Chapter 20
Async and file I/O

Graphical user interfaces have a little peculiarity that has far-reaching consequences: User input to an application must be processed sequentially. Regardless of whether user-input events come from a keyboard, a mouse, or touch, each event must be completely processed by an application—either directly or through user-interface objects such as buttons or sliders—before the application obtains the next user-input event from the operating system.

The rationale behind this restriction becomes clear after a little reflection and perhaps an example: Suppose a page contains two buttons, and the user quickly taps one and then the other. Might it be possible for the two buttons to process those two taps concurrently in two separate threads of execution? No, that would not work. It could be that the first button changes the meaning of the second button, perhaps disabling it entirely. For this reason, the first button must be allowed to completely finish processing its tap before the second button begins processing its own tap.

The consequences of this restriction are severe: All user input to a particular application must be processed in a single thread of execution. Moreover, user-interface objects are generally not thread-safe. They cannot be modified from a secondary thread of execution. All code connected with an application's user interface is therefore restricted to a single thread. This thread is known as the *main thread* or the *user-interface thread* or the *UI thread*.

As we users have become more accustomed to graphical user interfaces over the decades, we've become increasingly intolerant of even the slightest lapse in responsiveness. As application programmers, we therefore try our best to keep the user interface responsive to achieve maximum user satisfaction. This means that anything running on the UI thread must perform its processing as quickly as possible and return control back to the operating system. If an event handler running in the UI thread gets bogged down in a long processing job, the entire user interface will seem to freeze and certainly annoy the user.

For this reason, any lengthy jobs that an application must perform should be relegated to secondary threads of execution, often called *worker threads*. These worker threads are said to run "in the background" and do not interfere with the responsiveness of the UI thread.

You've already seen some examples in this book. Several sample programs—the **ImageBrowser** and **BitmapStreams** programs in Chapter 13, "Bitmaps," and the **SchoolOfFineArt** library and **RssFeed** program in Chapter 19, "Collection views"—use the `WebRequest` class to download files over the Internet. A call to the `BeginGetResponse` method of `WebRequest` starts a worker thread that accesses the web resource asynchronously. The `WebRequest` call returns quickly, and the program can handle other user input while the file is being downloaded. An argument to `BeginGetResponse` is a callback method that is invoked when the background process completes. Within this callback method the program calls `EndGetResponse` to get access to the downloaded data.

But the callback method passed to `BeginGetResponse` has a little problem. The callback method runs in the same worker thread that downloads the file, and in the general case, you can't access user-interface objects from anything other than the UI thread. Usually, this means that the callback method must access the UI thread. Each of the three platforms supported by Xamarin.Forms has its own native method for running code from a secondary thread on the UI thread, but in Xamarin.Forms these are all available through the `Device.BeginInvokeOnMainThread` method. (As you'll recall, however, there are some exceptions generally related to ViewModels: Although a secondary thread can't access a user-interface object directly, the secondary thread can set a property that is bound to a user-interface object through a data binding.)

In recent years, asynchronous processing has become more ubiquitous at the same time that it's become easier for programmers. This is an ongoing trend: The future of computing will undoubtedly involve a lot more asynchronous computing and parallel processing, particularly with the increasing use of multicore processor chips. Developers will need good operating-system support and language tools to work with asynchronous operations, and fortunately .NET and C# have been in the forefront of this support.

This chapter will explore some of the basics of working with asynchronous processing in Xamarin.Forms applications, including using the .NET `Task` class to help you define and work with asynchronous methods. The customary hassle of dealing with callback functions has been alleviated greatly with two keywords introduced in C# 5.0: `async` and `await`. The `await` operator has revolutionized asynchronous programming by simplifying the syntax of asynchronous calls, by clarifying program flow surrounding asynchronous calls, by easing the access of user-interface objects, by simplifying the handling of exceptions raised by worker threads, and by unifying the handling of these exceptions and cancellations of background jobs.

This chapter primarily demonstrates how to work with asynchronous processing to perform file input and output, and how to create your own worker threads for performing lengthy jobs.

But Xamarin.Forms itself contains several asynchronous methods.

From callbacks to await

The `Page` class defines three methods that let you display a visual object sometimes called an *alert* or a *message box*. Such a box pops up on the screen with some information or a question for the user. The alert box is modal, meaning that the rest of the application is unavailable while the alert is displayed. The user must dismiss it with the press of a button before returning to interact with the application.

Two of these three methods of the `Page` class are named `DisplayAlert`. The first simply displays some text with a single button to dismiss the box, while the second contains two buttons for yes or no responses. The `DisplayActionSheet` method is similar but displays any number of buttons.

In iOS, Android, and the Windows Runtime, these methods are implemented with platform-specific

objects that use events or callback methods to inform the application that the alert box has been dismissed and what button the user pressed to dismiss it. However, Xamarin.Forms has wrapped these objects with an asynchronous interface.

These three methods of the `Page` class are defined like this:

```
Task DisplayAlert (string title, string message, string cancel)

Task<bool> DisplayAlert (string title, string message, string accept, string cancel)

Task<string> DisplayActionSheet (string title, string cancel, string destruction,
                                 params string[] buttons)
```

They all return `Task` objects. The `Task` and `Task<T>` classes are defined in the `System.Threading.Tasks` namespace and they form the core of the Task-based Asynchronous Pattern, known as TAP. TAP is the recommended approach to handling asynchronous operations in .NET. The Task Parallel Library (TPL) builds on TAP.

In contrast, the `BeginGetResponse` and `EndGetResponse` methods of `WebRequest` represent an older approach to asynchronous operations involving `IAsyncResult`. This older approach is called the Asynchronous Programming Model or APM. You might also encounter code that uses the Event-based Asynchronous Model (EAP) to return information from asynchronous jobs through events.

You've already seen the simplest form of `DisplayAlert` in the **SetTimer** program in Chapter 15, "The interactive interface." **SetTimer** used an alert to indicate when a timer elapsed. The program didn't seem to care that `DisplayAlert` returned a `Task` object because the alert box was used strictly for notification purposes. It was not necessary to obtain a response from the user. However, the methods that return `Task<bool>` and `Task<string>` need to convey actual information back to the application indicating which button the user pressed to dismiss the alert.

A return value of `Task<T>` is sometimes referred to as a "promise." The actual value or object isn't available just yet, but it will be available in the future if nothing goes awry.

You can work with a `Task<T>` object in a few different ways. These approaches are fundamentally equivalent, but the C# syntax is quite different.

An alert with callbacks

The intended use of the `DisplayAlert` method that returns a `Task<bool>` is to ask the user a question with a yes or no answer. Obviously the answer isn't available until the user presses a button and the alert is dismissed, at which time a `true` value means Yes and `false` value means No.

One way to work with a `Task<T>` object is with callback methods. The **AlertCallbacks** program demonstrates that approach. It has a XAML file with a `Button` to invoke an alert and a `Label` for the program to display some information:

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

```
    <StackLayout>
        <Button Text="Invoke Alert"
                FontSize="Large"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Clicked="OnButtonClicked" />

        <Label x:Name="label"
               Text="Tap button to invoke alert"
               FontSize="Large"
               HorizontalTextAlignment="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

Here's the code-behind file with the `Clicked` event handler and two callback methods:

```
public partial class AlertCallbacksPage : ContentPage
{
    bool result;

    public AlertCallbacksPage()
    {
        InitializeComponent();
    }

    void OnButtonClicked(object sender, EventArgs args)
    {
        Task<bool> task = DisplayAlert("Simple Alert", "Decide on an option",
                                       "yes or ok", "no or cancel");
        task.ContinueWith(AlertDismissedCallback);
        label.Text = "Alert is currently displayed";
    }

    void AlertDismissedCallback(Task<bool> task)
    {
        result = task.Result;
        Device.BeginInvokeOnMainThread(DisplayResultCallback);
    }

    void DisplayResultCallback()
    {
        label.Text = String.Format("Alert {0} button was pressed",
                                   result ? "OK" : "Cancel");
    }
}
```

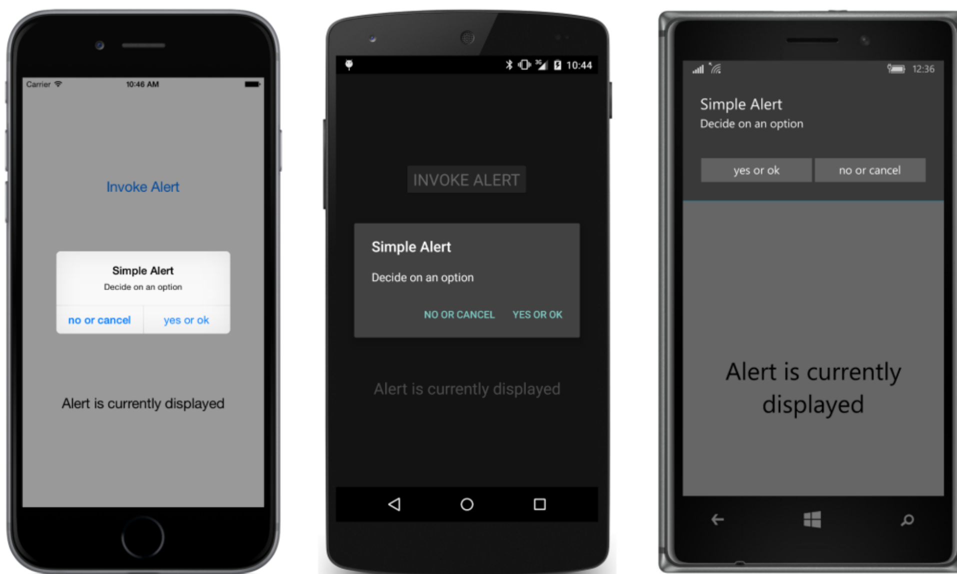
The `Clicked` handler calls `DisplayAlert` with arguments indicating a title, a question or statement, and the text for the two buttons. Generally, these two buttons are labeled "yes" and "no," or "ok" and "cancel," but you can put anything you want in those buttons as this program demonstrates.

If `DisplayAlert` were designed to be a synchronous method, the method would return a `bool` indicating which button the user pressed to dismiss the alert. However, `DisplayAlert` would not be

able to return that value until the alert were dismissed, which means that the application would be stuck in the `DisplayAlert` call during the entire time the alert is displayed. Depending on how the operating system handles user-input events, being stuck in the `DisplayAlert` call might not actually block other event handling by the user-interface thread during this time, but it might be a little strange for the UI thread to be seemingly in the `DisplayAlert` call while also handling other events.

Instead of returning a `bool` when the alert is dismissed, `DisplayAlert` returns a `Task<bool>` object that promises a `bool` result sometime in the future. To obtain that value, the `OnButtonClicked` handler in the **AlertCallbacks** program calls the `ContinueWith` method defined by `Task`. This method allows the program to specify a method that is called when the alert is dismissed. The `Clicked` handler concludes by setting some text to the `Label`, and then returns control back to the operating system.

The alert is then displayed:



Of course, the alert essentially disables the user interface of the application, but the application could still be doing some work while the alert is displayed. For example, the program could be using a timer, and that timer would continue to run. You can prove this to yourself by adding the following code to the constructor of the **AlertCallbacks** code-behind file:

```
Device.StartTimer(TimeSpan.FromSeconds(1), () =>
    {
        label.Text = DateTime.Now.ToString();
        return true;
    });
```

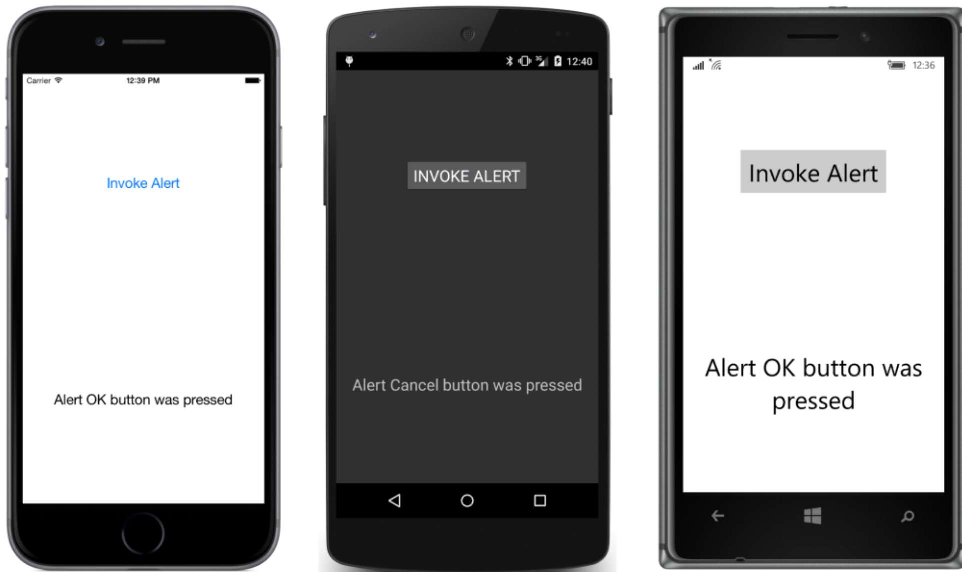
When the user dismisses the alert by tapping one of the buttons, the `AlertDismissedCallback` method is called:

```
void AlertDismissedCallback(Task<bool> task)
{
    result = task.Result;
    Device.BeginInvokeOnMainThread(DisplayResultCallback);
}
```

The argument is the same `Task` object originally returned from the `DisplayAlert` method. But now the `Result` property of the `Task` object has been set to `true` or `false` depending on what button the user pressed to dismiss the alert. The program wants to display that value, but unfortunately it cannot because this `AlertDismissedCallback` method is running in a secondary thread that Xamarin.Forms has created. This thread is not allowed to access any user-interface objects of the program. For that reason, the `AlertDismissedCallback` method saves the `bool` result in a field and calls `Device-.BeginInvokeOnMainThread` with a second callback method. That callback method runs in the UI thread:

```
void DisplayResultCallback()
{
    label.Text = String.Format("Alert {0} button was pressed",
                               result ? "OK" : "Cancel");
}
```

The `Label` then displays that text:



The **AlertCallbacks** program demonstrates one traditional way to handle asynchronous methods, but it has a distinct drawback: There are simply too many callbacks, and in one case, data must be passed from one callback to another by using a field.

An alert with lambdas

An obvious approach to simplify callbacks is with lambda functions. This is demonstrated with the **AlertLambdas** program. The XAML file is the same as in the **AlertCallbacks** method, but everything that happens in response to the button click is now inside that `Clicked` handler:

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

    void OnButtonClicked(object sender, EventArgs args)
    {
        Task<bool> task = DisplayAlert("Simple Alert", "Decide on an option",
                                       "yes or ok", "no or cancel");
        task.ContinueWith((Task<bool> taskResult) =>
            {
                Device.BeginInvokeOnMainThread(() =>
                    {
                        label.Text = String.Format("Alert {0} button was pressed",
                                                   taskResult.Result ? "OK" : "Cancel");
                    });
            });
        label.Text = "Alert is currently displayed";
    }
}
```

There is really no difference between this program and the previous one except that the callback methods have no name. They are anonymous. But sometimes lambda functions have the tendency to obscure program flow, and that is certainly the case here. The `Text` property of the `Label` is set to the text “Alert is currently displayed” right after the `ContinueWith` method is called and before the callback passed to `ContinueWith` executes, but that statement appears at the bottom of the method.

There should be a better way to denote what you want to happen without distorting program flow. That better way is called `await`.

An alert with await

The **AlertAwait** program has the same XAML file as **AlertCallbacks** and **AlertLambdas**, but the `OnButtonClicked` method is considerably simplified:

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

    async void OnButtonClicked(object sender, EventArgs args)
    {
```

```
        Task<bool> task = DisplayAlert("Simple Alert", "Decide on an option",
                                       "yes or ok", "no or cancel");
        label.Text = "Alert is currently displayed";
        bool result = await task;
        label.Text = String.Format("Alert {0} button was pressed",
                                   result ? "OK" : "Cancel");
    }
}
```

The key statement is this one:

```
bool result = await task;
```

That `task` variable is the `Task<bool>` object returned from `DisplayAlert`, but the `await` keyword seems to magically extract the Boolean result without any callbacks or lambdas.

The first thing you should know is that `await` doesn't actually wait for the alert to be dismissed! Instead, the C# compiler has performed a lot of surgery on the `OnButtonClicked` method. The method has basically been turned into a state machine. Part of the method is executed when the button is clicked, and part of the method is executed later. When the flow of execution hits the `await` keyword, the remainder of the `OnButtonClicked` method is skipped over for the moment. The `OnButtonClicked` method exits and returns control back to the operating system. From the perspective of the `Button`, the event handler has completed.

When the user dismisses the alert box, the remainder of the `OnButtonClicked` method resumes execution beginning with the assignment of the Boolean value to the `result` variable. In some circumstances, some optimizations can take place behind the scenes. For example, the flow of execution can just continue normally if the asynchronous operation completes immediately.

The `await` operator has another bonus: Notice that there's no use of `Device.BeginInvokeOnMainThread`. When the user dismisses the alert, the `OnButtonClicked` method automatically resumes execution in the user-interface thread, which means that it can access the `Label`. (In some cases, you might want to continue running in the background thread for performance reasons. If so, you can use the `ConfigureAwait` method of `Task` to do that. You'll see an example later in this chapter.)

The `await` keyword essentially converts asynchronous code into something that appears to be normal sequential imperative code. Of course, behind the scenes, there is really not much difference between this program and the two previous programs. In all three cases, the `OnButtonClicked` handler returns control back to the operating system when it displays the alert, and resumes execution when the alert is dismissed.

Simply for illustrative purposes, the three programs display some text immediately after the `DisplayAlert` method is called. If that isn't necessary, then the `DisplayAlert` call can be combined with the `await` operator to get rid of the explicit `Task<bool>` variable entirely:

```
bool result = await DisplayAlert("Simple Alert", "Decide on an option",
                                 "yes or ok", "no or cancel");
```

This is how await commonly appears in code. DisplayAlert returns Task<bool> but the await operator effectively extracts the bool result after the background task has completed.

Indeed, you can use await much like you can any other operator, and it can appear inside a more complex expression. For example, if you don't need the statement that displays the text after the DisplayAlert call, you can actually put both the await operator and DisplayAlert inside the final String.Format call:

```
async void OnButtonClicked(object sender, EventArgs args)
{
    label.Text = String.Format("Alert {0} button was pressed",
        await DisplayAlert("Simple Alert", "Decide on an option",
                           "yes or ok", "no or cancel") ? "OK" : "Cancel");
}
```

That might be a little difficult to read, but think of the combination of the await operator and the DisplayAlert method as a bool and the statement makes perfect sense.

You might have noticed that the OnButtonClicked method is marked with the async keyword. Any method in which you use await must be marked as async. However, the async keyword does not change the signature of the method. OnButtonClicked still qualifies as an event handler for the Clicked event.

But not every method can be an async method.

An alert with nothing

The simpler of the two DisplayAlert methods returns a Task object. It is intended to display some information to the user that doesn't require a response:

```
Task DisplayAlert (string title, string message, string cancel)
```

Generally, you'll want to use await with this simpler DisplayAlert method even though it doesn't return any information, and particularly if you need to perform some processing after it has been dismissed. The **NothingAlert** program has the same XAML file as the previous samples but displays this simpler alert box:

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

    async void OnButtonClicked(object sender, EventArgs args)
    {
        label.Text = "Displaying alert box";
        await DisplayAlert("Simple Alert", "Click 'dismiss' to dismiss", "dismiss");
        label.Text = "Alert has been dismissed";
    }
}
```

Nothing appears to the left of the `await` operator because the return value of `DisplayAlert` is `Task` rather than `Task<T>` and no information is returned.

The first program in this book that used this simpler form of `DisplayAlert` was the **SetTimer** program in Chapter 15. Here's the timer callback method from that program (with the oddly named `@switch` variable so that it doesn't conflict with the `switch` keyword):

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

The `DisplayAlert` call returns quickly, and the method continues to execute when the alert box is displayed. The `OnTimerTick` method then returns `true`, and a second later `OnTimerTick` is called again. Fortunately, the `Switch` is no longer toggled, so the program doesn't attempt to call `DisplayAlert` a second time. When the alert is dismissed, the user can again interact with the user interface, but no additional code is executed on its return.

What if you wanted to execute a little code after the alert box was dismissed? Try to put an `await` operator in front of `DisplayAlert` and identify the method with the `async` keyword:

```
// Will not compile!
async bool OnTimerTick()
{
    if (@switch.IsToggled && DateTime.Now >= triggerTime)
    {
        @switch.IsToggled = false;
        await DisplayAlert("Timer Alert",
                           "The '" + entry.Text + "' timer has elapsed",
                           "OK");
        // Some code to execute after the alert box is dismissed.
    }
    return true;
}
```

But as the comment says, this code will not compile.

Why not?

When the C# compiler encounters the `await` keyword, it constructs code so that the `OnTimerTick` callback returns to its caller. The remainder of the method then resumes execution when the alert box is dismissed. However, the `Device.StartTimer` method that invokes this callback is expecting the timer callback to return a Boolean value to determine whether it should call the callback again, and the C# compiler cannot construct code that returns a Boolean value because it doesn't know what that

Boolean value should be!

For this reason, methods that contain `await` operators are restricted to return types of `void`, `Task`, or `Task<T>`.

Event handlers usually have `void` return types. This is why the `Clicked` handler of a `Button` can contain `await` operators and be flagged with the `async` keyword. But the timer callback method returns a `bool`, and to use `await` within this method, the return value of the `OnTimerTick` method must be `Task<bool>`:

```
// Method compiles but Device.StartTimer does not!
async Task<bool> OnTimerTick()
{
    if (@switch.IsToggled && DateTime.Now >= triggerTime)
    {
        @switch.IsToggled = false;
        await DisplayAlert("Timer Alert",
                           "The '" + entry.Text + "' timer has elapsed",
                           "OK");
    }
    return true;
}
```

This method now contains entirely legal compilable code. When a method is defined to return `Task<T>`, the body of the method returns an object of type `T` and the compiler does the rest.

However, because the method now returns a `Task<bool>` object, code that calls this method must use `await` with the method (or call `ContinueWith` on the `Task` object) to obtain the Boolean value when the method completes execution. That's a problem for the `Device.StartTimer` call, which is not expecting the callback method to be asynchronous; it's expecting the callback method to return `bool` rather than `Task<bool>`.

If you really did want to execute some code after the alert is dismissed in the **SetTimer** program, you should use `ContinueWith` for that code. The `await` operator is very useful, but it is not a panacea for every asynchronous programming problem.

The `await` operator can only be used in a method, and the method must have a return type of `void`, `Task`, or `Task<T>`. That's it. The `get` accessors of properties cannot use `await`, and they shouldn't be performing asynchronous operations anyway. Constructors cannot use `await` because constructors are not methods and have no return type. You cannot use `await` in the body of a `lock` statement. C# 5 also prohibits using `await` in the `catch` or `finally` blocks of a `try-catch-finally` statement, but C# 6 lifts that restriction.

These restrictions turn out to be most severe for constructors. A constructor should complete promptly because nothing can really be done with an instance of a class until the constructor finishes. Although a constructor can call an asynchronous method that returns `Task`, the constructor can't use `await` with that call. The constructor finishes while the asynchronous method is still processing. (You'll see some examples in this chapter and the next.)

A constructor cannot call an asynchronous method that returns a value required by the constructor to complete. If a constructor needs to obtain an object from an asynchronous operation, it can use `ContinueWith`, in which case the constructor will finish before the object from the asynchronous operation is available. But that's unavoidable.

Saving program settings asynchronously

As you discovered in Chapter 6, "Button clicks," you can save program settings in a dictionary named `Properties` maintained by the `Application` class. Anything you put in the `Properties` dictionary is saved when the program goes into a sleep state and is restored when the program resumes or starts up again. Sometimes it's convenient to save settings in this dictionary as they are changed, and sometimes it's convenient to wait until the `OnSleep` method is called in your `App` class.

There's also another option: The `Application` class has a method named `SavePropertiesAsync` that lets your program take a more proactive role in saving program settings. This allows a program to save program settings whenever it wants to. If the program later crashes or is terminated through the Visual Studio or Xamarin Studio debugger, the settings are saved.

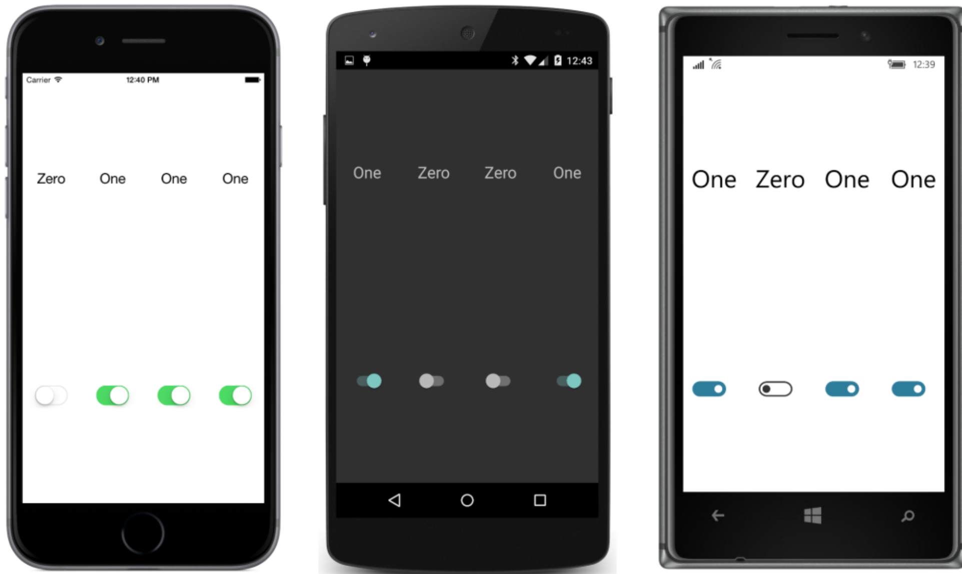
In conformance with recommended practice, the `Async` suffix on the `SavePropertiesAsync` method name identifies this as an asynchronous method. It returns quickly with a `Task` object and saves the settings in a secondary thread of execution.

A program named **SaveProgramSettings** demonstrates this technique. The XAML file contains four `Switch` views and four `Label` views that treat the `Switch` views as digits of a binary number:

```
<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="SaveProgramSettings.SaveProgramSettingsPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:BoolToStringConverter x:Key="boolToString"
                                           FalseText="Zero"
                                           TrueText="One" />
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <Style TargetType="Switch">
                <Setter Property="HorizontalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Grid VerticalOptions="CenterAndExpand">
            <Label Text="{Binding Source={x:Reference s3},
```

```
                               Path=IsToggled,
                               Converter={StaticResource boolToString}"
                   Grid.Column="0" />

            <Label Text="{Binding Source={x:Reference s2},
                               Path=IsToggled,
                               Converter={StaticResource boolToString}"
                   Grid.Column="1" />

            <Label Text="{Binding Source={x:Reference s1},
                               Path=IsToggled,
                               Converter={StaticResource boolToString}"
                   Grid.Column="2" />

            <Label Text="{Binding Source={x:Reference s0},
                               Path=IsToggled,
                               Converter={StaticResource boolToString}"
                   Grid.Column="3" />
        </Grid>

        <Grid x:Name="switchGrid"
              VerticalOptions="CenterAndExpand">
            <Switch x:Name="s3" Grid.Column="0"
                    Toggled="OnSwitchToggled" />

            <Switch x:Name="s2" Grid.Column="1"
                    Toggled="OnSwitchToggled" />

            <Switch x:Name="s1" Grid.Column="2"
                    Toggled="OnSwitchToggled" />

            <Switch x:Name="s0" Grid.Column="3"
                    Toggled="OnSwitchToggled" />
        </Grid>
    </StackLayout>
</ContentPage>
```

The data bindings on the `Label` elements allow them to track the values of the `Switch` views:



The saving and retrieving of program settings is handled in the code-behind file. Notice the handler assigned to the `Toggled` events of the `Switch` elements. The sole purpose of that handler is to store the settings in the `Properties` dictionary—and to save the `Properties` dictionary itself by using `SavePropertiesAsync`—whenever one of the `Switch` elements changes state. The dictionary key is the index of the `Switch` within the `Children` collection of the `Grid`:

```
public partial class SaveProgramSettingsPage : ContentPage
{
    bool isInitialized = false;

    public SaveProgramSettingsPage()
    {
        InitializeComponent();

        // Retrieve settings.
        IDictionary<string, object> properties = Application.Current.Properties;

        for (int index = 0; index < 4; index++)
        {
            Switch switcher = (Switch)(switchGrid.Children[index]);
            string key = index.ToString();

            if (properties.ContainsKey(key))
                switcher.IsToggled = (bool)(properties[key]);
        }
        isInitialized = true;
    }

    async void OnSwitchToggled(object sender, EventArgs args)
    {
        if (!isInitialized)
```

```
            return;

        Switch switcher = (Switch)sender;
        string key = switchGrid.Children.IndexOf(switcher).ToString();
        Application.Current.Properties[key] = switcher.IsToggled;

        // Save settings.
        foreach (View view in switchGrid.Children)
            view.IsEnabled = false;

        await Application.Current.SavePropertiesAsync();

        foreach (View view in switchGrid.Children)
            view.IsEnabled = true;
    }
}
```

One of the purposes of this exercise is to emphasize first, that using `await` doesn't completely solve problems involved with asynchronous operations, but second, that using `await` can help deal with those potential problems.

Here's the problem: The `Toggled` event handler is called every time a `Switch` changes state. It could be that a user toggles a couple of the `Switch` views in succession very quickly. And it could also be the case that the `SavePropertiesAsync` method is slow. Perhaps it saves much more information than four Boolean values. Because this method is asynchronous, there is a danger that it could be called again while it's still working to save the previous collection of settings.

Is `SavePropertiesAsync` reentrant? Can it safely be called again while it's still working? We don't know, and it's better to assume that it's not. For that reason, the handler disables all the `Switch` elements before calling `SavePropertiesAsync` and then reenables them after it's finished. Because `SavePropertiesAsync` returns `Task` rather than `Task<T>`, it's not necessary to use `await` (or `ContinueWith`) to get a value from the method, but it is necessary if you want to execute some code after the method has completed.

In reality, `SavePropertiesAsync` works so fast in this case that it's hard to tell whether this disabling and enabling of the `Switch` views is even working! For testing code such as this, a static method of the `Task` class is very useful. Try inserting this statement right after the `SavePropertiesAsync` call:

```
await Task.Delay(3000);
```

The `Switch` elements are disabled for another 3,000 milliseconds. Of course, if an asynchronous operation really took this long to complete and the user interface is disabled during this time, you'd want to display an `ActivityIndicator` or a `ProgressBar` if possible.

The `Task.Delay` method might seem reminiscent of the `Thread.Sleep` method that you possibly used in some .NET code many years ago. But the two static methods are very different. The `Thread.Sleep` method suspends the current thread, which in this case would be the user-interface thread. That's precisely what you *don't* want. The `Task.Delay` call, however, simulates a do-nothing secondary thread that runs for a specified period of time. The user-interface thread isn't blocked. If you

omit the `await` operator, `Task.Delay` would seemingly have no effect on the program at all. When used with the `await` operator, the code in the method that calls `Task.Delay` resumes after the specified period of time.

A platform-independent timer

So far in this book you've seen two ViewModels that have required timers: These are the `DateTimeViewModel` class used in the `MvvmClock` program in Chapter 18, "MVVM," and the `SchoolViewModel` class in the **SchoolOfFineArt** library, which used the timer to randomly alter the students' grade-point averages for several programs in Chapter 19, "Collection views."

These ViewModels used `Device.StartTimer`, but that's not a good practice. A ViewModel is supposed to be platform independent and usable in any .NET application, but `Device.StartTimer` is specific to Xamarin.Forms.

You can alternatively create your own timer by using `Task.Delay`. Because `Task.Delay` is part of .NET and can be used within Portable Class Libraries, it is much more platform independent than `Device.StartTimer`.

The **TaskDelayClock** demonstrates how to use `Task.Delay` for a timer. The XAML file consists of a `Label` in an `AbsoluteLayout`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TaskDelayClock.TaskDelayClockPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <AbsoluteLayout>
        <Label x:Name="label"
               FontSize="Large"
               AbsoluteLayout.LayoutFlags="PositionProportional" />
    </AbsoluteLayout>
</ContentPage>
```

The code-behind file contains a method called `InfiniteLoop`. Generally, infinite loops are avoided in programming, but this one runs in the user-interface thread for only a very brief period of time four times per second. For the bulk of the time, a `Task.Delay` call allows the user-interface thread to continue to interact with the user:

```
public partial class TaskDelayClockPage : ContentPage
{
    Random random = new Random();

    public TaskDelayClockPage()
    {
        InitializeComponent();
```

```
        InfiniteLoop();
    }

    async void InfiniteLoop()
    {
        while (true)
        {
            label.Text = DateTime.Now.ToString("T");
            label.FontSize = random.Next(12, 49);
            AbsoluteLayout.SetLayoutBounds(label, new Rectangle(random.NextDouble(),
                                                                random.NextDouble(),
                                                                AbsoluteLayout.AutoSize,
                                                                AbsoluteLayout.AutoSize));
            await Task.Delay(250);
        }
    }
}
```

Every 250 milliseconds, the code in the `while` loop runs to give the `Label` the current time, but also to randomly change its font size and its location within the `AbsoluteLayout`:

Yes, it's a rather annoying clock.

This is not truly an "infinite" loop, of course, but it will keep going until the application terminates. If you prefer, you can use a Boolean field as the `while` conditional and exit from the loop by just setting the field to `false`.

Notice how the `InfiniteLoop` method is simply called from the constructor as if it were a normal method. If this method used `Thread.Sleep` rather than `Task.Delay`, it would never return back to

the constructor, and the constructor would never finish, and that would not be good at all. This particular `InfiniteLoop` method returns back to the constructor when execution hits the `await` operator for the first time, and the constructor can finish execution. The program can do anything else it wants, but the user-interface thread will be required every 250 milliseconds when `InfiniteLoop` resumes.

Although the `Task.Delay` call simulates a do-nothing secondary thread, it's actually implemented using the `Timer` class from the `System.Threading` namespace. Curiously enough, that `Timer` class is not available in a Xamarin.Forms Portable Class Library, and if it were, it would be a little more difficult to use because the timer callback doesn't run in the user-interface thread.

File input/output

Traditionally, file input/output is one of the most basic programming tasks, but file I/O on mobile devices is a little different from that on the desktop. On the desktop, users and applications generally have access to an entire disk and perhaps additional drives, all of which are organized into directory structures. On mobile devices, several standard folders exist—for pictures or music, for example—but application-specific data is generally restricted to a storage area that is private to each application.

Programmers familiar with .NET know that the `System.IO` namespace contains the bulk of standard file I/O support. This is where you'll find the crucial `Stream` class that provides the basis of reading and writing data organized as a stream of bytes. Building upon this are several `Reader` and `Writer` classes and other classes that allow accessing files and directories. Perhaps the handiest of the file classes is `File` itself, which not only provides a collection of methods to create new files and open existing files but also includes several static methods capable of performing an entire file-read or file-write operation in a single method call.

Particularly if you're working with text files, these static methods of the `File` class can be very convenient. For example, the `File.WriteAllText` method has two arguments of type `string`—a filename and the file contents. The method creates the file (replacing an existing file with the same name if necessary), writes the contents to the file, and then closes it. The `File.ReadAllText` method is similar but returns the contents of the file in one big `string` object. These methods are ideal for writing and reading text files with a minimum of fuss.

At first, file I/O doesn't seem to require asynchronous operations, and in practice, sometimes you have a choice, and sometimes you can avoid asynchronous operations if you want to.

However, other times you do not have a choice. Some platforms require asynchronous functions for file I/O, and even when they're not required, it makes sense to avoid doing file I/O in the user-interface thread.

Good news and bad news

The Xamarin.iOS and Xamarin.Android libraries referenced by your Xamarin.Forms applications include

a version of .NET that Xamarin has expressly tailored for these two mobile platforms. The methods in the `File` class in the `System.IO` namespace map to appropriate file I/O functions in the iOS and Android platforms, and the static `Environment.GetFolderPath` method, when used with the `MyDocuments` enumeration member, returns a directory for the application's local storage. This means that you can use simple methods in the `File` class—including the static methods that perform entire file writing or reading operations in a single call—in your iOS and Android applications.

To verify the availability of these classes, let's experiment a little: Go into Visual Studio or Xamarin Studio and load any Xamarin.Forms solution created so far. Bring up one of the code files in the iOS or Android project. In a constructor or method, type the `System.IO` namespace name and then a period. You'll get a list of all the available types in the namespace. If you then type `File` and a period, you'll get all the static methods in the `File` class, including `WriteAllText` and `ReadAllText`.

In the Windows 8.1 and Windows Phone 8.1 projects, however, you're working with a version of .NET created by Microsoft specifically for these platforms. If you type `System.IO` and a period, you won't even see the `File` class at all! It doesn't exist! (However, you'll discover that it does exist in the UWP project.)

Now go into any code file in a Xamarin.Forms Portable Class Library project. As you'll recall, a PCL for Xamarin.Forms targets the following platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone 8.1
- Xamarin.Android
- Xamarin.iOS
- Xamarin.iOS (Classic)

As you might have already anticipated, the `System.IO` namespace in a PCL is also missing the `File` class. PCLs are configured to support multiple target platforms. Consequently, the APIs implemented within the PCL are necessarily an intersection of the APIs in these target platforms.

Beginning with Windows 8 and the Windows Runtime API, Microsoft completely revamped file I/O and created a whole new set of classes. Your Windows 8.1, Windows Phone 8.1, and UWP applications instead use classes in the `Windows.Storage` namespace for file I/O.

If you are targeting only iOS and Android in your Xamarin.Forms applications, you can share file I/O code between the two platforms. You can use the static `File` methods and everything else in `System.IO`.

If you also want to target one of the Windows or Windows Phone platforms, you'll want to make use of `DependencyService` (discussed in Chapter 9, "Platform-specific API calls") for different file I/O logic for each of the platforms.

A first shot at cross-platform file I/O

In the general case, you'll use DependencyService to give your Xamarin.Forms applications access to file I/O functions. As you know from the previous explorations into DependencyService, you can define the functions you want in an interface in the Portable Class Library project, while the code to implement these functions resides in separate classes in the individual platforms.

The file I/O functions developed in this chapter will be put to a good use in the **NoteTaker** application in Chapter 24, "Page navigation." For a first shot at file I/O, let's work with a much simpler solution, named **TextFileTryout**, that implements several functions to work with text files. Let's also restrict ourselves to getting this program running on iOS and Android and forget about the Windows platforms for the moment.

The first step in making use of DependencyService is creating an interface in the PCL that defines all the methods you'll need. Here is such an interface in the **TextFileTryout** project, named IFileHelper:

```
namespace TextFileTryout
{
    public interface IFileHelper
    {
        bool Exists(string filename);

        void WriteText(string filename, string text);

        string ReadText(string filename);

        IEnumerable<string> GetFiles();

        void Delete(string filename);
    }
}
```

The interface defines functions to determine whether a file exists, to write and read entire text files in one shot, to enumerate all the files created by the application, and to delete a file. In each platform implementation, these functions are restricted to the private file area associated with the application.

You then implement this interface in each of the platforms. Here's the FileHelper class in the iOS project, complete with using directives and the required Dependency attribute:

```
using System;
using System.Collections.Generic;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(TextFileTryout.iOS.FileHelper))]

namespace TextFileTryout.iOS
{
    class FileHelper : IFileHelper
    {
```

```
        public bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public void WriteText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public string ReadText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        public IEnumerable<string> GetFiles()
        {
            return Directory.GetFiles(GetDocsPath());
        }

        public void Delete(string filename)
        {
            File.Delete(GetFilePath(filename));
        }

        // Private methods.
        string GetFilePath(string filename)
        {
            return Path.Combine(GetDocsPath(), filename);
        }

        string GetDocsPath()
        {
            return Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        }
    }
}
```

It is essential that this class explicitly implements the `IFileHelper` interface and includes a `Dependency` attribute with the name of the class. These allow the `DependencyService` class in Xamarin.Forms to find this implementation of `IFileHelper` in the platform project. Two private methods at the bottom allow the program to construct a fully qualified filename using the directory of the application's private storage available from the `Environment.GetFolderPath` method.

In both Xamarin.iOS and Xamarin.Android, the implementation of `Environment.GetFolderPath` obtains the platform-specific area of the application's local storage, although the directory names that the method returns for the two platforms are very different.

As a result, the `FileHelper` class in the Android project is exactly the same as the one in the iOS

project apart from the different namespace names.

The iOS and Android versions of `FileHelper` make use of the static shortcut methods in the `File` class and a simple static method of `Directory` for obtaining all the files stored with the application. However, the implementation of `IFileHelper` in the Windows 8.1 and Windows Phone 8.1 projects can't use the shortcut methods in the `File` class because they are not available, and the `Environment.GetFolderPath` method isn't available in the UWP project.

Moreover, applications written for these Windows platforms should instead use file I/O functions implemented in the Windows Runtime API. Because the file I/O functions in the Windows Runtime are asynchronous, they do not fit into the interface established by the `IFileHelper` interface. For that reason, the version of `FileHelper` in the three Windows projects is forced to leave the crucial methods unimplemented. Here's the version in the **UWP** project:

```
using System;
using System.Collections.Generic;
using Xamarin.Forms;

[assembly: Dependency(typeof(TextFileTryout.UWP.FileHelper))]

namespace TextFileTryout.UWP
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            return false;
        }

        public void WriteText(string filename, string text)
        {
            throw new NotImplementedException("Writing files is not implemented");
        }

        public string ReadText(string filename)
        {
            throw new NotImplementedException("Reading files is not implemented");
        }

        public IEnumerable<string> GetFiles()
        {
            return new string[0];
        }

        public void Delete(string filename)
        {
        }
    }
}
```

The version of `FileHelper` in the Windows 8.1 and Windows Phone 8.1 projects is identical except for the namespace name.

Normally, an application needs to reference the methods in each platform by using the `DependencyService.Get` method. However, the **TextFileTryout** program has made things easy for itself by defining a class named `FileHelper` in the PCL project that also implements `IFileHelper`, but incorporates the call to the `Get` method of `DependencyService` to call the platform versions of these methods:

```
namespace TextFileTryout
{
    class FileHelper : IFileHelper
    {
        IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public bool Exists(string filename)
        {
            return fileHelper.Exists(filename);
        }

        public void WriteText(string filename, string text)
        {
            fileHelper.WriteText(filename, text);
        }

        public string ReadText(string filename)
        {
            return fileHelper.ReadText(filename);
        }

        public IEnumerable<string> GetFiles()
        {
            IEnumerable<string> filepaths = fileHelper.GetFiles();
            List<string> filenames = new List<string>();

            foreach (string filepath in filepaths)
            {
                filenames.Add(Path.GetFileName(filepath));
            }
            return filenames;
        }

        public void Delete(string filename)
        {
            fileHelper.Delete(filename);
        }
    }
}
```

Notice that the `GetFiles` method performs a little surgery on the filenames returned from the platform implementation. The filenames that are obtained from the platform implementations of `GetFiles` are fully qualified, and while it might be interesting to see the folder names that iOS and Android use for application local storage, those filenames are going to be displayed in a `ListView` where the folder names will just be a distraction, so this `GetFiles` method strips off the file path.

The `TextFileTryoutPage` class tests these functions. The XAML file includes an `Entry` for a filename, an `Editor` for the file contents, a `Button` labeled "Save", and a `ListView` with all the previously saved filenames:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TextFileTryout.TextFileTryoutPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

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

        <Entry x:Name="filenameEntry"
               Grid.Row="0"
               Placeholder="filename" />

        <Editor x:Name="fileEditor"
                Grid.Row="1">
            <Editor.BackgroundColor>
                <OnPlatform x:TypeArguments="Color"
                            WinPhone="#D0D0D0" />
            </Editor.BackgroundColor>
        </Editor>

        <Button x:Name="saveButton"
                Text="Save"
                Grid.Row="2"
                HorizontalOptions="Center"
                Clicked="OnSaveButtonClicked" />

        <ListView x:Name="fileListView"
                  Grid.Row="3"
                  ItemSelected="OnFileListViewItemSelected">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding}">
                        <TextCell.ContextActions>
                            <MenuItem Text="Delete"
                                      IsDestructive="True"
                                      Clicked="OnDeleteMenuItemClicked" />
                        </TextCell.ContextActions>
                    </TextCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </Grid>
```

```
</ContentPage>
```

Just to keep things simple, all processing is performed in the code-behind file without a ViewModel. The code-behind file implements all the event handlers from the XAML file. The **Save** button checks whether the file exists first and displays an alert box if it does. Selecting one of the files in the `ListView` loads it in. In addition, the `ListView` implements a context menu to delete a file. All the file I/O functions are methods of the `FileHelper` class defined in the PCL and instantiated as a field at the top of the class:

```
public partial class TextFileTryoutPage : ContentPage
{
    FileHelper fileHelper = new FileHelper();

    public TextFileTryoutPage()
    {
        InitializeComponent();

        RefreshListView();
    }

    async void OnSaveButtonClicked(object sender, EventArgs args)
    {
        string filename = filenameEntry.Text;

        if (fileHelper.Exists(filename))
        {
            bool okResponse = await DisplayAlert("TextFileTryout",
                                                 "File " + filename +
                                                 " already exists. Replace it?",
                                                 "Yes", "No");
            if (!okResponse)
                return;
        }

        string errorMessage = null;

        try
        {
            fileHelper.WriteText(filenameEntry.Text, fileEditor.Text);
        }
        catch (Exception exc)
        {
            errorMessage = exc.Message;
        }

        if (errorMessage == null)
        {
            filenameEntry.Text = "";
            fileEditor.Text = "";
            RefreshListView();
        }
        else
        {
```

```
            await DisplayAlert("TextFileTryout", errorMessage, "OK");
        }
    }

    async void OnFileListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem == null)
            return;

        string filename = (string)args.SelectedItem;
        string errorMessage = null;

        try
        {
            fileEditor.Text = fileHelper.ReadText((string)args.SelectedItem);
            filenameEntry.Text = filename;
        }
        catch (Exception exc)
        {
            errorMessage = exc.Message;
        }

        if (errorMessage != null)
        {
            await DisplayAlert("TextFileTryout", errorMessage, "OK");
        }
    }

    void OnDeleteMenuItemClicked(object sender, EventArgs args)
    {
        string filename = (string)((MenuItem)sender).BindingContext;
        fileHelper.Delete(filename);
        RefreshListView();
    }

    void RefreshListView()
    {
        fileListView.ItemsSource = fileHelper.GetFiles();
        fileListView.SelectedItem = null;
    }
}
```

The code-behind file calls `DisplayAlert` with the `await` operator on three occasions: The **Save** button uses `DisplayAlert` if the filename you specify already exists. This confirms that your real intention is to replace an existing file. The other two uses are for notification purposes for errors that occur when files are saved or loaded. The file save and file load operations are in `try` and `catch` blocks to catch any errors that might occur. The file save operation will fail for an illegal filename, for example. It is less likely that an error will be encountered on reading a file, but the program checks anyway.

The alerts that notify the user of an error could conceivably be displayed without the `await` operator, but they use `await` anyway to demonstrate a basic principle involved in exception handling: Although C# 6 allows using `await` in a `catch` block, C# 5 does not. To get around this restriction, the

catch block simply saves the error message in a variable called errorMessage, and then the code following the catch block uses DisplayAlert to display that text if it exists. This structure allows these event handlers to conclude with different processing depending on successful completion or an error.

Notice also that the constructor concludes with a call to RefreshListView to display all the existing files in the ListView, and the code-behind file also calls that method when a new file has been saved or a file has been deleted.

However, this program does not work on the Windows platforms. Let's fix that.

Accommodating Windows Runtime file I/O

The Windows Runtime API defined a whole new array of file I/O classes. Part of the impetus for this was the recognition of an industry-wide transition away from the relatively unconstrained file access of desktop applications toward a more sandboxed environment.

Much of the new file I/O API can be found in the Windows Runtime namespaces Windows.Storage and Windows.Storage.Streams. To store data that is private to an application, a Windows Runtime program first gets a special StorageFolder object:

```
StorageFolder localFolder = ApplicationData.Current.LocalFolder;
```

ApplicationData defines a static property named Current that returns the ApplicationData object for the application. LocalFolder is an instance property of ApplicationData.

StorageFolder defines methods named CreateFileAsync to create a new file and GetFileAsync to open an existing file. These two methods obtain objects of type StorageFile. With that object, a program can open the file for writing or reading with OpenAsync or OpenReadAsync. These methods obtain an IRandomAccessStream object. From this, DataWriter or DataReader objects are created to perform write or read operations.

This sounds a bit lengthy, and it is. Rather simpler approaches involve static methods of the FileIO class, which are similar to the static methods of the .NET File class. For text files, for example, FileIO.ReadTextAsync and FileIO.WriteTextAsync open a file, perform the read or write access, and close the file in one shot. The first argument to these methods is a StorageFile object.

At any rate, by this time you've undoubtedly noticed the frequent Async suffixes on these method names. Internally, all these methods spin off secondary threads of execution for doing the actual work and return quickly to the caller. The work takes place in the background, and the caller is notified of completion (or error) through callback functions.

Why is this?

When Windows 8 was first being created, the Microsoft developers took a good, hard look at timing and decided that any function call that requires more than 50 milliseconds to execute should be made asynchronous so that it would not interfere with the responsiveness of the user interface. APIs that require more than 50 milliseconds obviously include the file I/O functions, which often need to access

potentially slow pieces of hardware like disk drives or a network. Any Windows Runtime file I/O method that could possibly cause a physical storage device to be accessed was made asynchronous and given an `Async` suffix.

However, these asynchronous methods do *not* return `Task` objects. In the Windows Runtime, methods that return data have return types of `IAsyncOperation<TResult>`, while methods that do not return information have return types of `IAsyncAction`. These interfaces can all be found in the `System.Foundations` namespace.

Although these interfaces are not the same as `Task` and `Task<T>`, they are similar, and you can use `await` with them. You can also convert between the two asynchronous protocols. The **System.Runtime.WindowsRuntime** assembly includes a `System` namespace with a `WindowsRuntimeSystemExtensions` class that has extension methods named `AsAsyncAction`, `AsAsyncOpertion`, and `AsTask` that perform these conversions.

Let's rework the **TextFileTryout** program to accommodate asynchronous file I/O. The revised program is called **TextFileAsync and is developed in the next section**. Because asynchronous file I/O functions in the Windows projects will be accessed, all the file functions in the `IFileHelper` interface are defined to return `Task` or `Task<T>` objects.

Platform-specific libraries

Every programmer knows that potentially reusable code should be put in a library, and this is also the case for code used with dependency services. The asynchronous file I/O functions developed here will be reused in the **NoteTaker** program in Chapter 24, and you might want to use these functions in your own applications or perhaps develop your own functions.

However, these file I/O classes can't be put in just one library. Each of the various platform implementations of `FileHelper` must be in a library for that specific platform. This requires separate libraries for each platform.

The **Libraries** directory of the downloadable code for this book contains a solution named **Xamarin.FormsBook.Platform**. The **Platform** part of the name was inspired by the various **Xamarin.Forms.Platform** libraries. Each of the various platforms is a separate library in this solution.

The **Xamarin.FormsBook.Platform** solution contains no fewer than seven library projects, each of which was created somewhat differently:

- **Xamarin.FormsBook.Platform** is a normal Xamarin.Forms Portable Class Library with a profile of 111, which means that it can be accessed by all the platforms. You can create such a library in Visual Studio by selecting **Cross Platform** at the left of the **Add New Project** dialog, and **Class Library (Xamarin.Forms)** in the central area. In the Xamarin Studio **New Project** dialog, select **Multiplatform** and **Library** at the left, and **Xamarin.Forms** and **Class Library** in the central area.

- **Xamarin.FormsBook.Platform.iOS** was created in Visual Studio by selecting **iOS** in the left column of the **Add New Project** dialog, and **Class Library (iOS)** in the central section. In Xamarin Studio select **iOS** and **Library** in the **New Project** dialog, and **Class Library** in the central area.
- **Xamarin.FormsBook.Platform.Android** was created in Visual Studio by selecting **Android** at the left of the **Add New Project** dialog and **Class Library (Android)** in the central section. In Xamarin.Studio, select **Android** and **Library** at the left and **Class Library** in the central section.
- **Xamarin.FormsBook.Platform.UWP** is a library for Windows 10 and Windows 10 Mobile. It was created in Visual Studio by selecting **Windows** and **Universal** at the left, and then **Class Library (Universal Windows)**.
- **Xamarin.FormsBook.Platform.Windows** is a Portable Class Library just for Windows 8.1. It was created in Visual Studio by selecting **Windows**, **Windows 8**, and **Windows** at the left, and then **Class Library (Windows 8.1)**.
- **Xamarin.FormsBook.Platform.WinPhone** is a Portable Class Library just for Windows Phone 8.1. It was created in Visual Studio by selecting **Windows**, **Windows 8**, and **Windows Phone** at the left, and then **Class Library (Windows Phone)**.
- You'll often find that the three Windows platforms can share code because they all use variants of the Windows Runtime API. For this reason, a seventh project was created named **Xamarin.FormsBook.Platform.WinRT**. This is a shared project, and it was created in Visual Studio by searching for "Shared" in the **Add New Project** dialog, and selecting the **Shared Project** for C#.

If you're creating such a solution yourself, you'll also need to use the **Manage Packages for Solution** dialog to install the appropriate Xamarin.Forms NuGet packages for all these libraries.

You'll also need to establish references between the various projects in the solution. All the individual platform projects (with the exception of **Xamarin.FormsBook.Platform.WinRT**) need a reference to **Xamarin.FormsBook.Platform**. You set these references in the **Reference Manager** dialog by selecting **Solution** at the left. In addition, the three Windows projects (**UWP**, **Windows**, and **WinPhone**) all need references to the shared **Xamarin.FormsBook.Platform.WinRT** project. You set these references in the **Reference Manager** dialog by selecting **Shared Projects** at the left.

All the projects have a static `Toolkit.Init` method. Here's the one in the Xamarin.FormsBook.Platform library:

```
namespace Xamarin.FormsBook.Platform
{
    public static class Toolkit
    {
        public static void Init()
        {
        }
    }
```

```
}
```

Most of the others are similar except that the version in the Android library actually saves some information that might be useful to classes implemented in this library:

```
namespace Xamarin.FormsBook.Platform.Android
{
    public static class Toolkit
    {
        public static void Init(Activity activity, Bundle bundle)
        {
            Activity = activity;
        }

        public static Activity Activity { private set; get; }
    }
}
```

The `Toolkit.Init` method in each of the Windows platforms calls a do-nothing `Toolkit.Init` method in the shared **Xamarin.FormsBook.Platform.WinRT** project:

```
namespace Xamarin.FormsBook.Platform.UWP
{
    public static class Toolkit
    {
        public static void Init()
        {
            Xamarin.FormsBook.Platform.WinRT.Toolkit.Init();
        }
    }
}
```

The purpose of these methods is to ensure that the libraries are bound to the application even if the application does not directly access anything in the library. It is very often the case when you're working with dependency services and custom renderers that the application does not directly call any library function. However, if you later discover that you really do need to perform some library initialization, the method already exists for you to do so.

You'll discover that the version of the **Xamarin.FormsBook.Platform** libraries included with the downloadable code for this book already includes the `PlatformSoundPlayer` classes from Chapter 9, "Platform-specific API calls." You'll also see some classes beginning with the words `Ellipse` and `StepSlider`. These are discussed in Chapter 27, "Custom renderers."

Let's focus on the new asynchronous `FileHelper` classes. The **Xamarin.FormsBook.Platform** library contains the new `IFileHelper` interface:

```
using System.Collections.Generic;
using System.Threading.Tasks;

namespace Xamarin.FormsBook.Platform
{
    public interface IFileHelper
```

```
    {
        Task<bool> ExistsAsync(string filename);

        Task WriteTextAsync(string filename, string text);

        Task<string> ReadTextAsync(string filename);

        Task<IEnumerable<string>> GetFilesAsync();

        Task DeleteAsync(string filename);
    }
}
```

By convention, methods that return `Task` objects have a suffix of `Async`.

All three Windows platforms can share the same `FileHelper` class, so this shared class is implemented in the shared **Xamarin.FormsBook.Platform.WinRT** project. Each of the five methods in the `FileHelper` class begins with a call to obtain the `StorageFolder` associated with the application's local storage area. Each of them makes asynchronous calls using `await` and is flagged with the `async` keyword:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Windows.Storage;
using Xamarin.Forms;

[assembly: Dependency(typeof(Xamarin.FormsBook.Platform.WinRT.FileHelper))]

namespace Xamarin.FormsBook.Platform.WinRT
{
    class FileHelper : IFileHelper
    {
        public async Task<bool> ExistsAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;

            try
            {
                await localFolder.GetFileAsync(filename);
            }
            catch
            {
                return false;
            }
            return true;
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.CreateFileAsync(filename,
```

```
                                                    CreationCollisionOption.ReplaceExisting);
            await FileIO.WriteTextAsync(storageFile, text);
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.GetFileAsync(filename);
            return await FileIO.ReadTextAsync(storageFile);
        }

        public async Task<IEnumerable<string>> GetFilesAsync()
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;

            IEnumerable<string> filenames =
                from storageFile in await localFolder.GetFilesAsync()
                select storageFile.Name;

            return filenames;
        }

        public async Task DeleteAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            StorageFile storageFile = await localFolder.GetFileAsync(filename);
            await storageFile.DeleteAsync();
        }
    }
}
```

Although each of the methods is defined as returning a `Task` or a `Task<T>` object, the bodies of the methods don't have any reference to `Task` or `Task<T>`. Instead, the methods that return a `Task` object simply do some work and then end the method with an implicit `return` statement. The `ExistsAsync` method is defined as returning a `Task<bool>` but returns either `true` or `false`. (There is no `Exists` method in the `StorageFolder` class, so a workaround with `try` and `catch` is necessary.)

Similarly, the `ReadTextAsync` method is defined as returning a `Task<string>`, but the body returns a `string`, which is obtained from applying the `await` operator to the `IAsyncOperation<string>` return value of `File.ReadTextAsync`. The C# compiler performs the necessary conversions.

When a program calls this `ReadTextAsync` method, the method executes until the first `await` operator, and then it returns a `Task<string>` object to the caller. The caller can use either `ContinueWith` or `await` to obtain the string when the `FileIO.ReadTextAsync` method has completed.

For iOS and Android, however, we now have a problem. All the methods in `IFileHelper` are now defined as asynchronous methods that return `Task` or `Task<T>` objects, but we've already seen that the methods in the `System.IO` namespace are not asynchronous. What do we do?

The `FileHelper` class in the iOS namespace uses two strategies. In some cases, the `System.IO`

classes *do* include asynchronous methods. This is the case for the `WriteAsync` method of `StreamWriter` and the `ReadAsync` method of `StreamReader`. For the other methods, however, a static `FromResult` method of `Task<T>` is used to convert an object or value to a `Task<T>` object for the method return value. This does not actually convert the method to an asynchronous method, but simply allows the method to have the signature of an asynchronous method:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Xamarin.Forms;

[assembly: Dependency(typeof(Xamarin.FormsBook.Platform.iOS.FileHelper))]

namespace Xamarin.FormsBook.Platform.iOS
{
    class FileHelper : IFileHelper
    {
        public Task<bool> ExistsAsync(string filename)
        {
            string filepath = GetFilePath(filename);
            bool exists = File.Exists(filepath);
            return Task<bool>.FromResult(exists);
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            using (StreamWriter writer = File.CreateText(filepath))
            {
                await writer.WriteAsync(text);
            }
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            string filepath = GetFilePath(filename);
            using (StreamReader reader = File.OpenText(filepath))
            {
                return await reader.ReadToEndAsync();
            }
        }

        public Task<IEnumerable<string>> GetFilesAsync()
        {
            // Sort the filenames.
            IEnumerable<string> filenames =
                from filepath in Directory.EnumerateFiles(GetDocsFolder())
                select Path.GetFileName(filepath);

            return Task<IEnumerable<string>>.FromResult(filenames);
        }
```

```
        public Task DeleteAsync(string filename)
        {
            File.Delete(GetFilePath(filename));
            return Task.FromResult(true);
        }

        string GetDocsFolder()
        {
            return Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        }

        string GetFilePath(string filename)
        {
            return Path.Combine(GetDocsFolder(), filename);
        }
    }
}
```

The Android `FileHelper` class is the same as the iOS class but with a different namespace.

Notice that the only error checking within these platform implementations is for the `ExistsAsync` method in the Windows Runtime platforms, which uses the exception to determine whether the file exists or not. None of the other methods—and particularly the `WriteTextAsync` and `ReadTextAsync` methods—is performing any error checking. One of the nice features of using `await` is that any exception can be caught at a later time when you're actually calling these methods.

You might also have noticed that the individual `GetFilesAsync` methods are now removing the path from the fully qualified filename, so that job doesn't need to be performed by the `FileHelper` class in the **Xamarin.FormsBook.Platform** project:

```
namespace Xamarin.FormsBook.Platform
{
    class FileHelper
    {
        IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public Task<bool> ExistsAsync(string filename)
        {
            return fileHelper.ExistsAsync(filename);
        }

        public Task WriteTextAsync(string filename, string text)
        {
            return fileHelper.WriteTextAsync(filename, text);
        }

        public Task<string> ReadTextAsync(string filename)
        {
            return fileHelper.ReadTextAsync(filename);
        }

        public Task<IEnumerable<string>> GetFilesAsync()
```

```
        {
            return fileHelper.GetFilesAsync();
        }

        public Task DeleteAsync(string filename)
        {
            return fileHelper.DeleteAsync(filename);
        }
    }
}
```

Now that we have a library, we need to access this library from an application. The **TextFileAsync** solution was created normally. Then, all seven projects in the **Xamarin.FormsBook.Platform** solution were added to this solution. These projects must be added separately by using the **Add** and **Existing Project** menu item for the solution. There is no **Add All Projects from Solution** menu item, but if you use these libraries in your own projects, you'll wish there were!

At this point, the **TextFileAsync** solution contains 13 projects: Five application projects, a shared PCL with the application code, and seven library projects.

References must be established between these projects by using the **Reference Manager** for the following relationships:

- **TextFileAsync** has a reference to **Xamarin.FormsBook.Platform**.
- **TextFileAsync.iOS** has a reference to **Xamarin.FormsBook.Platform.iOS**.
- **TextFileAsync.Droid** has a reference to **Xamarin.FormsBook.Platform.Android**.
- **TextFileAsync.UWP** has a reference to **Xamarin.FormsBook.Platform.UWP**.
- **TextFileAsync.Windows** has a reference to **Xamarin.FormsBook.Platform.Windows**.
- **TextFileAsync.WinPhone** has a reference to **Xamarin.FormsBook.Platform.WinPhone**.

Of course, all the application projects have normal references to the **TextFileAsync** PCL, and, as you'll recall, the **Xamarin.FormsBook.Platform.UWP**, **Windows**, and **WinPhone** projects all have references to the shared **Xamarin.FormsBook.Platform.WinRT** project.

Also, all the **TextFileAsync** projects should make calls to the various `Toolkit.Init` methods in the libraries. In the `TextFileAsync` project itself, make the call in the constructor of the `App` class:

```
namespace TextFileAsync
{
    public class App : Application
    {
        public App()
        {
            Xamarin.FormsBook.Platform.Toolkit.Init();
            …
        }
        …
```

```
    }
}
```

In the iOS project, make the call after the normal `Forms.Init` call in the `AppDelegate` class:

```
namespace TextFileAsync.iOS
{
    …
    public partial class AppDelegate :
                            global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        …
        public override bool FinishedLaunching(UIApplication app, NSDictionary options)
        {
            global::Xamarin.Forms.Forms.Init();
            Xamarin.FormsBook.Platform.iOS.Toolkit.Init();
            LoadApplication(new App());
            …
        }
    }
}
```

In the Android project, call `Toolkit.Init` with the `MainActivity` and `Bundle` objects in the `MainActivity` class after the normal `Forms.Init` call:

```
namespace TextFileAsync.Droid
{
    …
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsApplicationActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            …
            global::Xamarin.Forms.Forms.Init(this, bundle);
            Xamarin.FormsBook.Platform.Android.Toolkit.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}
```

In the three Windows platforms, call `Toolkit.Init` right after `Forms.Init` in the App.xaml.cs file:

```
namespace TextFileAsync.UWP
{
    …
    sealed partial class App : Application
    {
        …
                Xamarin.Forms.Forms.Init(e);
                Xamarin.FormsBook.Platform.UWP.Toolkit.Init();
        …
    }
}
```

With that overhead out of the way, the actual writing of the application can begin. The XAML file

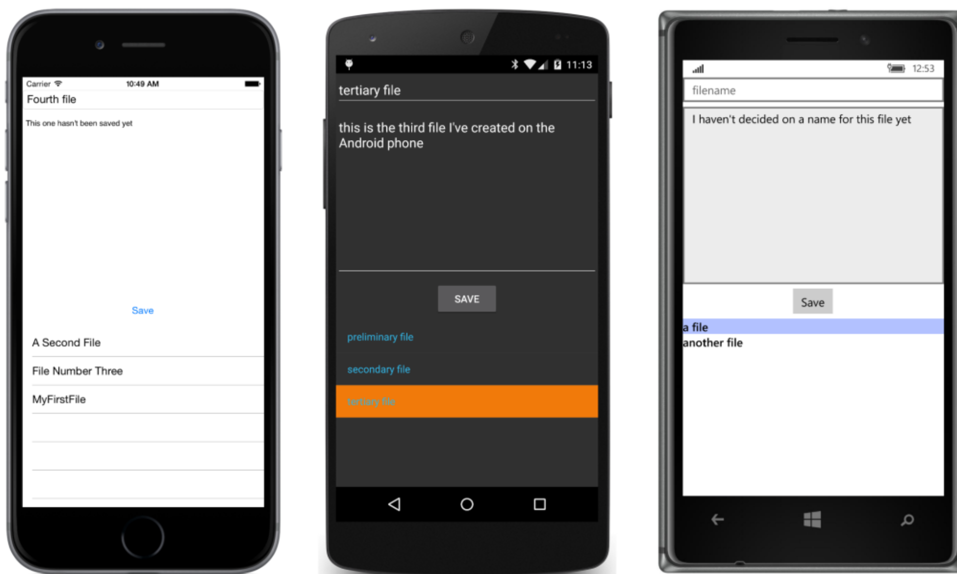
for `TextFileAsyncPage` is the same as `TextFileTryoutPage`, but the code-behind file must be fashioned to work with the asynchronous file I/O methods. Any exceptions that might occur in the file I/O functions must be caught here, which means that any method that can throw an exception must be in a `try` block along with the `await` operator:

```
public partial class TextFileAsyncPage : ContentPage
{
    FileHelper fileHelper = new FileHelper();

    public TextFileAsyncPage()
    {
        InitializeComponent();

        RefreshListView();
    }

    async void OnSaveButtonClicked(object sender, EventArgs args)
    {
        saveButton.IsEnabled = false;

        string filename = filenameEntry.Text;

        if (await fileHelper.ExistsAsync(filename))
        {
            bool okResponse = await DisplayAlert("TextFileTryout",
                                                 "File " + filename +
                                                 " already exists. Replace it?",
                                                 "Yes", "No");
            if (!okResponse)
                return;
        }

        string errorMessage = null;

        try
        {
            await fileHelper.WriteTextAsync(filenameEntry.Text, fileEditor.Text);
        }
        catch (Exception exc)
        {
            errorMessage = exc.Message;
        }

        if (errorMessage == null)
        {
            filenameEntry.Text = "";
            fileEditor.Text = "";
            RefreshListView();
        }
        else
        {
            await DisplayAlert("TextFileTryout", errorMessage, "OK");
        }
```

```
        saveButton.IsEnabled = true;
    }

    async void OnFileListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
    {
        if (args.SelectedItem == null)
            return;

        string filename = (string)args.SelectedItem;
        string errorMessage = null;

        try
        {
            fileEditor.Text = await fileHelper.ReadTextAsync((string)args.SelectedItem);
            filenameEntry.Text = filename;
        }
        catch (Exception exc)
        {
            errorMessage = exc.Message;
        }

        if (errorMessage != null)
        {
            await DisplayAlert("TextFileTryout", errorMessage, "OK");
        }
    }

    async void OnDeleteMenuItemClicked(object sender, EventArgs args)
    {
        string filename = (string)((MenuItem)sender).BindingContext;
        await fileHelper.DeleteAsync(filename);
        RefreshListView();
    }

    async void RefreshListView()
    {
        fileListView.ItemsSource = await fileHelper.GetFilesAsync();
        fileListView.SelectedItem = null;
    }
}
```

The result is that this code is structured very much like the previous code that used the synchronous file I/O functions. One difference, however, is that the `OnSaveButtonClicked` method disables the **Save** button when beginning processing and then reenables it when everything is finished. This is simply to prevent multiple presses of the **Save** button that might cause multiple overlapping calls to `FileIO.WriteFileAsync`.

Here's the program running on the three platforms:



Keeping it in the background

Some of the `FileHelper` methods in the Windows Runtime implementation have multiple `await` operators to deal with a series of asynchronous calls. This makes sense: Each step in the process must complete before the next step executes. However, one of the characteristics of `await` is that it resumes execution on the same thread that it was invoked on rather than the background thread. This is often convenient when you are obtaining a result to update the user interface. However, within the methods in the `FileHelper` implementations, this isn't necessary. Everything within the body of the `WriteTextAsync` and `ReadTextAsync` methods can occur in a secondary thread.

The `Task` class has a method named `ConfigureAwait` that can control which thread `await` resumes on. If you pass a `false` argument to `ConfigureAwait`, the completed task will resume on the same worker thread used to implement the function. If you'd like to use this in the `FileHelper` code, you'll need to convert the `IAsyncAction` and `IAsyncOperation` objects returned by the Windows Runtime methods to tasks by using `AsTask` and then call `ConfigureAwait` on that `Task` object.

For example, here's how the `WriteTextAsync` and `ReadTextAsync` methods are implemented in the existing **Xamarin.FormsBook.Platform.WinRT** project:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    class FileHelper : IFileHelper
    {
        …
        public async Task WriteTextAsync(string filename, string text)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
```

```
            IStorageFile storageFile = await localFolder.CreateFileAsync(filename,
                                                    CreationCollisionOption.ReplaceExisting);
            await FileIO.WriteTextAsync(storageFile, text);
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.GetFileAsync(filename);
            return await FileIO.ReadTextAsync(storageFile);
        }
        …
    }
}
```

These methods have two `await` operators each. To make these methods slightly more efficient, you can use `AsTask` and `ConfigureAwait` to change them to these:

```
namespace Xamarin.FormsBook.Platform.WinRT
{
    class FileHelper : IFileHelper
    {
    …
        public async Task WriteTextAsync(string filename, string text)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.CreateFileAsync(filename,
                                                    CreationCollisionOption.ReplaceExisting).
                                                    AsTask().ConfigureAwait(false);
            await FileIO.WriteTextAsync(storageFile, text).AsTask().ConfigureAwait(false);
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.GetFileAsync(filename).
                                                AsTask().ConfigureAwait(false);
            return await FileIO.ReadTextAsync(storageFile).AsTask().ConfigureAwait(false);
        }
        …
    }
}
```

Now the methods following the first `await` operator run in worker threads, and `await` doesn't need to switch back to the user-interface thread just to continue with the method. The switch back to the user-interface thread occurs when `await` is used to call these methods from `TextFileAsyncPage`.

You probably want to restrict this technique to underlying library functions, or for code in your page classes that contain a series of `await` operators that don't access user-interface objects. The technique doesn't make as much sense for functions that contain just one `await` operator that are called from the user-interface thread, because the switch back to the user-interface thread has to occur at some time, and if it doesn't occur in the library function, it will occur in the code that calls the library

function.

Don't block the UI thread!

Sometimes, there's a temptation to avoid the hassle of `ContinueWith` or even the lesser hassle of `await` simply by blocking the user-interface thread until a background process completes. Perhaps you know that the background process will complete very quickly and there's nothing much the user can do anyway until it finishes. What's the harm?

Don't do it! Not only is it impolite to the user, but it can introduce subtle bugs into your application.

Let's take an example: In the code-behind file of `TextFileAsyncPage`, the `OnFileListView-ItemSelected` handler has the following code to read the file and set the contents in the `Editor`:

```
fileEditor.Text = await fileHelper.ReadTextAsync((string)args.SelectedItem);
```

You might have discovered, perhaps accidentally or perhaps by experiment, that in a statement like this, you can leave out the `await` operator and just access the `Result` property of the `Task<string>` object returned from `ReadTextAsync`. That `Result` property is the content of the file being read:

```
fileEditor.Text = fileHelper.ReadTextAsync((string)args.SelectedItem).Result;
```

The code seems fine, and it might even work. But the way it works is not good. This statement will block the user-interface thread until the `ReadTextAsync` method has completed and `Result` is available. The user interface will be unresponsive during this time.

Moreover, if you haven't used `ConfigureAwait(false)` in the implementation of `ReadTex-tAsync` in `FileHelper`, then that `ReadTextAsync` method will require switching to the user-interface thread for resuming execution after each `await` operator. But when it tries to switch back to the user-interface thread, the UI thread will not be available because it's being blocked in the `ReadTextAsync` call in `TextFileAsyncPage`, and a classic deadlock results. The program will simply stop executing entirely.

The rule is simple: Use `ContinueWith` or `await` with every asynchronous method.

Your own awaitable methods

Aside from accessing files over the web or from the local file system, applications sometimes have the need to perform lengthy operations of their own. These operations should be run in the background on secondary threads of execution. While there are now several ways to do this, it's best (and certainly easiest) to use the same Task-based Asynchronous Pattern that is used within Xamarin.Forms and other .NET graphical environments and define your own asynchronous methods just like the others in these environments.

The easiest way to run some code on a worker thread is with the `Task.Run` and `Task.Run<T>`

static methods. The argument is an `Action` object, generally expressed as a lambda function, and the return value is a `Task`. The body of the lambda function is run on a worker thread from the thread pool, which (if you want to use the thread pool yourself) is accessible via the `ThreadPool` class. You can use the `await` operator directly with `Task.Run`:

```
await Task.Run(() =>
{
    // The code that runs in a background thread.
});
```

Although you can use `Task.Run` by itself with other code, generally it's used to construct asynchronous methods. By convention, an asynchronous method has a suffix of `Async`. The method returns either a `Task` object (if the method does not return any value or object) or a `Task<T>` object (if it does return something).

Here's how you can create an asynchronous method that returns `Task`:

```
Task MyMethodAsync(…)
{
    // Perhaps some initialization code.
    return Task.Run(() =>
    {
        // The code that runs in a background thread.
    });
}
```

The `Task.Run` method returns a `Task` object that your method also returns. The `Action` argument to `Task.Run` can use any arguments passed to the `MyMethodAsync`, but you shouldn't define any arguments using `ref` or `out`. Also, watch out for any reference types you pass to `MyMethodAsync`. These can be accessed both from inside the asynchronous code and from outside the method, so you might need to implement synchronization so that the object isn't accessed simultaneously from two threads.

The code within the `Task.Run` call can itself call asynchronous methods using `await`, but in that case you'll need to flag the lambda function passed to `Task.Run` with `async`:

```
return Task.Run(async () =>
{
    // The code that runs in a background thread.
});
```

If the asynchronous method returns something, you'll define the method using the generic form of `Task` and the generic form of `Task.Run`:

```
Task<SomeType> MyMethodAsync(…)
{
    // Perhaps some initialization code.
    return Task.Run<SomeType>(() =>
    {
        // The code that runs in a background thread.
        return anInstanceOfSomeType;
    });
```

```
}
```

The value or object returned from the lambda function becomes the `Result` property of the `Task<T>` object returned from `Task.Run` and from your method.

If you need to have more control over the background process, you can use `TaskFactory.StartNew` rather than `Task.Run` to define the asynchronous method.

There are some variations on the basic `Task.Run` patterns, as you'll see in the following several programs. These programs compute and display the famous Mandelbrot set.

The basic Mandelbrot set

The Polish-born French and American mathematician Benoit Mandelbrot (1924–2010) is best known for his work connected with complex self-similar surfaces that he called *fractals*. Among his work involving fractals was an investigation into a recursive formula that generates a fractal image that is now known as the *Mandelbrot set*.

The Mandelbrot set is graphed on the complex plane, where each coordinate is a complex number of the form:

$$c = x + yi$$

The real part x is graphed along the horizontal axis with negative values to the left and positive values to the right. The imaginary part y is graphed along the vertical axis, increasing from negative values on the bottom to positive values going up.

To calculate the Mandelbrot set, begin by taking any point on this plane and call it c, and initialize z to zero:

$$c = x + yi$$

$$z = 0$$

Now perform the following recursive operation:

$$z \leftarrow z^2 + c$$

The result will either diverge to infinity or it will not. If z does *not* diverge to infinity, then c is said to be a member of the Mandelbrot set. Otherwise, it is not a member of the Mandelbrot set.

You need to perform this calculation for every point of interest in the complex plane. Generally, the results are drawn on a bitmap, which means that each pixel in the bitmap corresponds to a particular complex coordinate. In its simplest rendition, points that belong to the Mandelbrot set are colored black and other pixels are colored white.

For some complex numbers, it's easy to determine whether the point belongs to the Mandelbrot set. For example, the complex number ($0 + 0i$) obviously belongs to the Mandelbrot set, and you can quickly establish that ($1 + 0i$) does not. But in general, you need to perform the recursive calculation.

And because this is a fractal, you can't take shortcuts. For example, if you know that two values c_1 and c_2 belong to the Mandelbrot set, you can't assume that all points between those two points belong to the Mandelbrot set as well. It is a fundamental characteristic of a fractal to defy interpolation.

How many iterations of the recursive calculation do you need to perform before you can assure yourself that the particular complex number does or does not belong to the Mandelbrot set? It turns out that if the absolute value of *z* in the recursive calculation ever becomes 2 or greater, then the values will eventually diverge to infinity and the point does not belong to the Mandelbrot set. (The absolute value of a complex number is also referred to as the *magnitude* of the number; it can be calculated as the square root of the sum of the squares of the *x* and *y* values, which is the Pythagorean theorem.)

However, if after a certain number of iterations the recursive calculation hasn't yet reached a magnitude of 2, there's no guarantee that it will not diverge with repeated iterations. For this reason, Mandelbrot sets are notoriously computation-intensive, and ideal for secondary threads of execution.

The **MandelbrotSet** program demonstrates how this is done. To render the image, the program makes use of the `BmpMaker` class (introduced in Chapter 13, "Bitmaps") from the **Xamarin.FormsBook.Toolkit** library. That library also contains the following structure to represent a complex number:

```
namespace Xamarin.FormsBook.Toolkit
{
    // Mostly a subset of System.Numerics.Complex.
    public struct Complex : IEquatable<Complex>, IFormattable
    {
        bool gotMagnitude, gotMagnitudeSquared;
        double magnitude, magnitudeSquared;

        public Complex(double real, double imaginary) : this()
        {
            Real = real;
            Imaginary = imaginary;
        }

        public double Real { private set; get; }

        public double Imaginary { private set; get; }

        // MagnitudeSquare and Magnitude calculated on demand and saved.
        public double MagnitudeSquared
        {
            get
            {
                if (gotMagnitudeSquared)
                {
                    return magnitudeSquared;
                }

                magnitudeSquared = Real * Real + Imaginary * Imaginary;
                gotMagnitudeSquared = true;
                return magnitudeSquared;
            }
```

```
}

public double Magnitude
{
    get
    {
        if (gotMagnitude)
        {
            return magnitude;
        }

        magnitude = Math.Sqrt(magnitudeSquared);
        gotMagnitude = true;
        return magnitude;
    }
}

public static Complex operator +(Complex left, Complex right)
{
    return new Complex(left.Real + right.Real, left.Imaginary + right.Imaginary);
}

public static Complex operator -(Complex left, Complex right)
{
    return new Complex(left.Real - right.Real, left.Imaginary - right.Imaginary);
}

public static Complex operator *(Complex left, Complex right)
{
    return new Complex(left.Real * right.Real - left.Imaginary * right.Imaginary,
                       left.Real * right.Imaginary + left.Imaginary * right.Real);
}

public static bool operator ==(Complex left, Complex right)
{
    return left.Real == right.Real && left.Imaginary == right.Imaginary;
}

public static bool operator !=(Complex left, Complex right)
{
    return !(left == right);
}

public static implicit operator Complex(double value)
{
    return new Complex(value, 0);
}

public static implicit operator Complex(int value)
{
    return new Complex(value, 0);
}

public override int GetHashCode()
```

```
        {
            return Real.GetHashCode() + Imaginary.GetHashCode();
        }

        public override bool Equals(Object value)
        {
            return Real.Equals(((Complex)value).Real) &&
                   Imaginary.Equals(((Complex)value).Imaginary);
        }

        public bool Equals(Complex value)
        {
            return Real.Equals(value) && Imaginary.Equals(value);
        }

        public override string ToString()
        {
            return String.Format("{0} {1} {2}i", Real,
                                                RealImaginaryConnector(Imaginary),
                                                Math.Abs(Imaginary));
        }

        public string ToString(string format)
        {
            return String.Format("{0} {1} {2}i", Real.ToString(format),
                                                RealImaginaryConnector(Imaginary),
                                                Math.Abs(Imaginary).ToString(format));
        }

        public string ToString(IFormatProvider formatProvider)
        {
            return String.Format("{0} {1} {2}i", Real.ToString(formatProvider),
                                                RealImaginaryConnector(Imaginary),
                                                Math.Abs(Imaginary).ToString(formatProvider));
        }

        public string ToString(string format, IFormatProvider formatProvider)
        {
            return String.Format("{0} {1} {2}i", Real.ToString(format, formatProvider),
                                                RealImaginaryConnector(Imaginary),
                                        Math.Abs(Imaginary).ToString(format, formatProvider));
        }

        string RealImaginaryConnector(double value)
        {
            return Math.Sign(value) > 0 ? "+" : "\u2013";
        }
    }
}
```

As the comment at the top indicates, this is *mostly* a subset of the `Complex` structure in the .NET `System.Numerics` namespace, which unfortunately is not available to a Portable Class Library in a Xamarin.Forms project. The `ToString` methods in this `Complex` structure work a little differently, however,

and the original `Complex` structure does not have a `MagnitudeSquared` property. A `MagnitudeSquared` property is handy for a Mandelbrot calculation: Checking if the `Magnitude` property is less than 2 is the same as checking if the `MagnitudeSquared` property is less than 4, but without the square root calculation.

The **MandelbrotSet** program has the following XAML file:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MandelbrotSet.MandelbrotSetPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <ContentView Padding="10, 0"
                         VerticalOptions="Center">
                <ActivityIndicator x:Name="activityIndicator" />
            </ContentView>

            <Image x:Name="image" />
        </Grid>

        <Button x:Name="calculateButton"
                Text="Calculate"
                FontSize="Large"
                HorizontalOptions="Center"
                Clicked="OnCalculateButtonClicked" />
    </StackLayout>
</ContentPage>
```

The `ActivityIndicator` informs the user that the program is busy with the background job. The `Image` element and that `ActivityIndicator` share a single-cell `Grid` so that the `ActivityIndicator` can be more toward the vertical center of the screen and then become covered when the bitmap appears. At the bottom is a `Button` to begin the calculation.

The code-behind file below begins by defining several constants. The first four constants relate to the bitmap that the program constructs to display the image of the Mandelbrot set. Throughout this exercise, these bitmaps will always be square, but the code itself is more generalized and should be able to accommodate rectangular dimensions.

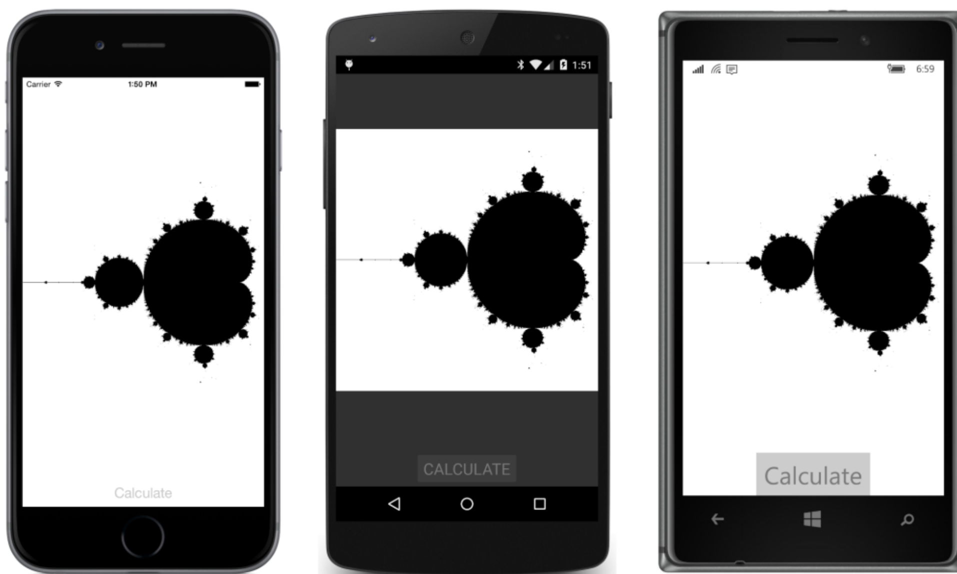
The `center` field is the `Complex` point that corresponds to the center of the bitmap, while the `size` field indicates the extent of the real and imaginary coordinates on the bitmaps. These particular `center` and `size` fields imply that the real coordinates range from –2 on the left of the bitmap to 0.5 on the right, and the imaginary coordinates range from –1.25 on the bottom to 1.25 on the top. The `pixelWidth` and `pixelHeight` values indicate the width and height of the bitmap in pixels. The `iterations` field is the maximum number of iterations of the recursive formula before the program assumes that the point belongs to the Mandelbrot set:

```
public partial class MandelbrotSetPage : ContentPage
{
    static readonly Complex center = new Complex(-0.75, 0);
    static readonly Size size = new Size(2.5, 2.5);
    const int pixelWidth = 1000;
    const int pixelHeight = 1000;
    const int iterations = 100;

    public MandelbrotSetPage()
    {
        InitializeComponent();
    }

    async void OnCalculateButtonClicked(object sender, EventArgs args)
    {
        calculateButton.IsEnabled = false;
        activityIndicator.IsRunning = true;

        BmpMaker bmpMaker = new BmpMaker(pixelWidth, pixelHeight);
        await CalculateMandelbrotAsync(bmpMaker);
        image.Source = bmpMaker.Generate();

        activityIndicator.IsRunning = false;
    }

    Task CalculateMandelbrotAsync(BmpMaker bmpMaker)
    {
        return Task.Run(() =>
        {
            for (int row = 0; row < pixelHeight; row++)
            {
                double y = center.Imaginary - size.Height / 2 + row * size.Height / pixelHeight;

                for (int col = 0; col < pixelWidth; col++)
                {
                    double x = center.Real - size.Width / 2 + col * size.Width / pixelWidth;
                    Complex c = new Complex(x, y);
                    Complex z = 0;
                    int iteration = 0;

                    do
                    {
                        z = z * z + c;
                        iteration++;
                    }
                    while (iteration < iterations && z.MagnitudeSquared < 4);

                    bool isMandelbrotSet = iteration == iterations;
                    bmpMaker.SetPixel(row, col, isMandelbrotSet ? Color.Black : Color.White);
                }
            }
        });
    }
}
```

The `OnCalculateButtonClicked` handler is flagged as `async`. It begins by disabling the `Button` to avoid multiple simultaneous calculations and starts the `ActivityIndicator` display. It then creates a `BmpMaker` object with the desired pixel size and passes it to `CalculateMandelbrotAsync`. When that method is finished, the `Clicked` handler continues by setting the bitmap to the `Image` object and turning off the `ActivityIndicator`. The `Button` is not reenabled.

The lambda function passed to the `Task.Run` method loops through the rows and columns of the bitmap created by `BmpMaker`, and for each pixel, it calculates a complex number `c` from the `x` and `y` coordinate values. The little `do-while` loop continues until the maximum number of iterations is reached or the magnitude is 2 or greater. At that point, a pixel can be set to black or white.

After you press the button, your phone might take a minute or so to loop through all the pixels, but then you'll see the classic image:



There's a little danger in the way the `CalculateMandelbrotAsync` method is structured. It is passed a `BmpMaker` object that the background thread fills with pixels, but the main thread also has access to this `BmpMaker` object. If this object were saved as a field, the main thread might contain some code that alters or sets pixels as the background thread is working. That would probably be a bug, of course, but in general you can make your asynchronous methods more bulletproof if arguments are restricted to value types rather than reference types. Don't worry too much if that's not quite possible or convenient, but in the next version of the program, the `CalculateMandelbrotAsync` method will itself create the `BmpMaker` object and return it.

Marking progress

As you've undoubtedly discovered, it's somewhat disconcerting to press the **Calculate** button in **MandelbrotSet** and wait for the bitmap to show up. There's no indication at all how far along the program has gotten in completing the job, or how much longer you need to wait.

If possible, asynchronous methods should report progress. I'm sure you can rig something up yourself to do the job, but there is a standard way of reporting progress for methods that return `Task` objects. This involves the `IProgress<T>` interface and the `Progress<T>` class that implements that interface, both of which are defined in the `System` namespace. `IProgress` is defined like so:

```
public interface IProgress<T>
{
    void Report(T value);
}
```

To make use of this facility, you define an argument to your asynchronous method of type `IProgress`. The asynchronous method then periodically calls `Report` as it's doing the background job. Generally, `T` is either `int`, in which case the values passed to `Report` usually range from 1 to 100, or `double`, for values ranging from 0 to 1. It's your choice. For consistency with the Xamarin.Forms `ProgressBar`, `double` values from 0 to 1 are ideal.

The code that calls the asynchronous method instantiates a `Progress` object and passes to its constructor a lambda function that is called whenever the asynchronous method calls `Report`. (Or you can attach a handler to the `Progress` object's `ProgressChanged` event.) Although `Report` is called on a background thread, the lambda function or event handler is called on the thread that instantiated the `Progress` object, which means that the lambda function or event handler can safely access user-interface objects.

The XAML file for the **MandelbrotProgress** program is the same as the previous XAML file except that a `ProgressBar` has replaced the `ActivityIndicator`:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MandelbrotProgress.MandelbrotProgressPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <ContentView Padding="10, 0"
                         VerticalOptions="Center">
                <ProgressBar x:Name="progressBar" />
            </ContentView>

            <Image x:Name="image" />
        </Grid>
```

```
        <Button x:Name="calculateButton"
                Text="Calculate"
                FontSize="Large"
                HorizontalOptions="Center"
                Clicked="OnCalculateButtonClicked" />
    </StackLayout>
</ContentPage>
```

The code-behind file is very similar, except that a `Progress` object named `progressReporter` is defined as a field and the constructor instantiates it with a lambda function that simply sets the argument to the `Progress` property of the `ProgressBar`. This `Progress` object is passed to the `CalculateMandelbrotAsync` method, which in this new version now takes over the responsibility of creating and returning the `BmpMaker` object:

```
public partial class MandelbrotProgressPage : ContentPage
{
    static readonly Complex center = new Complex(-0.75, 0);
    static readonly Size size = new Size(2.5, 2.5);
    const int pixelWidth = 1000;
    const int pixelHeight = 1000;
    const int iterations = 100;

    Progress<double> progressReporter;

    public MandelbrotProgressPage()
    {
        InitializeComponent();

        progressReporter = new Progress<double>((double value) =>
            {
                progressBar.Progress = value;
            });
    }

    async void OnCalculateButtonClicked(object sender, EventArgs args)
    {
        // Configure the UI for a background process.
        calculateButton.IsEnabled = false;

        // Render the Mandelbrot set on a bitmap.
        BmpMaker bmpMaker = await CalculateMandelbrotAsync(progressReporter);
        image.Source = bmpMaker.Generate();
    }

    Task<BmpMaker> CalculateMandelbrotAsync(IProgress<double> progress)
    {
        return Task.Run<BmpMaker>(() =>
        {
            BmpMaker bmpMaker = new BmpMaker(pixelWidth, pixelHeight);

            for (int row = 0; row < pixelHeight; row++)
            {
                double y = center.Imaginary - size.Height / 2 + row * size.Height / pixelHeight;
```

```
                    // Report the progress.
                    progress.Report((double)row / pixelHeight);

                    for (int col = 0; col < pixelWidth; col++)
                    {
                        double x = center.Real - size.Width / 2 + col * size.Width / pixelWidth;
                        Complex c = new Complex(x, y);
                        Complex z = 0;
                        int iteration = 0;
                        bool isMandelbrotSet = false;

                        if ((c - new Complex(-1, 0)).MagnitudeSquared < 1.0 / 16)
                        {
                            isMandelbrotSet = true;
                        }
                        else
                        {
                            do
                            {
                                z = z * z + c;
                                iteration++;
                            }
                            while (iteration < iterations && z.MagnitudeSquared < 4);

                            isMandelbrotSet = iteration == iterations;
                        }
                        bmpMaker.SetPixel(row, col, isMandelbrotSet ? Color.Black : Color.White);
                    }
                }
                return bmpMaker;
            });
    }
}
```

The asynchronous method reports its progress with every new row:

```
progress.Report((double)row / pixelHeight);
```

Watch out: You don't want to report progress so frequently that you slow down the method! A hundred calls to the `Report` method during the whole operation is plenty, and you can probably reduce that number considerably before the `ProgressBar` begins looking jittery.

If you pay close attention to the `ProgressBar` in **MandelbrotProgress**, you'll see that it moves fast at the start and then slows down. The problem area is the large cardioid—and to a lesser extent, the circle to its left—that makes up the bulk of the Mandelbrot set. For points within these areas, the recursive calculation must run to the maximum iteration count before the point is identified as a member of the set. This new method attempts to reduce the work somewhat by detecting when the point is within the circle. The center of this circle is the complex point –1, and the radius is 1/4:

```
if ((c - new Complex(-1, 0)).MagnitudeSquared < 1.0 / 16)
{
    isMandelbrotSet = true;
```

```
}
```

But the cardioid is a more complex object (although that too can be identified, as the next version of the program demonstrates).

When the asynchronous method creates and returns that `BmpMaker` object, the code to obtain that object and set the bitmap to the `Image` object reduces to just two statements:

```
BmpMaker bmpMaker = await CalculateMandelbrotAsync(progressReporter);
image.Source = bmpMaker.Generate();
```

But if two statements are too many, keep in mind that `await` is pretty much just an ordinary operator and can be part of a more complex statement:

```
image.Source = (await CalculateMandelbrotAsync(progressReporter)).Generate();
```

Cancelling the job

The two Mandelbrot programs shown so far exist for the sole purpose of generating a single image, so it's unlikely that you would want to cancel that job once it's started. However, in the general case, you'll want to provide a facility for the user to bail out of lengthy background jobs.

Although you can probably put together a little cancellation system of your own, the `System.Threading` namespace already has you covered with a class named `CancellationTokenSource` and a structure named `CancellationToken`.

Here's how it works:

A program creates a `CancellationTokenSource` for use with a particular asynchronous method. The `CancellationTokenSource` class defines a property named `Token` that returns a `CancellationToken`. This `CancellationToken` value is passed to the asynchronous method. The asynchronous method periodically calls the `IsCancellationRequested` method of the `CancellationToken`. This method usually returns `false`.

When the program wants to cancel the asynchronous operation (probably in response to some user input), it calls the `Cancel` method of the `CancellationTokenSource`. The next time the asynchronous method calls the `IsCancellationRequested` method of the `CancellationToken`, the method returns `true` because a cancellation has been requested. The asynchronous method can choose how to stop running, perhaps with a simple `return` statement.

Usually, however, a different approach is taken. Rather than calling the `IsCancellationRequested` method of `CancellationToken`, the asynchronous method can instead simply call the `ThrowIfCancellationRequested` method. If a cancellation has been requested, the asynchronous method stops executing by raising an `OperationCanceledException`.

This means that the `await` operator must be part of a `try` block, but as you've seen, this is generally the case when working with files, so it doesn't add much additional code, and the program can process a cancellation as simply another form of exception.

The **MandelbrotCancellation** program demonstrates this technique. The XAML file now has a second button, labeled “Cancel”, which is initially disabled:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MandelbrotCancellation.MandelbrotCancellationPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <ContentView Padding="10, 0"
                         VerticalOptions="Center">
                <ProgressBar x:Name="progressBar" />
            </ContentView>

            <Image x:Name="image" />
        </Grid>

        <Grid>
            <Button x:Name="calculateButton"
                    Grid.Column="0"
                    Text="Calculate"
                    FontSize="Large"
                    HorizontalOptions="Center"
                    Clicked="OnCalculateButtonClicked" />

            <Button x:Name="cancelButton"
                    Grid.Column="1"
                    Text="Cancel"
                    FontSize="Large"
                    IsEnabled="False"
                    HorizontalOptions="Center"
                    Clicked="OnCancelButtonClicked" />
        </Grid>
    </StackLayout>
</ContentPage>
```

The code-behind file now has a more extensive `OnCalculateButtonClicked` method. It begins by disabling the **Calculate** button and enabling the **Cancel** button. It creates a new `CancellationTokenSource` object and passes the `Token` property to `CalculateMandelbrotAsync`. The `OnCancelButtonClicked` method is responsible for calling `Cancel` on the `CancellationTokenSource` object. The `CalculateMandelbrotAsync` method calls the `ThrowIfCancellationRequested` method at the same rate that it reports progress. The exception is caught by the `OnCalculateButtonClicked` method, which responds by reenabling the **Calculate** button for another try:

```
public partial class MandelbrotCancellationPage : ContentPage
{
    static readonly Complex center = new Complex(-0.75, 0);
    static readonly Size size = new Size(2.5, 2.5);
    const int pixelWidth = 1000;
```

```
const int pixelHeight = 1000;
const int iterations = 100;

Progress<double> progressReporter;
CancellationTokenSource cancelTokenSource;

public MandelbrotCancellationPage()
{
    InitializeComponent();

    progressReporter = new Progress<double>((double value) =>
        {
            progressBar.Progress = value;
        });
}

async void OnCalculateButtonClicked(object sender, EventArgs args)
{
    // Configure the UI for a background process.
    calculateButton.IsEnabled = false;
    cancelButton.IsEnabled = true;

    cancelTokenSource = new CancellationTokenSource();

    try
    {
        // Render the Mandelbrot set on a bitmap.
        BmpMaker bmpMaker = await CalculateMandelbrotAsync(progressReporter,
                                                           cancelTokenSource.Token);
        image.Source = bmpMaker.Generate();
    }
    catch (OperationCanceledException)
    {
        calculateButton.IsEnabled = true;
        progressBar.Progress = 0;
    }
    catch (Exception)
    {
        // Shouldn't occur in this case.
    }

    cancelButton.IsEnabled = false;
}

void OnCancelButtonClicked(object sender, EventArgs args)
{
    cancelTokenSource.Cancel();
}

Task<BmpMaker> CalculateMandelbrotAsync(IProgress<double> progress,
                                        CancellationToken cancelToken)
{
    return Task.Run<BmpMaker>(() =>
    {
```

```
            BmpMaker bmpMaker = new BmpMaker(pixelWidth, pixelHeight);

            for (int row = 0; row < pixelHeight; row++)
            {
                double y = center.Imaginary - size.Height / 2 + row * size.Height / pixelHeight;

                // Report the progress.
                progress.Report((double)row / pixelHeight);

                // Possibly cancel.
                cancelToken.ThrowIfCancellationRequested();

                for (int col = 0; col < pixelWidth; col++)
                {
                    double x = center.Real - size.Width / 2 + col * size.Width / pixelWidth;
                    Complex c = new Complex(x, y);
                    Complex z = 0;
                    int iteration = 0;
                    bool isMandelbrotSet = false;

                    if ((c - new Complex(-1, 0)).MagnitudeSquared < 1.0 / 16)
                    {
                        isMandelbrotSet = true;
                    }
                    // http://www.reenigne.org/blog/algorithm-for-mandelbrot-cardioid/
                    else if (c.MagnitudeSquared * (8 * c.MagnitudeSquared - 3) <
                                                               3.0 / 32 - c.Real)
                    {
                        isMandelbrotSet = true;
                    }
                    else
                    {
                        do
                        {
                            z = z * z + c;
                            iteration++;
                        }
                        while (iteration < iterations && z.MagnitudeSquared < 4);

                        isMandelbrotSet = iteration == iterations;
                    }
                    bmpMaker.SetPixel(row, col, isMandelbrotSet ? Color.Black : Color.White);
                }
            }
            return bmpMaker;
        }, cancelToken);
    }
}
```

The `CancellationToken` is also passed as the second argument to `Task.Run`. This isn't required, but it allows the `Task.Run` method to skip a lot of work if cancellation has already been requested before it even gets started.

Also notice that the code now skips the large cardioid. A comment references a web page that derives the formula in case you want to check the math.

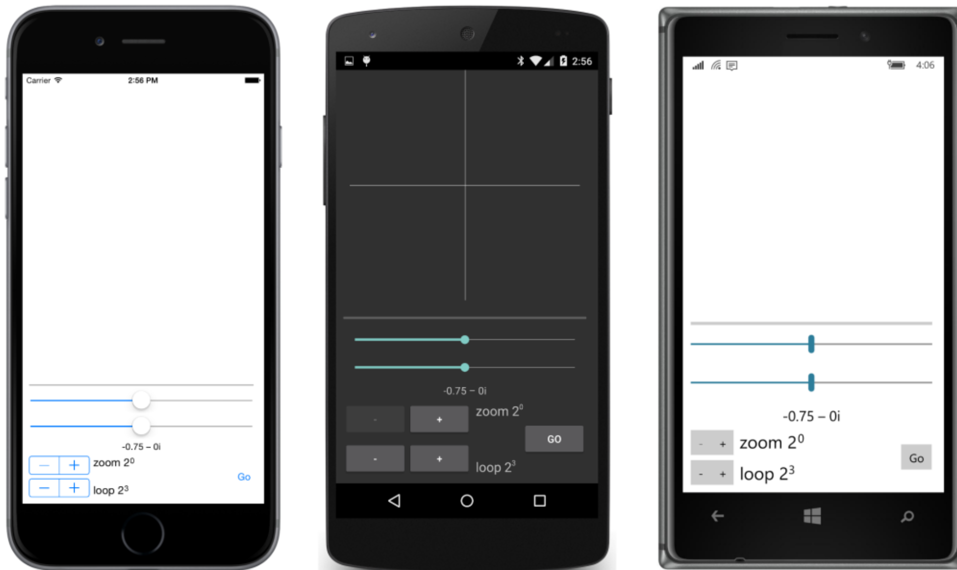
An MVVM Mandelbrot

Although the black-and-white Mandelbrot set is the classic image, most Mandelbrot programs color pixels that are not in the Mandelbrot set based on the number of iterations required for that determination. The penultimate program in this chapter is called **MandelbrotXF**—the XF prefix stands for Xamarin.Forms—and colors the pixels in that way. The program also allows zooming in on specific locations. It is a characteristic of a Mandelbrot set that the image remains interesting no matter how far you zoom. Unfortunately, there is a practical limit to zooming based on the resolution of double-precision floating-point numbers.

The program is architected using MVVM principles, although after seeing the somewhat odd user interface—and how the ViewModel deals with that user interface—you might question the wisdom of that decision.

The odd user interface of **MandelbrotXF** results from a decision to avoid any platform-specific code. At the time this program was originally written, Xamarin.Forms did not support touch operations such as dragging and pinching that might have been helpful in zooming into a particular location. Instead, the program's entire user interface is implemented with two `Slider` elements, two `Stepper` elements, two `Button` elements, a `ProgessBar`, and visuals implemented with `BoxView`.

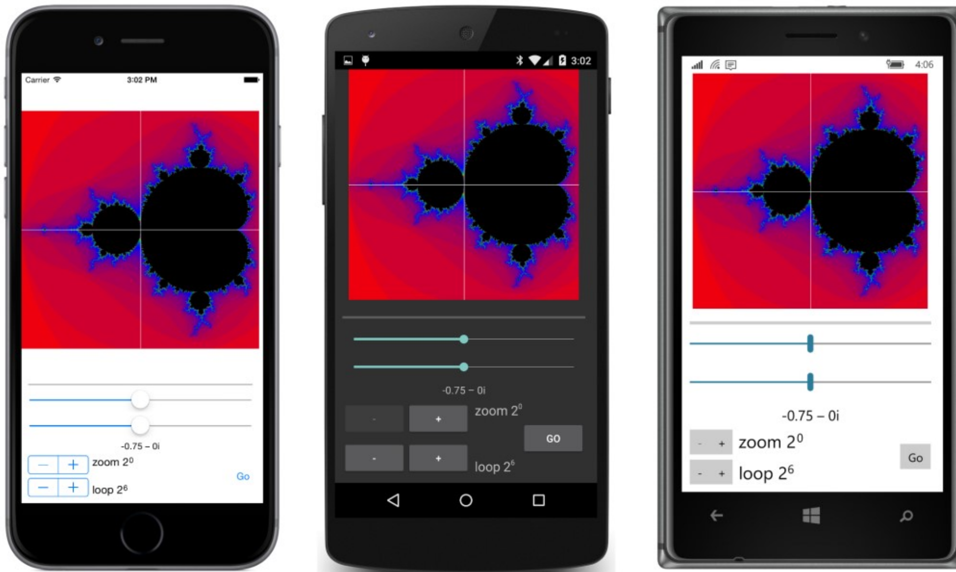
When you first run the program, here's what you'll see:



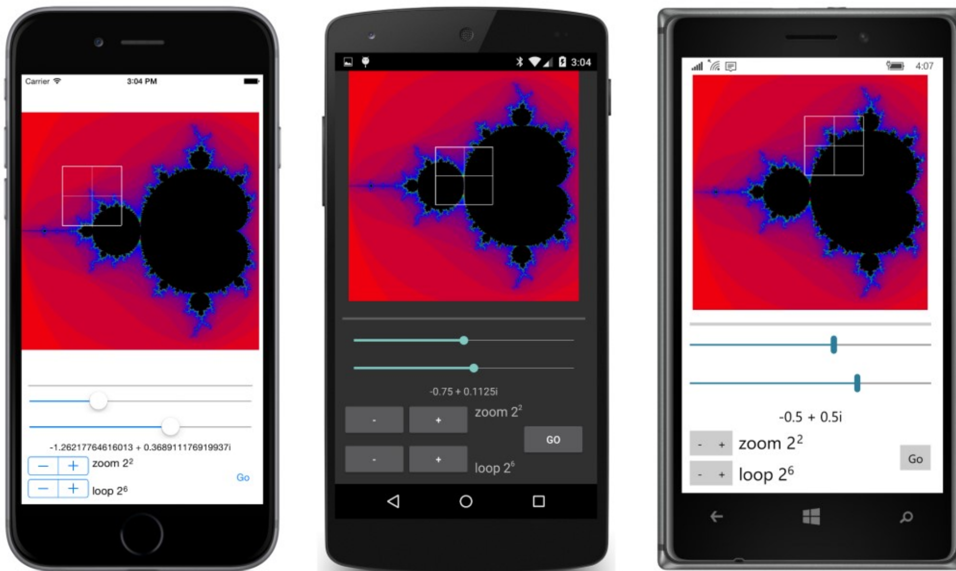
The white crosshairs—which don't show up against the white background of the blank iOS and Windows 10 Mobile screens—fade out over the course of 10 seconds so that they won't obscure the pretty pictures that you'll soon be admiring, but you can bring them back by manipulating either of the sliders or the steppers.

But the first thing you'll want to do is press the **Go** button. The button is replaced with a **Cancel** button and the `ProgressBar` indicates progress. When it's finished, you'll see a colored Mandelbrot set:

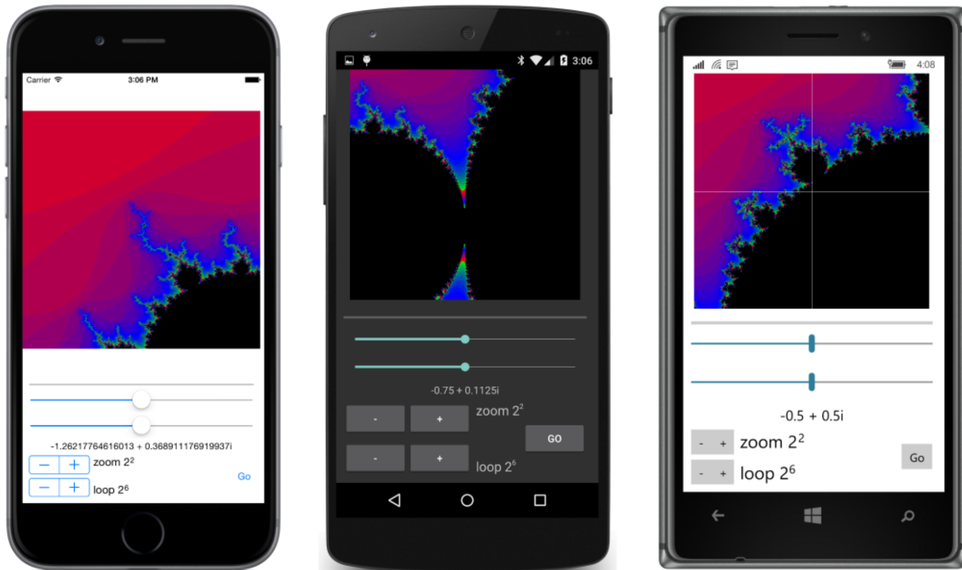
It finishes quickly because the maximum iteration count (indicated by the bottom `Stepper` labeled **loop**) is only 2 to the third power, or 8. As a result, the outline of the black Mandelbrot set is not nearly as sharp as the earlier programs. Many more points are flagged as being a member of the set than would be with a higher maximum iteration count. You can increase that iteration count by powers of 2. Here's a sharper image with a maximum iteration count of 64:



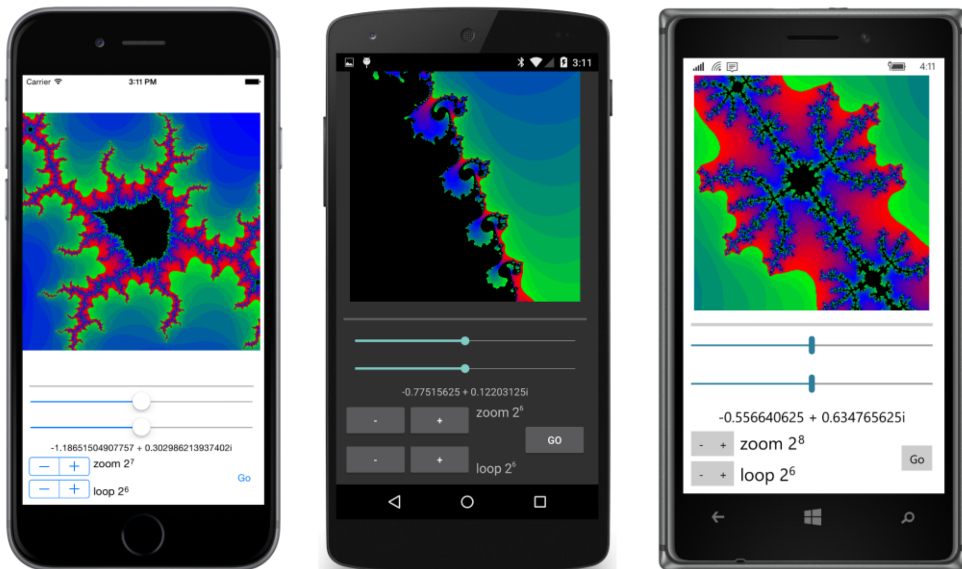
The two `Slider` views allow you to select a new center, which is displayed as a complex number right below the sliders. The first `Stepper` element (labeled **zoom**) allows you to select a magnification factor, also in powers of 2. As you manipulate these three elements, you'll see a box with crosshairs constructed with six thin `BoxView` elements. That box marks the area that will be magnified the next time you press the **Go** button:



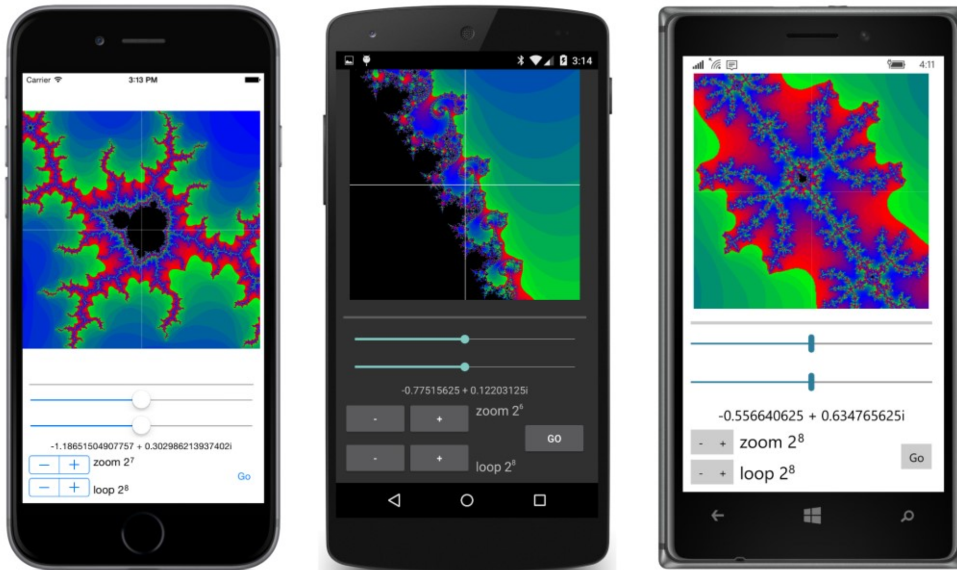
Now press the **Go** button again and wait. Now that previously boxed area fills the bitmap:



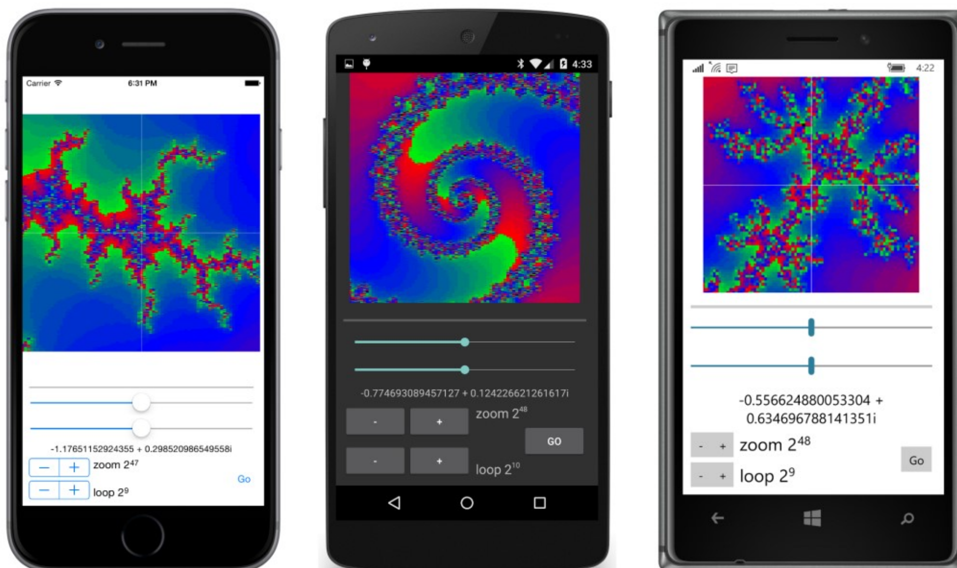
After the new image is calculated, the crosshairs are recentered, and you can reposition the center and zoom in again, and again, and again.



However, generally the more you zoom in, the greater the maximum iterations you'll need to see all the detail. For each device, the image in the previous screenshots acquires visibly more detail with four times as many iterations:



It is a characteristic of the Mandelbrot set that you can just keep zooming in as much as you want and you'll still see just as much detail. However, generally you will need to keep increasing the maximum iteration count as well, and by the time you hit a magnification factor of 2 to the forty-eighth power or so, you've hit a ceiling involving the resolution of double-precision floating-point numbers. Adjacent pixels are no longer associated with distinct complex numbers, and the image begins looking blocky:



That's not an easy obstacle to transcend. There exist implementations of variable-precision floating-point numbers, but because they are not directly handled by the computer's math coprocessor, calculations involving these numbers are necessarily much slower than `float` or `double` types, and it's likely you're not going to want the Mandelbrot calculation to go any slower.

The **MandelbrotXF** program has both a ViewModel and an underlying Model. The Model does the actual number crunching and returns an object of type `BitmapInfo`, which indicates a pixel width and height and an array of integers. The size of the integer array is the product of the pixel width and height, and the elements of the array are iteration counts. A value of –1 indicates a member of the Mandelbrot set:

```
namespace MandelbrotXF
{
    class BitmapInfo
    {
        public BitmapInfo(int pixelWidth, int pixelHeight, int[] iterationCounts)
        {
            PixelWidth = pixelWidth;
            PixelHeight = pixelHeight;
            IterationCounts = iterationCounts;
        }

        public int PixelWidth { private set; get; }

        public int PixelHeight { private set; get; }

        public int[] IterationCounts { private set; get; }
    }
}
```

The `MandelbrotModel` class contains a single asynchronous method. Aside from the `IProgress` object, all the arguments are value types, so there is no danger of any argument changing while the calculation is in progress:

```
namespace MandelbrotXF
{
    class MandelbrotModel
    {
        public Task<BitmapInfo> CalculateAsync(Complex Center,
                                               double width, double height,
                                               int pixelWidth, int pixelHeight,
                                               int iterations,
                                               IProgress<double> progress,
                                               CancellationToken cancelToken)
        {
            return Task.Run(() =>
            {
                int[] iterationCounts = new int[pixelWidth * pixelHeight];
                int index = 0;

                for (int row = 0; row < pixelHeight; row++)
                {
```

```
                    progress.Report((double)row / pixelHeight);
                    cancelToken.ThrowIfCancellationRequested();

                    double y = Center.Imaginary - height / 2 + row * height / pixelHeight;

                    for (int col = 0; col < pixelWidth; col++)
                    {
                        double x = Center.Real - width / 2 + col * width / pixelWidth;
                        Complex c = new Complex(x, y);

                        if ((c - new Complex(-1, 0)).MagnitudeSquared < 1.0 / 16)
                        {
                            iterationCounts[index++] = -1;
                        }
                        // http://www.reenigne.org/blog/algorithm-for-mandelbrot-cardioid/
                        else if (c.MagnitudeSquared * (8 * c.MagnitudeSquared - 3) <
                                                                    3.0 / 32 - c.Real)
                        {
                            iterationCounts[index++] = -1;
                        }
                        else
                        {
                            Complex z = 0;
                            int iteration = 0;

                            do
                            {
                                z = z * z + c;
                                iteration++;
                            }
                            while (iteration < iterations && z.MagnitudeSquared < 4);

                            if (iteration == iterations)
                            {
                                iterationCounts[index++] = -1;
                            }
                            else
                            {
                                iterationCounts[index++] = iteration;
                            }
                        }
                    }
                }
                return new BitmapInfo(pixelWidth, pixelHeight, iterationCounts);
            }, cancelToken);
        }
    }
}
```

This `CalculateAsync` method is called only from the ViewModel. The ViewModel is also intended to provide data-binding sources for the XAML file and to assist the code-behind file in performing those jobs that the XAML data bindings cannot handle. (Drawing the crosshairs and magnification box is a job for that code-behind file.)

For this reason, the MandelbrotViewModel class has many properties, but probably not the same properties you'd define if you weren't thinking about the user interface. The CurrentCenter property is the complex number for the center of the image currently displayed by the program, and the CurrentMagnification also applies to that image. But the TargetMagnification is bound to the current setting of the Stepper, which will apply to the next calculated image. The RealOffset and ImaginaryOffset properties are bound to the two Slider elements and can range from 0 to 1. From the CurrentCenter, CurrentMagnification, RealOffset, and ImaginaryOffset properties, the ViewModel can calculate the TargetCenter property. This is the center for the next calculated image. As you'll see, that TargetCenter property is used to display the complex number below the two sliders:

```
namespace MandelbrotXF
{
    class MandelbrotViewModel : ViewModelBase
    {
        // Set via constructor arguments.
        readonly double baseWidth;
        readonly double baseHeight;

        // Backing fields for properties.
        Complex currentCenter, targetCenter;
        int pixelWidth, pixelHeight;
        double currentMagnification, targetMagnification;
        int iterations;
        double realOffset, imaginaryOffset;
        bool isBusy;
        double progress;
        BitmapInfo bitmapInfo;

        public MandelbrotViewModel(double baseWidth, double baseHeight)
        {
            this.baseWidth = baseWidth;
            this.baseHeight = baseHeight;

            // Create MandelbrotModel object.
            MandelbrotModel model = new MandelbrotModel();

            // Progress reporter
            Progress<double> progressReporter = new Progress<double>((double progress) =>
            {
                Progress = progress;
            });

            CancellationTokenSource cancelTokenSource = null;

            // Define CalculateCommand and CancelCommand.
            CalculateCommand = new Command(
                execute: async () =>
                {
                    // Disable this button and enable Cancel button.
                    IsBusy = true;
                    ((Command)CalculateCommand).ChangeCanExecute();
```

```
                ((Command)CancelCommand).ChangeCanExecute();

                // Create CancellationToken.
                cancelTokenSource = new CancellationTokenSource();
                CancellationToken cancelToken = cancelTokenSource.Token;

                try
                {
                    // Perform the calculation.
                    BitmapInfo = await model.CalculateAsync(TargetCenter,
                                                    baseWidth / TargetMagnification,
                                                    baseHeight / TargetMagnification,
                                                            PixelWidth, PixelHeight,
                                                            Iterations,
                                                            progressReporter,
                                                            cancelToken);

                    // Processing only for a successful completion.
                    CurrentCenter = TargetCenter;
                    CurrentMagnification = TargetMagnification;
                    RealOffset = 0.5;
                    ImaginaryOffset = 0.5;
                }
                catch (OperationCanceledException)
                {
                    // Operation cancelled!
                }
                catch
                {
                    // Another type of exception? This should not occur.
                }

                // Processing regardless of success or cancellation.
                Progress = 0;
                IsBusy = false;

                // Disable Cancel button and enable this button.
                ((Command)CalculateCommand).ChangeCanExecute();
                ((Command)CancelCommand).ChangeCanExecute();
            },
        canExecute: () =>
            {
                return !IsBusy;
            });

    CancelCommand = new Command(
        execute: () =>
            {
                cancelTokenSource.Cancel();
            },
        canExecute: () =>
            {
                return IsBusy;
            });
```

```
}

public int PixelWidth
{
    set { SetProperty(ref pixelWidth, value); }
    get { return pixelWidth; }
}

public int PixelHeight
{
    set { SetProperty(ref pixelHeight, value); }
    get { return pixelHeight; }
}

public Complex CurrentCenter
{
    set
    {
        if (SetProperty(ref currentCenter, value))
            CalculateTargetCenter();
    }
    get { return currentCenter; }
}

public Complex TargetCenter
{
    private set { SetProperty(ref targetCenter, value); }
    get { return targetCenter; }
}

public double CurrentMagnification
{
    set { SetProperty(ref currentMagnification, value); }
    get { return currentMagnification; }
}

public double TargetMagnification
{
    set { SetProperty(ref targetMagnification, value); }
    get { return targetMagnification; }
}

public int Iterations
{
    set { SetProperty(ref iterations, value); }
    get { return iterations; }
}

// These two properties range from 0 to 1.
// They indicate a new center relative to the
//  current width and height, which is the baseWidth
//  and baseHeight divided by CurrentMagnification.
public double RealOffset
{
```

```
            set
            {
                if (SetProperty(ref realOffset, value))
                    CalculateTargetCenter();
            }
            get { return realOffset; }
        }

        public double ImaginaryOffset
        {
            set
            {
                if (SetProperty(ref imaginaryOffset, value))
                    CalculateTargetCenter();
            }
            get { return imaginaryOffset; }
        }

        void CalculateTargetCenter()
        {
            double width = baseWidth / CurrentMagnification;
            double height = baseHeight / CurrentMagnification;

            TargetCenter = new Complex(CurrentCenter.Real + (RealOffset - 0.5) * width,
                                       CurrentCenter.Imaginary + (ImaginaryOffset - 0.5) *
                                       height);
        }

        public bool IsBusy
        {
            private set { SetProperty(ref isBusy, value); }
            get { return isBusy; }
        }

        public double Progress
        {
            private set { SetProperty(ref progress, value); }
            get { return progress; }
        }

        public BitmapInfo BitmapInfo
        {
            private set { SetProperty(ref bitmapInfo, value); }
            get { return bitmapInfo; }
        }

        public ICommand CalculateCommand { private set; get; }

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

`MandelbrotViewModel` also defines two properties of type `ICommand` for the **Calculate** and **Cancel** buttons, a `Progress` property, and an `IsBusy` property. As you'll see, the `IsBusy` property is used

to display one of those two buttons and hide the other and to disable the rest of the user interface during the calculations. The two `ICommand` properties are implemented with lambda functions in the class's constructor.

The data bindings in the XAML file to the properties in `MandelbrotViewModel` require two new binding converters in the **Xamarin.FormsBook.Toolkit** library. The first simply negates a `bool` value:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class BooleanNegationConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            return !(bool)value;
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            return !(bool)value;
        }
    }
}
```

This is used in conjunction with the `IsBusy` property of the ViewModel. When `IsBusy` is `true`, the `IsEnabled` properties of several elements and the `IsVisible` property of the **Go** button need to be set to `false`.

Both `Stepper` elements actually control an exponent of a value in the ViewModel. A `Stepper` value of 8, for example, corresponds to an `Iterations` or `TargetMagnification` value of 256. That conversion requires a base-2 logarithm converter:

```
namespace Xamarin.FormsBook.Toolkit
{
    public class BaseTwoLogConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            if (value is int)
            {
                return Math.Log((int)value) / Math.Log(2);
            }
            return Math.Log((double)value) / Math.Log(2);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            double returnValue = Math.Pow(2, (double)value);

            if (targetType == typeof(int))
```

```
            {
                return (int) returnValue;
            }
            return returnValue;
        }
    }
}
```

Here's the XAML file, with bindings to the `Progress`, `RealOffset`, `ImaginaryOffset`, `TargetCenter`, `TargetMagnification`, `Iterations`, `IsBusy`, `CalculateCommand`, and `CancelCommand` properties of the ViewModel:

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

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:BooleanNegationConverter x:Key="negate" />
            <toolkit:BaseTwoLogConverter x:Key="base2log" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid x:Name="mainGrid">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

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

        <!-- Image for determining pixels per unit. -->
        <Image x:Name="testImage"
               Grid.Row="0" Grid.Column="0"
               Opacity="0"
               HorizontalOptions="Center"
               VerticalOptions="Center" />

        <!-- Image for Mandelbrot Set. -->
        <Image x:Name="image"
               Grid.Row="0" Grid.Column="0"
               HorizontalOptions="FillAndExpand"
               VerticalOptions="FillAndExpand"
               SizeChanged="OnImageSizeChanged" />
```

```
<AbsoluteLayout x:Name="crossHairLayout"
                Grid.Row="0" Grid.Column="0"
                HorizontalOptions="Center"
                VerticalOptions="Center"
                SizeChanged="OnCrossHairLayoutSizeChanged">

    <AbsoluteLayout.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="Color" Value="White" />
                <Setter Property="AbsoluteLayout.LayoutBounds" Value="0,0,0,0" />
            </Style>
        </ResourceDictionary>
    </AbsoluteLayout.Resources>

    <BoxView x:Name="realCrossHair" />
    <BoxView x:Name="imagCrossHair" />
    <BoxView x:Name="topBox" />
    <BoxView x:Name="bottomBox" />
    <BoxView x:Name="leftBox" />
    <BoxView x:Name="rightBox" />
</AbsoluteLayout>

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

    <ProgressBar Progress="{Binding Progress}"
                 VerticalOptions="CenterAndExpand" />

    <StackLayout VerticalOptions="CenterAndExpand">
        <Slider Value="{Binding RealOffset, Mode=TwoWay}"
                IsEnabled="{Binding IsBusy, Converter={StaticResource negate}}" />

        <Slider Value="{Binding ImaginaryOffset, Mode=TwoWay}"
                IsEnabled="{Binding IsBusy, Converter={StaticResource negate}}" />

        <Label Text="{Binding TargetCenter, StringFormat='{0}'}"
               FontSize="Small"
               HorizontalTextAlignment="Center" />
    </StackLayout>

    <Grid VerticalOptions="CenterAndExpand">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

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

```
        <!-- Magnification factor stepper and display. -->
        <Stepper x:Name="magnificationStepper"
                 Grid.Row="0" Grid.Column="0"
                 Value="{Binding TargetMagnification,
                                 Converter={StaticResource base2log}}"
                 IsEnabled="{Binding IsBusy, Converter={StaticResource negate}}"
                 VerticalOptions="Center" />

        <StackLayout Grid.Row="0" Grid.Column="1"
                     Orientation="Horizontal"
                     Spacing="0"
                     VerticalOptions="Start">
            <Label Text="zoom 2"
                   FontSize="Medium" />
            <Label Text="{Binding Source={x:Reference magnificationStepper},
                                  Path=Value,
                                  StringFormat='{0}'}"
                   FontSize="Micro" />
        </StackLayout>

        <!-- Iterations factor stepper and display. -->
        <Stepper x:Name="iterationsStepper"
                 Grid.Row="1" Grid.Column="0"
                 Value="{Binding Iterations, Converter={StaticResource base2log}}"
                 IsEnabled="{Binding IsBusy, Converter={StaticResource negate}}"
                 VerticalOptions="Center" />

        <StackLayout Grid.Row="1" Grid.Column="1"
                     Orientation="Horizontal"
                     Spacing="0"
                     VerticalOptions="End">
            <Label Text="loop 2"
                   FontSize="Medium" />
            <Label Text="{Binding Source={x:Reference iterationsStepper},
                                  Path=Value,
                                  StringFormat='{0}'}"
                   FontSize="Micro" />
        </StackLayout>

        <!-- Go / Cancel buttons. -->
        <Grid Grid.Row="0" Grid.Column="1" Grid.RowSpan="2"
              HorizontalOptions="End"
              VerticalOptions="Center">

            <Button Text="Go"
                    Command="{Binding CalculateCommand}"
                    IsVisible="{Binding IsBusy, Converter={StaticResource negate}}" />

            <Button Text="Cancel"
                    Command="{Binding CancelCommand}"
                    IsVisible="{Binding IsBusy}" />
        </Grid>
    </Grid>
</StackLayout>
```

```
    </Grid>
</ContentPage>
```

This XAML file only installs three event handlers, and they are all `SizeChanged` handlers.

The first `SizeChanged` handler is on the page itself. This handler is used by the code-behind file to adapt `mainGrid` and its children for portrait or landscape mode using techniques you've seen in previous samples.

The second `SizeChanged` handler is on the `Image` element. The code-behind file uses this to size the `AbsoluteLayout` that displays the crosshairs and magnification box. This `AbsoluteLayout` must be made the same size as the bitmap displayed by the `Image` under the assumption that the `Image` will display a square bitmap.

The third `SizeChanged` handler is on that `AbsoluteLayout`, so the crosshairs and magnification box can be redrawn for a change in size.

The **MandelbrotXF** program also performs a little trick of sorts to ensure that the bitmap contains the optimum number of pixels, which happens when there is a one-to-one mapping between the pixels of the bitmap and the pixels of the display. The XAML file contains a second `Image` element named `testImage`. This `Image` is invisible because the `Opacity` is set to zero, and it is horizontally and vertically centered, which means that it will be displayed with a one-to-one pixel mapping. The code-behind file creates a 120-pixel square bitmap that is set to this `Image`. The resultant size of the `Image` lets the program know how many pixels there are to the device-independent unit, and it can use that to calculate an optimum pixel size for the Mandelbrot bitmap. (Unfortunately it doesn't work for the Windows Runtime platforms.)

Here's roughly the first half of the code-behind file for `MandelbrotXFPage`, showing mostly the instantiation of the `MandelbrotViewModel` class and the interaction of these `SizeChanged` handlers:

```
namespace MandelbrotXF
{
    public partial class MandelbrotXFPage : ContentPage
    {
        MandelbrotViewModel mandelbrotViewModel;
        double pixelsPerUnit = 1;

        public MandelbrotXFPage()
        {
            InitializeComponent();

            // Instantiate ViewModel and get saved values.
            mandelbrotViewModel = new MandelbrotViewModel(2.5, 2.5)
            {
                PixelWidth = 1000,
                PixelHeight = 1000,
                CurrentCenter = new Complex(GetProperty("CenterReal", -0.75),
                                            GetProperty("CenterImaginary", 0.0)),
                CurrentMagnification = GetProperty("Magnification", 1.0),
                TargetMagnification = GetProperty("Magnification", 1.0),
```

```
                Iterations = GetProperty("Iterations", 8),
                RealOffset = 0.5,
                ImaginaryOffset = 0.5
            };

            // Set BindingContext on page.
            BindingContext = mandelbrotViewModel;

            // Set PropertyChanged handler on ViewModel for "manual" processing.
            mandelbrotViewModel.PropertyChanged += OnMandelbrotViewModelPropertyChanged;

            // Create test image to obtain pixels per device-independent unit.
            BmpMaker bmpMaker = new BmpMaker(120, 120);

            testImage.SizeChanged += (sender, args) =>
                {
                    pixelsPerUnit = bmpMaker.Width / testImage.Width;
                    SetPixelWidthAndHeight();
                };
            testImage.Source = bmpMaker.Generate();

            // Gradually reduce opacity of crosshairs.
            Device.StartTimer(TimeSpan.FromMilliseconds(100), () =>
                {
                    realCrossHair.Opacity -= 0.01;
                    imagCrossHair.Opacity -= 0.01;
                    return true;
                });
        }

        // Method for accessing Properties dictionary if key is not yet present.
        T GetProperty<T>(string key, T defaultValue)
        {
            IDictionary<string, object> properties = Application.Current.Properties;

            if (properties.ContainsKey(key))
            {
                return (T)properties[key];
            }
            return defaultValue;
        }

        // Switch between portrait and landscape mode.
        void OnPageSizeChanged(object sender, EventArgs args)
        {
            if (Width == -1 || Height == -1)
                return;

            // Portrait mode.
            if (Width < Height)
            {
                mainGrid.RowDefinitions[1].Height = GridLength.Auto;
                mainGrid.ColumnDefinitions[1].Width = new GridLength(0, GridUnitType.Absolute);
                Grid.SetRow(controlPanelStack, 1);
```

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

    void OnImageSizeChanged(object sender, EventArgs args)
    {
        // Assure that crosshair layout is same size as Image.
        double size = Math.Min(image.Width, image.Height);
        crossHairLayout.WidthRequest = size;
        crossHairLayout.HeightRequest = size;

        // Calculate the pixel size of the Image element.
        SetPixelWidthAndHeight();
    }

    // Sets the Mandelbrot bitmap to optimum pixel width and height.
    void SetPixelWidthAndHeight()
    {
        int pixels = (int)(pixelsPerUnit * Math.Min(image.Width, image.Height));
        mandelbrotViewModel.PixelWidth = pixels;
        mandelbrotViewModel.PixelHeight = pixels;
    }


    // Redraw crosshairs if the crosshair layout changes size.
    void OnCrossHairLayoutSizeChanged(object sender, EventArgs args)
    {
        SetCrossHairs();
    }

    …

    }
}
```

Rather than attach a bunch of event handlers to user-interface elements in the XAML file, the constructor of the code-behind file instead attaches a `PropertyChanged` handler to the `MandelbrotViewModel` instance. Changes to several properties require that the crosshairs and sizing box be redrawn, and any change to any property brings the crosshairs back into view:

```
namespace MandelbrotXF
{
    {
        …
        async void OnMandelbrotViewModelPropertyChanged(object sender,
                                                        PropertyChangedEventArgs args)
```

```
{
    // Set opacity back to 1.
    realCrossHair.Opacity = 1;
    imagCrossHair.Opacity = 1;

    switch (args.PropertyName)
    {
        case "RealOffset":
        case "ImaginaryOffset":
        case "CurrentMagnification":
        case "TargetMagnification":
            // Redraw crosshairs if these properties change
            SetCrossHairs();
            break;

        case "BitmapInfo":
            // Create bitmap based on the iteration counts.
            DisplayNewBitmap(mandelbrotViewModel.BitmapInfo);

            // Save properties for the next time program is run.
            IDictionary<string, object> properties = Application.Current.Properties;
            properties["CenterReal"] = mandelbrotViewModel.TargetCenter.Real;
            properties["CenterImaginary"] = mandelbrotViewModel.TargetCenter.Imaginary;
            properties["Magnification"] = mandelbrotViewModel.TargetMagnification;
            properties["Iterations"] = mandelbrotViewModel.Iterations;
            await Application.Current.SavePropertiesAsync();
            break;
    }
}

void SetCrossHairs()
{
    // Size of the layout for the crosshairs and zoom box.
    Size layoutSize = new Size(crossHairLayout.Width, crossHairLayout.Height);

    // Fractional position of center of crosshair.
    double xCenter = mandelbrotViewModel.RealOffset;
    double yCenter = 1 - mandelbrotViewModel.ImaginaryOffset;

    // Calculate dimension of zoom box.
    double boxSize = mandelbrotViewModel.CurrentMagnification /
                     mandelbrotViewModel.TargetMagnification;

    // Fractional positions of zoom box corners.
    double xLeft = xCenter - boxSize / 2;
    double xRight = xCenter + boxSize / 2;
    double yTop = yCenter - boxSize / 2;
    double yBottom = yCenter + boxSize / 2;

    // Set all the layout bounds.
    SetLayoutBounds(realCrossHair,
                    new Rectangle(xCenter, yTop, 0, boxSize),
                    layoutSize);
    SetLayoutBounds(imagCrossHair,
```

```
                                new Rectangle(xLeft, yCenter, boxSize, 0),
                                layoutSize);
            SetLayoutBounds(topBox, new Rectangle(xLeft, yTop, boxSize, 0), layoutSize);
            SetLayoutBounds(bottomBox, new Rectangle(xLeft, yBottom, boxSize, 0), layoutSize);
            SetLayoutBounds(leftBox, new Rectangle(xLeft, yTop, 0, boxSize), layoutSize);
            SetLayoutBounds(rightBox, new Rectangle(xRight, yTop, 0, boxSize), layoutSize);
        }

        void SetLayoutBounds(View view, Rectangle fractionalRect, Size layoutSize)
        {
            if (layoutSize.Width == -1 || layoutSize.Height == -1)
            {
                AbsoluteLayout.SetLayoutBounds(view, new Rectangle());
                return;
            }

            const double thickness = 1;
            Rectangle absoluteRect = new Rectangle();

            // Horizontal lines.
            if (fractionalRect.Height == 0 && fractionalRect.Y > 0 && fractionalRect.Y < 1)
            {
                double xLeft = Math.Max(0, fractionalRect.Left);
                double xRight = Math.Min(1, fractionalRect.Right);
                absoluteRect = new Rectangle(layoutSize.Width * xLeft,
                                             layoutSize.Height * fractionalRect.Y,
                                             layoutSize.Width * (xRight - xLeft),
                                             thickness);
            }
            // Vertical lines.
            else if (fractionalRect.Width == 0 && fractionalRect.X > 0 && fractionalRect.X < 1)
            {
                double yTop = Math.Max(0, fractionalRect.Top);
                double yBottom = Math.Min(1, fractionalRect.Bottom);
                absoluteRect = new Rectangle(layoutSize.Width * fractionalRect.X,
                                             layoutSize.Height * yTop,
                                             thickness,
                                             layoutSize.Height * (yBottom - yTop));
            }
            AbsoluteLayout.SetLayoutBounds(view, absoluteRect);
        }

        …

    }
}
```

Early versions of the program attempted to use the proportional sizing and positioning facility of `AbsoluteLayout` for the six `BoxView` elements, but it became too difficult. Fractional values are passed to the `SetLayoutBounds` method, but those are used to calculate coordinates based on the size of the `AbsoluteLayout`.

Because Models and ViewModels are supposed to be platform independent, neither `MandelbrotModel` nor `MandelbrotViewModel` get involved with creating the actual bitmap. These classes express the image as a `BitmapInfo` value, which is simply a pixel width and height and an array of integers that correspond to iteration counts. Creating and displaying that bitmap mostly involves using `BmpMaker` and applying a color scheme based on the iteration count:

```
namespace MandelbrotXF
{
    {
        …
        void DisplayNewBitmap(BitmapInfo bitmapInfo)
        {
            // Create the bitmap.
            BmpMaker bmpMaker = new BmpMaker(bitmapInfo.PixelWidth, bitmapInfo.PixelHeight);

            // Set the colors.
            int index = 0;
            for (int row = 0; row < bitmapInfo.PixelHeight; row++)
            {
                for (int col = 0; col < bitmapInfo.PixelWidth; col++)
                {
                    int iterationCount = bitmapInfo.IterationCounts[index++];

                    // In the Mandelbrot set: Color black.
                    if (iterationCount == -1)
                    {
                        bmpMaker.SetPixel(row, col, 0, 0, 0);
                    }
                    // Not in the Mandelbrot set: Pick a color based on count.
                    else
                    {
                        double proportion = (iterationCount / 32.0) % 1;

                        if (proportion < 0.5)
                        {
                            bmpMaker.SetPixel(row, col, (int)(255 * (1 - 2 * proportion)),
                                                        0,
                                                        (int)(255 * 2 * proportion));
                        }
                        else
                        {
                            proportion = 2 * (proportion - 0.5);
                            bmpMaker.SetPixel(row, col, 0,
                                                        (int)(255 * proportion),
                                                        (int)(255 * (1 - proportion)));
                        }
                    }
                }
            }
            image.Source = bmpMaker.Generate();
        }
    }
}
```

Feel free to experiment with the color scheme. One easy alternative is to vary the hue of an HSL color with the iteration count:

```
double hue = (iterationCount / 64.0) % 1;
bmpMaker.SetPixel(row, col, Color.FromHsla(hue, 1, 0.5));
```

Back to the web

Prior to this chapter, the only asynchronous code in this book involved web accesses using the only reasonable class available for that purpose in the Portable Class Library, `WebRequest`. The `WebRequest` class uses an older asynchronous protocol called the Asynchronous Programming Model or APM. APM involves two methods, in the case of `WebRequest`, these are called `BeginGetResponse` and `EndGetResponse`.

You can convert this pair of method calls into the Task-based Asynchronous Pattern (TAP) by using the `FromAsync` method of `TaskFactory`, and the **ApmToTap** program demonstrates how. The program uses a web access and `ImageSource.FromStream` to load a bitmap and display it. This technique was shown in Chapter 13 as an alternative to `ImageSource.FromUri`.

The XAML file contains an `Image` element awaiting a bitmap, an `ActivityIndicator` that runs when the bitmap is loading, a `Label` to display a possible error message, and a `Button` to start the download:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ApmToTap.ApmToTapPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="0, 20, 0, 0" />
    </ContentPage.Padding>

    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <Label x:Name="errorLabel"
                   HorizontalOptions="Center"
                   VerticalOptions="Center" />

            <ActivityIndicator IsRunning="{Binding Source={x:Reference image},
                                                   Path=IsLoading}" />

            <Image x:Name="image" />
        </Grid>

        <Button Text="Load Bitmap"
                HorizontalOptions="Center"
                Clicked="OnLoadButtonClicked" />
    </StackLayout>
</ContentPage>
```

The code-behind file consolidates all the `WebRequest` code in an asynchronous method named `GetStreamAsync`. After the `TaskFactory` and `WebRequest` objects are instantiated, the method passes the `BeginGetResponse` and `EndGetResponse` methods to the `FromAsync` method of `TaskFactory`, which then returns a `WebResponse` object from which a `Stream` is available:

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

    async void OnLoadButtonClicked(object sender, EventArgs args)
    {
        try
        {
            Stream stream =
                      await GetStreamAsync("https://developer.xamarin.com/demo/IMG_1996.JPG");
            image.Source = ImageSource.FromStream(() => stream);
        }
        catch (Exception exc)
        {
            errorLabel.Text = exc.Message;
        }
    }

    async Task<Stream> GetStreamAsync(string uri)
    {
        TaskFactory factory = new TaskFactory();
        WebRequest request = WebRequest.Create(uri);
        WebResponse response = await factory.FromAsync<WebResponse>(request.BeginGetResponse,
                                                                     request.EndGetResponse,
                                                                     null);
        return response.GetResponseStream();
    }
}
```

The `Clicked` handler for the `Button` can then get that `Stream` object by calling `GetStreamAsync` with a URI. As usual, the code with the `await` operator is in a `try` block to catch any possible errors. You can experiment a bit by deliberately misspelling the domain or filename to see what kind of errors you get.

Another option for web accesses is a class named `HttpClient` in the `System.Net.Http` namespace. This class is not available in the version of .NET included in the Portable Class Library in a Xamarin.Forms solution, but Microsoft has made the class available as a NuGet package:

https://www.nuget.org/packages/Microsoft.Net.Http

From the NuGet manager in Visual Studio or Xamarin Studio, just search for “HttpClient”.

`HttpClient` is based on TAP. The asynchronous methods return `Task` and `Task<T>` objects, and some of the methods also have `CancellationToken` arguments.

None of the methods report progress, however, which suggests that a first-rate modern class for web accesses is still not yet available to Portable Class Libraries.

In the next chapter you'll see many more uses of `await` and explore some other features of the Task-based Asynchronous Pattern in connection with the exciting Xamarin.Forms implementation of animation.