor of the home page subscribes to a message identified by the text string "InformationReady." The generic arguments to `Subscribe` indicate what object type sends this message—an object of type `DataTransfer2InfoPage`—and the type of the data, which is `Information`. The `Subscribe` method arguments indicate the object receiving the message (`this`), the message name,

and a lambda function. The body of this lambda function is the same as the body of the `InformationReady` method in the previous program:

```
public partial class DataTransfer2HomePage : ContentPage
{
    ObservableCollection<Information> list = new ObservableCollection<Information>();

    public DataTransfer2HomePage()
    {
        InitializeComponent();

        // Set collection to ListView.
        listView.ItemsSource = list;

        // Subscribe to "InformationReady" message.
        MessagingCenter.Subscribe<DataTransfer2InfoPage, Information>
            (this, "InformationReady", (sender, info) =>
            {
                // If the object has already been added, replace it.
                int index = list.IndexOf(info);

                if (index != -1)
                {
                    list[index] = info;
                }
                // Otherwise, add it.
                else
                {
                    list.Add(info);
                }
            });
    }

    // Button Clicked handler.
    async void OnGetInfoButtonClicked(object sender, EventArgs args)
    {
        await Navigation.PushAsync(new DataTransfer2InfoPage());
    }

    // ListView ItemSelected handler.
    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            listView.SelectedItem = null;

            DataTransfer2InfoPage infoPage = new DataTransfer2InfoPage();
            await Navigation.PushAsync(infoPage);

            // Send "InitializeInfo" message to info page.
            MessagingCenter.Send<DataTransfer2HomePage, Information>
                (this, "InitializeInfo", (Information)args.SelectedItem);
        }
```

```
    }
}
```

The `ItemSelected` handler of the `ListView` contains a call to `MessagingCenter.Send`. The generic arguments indicate the type of the message sender and the type of the data. The arguments to the method indicate the object sending the message, the message name, and the data, which is the `SelectedItem` of the `ListView`.

The `DataTransfer2InfoPage` code-behind file contains complementary calls to `MessagingCenter.Subscribe` and `MessageCenter.Send`. The info page constructor subscribes to the "InitializeInfo" message; the body of the lambda function is the same as the `InitializeInfo` method in the previous program except that it ends with a call to unsubscribe from the message. Unsubscribing ensures that there is no longer a reference to the info page object and allows the info page object to be garbage collected. Strictly speaking, however, unsubscribing shouldn't be necessary because the `MessagingCenter` maintains `WeakReference` objects for subscribers:

```
public partial class DataTransfer2InfoPage : ContentPage
{
    // Instantiate an Information object for this page instance.
    Information info = new Information();

    public DataTransfer2InfoPage()
    {
        InitializeComponent();

        // Subscribe to "InitializeInfo" message.
        MessagingCenter.Subscribe<DataTransfer2HomePage, Information>
            (this, "InitializeInfo", (sender, info) =>
            {
                // Replace the instance.
                this.info = info;

                // Initialize the views.
                nameEntry.Text = info.Name ?? "";
                emailEntry.Text = info.Email ?? "";

                if (!String.IsNullOrWhiteSpace(info.Language))
                {
                    languagePicker.SelectedIndex = languagePicker.Items.IndexOf(info.Language);
                }
                datePicker.Date = info.Date;

                // Don't need "InitializeInfo" any more so unsubscribe.
                MessagingCenter.Unsubscribe<DataTransfer2HomePage, Information>
                    (this, "InitializeInfo");
            });
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();
```

```
        // Set properties of Information object.
        info.Name = nameEntry.Text;
        info.Email = emailEntry.Text;

        int index = languagePicker.SelectedIndex;
        info.Language = index == -1 ? null : languagePicker.Items[index];

        info.Date = datePicker.Date;

        // Send "InformationReady" message back to home page.
        MessagingCenter.Send<DataTransfer2InfoPage, Information>
            (this, "InformationReady", info);
    }
}
```

The `OnDisappearing` override is considerably shorter than the version in the previous program. To call a method in the home page, the previous program had to go into the `NavigationStack` collection. In this version, all that's necessary is to use `MessagingCenter.Send` to send an "Information-Ready" message to whoever has subscribed to it, and that happens to be the home page.

Events

In both the method-call approach and the messaging-center approach to interclass communication, the info page needs to know the type of the home page. This is sometimes undesirable if the same info page can be called from different types of pages.

One solution to this problem is for the info class to implement an event, and that's the approach taken in **DataTransfer3**. The `Information` class and XAML files are the same as the previous programs, but `DataTransfer3InfoPage` now implements a public event named `InformationReady`:

```
public partial class DataTransfer3InfoPage : ContentPage
{
    // Define a public event for transferring data.
    public EventHandler<Information> InformationReady;

    // Instantiate an Information object for this page instance.
    Information info = new Information();

    public DataTransfer3InfoPage()
    {
        InitializeComponent();
    }

    public void InitializeInfo(Information info)
    {
        // Replace the instance.
        this.info = info;

        // Initialize the views.
        nameEntry.Text = info.Name ?? "";
        emailEntry.Text = info.Email ?? "";
```

```
        if (!String.IsNullOrWhiteSpace(info.Language))
        {
            languagePicker.SelectedIndex = languagePicker.Items.IndexOf(info.Language);
        }
        datePicker.Date = info.Date;
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        // Set properties of Information object.
        info.Name = nameEntry.Text;
        info.Email = emailEntry.Text;

        int index = languagePicker.SelectedIndex;
        info.Language = index == -1 ? null : languagePicker.Items[index];

        info.Date = datePicker.Date;

        // Raise the InformationReady event.
        EventHandler<Information> handler = InformationReady;

        if (handler != null)
            handler(this, info);
    }
}
```

During the `OnDisappearing` override, the class sets the `Information` properties from the elements and raises an `InformationReady` event with the `Information` object.

The home page can set a handler for the `InformationReady` event either after it instantiates the info page or after it navigates to the page. The event handler adds the `Information` object to the `ListView` or replaces an existing item:

```
public partial class DataTransfer3HomePage : ContentPage
{
    ObservableCollection<Information> list = new ObservableCollection<Information>();

    public DataTransfer3HomePage()
    {
        InitializeComponent();

        // Set collection to ListView.
        listView.ItemsSource = list;
    }

    // Button Clicked handler.
    async void OnGetInfoButtonClicked(object sender, EventArgs args)
    {
        DataTransfer3InfoPage infoPage = new DataTransfer3InfoPage();
        await Navigation.PushAsync(infoPage);

        // Set event handler for obtaining information.
```

```
        infoPage.InformationReady += OnInfoPageInformationReady;
    }

    // ListView ItemSelected handler.
    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            listView.SelectedItem = null;

            DataTransfer3InfoPage infoPage = new DataTransfer3InfoPage();
            await Navigation.PushAsync(infoPage);
            infoPage.InitializeInfo((Information)args.SelectedItem);

            // Set event handler for obtaining information.
            infoPage.InformationReady += OnInfoPageInformationReady;
        }
    }

    void OnInfoPageInformationReady(object sender, Information info)
    {
        // If the object has already been added, replace it.
        int index = list.IndexOf(info);

        if (index != -1)
        {
            list[index] = info;
        }
        // Otherwise, add it.
        else
        {
            list.Add(info);
        }
    }
}
```

There are a couple of problems with this approach. The first problem is that there is no convenient place to detach the event handler. The info page raises the event in its `OnDisappearing` override. If you are not confident that `OnDisappearing` is called only when navigation is occurring, then the home page can't detach the event hander in the handler itself.

Nor can the home page detach the event handler in its `OnAppearing` override because when the info page returns back to the home page, the order in which the `OnAppearing` and `OnDisappearing` overrides are called is platform dependent.

If the home page can't detach the handler from the info page, then each instance of info page will continue to maintain a reference to the home page and can't be garbage collected.

The event-handler approach is also not good when an application needs to save and restore page state. The info page cannot save the state of the event to restore it when the program executes again.

The App class intermediary

In a Xamarin.Forms application, the first code that executes in the common code project is the constructor of a class customarily named `App` that derives from `Application`. This `App` object remains constant until the program terminates, and it is always available to any code in the program through the static `Application.Current` property. The return value of that property is of type `Application`, but it's simple to cast it to `App`.

This implies that the `App` class is a great place to store data that must be accessed throughout the application, including data that is transferred from one page to another.

The `Information` class in the **DataTransfer4** version of the program is just a little different from the version you've seen previously:

```
public class Information
{
    public Information()
    {
        Date = DateTime.Today;
    }

    public string Name { set; get; }

    public string Email { set; get; }

    public string Language { set; get; }

    public DateTime Date { set; get; }

    public override string ToString()
    {
        return String.Format("{0} / {1} / {2} / {3:d}",
                             String.IsNullOrWhiteSpace(Name) ? "???" : Name,
                             String.IsNullOrWhiteSpace(Email) ? "???" : Email,
                             String.IsNullOrWhiteSpace(Language) ? "???" : Language,
                             Date);
    }
}
```

The constructor of this version sets the `Date` property to today's date. In previous versions of the program, the properties of an `Information` instance are set from the various elements on the info page. In that case, the `Date` property is set from the `DatePicker`, which by default sets its `Date` property to the current date. In **DataTransfer4**, as you'll see, the elements on the info page are initialized from the properties in the `Information` object, so setting the `Date` property in the `Information` class merely keeps the functionality of the programs consistent.

Here's the `App` class in **DataTransfer4**. Notice the public properties named `InfoCollection` and `CurrentInfoItem`. The constructor initializes `InfoCollection` to an `ObservableCollection<Information>` object before creating `DataTransfer4HomePage`:

```
public class App : Application
```

```
{
    public App()
    {
        // Create the ObservableCollection for the Information items.
        InfoCollection = new ObservableCollection<Information>();

        MainPage = new NavigationPage(new DataTransfer4HomePage());
    }

    public IList<Information> InfoCollection { private set; get; }

    public Information CurrentInfoItem { set; get; }
    …
}
```

The availability of the `InfoCollection` property in `App` allows `DataTransfer4HomePage` to set it directly to the `ItemsSource` property of the `ListView`:

```
public partial class DataTransfer4HomePage : ContentPage
{
    App app = (App)Application.Current;

    public DataTransfer4HomePage()
    {
        InitializeComponent();

        // Set collection to ListView.
        listView.ItemsSource = app.InfoCollection;
    }

    // Button Clicked handler.
    async void OnGetInfoButtonClicked(object sender, EventArgs args)
    {
        // Create new Information item.
        app.CurrentInfoItem = new Information();

        // Navigate to info page.
        await Navigation.PushAsync(new DataTransfer4InfoPage());
    }

    // ListView ItemSelected handler.
    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            listView.SelectedItem = null;

            // Get existing Information item.
            app.CurrentInfoItem = (Information)args.SelectedItem;

            // Navigate to info page.
            await Navigation.PushAsync(new DataTransfer4InfoPage());
        }
```

```
    }
}
```

Notice the two different but similar ways that the `Clicked` handler for the `Button` and the `ItemSelected` handler for the `ListView` are implemented. Before navigating to `DataTransfer4InfoPage`, both handlers set the `CurrentInfoItem` property of the `App` class to an instance of `Information`. But the `Clicked` handler sets the `CurrentInfoItem` property to a new instance, whereas the `ItemSelected` handler sets it to the selected item in the `ListView`.

Everything else is handled by `DataTransfer4InfoPage`. The info page can initialize the elements on the page from the `Information` object stored in the `CurrentInfoItem` property of the `App` class:

```
public partial class DataTransfer4InfoPage : ContentPage
{
    App app = (App)Application.Current;

    public DataTransfer4InfoPage()
    {
        InitializeComponent();

        // Initialize the views.
        Information info = app.CurrentInfoItem;

        nameEntry.Text = info.Name ?? "";
        emailEntry.Text = info.Email ?? "";

        if (!String.IsNullOrWhiteSpace(info.Language))
        {
            languagePicker.SelectedIndex = languagePicker.Items.IndexOf(info.Language);
        }
        datePicker.Date = info.Date;
    }

    protected override void OnDisappearing()
    {
        base.OnDisappearing();

        // Set properties of Information object.
        Information info = app.CurrentInfoItem;

        info.Name = nameEntry.Text;
        info.Email = emailEntry.Text;

        int index = languagePicker.SelectedIndex;
        info.Language = index == -1 ? null : languagePicker.Items[index];

        info.Date = datePicker.Date;

        // If the object has already been added to the collection, replace it.
        IList<Information> list = app.InfoCollection;

        index = list.IndexOf(info);
```

```
        if (index != -1)
        {
            list[index] = info;
        }
        // Otherwise, add it.
        else
        {
            list.Add(info);
        }
    }
}
```

The info page still needs to override its `OnDisappearing` method to set the properties of the `Information` object and possibly add it to the `ListView` collection or replace the same object to trigger a redraw. But the info page doesn't need to directly access the `ListView` because it can obtain the `ObservableCollection` from the `InfoCollection` property of the `App` class.

Moreover, if you need to save and restore page state, everything is available right in the `App` class.

Let's see how that might work.

Switching to a ViewModel

At this point it should be obvious that the `Information` class should really implement `INotifyPropertyChanged`. In **DataTransfer5**, the `Information` class has become an `InformationViewModel` class. It derives from `ViewModelBase` in the **Xamarin.FormsBook.Toolkit** library to reduce the overhead:

```
public class InformationViewModel : ViewModelBase
{
    string name, email, language;
    DateTime date = DateTime.Today;

    public string Name
    {
        set { SetProperty(ref name, value); }
        get { return name; }
    }

    public string Email
    {
        set { SetProperty(ref email, value); }
        get { return email; }
    }

    public string Language
    {
        set { SetProperty(ref language, value); }
        get { return language; }
    }

    public DateTime Date
```

```
    {
        set { SetProperty(ref date, value); }
        get { return date; }
    }
}
```

A new class has been added to **DataTransfer5** called `AppData`. This class includes an `ObservableCollection` of `Information` objects for the `ListView` as well as a separate `Information` instance for the info page:

```
public class AppData
{
    public AppData()
    {
        InfoCollection = new ObservableCollection<InformationViewModel>();
    }

    public IList<InformationViewModel> InfoCollection { private set; get; }

    public InformationViewModel CurrentInfo { set; get; }
}
```

The `App` class instantiates `AppData` before instantiating the home page and makes it available as a public property:

```
public class App : Application
{
    public App()
    {
        // Ensure link to Toolkit library.
        new Xamarin.FormsBook.Toolkit.ObjectToIndexConverter<object>();

        // Instantiate AppData and set property.
        AppData = new AppData();

        // Go to the home page.
        MainPage = new NavigationPage(new DataTransfer5HomePage());
    }

    public AppData AppData { private set; get; }
    …
}
```

The XAML file of `DataTransfer5HomePage` sets the `BindingContext` for the page with a binding that incorporates the static `Application.Current` property (which returns the `App` object) and the `AppData` instance. This means that the `ListView` can bind its `ItemsSource` property to the `InfoCollection` property of `AppData`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataTransfer5.DataTransfer5HomePage"
             Title="Home Page"
             BindingContext="{Binding Source={x:Static Application.Current},
```

```
                                    Path=AppData}">
    <Grid>
        <Button Text="Add New Item"
                Grid.Row="0"
                FontSize="Large"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                Clicked="OnGetInfoButtonClicked" />

        <ListView x:Name="listView"
                  Grid.Row="1"
                  ItemsSource="{Binding InfoCollection}"
                  ItemSelected="OnListViewItemSelected">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <Label Text="{Binding Name}" />
                            <Label Text=" / " />
                            <Label Text="{Binding Email}" />
                            <Label Text=" / " />
                            <Label Text="{Binding Language}" />
                            <Label Text=" / " />
                            <Label Text="{Binding Date, StringFormat='{0:d}'}" />
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </Grid>
</ContentPage>
```

Previous versions of the program relied upon the `ToString` override in `Information` to display the items. Now that `Information` has been replaced with `InformationViewModel`, the `ToString` method isn't adequate because there's no notification that the `ToString` method might return something different. Instead, the `ListView` uses a `ViewCell` containing elements with bindings to properties of `InformationViewModel`.

The code-behind file continues to implement the `Clicked` handler for the `Button` and the `ItemSelected` handler for the `ListView`, but they are now so similar they can make use of a common method named `GoToInfoPage`:

```
public partial class DataTransfer5HomePage : ContentPage
{
    public DataTransfer5HomePage()
    {
        InitializeComponent();
    }

    // Button Clicked handler.
    void OnGetInfoButtonClicked(object sender, EventArgs args)
    {
        // Navigate to the info page.
```

```
        GoToInfoPage(new InformationViewModel(), true);
    }

    // ListView ItemSelected handler.
    void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            listView.SelectedItem = null;

            // Navigate to the info page.
            GoToInfoPage((InformationViewModel)args.SelectedItem, false);
        }
    }

    async void GoToInfoPage(InformationViewModel info, bool isNewItem)
    {
        // Get AppData object (set to BindingContext in XAML file).
        AppData appData = (AppData)BindingContext;

        // Set info item to CurrentInfo property of AppData.
        appData.CurrentInfo = info;

        // Navigate to the info page.
        await Navigation.PushAsync(new DataTransfer5InfoPage());

        // Add new info item to the collection.
        if (isNewItem)
        {
            appData.InfoCollection.Add(info);
        }
    }
}
```

For both cases, the `GoToInfoPage` method sets the `CurrentInfo` property of `AppData`. For a `Clicked` event, it's set to a new `InformationViewModel` object. For the `ItemSelected` event, it's set to an existing `InformationViewModel` from the `ListView` collection. The `isNewItem` parameter of the `GoToInfoPage` method indicates whether this `InformationViewModel` object should also be added to the `InfoCollection` of `AppData`.

Notice that the new item is added to the `InfoCollection` after the `PushAsync` task completes. If the item is added prior to the `PushAsync` call, then—depending on the platform—you might notice this new item suddenly appearing in the `ListView` immediately before the page transition. That could be a bit disturbing!

The XAML file for the `DataTransfer5InfoPage` sets the `BindingContext` for the page to the `CurrentInfo` property of `AppData`. (The home page sets the `CurrentInfo` property of `AppData` prior to instantiating the info page, so it's not necessary for `AppData` to implement `INotifyPropertyChanged`.) The setting of the `BindingContext` allows all the visual elements on the page to be bound to properties in the `InformationViewModel` class:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit=
                "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="DataTransfer5.DataTransfer5InfoPage"
             Title="Info Page"
             BindingContext="{Binding Source={x:Static Application.Current},
                                      Path=AppData.CurrentInfo}">

    <StackLayout Padding="20, 0"
                 Spacing="20">
        <Entry Text="{Binding Name}"
               Placeholder="Enter Name" />

        <Entry Text="{Binding Email}"
               Placeholder="Enter Email Address" />

        <Picker x:Name="languagePicker"
                Title="Favorite Programming Language">
            <Picker.Items>
                <x:String>C#</x:String>
                <x:String>F#</x:String>
                <x:String>Objective C</x:String>
                <x:String>Swift</x:String>
                <x:String>Java</x:String>
            </Picker.Items>

            <Picker.SelectedIndex>
                <Binding Path="Language">
                    <Binding.Converter>
                        <toolkit:ObjectToIndexConverter x:TypeArguments="x:String">
                            <x:String>C#</x:String>
                            <x:String>F#</x:String>
                            <x:String>Objective C</x:String>
                            <x:String>Swift</x:String>
                            <x:String>Java</x:String>
                        </toolkit:ObjectToIndexConverter>
                    </Binding.Converter>
                </Binding>
            </Picker.SelectedIndex>
        </Picker>

        <DatePicker Date="{Binding Date}" />
    </StackLayout>
</ContentPage>
```

Notice the use of the `ObjectToIndexConverter` in the binding between the `SelectedIndex` property of the `Picker` and the string `Language` property of `InformationViewModel`. This binding converter was introduced in Chapter 19, "Collection views," in the section "Data binding the Picker."

The code-behind file of `DataTransfer5InfoPage` achieves the MVVM goal of being nothing but a call to `InitializeComponent`:

```
public partial class DataTransfer5InfoPage : ContentPage
```

```
{
    public DataTransfer5InfoPage()
    {
        InitializeComponent();
    }
}
```

The other convenient aspect of **DataTransfer5** is that there is no longer a need to override the `OnAppearing` and `OnDisappearing` methods, and no need to wonder about the order of these method calls during page navigation.

But what's really nice is that it's easy to migrate **DataTransfer5** to a version that saves application data when the program is terminated and restores it the next time the program is run.

Saving and restoring page state

Particularly as you begin working more with multipage applications, it's very beneficial to treat the pages of your application *not* as the primary repositories of data, but merely as temporary visual and interactive views of underlying data. The key word here is *temporary*. If you keep the underlying data up to date as the user interacts with it, then pages can appear and disappear without worry.

The final program in this series is **DataTransfer6**, which saves the contents of `AppData` (and some other information) in application local storage when the program is suspended—and hence when the program is terminated—and then retrieves that data the next time the program starts up.

Besides saving data that the user has painstakingly entered, you'll probably also want to save the state of the page navigation stack. This means that if the user is entering data on the info page and the program terminates, then the next time the program runs, it navigates to that info page with the partially entered data restored.

As you'll recall, the `Application` class defines a property named `Properties` that is a dictionary with `string` keys and `object` values. You can set items in the `Properties` dictionary either before or during the `OnSleep` override in your `App` class. The items will then be available the next time the `App` constructor executes.

The underlying platform serializes objects in the `Properties` dictionary by converting the objects to a form in which they can be saved to a file. It doesn't matter to the application programmer whether this is a binary form or a string form, perhaps XML or JSON.

For integer or floating-point numbers, for `DateTime` values, or for strings, the serialization is straightforward. On some platforms, it might be possible to save an instance of a more complex class, such as `InformationViewModel`, directly to the `Properties` collection. However, this doesn't work on all the platforms. It's much safer to serialize classes yourself to XML or JSON strings and then save the resultant strings in the `Properties` collection. With the version of .NET available to Xamarin.Forms Portable Class Libraries, XML serialization is a bit easier than JSON serialization, and that's what **DataTransfer6** uses.

When performing serialization and deserialization, you need to watch out for object references. Serialization does not preserve object equality. Let's see how this can be an issue:

The version of `AppData` introduced in **DataTransfer5** has two properties: `InfoCollection`, which is a collection of `InformationViewModel` objects, and `CurrentInfo`, which is an `InformationViewModel` object that is currently being edited.

The program relies on the fact that the `CurrentInfo` object is also an item in the `InfoCollection`. The `CurrentInfo` becomes the `BindingContext` for the info page, and the properties of that `InformationViewModel` instance are interactively altered by the user. But only because that same object is part of `InfoCollection` will the new values show up in the `ListView`.

What happens when you serialize the `InfoCollection` and `CurrentInfo` properties of `AppData` and then deserialize to create a new `AppData`?

In the deserialized version, the `CurrentInfo` object will have the exact same properties as one of the items in the `InfoCollection`, but it won't be the same instance. If the program is restored to allow the user to continue editing an item on the info page, none of those edits will be reflected in the object in the `ListView` collection.

With that mental preparation, it is now time to look at the version of `AppData` in **DataTransfer6**.

```
public class AppData
{
    public AppData()
    {
        InfoCollection = new ObservableCollection<InformationViewModel>();
        CurrentInfoIndex = -1;
    }

    public ObservableCollection<InformationViewModel> InfoCollection { private set; get; }

    [XmlIgnore]
    public InformationViewModel CurrentInfo { set; get; }

    public int CurrentInfoIndex { set; get; }

    public string Serialize()
    {
        // If the CurrentInfo is valid, set the CurrentInfoIndex.
        if (CurrentInfo != null)
        {
            CurrentInfoIndex = InfoCollection.IndexOf(CurrentInfo);
        }
        XmlSerializer serializer = new XmlSerializer(typeof(AppData));
        using (StringWriter stringWriter = new StringWriter())
        {
            serializer.Serialize(stringWriter, this);
            return stringWriter.GetStringBuilder().ToString();
        }
    }
```

```
    public static AppData Deserialize(string strAppData)
    {
        XmlSerializer serializer = new XmlSerializer(typeof(AppData));
        using (StringReader stringReader = new StringReader(strAppData))
        {
            AppData appData = (AppData)serializer.Deserialize(stringReader);

            // If the CurrentInfoIndex is valid, set the CurrentInfo.
            if (appData.CurrentInfoIndex != -1)
            {
                appData.CurrentInfo = appData.InfoCollection[appData.CurrentInfoIndex];
            }
            return appData;
        }
    }
}
```

This version has an `InfoCollection` property and a `CurrentInfo` property like the previous version, but it also includes a `CurrentInfoIndex` property of type `int`, and the `CurrentInfo` property is flagged with the `XmlIgnore` attribute, which means that it won't be serialized.

The class also has two methods, named `Serialize` and `Deserialize`. `Serialize` begins by setting the `CurrentInfoIndex` property to the index of `CurrentInfo` within the `InfoCollection`. It then converts the instance of the class to an XML string and returns that string.

`Deserialize` does the opposite. It is a static method with a `string` argument. The string is assumed to be the XML representation of an `AppData` object. After it's converted into an `AppData` instance, the method sets the `CurrentInfo` property based on the `CurrentInfoIndex` property. Now `CurrentInfo` is once again the identical object to one of the members of the `InfoCollection`. The method returns that `AppData` instance.

The only other change from **DataTransfer5** to **DataTransfer6** is the `App` class. The `OnSleep` override serializes the `AppData` object and saves it in the `Properties` dictionary with a key of "appData". But it also saves a Boolean value with the key "isInfoPageActive" if the user has navigated to `DataTransfer6InfoPage` and is possibly in the process of entering or editing information.

The `App` constructor deserializes the string available from the "appData" `Properties` entry or sets the `AppData` property to a new instance if that dictionary entry doesn't exist. If the "isInfoPageActive" entry is `true`, it must not only instantiate `DataTransfer6MainPage` as the argument to the `NavigationPage` constructor (as usual), but must also navigate to `DataTransfer6InfoPage`:

```
public class App : Application
{
    public App()
    {
        // Ensure link to Toolkit library.
        Xamarin.FormsBook.Toolkit.Toolkit.Init;

        // Load previous AppData if it exists.
```

```
        if (Properties.ContainsKey("appData"))
        {
            AppData = AppData.Deserialize((string)Properties["appData"]);
        }
        else
        {
            AppData = new AppData();
        }

        // Launch home page.
        Page homePage = new DataTransfer6HomePage();
        MainPage = new NavigationPage(homePage);

        // Possibly navigate to info page.
        if (Properties.ContainsKey("isInfoPageActive") &&
            (bool)Properties["isInfoPageActive"])
        {
            homePage.Navigation.PushAsync(new DataTransfer6InfoPage(), false);
        }
    }

    public AppData AppData { private set; get; }

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

    protected override void OnSleep()
    {
        // Save AppData serialized into string.
        Properties["appData"] = AppData.Serialize();

        // Save Boolean for info page active.
        Properties["isInfoPageActive"] =
            MainPage.Navigation.NavigationStack.Last() is DataTransfer6InfoPage;
    }

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

To test this program, it is necessary to terminate the program in such a way that the `App` class gets a call to its `OnSleep` method. If you're running the program under the Visual Studio or Xamarin Studio debugger, do *not* terminate the program from the debugger. Instead, terminate the application on the phone.

Perhaps the best way of terminating a program on phones and phone emulators is to first display all the currently running programs:

- On iOS, double tap the **Home** button.

- On Android, tap the (rightmost) **MultiTask** button.
- On Windows Phone, hold down the (leftmost) **Back** button.

This action causes the `OnSleep` method to be called. You can then terminate the program:

- On iOS, swipe the application up.
- On Android, swipe it to the side.
- On Windows Phone, swipe it down.

When running the Windows program in a window, you can terminate the program simply by clicking the **Close** button. In tablet mode, swipe the program down from the top.

You can then use Visual Studio or Xamarin Studio to stop debugging the application (if necessary). Then run the program again to see whether it "remembers" where it left off.

Saving and restoring the navigation stack

Many multipage applications have a page architecture that is more complex than **DataTransfer6**, and you'll want a generalized way to save and restore the entire navigation stack. Moreover, you'll probably want to integrate the preservation of the navigation stack with a systematic way to save and restore the state of each page, particularly if you're not using MVVM.

In an MVVM application, generally a ViewModel is responsible for saving the data that underlies the various pages of an application. But in the absence of a ViewModel, that job is left up to each individual page, generally involving the `Properties` dictionary implemented by the `Application` class. However, you need to be careful not to have duplicate dictionary keys in two or more pages. Duplicate keys are particularly likely if a particular page type might have multiple instances in the navigation stack.

The problem of duplicate dictionary keys can be avoided if each page in the navigation stack uses a unique prefix for its dictionary keys. For example, the home page might use a prefix of "0" for all its dictionary keys, the next page in the navigation stack might use a prefix of "1" and so forth.

The **Xamarin.FormsBook.Toolkit** library has an interface and a class that work together to help you with saving and restoring the navigation stack, and saving and restoring page state using unique dictionary key prefixes. This interface and class do not preclude the use of MVVM with your application.

The interface is called `IPersistentPage`, and it has methods named `Save` and `Restore` that include the dictionary-key prefix as an argument:

```
namespace Xamarin.FormsBook.Toolkit
{
    public interface IPersistentPage
```

```
    {
        void Save(string prefix);

        void Restore(string prefix);
    }
}
```

Any page in your application can implement `IPersistentPage`. The `Save` and `Restore` methods are responsible for using the `prefix` parameter when adding items to the `Properties` dictionary or accessing those items. You'll see examples shortly.

These `Save` and `Restore` methods are called from a class named `MultiPageRestorableApp`, which derives from `Application` and is intended to be a base class for the `App` class. When you derive `App` from `MultiPageRestorableApp`, you have two responsibilities:

- From the `App` class's constructor, call the `Startup` method of `MultiPageRestorableApp` with the type of the application's home page.
- Call the base class's `OnSleep` method from the `OnSleep` override of the `App` class.

There are also two requirements when using `MultiPageRestoreableApp`:

- Each page in the application must have a parameterless constructor.
- When you derive `App` from `MultiPageRestorableApp`, this base class becomes a public type exposed from the application's Portable Class Library. This means that all the individual platform projects also require a reference to the **Xamarin.FormsBook.Toolkit** library.

`MultiPageRestorableApp` implements its `OnSleep` method by looping through the contents of `NavigationStack` and `ModalStack`. Each page is given a unique index starting at 0, and each page is reduced to a short string that includes the page type, the page's index, and a Boolean indicating whether the page is modal:

```
namespace Xamarin.FormsBook.Toolkit
{
    // Derived classes must call Startup(typeof(YourStartPage));
    // Derived classes must call base.OnSleep() in override
    public class MultiPageRestorableApp : Application
    {
        …
        protected override void OnSleep()
        {
            StringBuilder pageStack = new StringBuilder();
            int index = 0;

            // Accumulate the modeless pages in pageStack.
            IReadOnlyList<Page> stack = (MainPage as NavigationPage).Navigation.NavigationStack;
            LoopThroughStack(pageStack, stack, ref index, false);

            // Accumulate the modal pages in pageStack.
            stack = (MainPage as NavigationPage).Navigation.ModalStack;
            LoopThroughStack(pageStack, stack, ref index, true);
```

```
            // Save the list of pages.
            Properties["pageStack"] = pageStack.ToString();
        }

        void LoopThroughStack(StringBuilder pageStack, IReadOnlyList<Page> stack,
                              ref int index, bool isModal)
        {
            foreach (Page page in stack)
            {
                // Skip the NavigationPage that's often at the bottom of the modal stack.
                if (page is NavigationPage)
                    continue;

                pageStack.AppendFormat("{0} {1} {2}", page.GetType().ToString(),
                                                      index, isModal);
                pageStack.AppendLine();

                if (page is IPersistentPage)
                {
                    string prefix = index.ToString() + ' ';
                    ((IPersistentPage)page).Save(prefix);
                }
                index++;
            }
        }
    }
}
```

In addition, each page that implements `IPersistentPage` gets a call to its `Save` method with the integer prefix converted to a string.

The `OnSleep` method concludes by saving the composite string containing one line per page to the `Properties` dictionary with the key "pageStack".

An `App` class that derives from `MultiPageRestorableApp` must call the `Startup` method from its constructor. The `Startup` method accesses the "pageStack" entry in the `Properties` dictionary. For each line, it instantiates a page of that type. If the page implements `IPersistentPage`, then the `Restore` method is called. Each page is added to the navigation stack with a call to `PushAsync` or `PushModalAsync`. Notice the second argument to `PushAsync` and `PushModalAsync` is set to `false` to suppress any page-transition animation the platform might implement:

```
namespace Xamarin.FormsBook.Toolkit
{
    // Derived classes must call Startup(typeof(YourStartPage));
    // Derived classes must call base.OnSleep() in override
    public class MultiPageRestorableApp : Application
    {
        protected void Startup(Type startPageType)
        {
            object value;

            if (Properties.TryGetValue("pageStack", out value))
```

```
        {
            MainPage = new NavigationPage();
            RestorePageStack((string)value);
        }
        else
        {
            // First time the program is run.
            Assembly assembly = this.GetType().GetTypeInfo().Assembly;
            Page page = (Page)Activator.CreateInstance(startPageType);
            MainPage = new NavigationPage(page);
        }
    }

    async void RestorePageStack(string pageStack)
    {
        Assembly assembly = GetType().GetTypeInfo().Assembly;
        StringReader reader = new StringReader(pageStack);
        string line = null;

        // Each line is a page in the navigation stack.
        while (null != (line = reader.ReadLine()))
        {
            string[] split = line.Split(' ');
            string pageTypeName = split[0];
            string prefix = split[1] + ' ';
            bool isModal = Boolean.Parse(split[2]);

            // Instantiate the page.
            Type pageType = assembly.GetType(pageTypeName);
            Page page = (Page)Activator.CreateInstance(pageType);

            // Call Restore on the page if it's available.
            if (page is IPersistentPage)
            {
                ((IPersistentPage)page).Restore(prefix);
            }

            if (!isModal)
            {
                // Navigate to the next modeless page.
                await MainPage.Navigation.PushAsync(page, false);

                // HACK: to allow page navigation to complete!
                if (Device.OS == TargetPlatform.Windows &&
                        Device.Idiom != TargetIdiom.Phone)
                    await Task.Delay(250);
            }
            else
            {
                // Navigate to the next modal page.
                await MainPage.Navigation.PushModalAsync(page, false);

                // HACK: to allow page navigation to complete!
                if (Device.OS == TargetPlatform.iOS)
```

```
                    await Task.Delay(100);
                }
            }
        }
        …
    }
}
```

This code contains two comments that begin with the word "HACK". These indicate statements that are intended to fix two problems encountered in Xamarin.Forms:

- On iOS, nested modal pages don't properly restore unless a little time separates the `PushModalAsync` calls.
- On Windows 8.1, modeless pages do not contain left arrow **Back** buttons unless a little time separates the calls to `PushAsync`.

Let's try it out!

The **StackRestoreDemo** program has three pages, named `DemoMainPage`, `DemoModelessPage`, and `DemoModalPage`, each of which contains a `Stepper` and implements `IPersistentPage` to save and restore the `Value` property associated with that `Stepper`. You can set different `Stepper` values on each page and then check whether they're restored correctly.

The `App` class derives from `MultiPageRestorableApp`. It calls `Startup` from its constructor and calls the base class `OnSleep` method from its `OnSleep` override:

```
public class App : Xamarin.FormsBook.Toolkit.MultiPageRestorableApp
{
    public App()
    {
        // Must call Startup with type of start page!
        Startup(typeof(DemoMainPage));
    }

    protected override void OnSleep()
    {
        // Must call base implementation!
        base.OnSleep();
    }
}
```

The XAML for `DemoMainPage` instantiates a `Stepper`, a `Label` showing the value of that `Stepper`, and two `Button` elements:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StackRestoreDemo.DemoMainPage"
             Title="Main Page">

    <StackLayout>
        <Label Text="Main Page"
```

```
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center" />

        <Grid VerticalOptions="CenterAndExpand">
            <Stepper x:Name="stepper"
                     Grid.Column="0"
                     VerticalOptions="Center"
                     HorizontalOptions="Center" />

            <Label Grid.Column="1"
                   Text="{Binding Source={x:Reference stepper},
                                  Path=Value,
                                  StringFormat='{0:F0}'}"
                   FontSize="Large"
                   VerticalOptions="Center"
                   HorizontalOptions="Center" />
        </Grid>

        <Button Text="Go to Modeless Page"
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnGoToModelessPageClicked" />

        <Button Text="Go to Modal Page"
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnGoToModalPageClicked" />

    </StackLayout>
</ContentPage>
```

The event handlers for the two `Button` elements navigate to `DemoModelessPage` and `DemoModalPage`. The implementation of `IPersistentPage` saves and restores the `Value` property of the `Stepper` element by using the `Properties` dictionary. Notice the use of the `prefix` parameter in defining the dictionary key:

```
public partial class DemoMainPage : ContentPage, IPersistentPage
{
    public DemoMainPage()
    {
        InitializeComponent();
    }

    async void OnGoToModelessPageClicked(object sender, EventArgs args)
    {
        await Navigation.PushAsync(new DemoModelessPage());
    }

    async void OnGoToModalPageClicked(object sender, EventArgs args)
    {
        await Navigation.PushModalAsync(new DemoModalPage());
```

```
    }

    public void Save(string prefix)
    {
        App.Current.Properties[prefix + "stepperValue"] = stepper.Value;
    }

    public void Restore(string prefix)
    {
        object value;
        if (App.Current.Properties.TryGetValue(prefix + "stepperValue", out value))
            stepper.Value = (double)value;
    }
}
```

The `DemoModelessPage` class is essentially the same as `DemoMainPage` except for the `Title` property and the `Label` that displays the same text as the `Title`.

The `DemoModalPage` is somewhat different. It also has a `Stepper` and a `Label` that displays the value of the `Stepper`, but one `Button` returns to the previous page and the other `Button` navigates to another modal page:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StackRestoreDemo.DemoModalPage"
             Title="Modal Page">

    <StackLayout>
        <Label Text="Modal Page"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Grid VerticalOptions="CenterAndExpand">
            <Stepper x:Name="stepper"
                     Grid.Column="0"
                     VerticalOptions="Center"
                     HorizontalOptions="Center" />

            <Label Grid.Column="1"
                   Text="{Binding Source={x:Reference stepper},
                                  Path=Value,
                                  StringFormat='{0:F0}'}"
                   FontSize="Large"
                   VerticalOptions="Center"
                   HorizontalOptions="Center" />
        </Grid>

        <Button Text="Go Back"
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnGoBackClicked" />
```

```
        <Button x:Name="gotoModalButton"
                Text="Go to Modal Page"
                FontSize="Large"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnGoToModalPageClicked" />

    </StackLayout>
</ContentPage>
```

The code-behind file contains handlers for those two buttons and also implements `IPersistentPage`:

```
public partial class DemoModalPage : ContentPage, IPersistentPage
{
    public DemoModalPage()
    {
        InitializeComponent();
    }

    async void OnGoBackClicked(object sender, EventArgs args)
    {
        await Navigation.PopModalAsync();
    }

    async void OnGoToModalPageClicked(object sender, EventArgs args)
    {
        await Navigation.PushModalAsync(new DemoModalPage());
    }

    public void Save(string prefix)
    {
        App.Current.Properties[prefix + "stepperValue"] = stepper.Value;
    }

    public void Restore(string prefix)
    {
        object value;
        if (App.Current.Properties.TryGetValue(prefix + "stepperValue", out value))
            stepper.Value = (double)value;
    }
}
```

One easy way to test the program is to progressively navigate to several modeless and then modal pages, setting a different value on the `Stepper` on each page. Then terminate the application from the phone or emulator (as described earlier) and start it up again. You should be on the same page as the page you left and see the same `Stepper` values as you go back through the pages.

Something like a real-life app

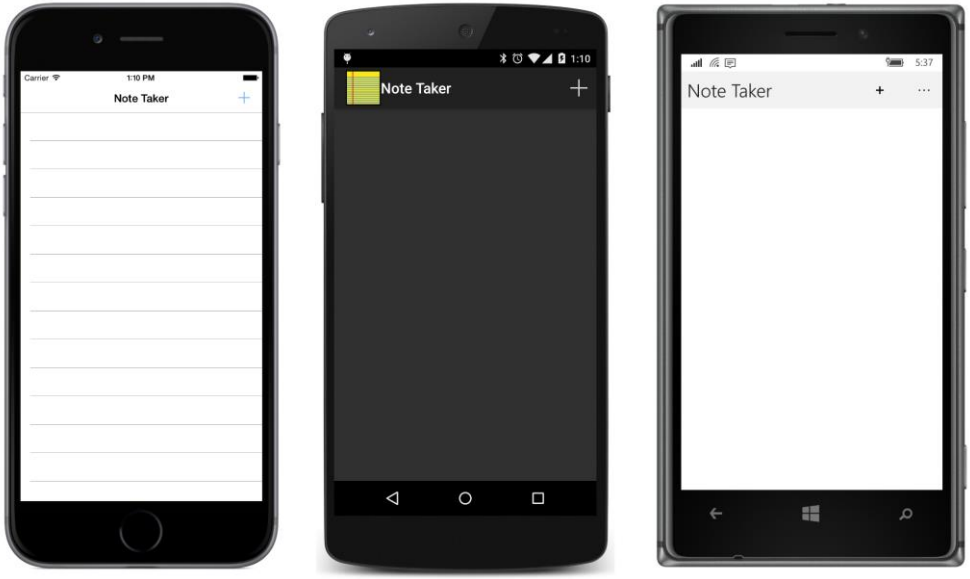
Ideally, users should not be aware when an application is terminated and restarted. The application experience should be continuous and seamless. A half-entered `Entry` that was never completed should

still be in the same state a week later even if the program hasn't been running all that time.

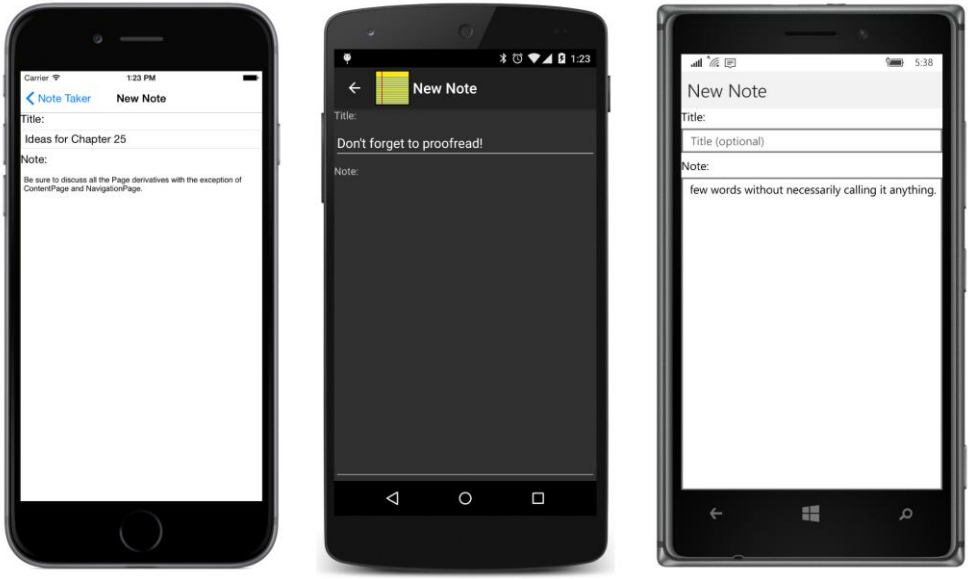
The **NoteTaker** program allows a user to take notes consisting of a title and some text. Because there may be quite a few of these notes, and they have the potential of becoming quite long, they are not stored in the `Properties` dictionary. Instead, the program makes use of the `IFileHelper` interface and `FileHelper` classes demonstrated in the **TextFileAsync** program in Chapter 20, "Async and file I/O." Each note is a separate file. Like **TextFileAsync**, the **NoteTaker** solution also contains all the projects in the **Xamarin.FormsBook.Platform** solution and references to those library projects.

NoteTaker is structured much like the `DataTransfer` programs earlier in this chapter, with pages named `NoteTakerHomePage` and `NoteTakerNotePage`.

The home page consists of an `ItemsView` that dominates the page and an **Add** button. This **Add** button is a `ToolbarItem` that takes the form of a plus sign in the upper-right corner of the iOS, Android, and Windows Phone screens:

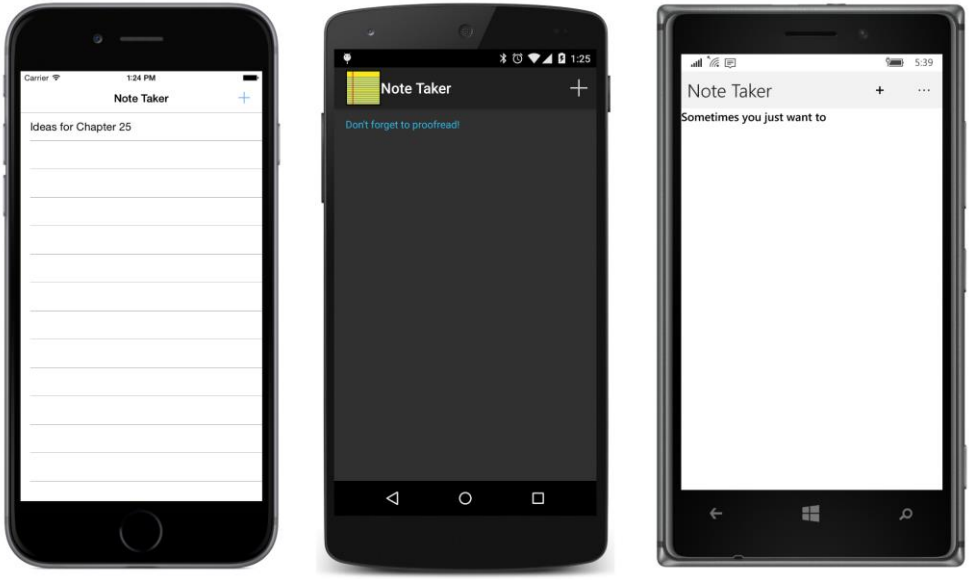


Pressing that button causes the program to navigate to the `NoteTakerNotePage`. At the top is an `Entry` for a title, but most of the page is occupied by an `Editor` for the text of the note itself. You can now type in a title and note:

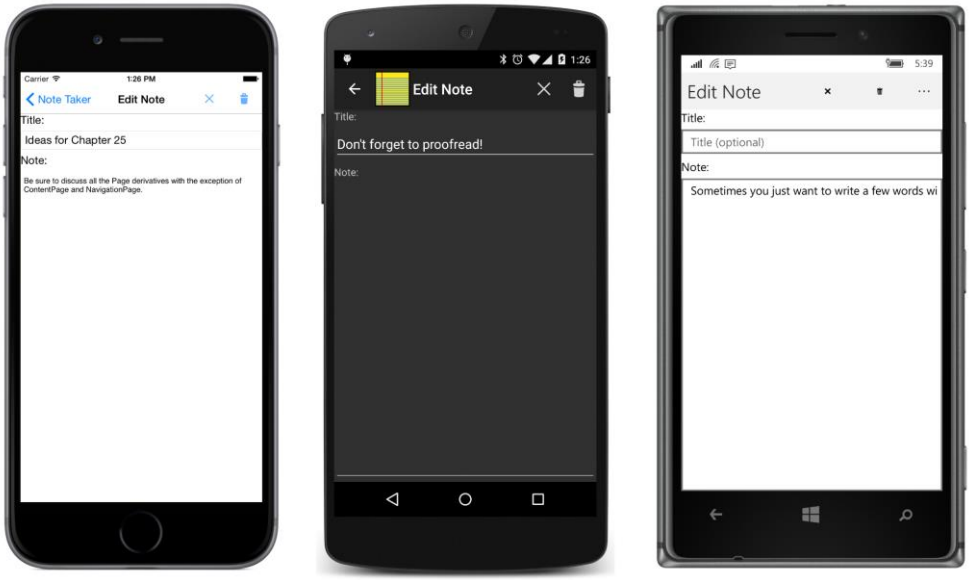


It is not necessary to enter a title. If there is none, an identifier is constructed that consists of the beginning of the text. Nor is it necessary to enter the note text. A note can consist solely of a title. (At the time this chapter was written, the Windows Runtime `Editor` didn't properly wrap text.)

If either the title or note isn't blank, the note is considered to be a valid note. When you go back to the home page by using the standard **Back** button either in the navigation bar or at the bottom of the screen, the new note is added to the `ListView`:



You can now add more new notes or edit an existing note by tapping the entry in the `ListView`. The `ListView` tap navigates to the same page as the **Add** button, but notice that the `Title` property on the second page is now "Edit Note" rather than "New Note":



You can now make changes to the note and return back to the home page. Two toolbar items are also available on this page: The first is a **Cancel** button that allows you to abandon any edits you've made. An alert asks whether you're sure. You can also tap the **Delete** item to delete the note, also with an alert for confirmation.

One of the tricky aspects of this program involves the **Cancel** button. Suppose you're in the middle of editing a note and you get distracted, and eventually the program is terminated. The next time you start up the program, you should return to the edit screen and see your edits. If you then invoke the **Cancel** command, your edits should be abandoned.

This means that when the application is suspended while a note is being edited, the note must essentially be saved in two different forms: The pre-edit note and the note with the edits. The program handles this by saving each note to a file only when `NoteTakerNotePage` gets a call to its `OnDisappearing` override. (However, some special consideration needs to accommodate the case in iOS when the page gets a call to `OnDisappearing` when the program terminates.) The file version of the `Note` object is the one without the edits. The edited version is reflected by the current contents of the `Entry` and `Editor`; `NoteTakerNotePage` saves those two text strings in the `Save` method of its `IPersistantPage` implementation.

The `Note` class implements `INotifyPropertyChanged` by virtue of deriving from `ViewModelBase` in the **Xamarin.FormsBook.Toolkit** library. The class defines four public properties: `Filename`, `Title`, `Text`, and `Identifier`, which is either the same as `Title` or the first 30 characters of `Text`, truncated

to display complete words only. The `Filename` property is set from the constructor and never changes:

```
public class Note : ViewModelBase, IEquatable<Note>
{
    string title, text, identifier;
    FileHelper fileHelper = new FileHelper();

    public Note(string filename)
    {
        Filename = filename;
    }

    public string Filename { private set; get; }

    public string Title
    {
        set
        {
            if (SetProperty(ref title, value))
            {
                Identifier = MakeIdentifier();
            }
        }
        get { return title; }
    }

    public string Text
    {
        set
        {
            if (SetProperty(ref text, value) && String.IsNullOrWhiteSpace(Title))
            {
                Identifier = MakeIdentifier();
            }
        }
        get { return text; }
    }

    public string Identifier
    {
        private set { SetProperty(ref identifier, value); }
        get { return identifier; }
    }

    string MakeIdentifier()
    {
        if (!String.IsNullOrWhiteSpace(this.Title))
            return Title;

        int truncationLength = 30;

        if (Text == null || Text.Length <= truncationLength)
        {
            return Text;
```

```
        }

        string truncated = Text.Substring(0, truncationLength);

        int index = truncated.LastIndexOf(' ');

        if (index != -1)
            truncated = truncated.Substring(0, index);

        return truncated;
    }

    public Task SaveAsync()
    {
        string text = Title + Environment.NewLine + Text;
        return fileHelper.WriteTextAsync(Filename, text);
    }

    public async Task LoadAsync()
    {
        string text = await fileHelper.ReadTextAsync(Filename);

        // Break string into Title and Text.

        int index = text.IndexOf(Environment.NewLine);
        Title = text.Substring(0, index);
        Text = text.Substring(index + Environment.NewLine.Length);
    }

    public async Task DeleteAsync()
    {
        await fileHelper.DeleteAsync(Filename);
    }

    public bool Equals(Note other)
    {
        return other == null ? false : Filename == other.Filename;
    }
}
```

The `Note` class also defines methods to save, load, or delete the file associated with the particular instance of the class. The first line of the file is the `Title` property, and the remainder of the file is the `Text` property.

In most cases, there is a one-to-one correspondence between `Note` files and instances of the `Note` class. However, if the `DeleteAsync` method is called, then the `Note` object still exists, but the file does not. (However, as you'll see, all references to a `Note` object whose `DeleteAsync` method is called are quickly detached and the object become eligible for garbage collection.)

The program does not maintain a list of these files when the program isn't running. Instead, the `NoteFolder` class obtains all the files in the application's local storage with a filename extension of ".note" and creates a collection of `Note` objects from these files:

```
public class NoteFolder
{
    public NoteFolder()
    {
        this.Notes = new ObservableCollection<Note>();
        GetFilesAsync();
    }

    public ObservableCollection<Note> Notes { private set; get; }

    async void GetFilesAsync()
    {
        FileHelper fileHelper = new FileHelper();

        // Sort the filenames.
        IEnumerable<string> filenames =
            from filename in await fileHelper.GetFilesAsync()
            where filename.EndsWith(".note")
            orderby (filename)
            select filename;

        // Store them in the Notes collection.
        foreach (string filename in filenames)
        {
            Note note = new Note(filename);
            await note.LoadAsync();
            Notes.Add(note);
        }
    }
}
```

As you'll see, the filename is constructed from a `DateTime` object at the time the note is first created, consisting of the year followed by the month, day, and time, which means that when these `Note` files are sorted by filename, they appear in the collection in the same order in which they are created.

The `App` class instantiates `NoteFolder` and makes it available as a public property. `App` derives from `MultiPageRestorableApp`, so it calls `Startup` with the `NoteTakerHomePage` type, and also implements the `OnSleep` override by calling the base class implementation:

```
public class App : MultiPageRestorableApp
{
    public App()
    {
        // This loads all the existing .note files.
        NoteFolder = new NoteFolder();

        // Make call to method in MultiPageRestorableApp.
        Startup(typeof(NoteTakerHomePage));
    }

    public NoteFolder NoteFolder { private set; get; }

    protected override void OnSleep()
```

```
    {
        // Required call when deriving from MultiPageRestorableApp.
        base.OnSleep();
    }

    // Special processing for iOS.
    protected override void OnResume()
    {
        NoteTakerNotePage notePage =
            ((NavigationPage)MainPage).CurrentPage as NoteTakerNotePage;

        if (notePage != null)
            notePage.OnResume();
    }
}
```

The `App` class also overrides the `OnResume` method. If `NoteTakerNotePage` is currently active, the method calls an `OnResume` method in the note page. This is some special processing for iOS. As you'll see, `NoteTakerNotePage` saves a `Note` object to a file during its `OnDisappearing` override, but it shouldn't do that if the `OnDisappearing` override indicates that the application is terminating.

The XAML file for the `NoteTakerHomePage` instantiates the `ListView` for displaying all the `Note` objects. The `ItemsSource` is bound to the `Notes` collection of the `NoteFolder` that is stored in the `App` class. Each `Note` object is displayed in the `ListView` with its `Identifier` property:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NoteTaker.NoteTakerHomePage"
             Title="Note Taker">

    <ListView ItemsSource="{Binding Source={x:Static Application.Current},
                                    Path=NoteFolder.Notes}"
              ItemSelected="OnListViewItemSelected"
              VerticalOptions="FillAndExpand">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding Identifier}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>

    <ContentPage.ToolbarItems>
        <ToolbarItem Name="Add Note"
                     Order="Primary"
                     Activated="OnAddNoteActivated">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="new.png"
                            Android="ic_action_new.png"
                            WinPhone="Images/add.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
```

```
</ContentPage>
```

The code-behind file is dedicated to handling just two events: The `ItemSelected` event of the `ListView` for editing an existing `Note`, and the `Activated` event of the `ToolbarItem` for creating a new `Note`:

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

    async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem != null)
        {
            // Deselect the item.
            ListView listView = (ListView)sender;
            listView.SelectedItem = null;

            // Navigate to NotePage.
            await Navigation.PushAsync(new NoteTakerNotePage
                {
                    Note = (Note)args.SelectedItem,
                    IsNoteEdit = true
                });
        }
    }

    async void OnAddNoteActivated(object sender, EventArgs args)
    {
        // Create unique filename.
        DateTime dateTime = DateTime.UtcNow;
        string filename = dateTime.ToString("yyyyMMddHHmmssfff") + ".note";

        // Navigate to NotePage.
        await Navigation.PushAsync(new NoteTakerNotePage
            {
                Note = new Note(filename),
                IsNoteEdit = false
            });
    }
}
```

In both cases, the event handler instantiates `NoteTakerNotePage`, sets two properties, and navigates to that page. For a new note, a filename is constructed and a `Note` object is instantiated. For an existing note, the `Note` object is simply the selected item from the `ListView`. Notice that the new note has a filename but is not yet saved to a file or made part of the `Notes` collection in `NoteFolder`.

The XAML file for `NoteTakerNotePage` has an `Entry` and `Editor` for entering a note's title and text. The data bindings on the `Text` properties of these elements imply that the `BindingContext` for

the page is a `Note` object:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NoteTaker.NoteTakerNotePage"
             Title="New Note">

    <StackLayout>
        <Label Text="Title:" />

        <Entry Text="{Binding Title}"
               Placeholder="Title (optional)" />

        <Label Text="Note:" />

        <Editor Text="{Binding Text}"
                Keyboard="Text"
                VerticalOptions="FillAndExpand" />
    </StackLayout>

    <ContentPage.ToolbarItems>
        <ToolbarItem Name="Cancel"
                     Order="Primary"
                     Activated="OnCancelActivated">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="cancel.png"
                            Android="ic_action_cancel.png"
                            WinPhone="Images/cancel.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>

        <ToolbarItem Name="Delete"
                     Order="Primary"
                     Activated="OnDeleteActivated">
            <ToolbarItem.Icon>
                <OnPlatform x:TypeArguments="FileImageSource"
                            iOS="discard.png"
                            Android="ic_action_discard.png"
                            WinPhone="Images/delete.png" />
            </ToolbarItem.Icon>
        </ToolbarItem>
    </ContentPage.ToolbarItems>
</ContentPage>
```

The two `ToolbarItem` elements toward the bottom should be visible only when an existing note is being edited. The removal of these toolbar items occurs in the code-behind file when the `IsNoteEdit` property is set from the home page. That code also changes the `Title` property for the page. The `set` accessor for the `Note` property is responsible for setting the page's `BindingContext`:

```
public partial class NoteTakerNotePage : ContentPage, IPersistentPage
{
    Note note;
    bool isNoteEdit;
```

```
    …
    public NoteTakerNotePage()
    {
        InitializeComponent();
    }

    public Note Note
    {
        set
        {
            note = value;
            BindingContext = note;
        }
        get { return note; }
    }

    public bool IsNoteEdit
    {
        set
        {
            isNoteEdit = value;
            Title = IsNoteEdit ? "Edit Note" : "New Note";

            // No toolbar items if it's a new Note!
            if (!IsNoteEdit)
            {
                ToolbarItems.Clear();
            }
        }
        get { return isNoteEdit; }
    }
    …
}
```

The `NoteTakerNotePage` class implements the `IPersistentPage` interface, which means that it has methods named `Save` and `Restore` for saving and restoring the page state. These methods use the `Properties` dictionary to save and restore the three properties of `Note` that define a `Note` object—the `Filename`, `Title`, and `Text` properties—and the `IsNoteEdit` property of `NoteTakerNotePage`. This is the `Note` object in its current edited state:

```
public partial class NoteTakerNotePage : ContentPage, IPersistentPage
{
    …
    // Special field for iOS.
    bool isInSleepState;
    …
    // IPersistent implementation
    public void Save(string prefix)
    {
        // Special code for iOS.
        isInSleepState = true;

        Application app = Application.Current;
```

```
        app.Properties["fileName"] = Note.Filename;
        app.Properties["title"] = Note.Title;
        app.Properties["text"] = Note.Text;
        app.Properties["isNoteEdit"] = IsNoteEdit;
    }

    public void Restore(string prefix)
    {
        Application app = Application.Current;

        // Create a new Note object.
        Note note = new Note((string)app.Properties["fileName"])
        {
            Title = (string)app.Properties["title"],
            Text = (string)app.Properties["text"]
        };

        // Set the properties of this class.
        Note = note;
        IsNoteEdit = (bool)app.Properties["isNoteEdit"];
    }

    // Special code for iOS.
    public void OnResume()
    {
        isInSleepState = false;
    }
    …
}
```

The class also defines a method named `OnResume` that is called from the `App` class. Thus, the `isInSleepState` field is `true` when the application has been suspended.

The purpose of the `isInSleepState` field is to avoid saving the `Note` to a file when the `OnDisappearing` override is called as the application is being terminated under iOS. Saving this `Note` object to a file would not allow the user to later abandon edits of the `Note` by pressing the **Cancel** button on this page.

If the `OnDisappearing` override indicates that the user is returning back to the home page—as it otherwise does in this application—then the `Note` object can be saved to a file, and possibly added to the `Notes` collection in `NoteFolder`:

```
public partial class NoteTakerNotePage : ContentPage, IPersistentPage
{
    …
    async protected override void OnDisappearing()
    {
        base.OnDisappearing();

        // Special code for iOS:
        //      Do not save note when program is terminating.
        if (isInSleepState)
            return;
```

```
        // Only save the note if there's some text somewhere.
        if (!String.IsNullOrWhiteSpace(Note.Title) ||
            !String.IsNullOrWhiteSpace(Note.Text))
        {
            // Save the note to a file.
            await Note.SaveAsync();

            // Add it to the collection if it's a new note.
            NoteFolder noteFolder = ((App)App.Current).NoteFolder;

            // IndexOf method finds match based on Filename property
            //      based on implementation of IEquatable in Note.
            int index = noteFolder.Notes.IndexOf(note);
            if (index == -1)
            {
                // No match -- add it.
                noteFolder.Notes.Add(note);
            }
            else
            {
                // Match -- replace it.
                noteFolder.Notes[index] = note;
            }
        }
    }
    …
}
```

The `Note` class implements the `IEquatable` interface and defines two `Note` objects to be equal if their `Filename` properties are the same. The `OnDisappearing` override relies on that definition of equality to avoid adding a `Note` object to the collection if another one already exists with the same `Filename` property.

Finally, the `NoteTakerNotePage` code-behind file has handlers for the two `ToolbarItem` elements. In both cases, the processing begins with a call to `DisplayAlert` to get user confirmation, and either reloads the `Note` object from the file (effectively overwriting any edits), or deletes the file and removes it from the `Notes` collection:

```
public partial class NoteTakerNotePage : ContentPage, IPersistentPage
{
    …
    async void OnCancelActivated(object sender, EventArgs args)
    {
        if (await DisplayAlert("Note Taker", "Cancel note edit?",
                                             "Yes", "No"))
        {
            // Reload note.
            await Note.LoadAsync();

            // Return to home page.
            await Navigation.PopAsync();
        }
```

```
    }

    async void OnDeleteActivated(object sender, EventArgs args)
    {
        if (await DisplayAlert("Note Taker", "Delete this note?",
                                             "Yes", "No"))
        {
            // Delete Note file and remove from collection.
            await Note.DeleteAsync();
            ((App)App.Current).NoteFolder.Notes.Remove(Note);

            // Wipe out Entry and Editor so the Note
            //  won't be saved during OnDisappearing.
            Note.Title = "";
            Note.Text = "";

            // Return to home page.
            await Navigation.PopAsync();
        }
    }
}
```

Of course, this is not the only way to write a program like this. It's possible to move a lot of the logic for creating, editing, and deleting notes into `AppData` and make it a proper ViewModel. `AppData` would probably need a new property of type `Note` called `CurrentNote`, and several properties of type `ICommand` for binding to the `Command` property of each of the `ToolbarItem` elements.

Some programmers even try to move page-navigation logic into ViewModels, but not everyone agrees that this is a proper approach to MVVM. Is a page part of the user interface and hence part of the View? Or are pages really more like a collection of related data items?

Philosophical questions such as those might become even more vexing as the varieties of page types in Xamarin.Forms are explored in the next chapter.