Chapter 18 MVVM

Can you remember your earliest experiences with programming? It's likely that your main goal was just getting the program working, and then getting it working correctly. You probably didn't think much about the organization or structure of the program. That was something that came later.

The computer industry as a whole has gone through a similar evolution. As developers, we all now realize that once an application begins growing in size, it's usually a good idea to impose some kind of structure or architecture on the code. Experience with this process suggests that it's often best to start thinking about this architecture perhaps before any code is written at all. In most cases, a desirable program structure strives for a "separation of concerns" through which different pieces of the program focus on different sorts of tasks.

In a graphically interactive program, one obvious technique is to separate the user interface from underlying non-user-interface logic, sometimes called *business logic*. The first formal description of such an architecture for graphical user interfaces was called Model-View-Controller (MVC), but this architecture has since given rise to others derived from it.

To some extent, the nature of the programming interface itself influences the application architecture. For example, a programming interface that includes a markup language with data bindings might suggest a particular way to structure an application.

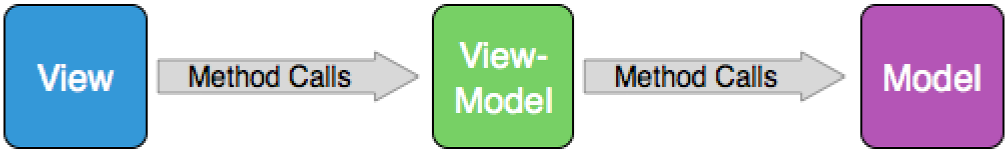
There is indeed an architectural model that was designed specifically with XAML in mind. This is known as Model-View-ViewModel or MVVM. This chapter covers the basics of MVVM (including the command interface), but you'll see more about MVVM in the next chapter, which covers collection views. Also, some other features of Xamarin.Forms are often used in conjunction with MVVM; these features include *triggers* and *behaviors*, and they are the subject of Chapter 23.

MVVM interrelationships

MVVM divides an application into three layers:

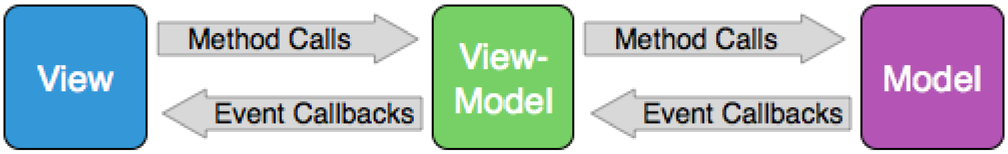
- The Model provides underlying data, sometimes involving file or web accesses.
- The ViewModel connects the Model and the View. It helps to manage the data from the Model to make it more amenable to the View, and vice versa.
- The View is the user interface or presentation layer, generally implemented in XAML.

The Model is ignorant of the ViewModel. In other words, the Model knows nothing about the public properties and methods of the ViewModel, and certainly nothing about its internal workings. Similarly, the ViewModel is ignorant of the View. If all the communication between the three layers occurs through method calls and property accesses, then calls in only one direction are allowed. The View only makes calls into the ViewModel or accesses properties of the ViewModel, and the ViewModel similarly only makes calls into the Model or accesses Model properties:

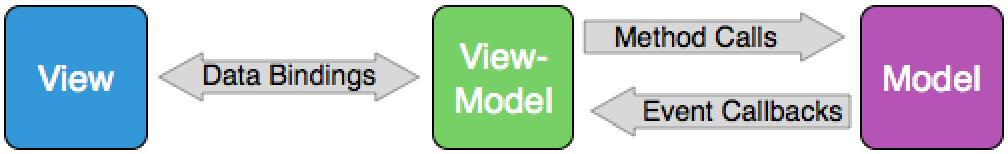


These method calls allow the View to get information from the ViewModel, which in turn gets information from the Model.

In modern environments, however, data is often dynamic. Often the Model will obtain more or newer data that must be communicated to the ViewModel and eventually to the View. For this reason, the View can attach handlers to events that are implemented in the ViewModel, and the ViewModel can attach handlers to events defined by the Model. This allows two-way communication while continuing to hide the View from the ViewModel, and the ViewModel from the Model:



MVVM was designed to take advantage of XAML and particularly XAML-based data bindings. Generally, the View is a page class that uses XAML to construct the user interface. Therefore, the connection between the View and the ViewModel consists largely—and perhaps exclusively—of XAML-based data bindings:



Programmers who are very passionate about MVVM often have an informal goal of expressing all interactions between the View and the ViewModel in a page class with XAML-based data bindings, and in the process reducing the code in the page's code-behind file to a simple `InitializeComponent` call. This goal is difficult to achieve in real-life programming, but it's a pleasure when it happens.

Small programs—such as those in a book like this—often become larger when MVVM is introduced. Do not let this discourage your use of MVVM! Use the examples here to help you determine how

MVVM can be used in a larger program, and you'll eventually see that it helps enormously in architecting your applications.

ViewModels and data binding

In many fairly simple demonstrations of MVVM, the Model is absent or only implied, and the ViewModel contains all the business logic. The View and the ViewModel communicate through XAML-based data bindings. The visual elements in the View are data-binding targets, and properties in the ViewModel are data-binding sources.

Ideally, a ViewModel should be independent of any particular platform. This independence allows ViewModels to be shared among other XAML-based environments (such as Windows) in addition to Xamarin.Forms. For this reason, you should try to avoid using the following statement in your ViewModels:

```
using Xamarin.Forms;
```

That rule is frequently broken in this chapter! One of the ViewModels is based on the Xamarin.Forms `Color` structure, and another uses `Device.StartTimer`. So let's call the avoidance of anything specific to Xamarin.Forms in the ViewModel a "suggestion" rather than a "rule."

Visual elements in the View qualify as data-binding targets because the properties of these visual elements are backed by bindable properties. To be a data-binding source, a ViewModel must implement a notification protocol to signal when a property in the ViewModel has changed. This notification protocol is the `INotifyPropertyChanged` interface, which is defined in the `System.ComponentModel` namespace very simply with just one event:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

The `INotifyPropertyChanged` interface is so central to MVVM that in informal discussions the interface is often abbreviated INPC.

The `PropertyChanged` event in the `INotifyPropertyChanged` interface is of type `Property-Changed-EventHandler`. A handler for this `PropertyChanged` event handler gets an instance of the `PropertyChangedEventArgs` class, which defines a single property named `PropertyName` of type `string` indicating what property in the ViewModel has changed. The event handler can then access that property.

A class that implements `INotifyPropertyChanged` should fire a `PropertyChanged` event whenever a public property changes, but the class should *not* fire the event when the property is merely set but not changed.

Some classes define immutable properties—properties that are initialized in the constructor and then never change. Those properties do not need to fire PropertyChanged events because a PropertyChanged handler can be attached only after the code in the constructor finishes, and the immutable properties never change after that time.

In theory, a ViewModel class can be derived from BindableObject and implement its public properties as BindableProperty objects. BindableObject implements INotifyPropertyChanged and automatically fires a PropertyChanged event when any property backed by a BindableProperty changes. But deriving from BindableObject is overkill for a ViewModel. Because BindableObject and BindableProperty are specific to Xamarin.Forms, such a ViewModel is no longer platform independent, and the technique provides no real advantages over a simpler implementation of INotifyPropertyChanged.

A ViewModel clock

Suppose you are writing a program that needs access to the current date and time, and you'd like to use that information through data bindings. The .NET base class library provides date and time information through the DateTime structure. To get the current date and time, just access the DateTime.Now property. That's the customary way to write a clock application.

But for data-binding purposes, DateTime has a severe flaw: It provides just static information with no notification when the date or time has changed.

In the context of MVVM, the DateTime structure perhaps qualifies as a Model in the sense that DateTime provides all the data we need but not in a form that's conducive to data bindings. It's necessary to write a ViewModel that makes use of DateTime but provides notifications when the date or time has changed.

The **Xamarin.FormsBook.Toolkit** library contains the DateTimeViewModel class shown below. The class has only one property, which is named DateTime of type DateTime, but this property dynamically changes as a result of frequent calls to DateTime.Now in a Device.StartTimer callback.

Notice that the DateTimeViewModel class is based on the INotifyPropertyChanged interface and includes a using directive for the System.ComponentModel namespace that defines this interface. To implement this interface, the class defines a public event named PropertyChanged.

Watch out: It is very easy to define a PropertyChanged event in your class without explicitly specifying that the class implements INotifyPropertyChanged! The notifications will be ignored if you don't explicitly specify that the class is based on the INotifyPropertyChanged interface:

```
using System;
using System.ComponentModel;
using Xamarin.Forms;

namespace Xamarin.FormsBook.Toolkit
{
    public class DateTimeViewModel : INotifyPropertyChanged
    {
        DateTime dateTime = DateTime.Now;

        public event PropertyChangedEventHandler PropertyChanged;

        public DateTimeViewModel()
        {
            Device.StartTimer(TimeSpan.FromMilliseconds(15), OnTimerTick);
        }

        bool OnTimerTick()
        {
            DateTime = DateTime.Now;
            return true;
        }

        public DateTime DateTime
        {
            private set
            {
                if (dateTime != value)
                {
                    dateTime = value;

                    // Fire the event.
                    PropertyChangedEventHandler handler = PropertyChanged;

                    if (handler != null)
                    {
                        handler(this, new PropertyChangedEventArgs("DateTime"));
                    }
                }
            }

            get
            {
                return dateTime;
            }
        }
    }
}
```

The only public property in this class is called `DateTime` of type `DateTime`, and it is associated with a private backing field named `dateTime`. Public properties in ViewModels usually have private backing fields. The `set` accessor of the `DateTime` property is private to the class, and it's updated every 15 milliseconds from the timer callback.

Other than that, the `set` accessor is constructed in a very standard way for ViewModels: It first checks whether the value being set to the property is different from the `dateTime` backing field. If not, it sets that backing field from the incoming value and fires the `PropertyChanged` handler with the name of the property. It is considered very bad practice to fire the `PropertyChanged` handler if the property is merely being set to its existing value, and it might even lead to problems involving infinite cycles of recursive property settings in two-way bindings.

This is the code in the `set` accessor that fires the event:

```
PropertyChangedEventHandler handler = PropertyChanged;

if (handler != null)
{
    handler(this, new PropertyChangedEventArgs("DateTime"));
}
```

That form is preferable to code such as this, which doesn't save the handler in a separate variable:

```
if (PropertyChanged != null)
{
    PropertyChanged(this, new PropertyChangedEventArgs("DateTime"));
}
```

In a multithreaded environment, a `PropertyChanged` handler might be detached between the `if` statement that checks for a `null` value and the actual firing of the event. Saving the handler in a separate variable prevents that from causing a problem, so it's a good habit to adopt even if you're not yet working in a multithreaded environment.

The `get` accessor simply returns the `dateTime` backing field.

The **MvvmClock** program demonstrates how the `DateTimeViewModel` class is capable of providing updated date and time information to the user interface through data bindings:

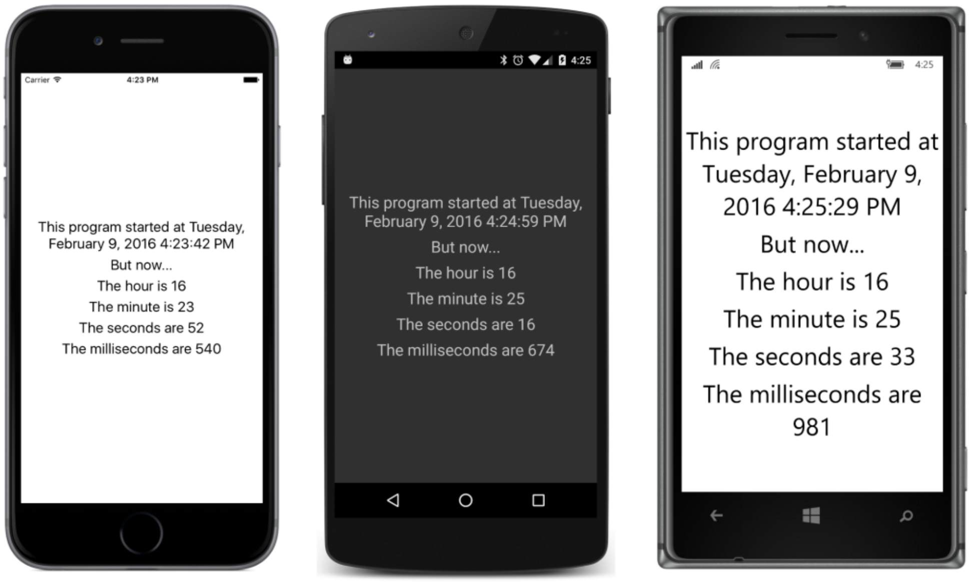
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             xmlns:toolkit=
                 "clr-namespace:Xamarin.FormsBook.Toolkit;assembly=Xamarin.FormsBook.Toolkit"
             x:Class="MvvmClock.MvvmClockPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <toolkit:DateTimeViewModel x:Key="dateTimeViewModel" />

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

    <StackLayout VerticalOptions="Center">
        <Label Text="{Binding Source={x:Static sys:DateTime.Now},
                              StringFormat='This program started at {0:F}'}" />

        <Label Text="But now..." />
        <Label Text="{Binding Source={StaticResource dateTimeViewModel},
                              Path=DateTime.Hour,
                              StringFormat='The hour is {0}'}" />

        <Label Text="{Binding Source={StaticResource dateTimeViewModel},
                              Path=DateTime.Minute,
                              StringFormat='The minute is {0}'}" />

        <Label Text="{Binding Source={StaticResource dateTimeViewModel},
                              Path=DateTime.Second,
                              StringFormat='The seconds are {0}'}" />

        <Label Text="{Binding Source={StaticResource dateTimeViewModel},
                              Path=DateTime.Millisecond,
                              StringFormat='The milliseconds are {0}'}" />
    </StackLayout>
</ContentPage>
```

The `Resources` section for the page instantiates the `DateTimeViewModel` and also defines an implicit `Style` for the `Label`.

The first of the six `Label` elements sets its `Text` property to a `Binding` object that involves the actual .NET `DateTime` structure. The `Source` property of that binding is an `x:Static` markup extension that references the static `DateTime.Now` property to obtain the date and time when the program first starts running. No `Path` is required in this binding. The “F” formatting specification is for the full date/time pattern, with long versions of the date and time strings. Although this `Label` displays the date and time when the program starts up, it will never get updated.

The final four data bindings *will* be updated. In these data bindings, the `Source` property is set to a `StaticResource` markup extension that references the `DateTimeViewModel` object. The `Path` is set to various subproperties of the `DateTime` property of that ViewModel. Behind the scenes, the binding infrastructure attaches a handler on the `PropertyChanged` event in the `DateTimeViewModel`. This handler checks for a change in the `DateTime` property and updates the `Text` property of the `Label` whenever that property changes.

The code-behind file is empty except for an `InitializeComponent` call. The data bindings of the final four labels display an updated time that changes as fast as the video refresh rate:



This program started at Tuesday, February 9, 2016 4:23:42 PM
But now...
The hour is 16
The minute is 23
The seconds are 52
The milliseconds are 540

This program started at Tuesday, February 9, 2016 4:24:59 PM
But now...
The hour is 16
The minute is 25
The seconds are 16
The milliseconds are 674

This program started at Tuesday, February 9, 2016 4:25:29 PM
But now...
The hour is 16
The minute is 25
The seconds are 33
The milliseconds are 981

The markup in this XAML file can be simplified by setting the `BindingContext` property of the `StackLayout` to a `StaticResource` markup extension that references the ViewModel. That `BindingContext` is propagated through the visual tree so that you can remove the `Source` settings on the final four `Label` elements:

```
<StackLayout VerticalOptions="Center"
             BindingContext="{StaticResource dateTimeViewModel}">

    <Label Text="{Binding Source={x:Static sys:DateTime.Now},
                          StringFormat='This program started at {0:F}'}" />

    <Label Text="But now..." />

    <Label Text="{Binding Path=DateTime.Hour,
                          StringFormat='The hour is {0}'}" />

    <Label Text="{Binding Path=DateTime.Minute,
                          StringFormat='The minute is {0}'}" />

    <Label Text="{Binding Path=DateTime.Second,
                          StringFormat='The seconds are {0}'}" />

    <Label Text="{Binding Path=DateTime.Millisecond,
                          StringFormat='The milliseconds are {0}'}" />
</StackLayout>
```

The `Binding` on the first `Label` overrides that `BindingContext` with its own `Source` setting.

You can even remove the `DateTimeViewModel` item from the `ResourceDictionary` and instantiate it right in the `StackLayout` between `BindingContext` property-element tags:

```
<StackLayout VerticalOptions="Center">
    <StackLayout.BindingContext>
        <toolkit:DateTimeViewModel />
    </StackLayout.BindingContext>

    <Label Text="{Binding Source={x:Static sys:DateTime.Now},
                          StringFormat='This program started at {0:F}'}" />

    <Label Text="But now..." />

    <Label Text="{Binding Path=DateTime.Hour,
                          StringFormat='The hour is {0}'}" />

    <Label Text="{Binding Path=DateTime.Minute,
                          StringFormat='The minute is {0}'}" />

    <Label Text="{Binding Path=DateTime.Second,
                          StringFormat='The seconds are {0}'}" />

    <Label Text="{Binding Path=DateTime.Millisecond,
                          StringFormat='The milliseconds are {0}'}" />
</StackLayout>
```

Or, you can set the `BindingContext` property of the `StackLayout` to a `Binding` that includes the `DateTime` property. The `BindingContext` then becomes the `DateTime` value, which allows the individual bindings to simply reference properties of the .NET `DateTime` structure:

```
<StackLayout VerticalOptions="Center"
             BindingContext="{Binding Source={StaticResource dateTimeViewModel},
                                      Path=DateTime}">

    <Label Text="{Binding Source={x:Static sys:DateTime.Now},
                          StringFormat='This program started at {0:F}'}" />

    <Label Text="But now..." />

    <Label Text="{Binding Path=Hour,
                          StringFormat='The hour is {0}'}" />

    <Label Text="{Binding Path=Minute,
                          StringFormat='The minute is {0}'}" />

    <Label Text="{Binding Path=Second,
                          StringFormat='The seconds are {0}'}" />

    <Label Text="{Binding Path=Millisecond,
                          StringFormat='The milliseconds are {0}'}" />
</StackLayout>
```

You might have doubts that this will work! Behind the scenes, a data binding normally installs a `PropertyChanged` event handler and watches for particular properties being changed, but it can't in this case because the source of the data binding is a `DateTime` value, and `DateTime` doesn't implement `INotifyPropertyChanged`. However, the `BindingContext` of these `Label` elements changes with each change to the `DateTime` property in the ViewModel, so the binding infrastructure accesses new values of these properties at that time.

As the individual bindings on the `Text` properties decrease in length and complexity, you can remove the `Path` attribute name and put everything on one line and nobody will be confused:

```
<StackLayout VerticalOptions="Center">
    <StackLayout.BindingContext>
        <Binding Path="DateTime">
            <Binding.Source>
                <toolkit:DateTimeViewModel />
            </Binding.Source>
        </Binding>
    </StackLayout.BindingContext>

    <Label Text="{Binding Source={x:Static sys:DateTime.Now},
                          StringFormat='This program started at {0:F}'}" />

    <Label Text="But now..." />

    <Label Text="{Binding Hour, StringFormat='The hour is {0}'}" />
    <Label Text="{Binding Minute, StringFormat='The minute is {0}'}" />
    <Label Text="{Binding Second, StringFormat='The seconds are {0}'}" />
    <Label Text="{Binding Millisecond, StringFormat='The milliseconds are {0}'}" />
</StackLayout>
```

In future programs in this book, the individual bindings will mostly be as short and as elegant as possible.

Interactive properties in a ViewModel

The second example of a ViewModel does something so basic that you'd never write a ViewModel for this purpose. The `SimpleMultiplierViewModel` class simply multiplies two numbers together. But it's a good example for demonstrating the overhead and mechanics of a ViewModel that has multiple interactive properties. (And although you'd never write a ViewModel for multiplying two numbers together, you might write a ViewModel for solving quadratic equations or something much more complex.)

The `SimpleMultiplierViewModel` class is part of the **SimpleMultiplier** project:

```
using System;
using System.ComponentModel;

namespace SimpleMultiplier
{
    class SimpleMultiplierViewModel : INotifyPropertyChanged
    {
        double multiplicand, multiplier, product;

        public event PropertyChangedEventHandler PropertyChanged;
        public double Multiplicand
        {
            set
            {
                if (multiplicand != value)
                {
                    multiplicand = value;
                    OnPropertyChanged("Multiplicand");
                    UpdateProduct();
                }
            }
            get
            {
                return multiplicand;
            }
        }

        public double Multiplier
        {
            set
            {
                if (multiplier != value)
                {
                    multiplier = value;
                    OnPropertyChanged("Multiplier");
                    UpdateProduct();
                }
            }
            get
            {
                return multiplier;
            }
        }

        public double Product
        {
            protected set
            {
                if (product != value)
                {
                    product = value;
                    OnPropertyChanged("Product");
                }
            }
            get
            {
                return product;
            }
        }

        void UpdateProduct()
        {
            Product = Multiplicand * Multiplier;
        }
        protected void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;

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

The class defines three public properties of type `double`, named `Multiplicand`, `Multiplier`, and `Product`. Each property is backed by a private field. The `set` and `get` accessors of the first two properties are public, but the `set` accessor of the `Product` property is protected to prevent it from being set outside the class while still allowing a descendant class to change it.

The `set` accessor of each property begins by checking whether the property value is actually changing, and if so, it sets the backing field to that value and calls a method named `OnPropertyChanged` with that property name.

The `INotifyPropertyChanged` interface does not require an `OnPropertyChanged` method, but ViewModel classes often include one to cut down the code repetition. It's usually defined as `protected` in case you need to derive one ViewModel from another and fire the event in the derived class. Later in this chapter, you'll see techniques to cut down the code repetition in `INotifyPropertyChanged` classes even more.

The `set` accessors for both the `Multiplicand` and `Multiplier` properties conclude by calling the `UpdateProduct` method. This is the method that performs the job of multiplying the values of the two properties and setting a new value for the `Product` property, which then fires its own `PropertyChanged` event.

Here's the XAML file that makes use of this ViewModel:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:SimpleMultiplier"
             x:Class="SimpleMultiplier.SimpleMultiplierPage"
             Padding="10, 0">

    <ContentPage.Resources>
        <ResourceDictionary>
            <local:SimpleMultiplierViewModel x:Key="viewModel" />

            <Style TargetType="Label">
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <StackLayout BindingContext="{StaticResource viewModel}">

        <StackLayout VerticalOptions="CenterAndExpand">
            <Slider Value="{Binding Multiplicand}" />
            <Slider Value="{Binding Multiplier}" />
        </StackLayout>

        <StackLayout Orientation="Horizontal"
                     Spacing="0"
                     VerticalOptions="CenterAndExpand"
                     HorizontalOptions="Center">
            <Label Text="{Binding Multiplicand, StringFormat='{0:F3}'}" />
            <Label Text="{Binding Multiplier, StringFormat=' x {0:F3}'}" />
            <Label Text="{Binding Product, StringFormat=' = {0:F3}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The `SimpleMultiplierViewModel` is instantiated in the `Resources` dictionary and set to the `BindingContext` property of the `StackLayout` by using a `StaticResource` markup extension. That `BindingContext` is inherited by all the children and grandchildren of the `StackLayout`, which includes two `Slider` and three `Label` elements. The use of the `BindingContext` allows these bindings to be as simple as possible.

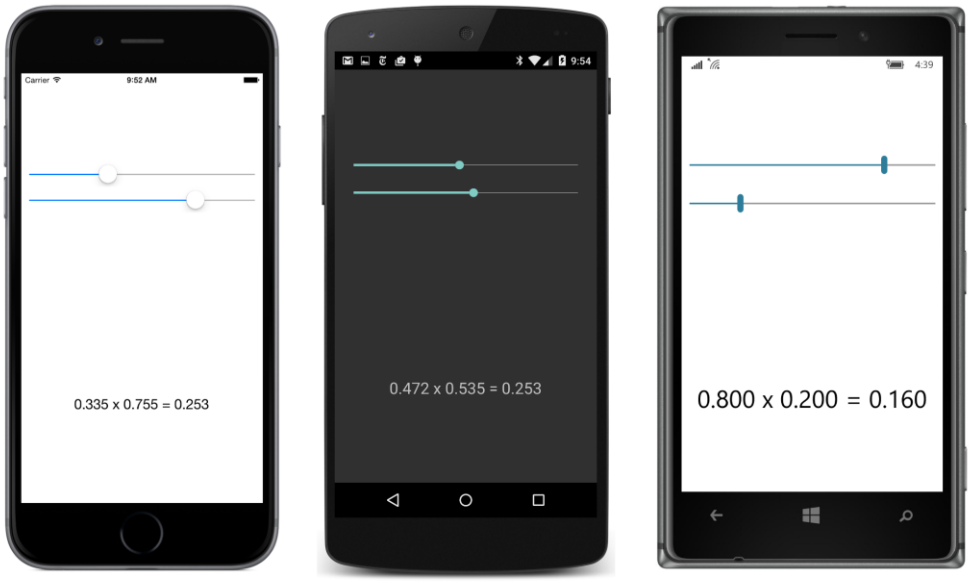
The default binding mode of the `Value` property of the `Slider` is `TwoWay`. Changes in the `Value` property of each `Slider` cause changes to the properties of the ViewModel.

The three `Label` elements display the values of all three properties of the ViewModel with some formatting that inserts times and equals signs with the numbers:

```
<Label Text="{Binding Multiplicand, StringFormat='{0:F3}'}" />
<Label Text="{Binding Multiplier, StringFormat=' x {0:F3}'}" />
<Label Text="{Binding Product, StringFormat=' = {0:F3}'}" />
```

For the first two, you can alternatively bind the `Text` property of the `Label` elements directly to the `Value` property of the corresponding `Slider`, but that would require that you give each `Slider` a name with `x:Name` and reference that name in a `Source` argument by using the `x:Reference` markup extension. The approach used in this program is much cleaner and verifies that data is making a full trip through the ViewModel from each `Slider` to each `Label`.

There is nothing in the code-behind file except a call to `InitializeComponent` in the constructor. All the business logic is in the ViewModel, and the whole user interface is defined in XAML:



If you'd like to, you can initialize the ViewModel as it is instantiated in the `Resources` dictionary:

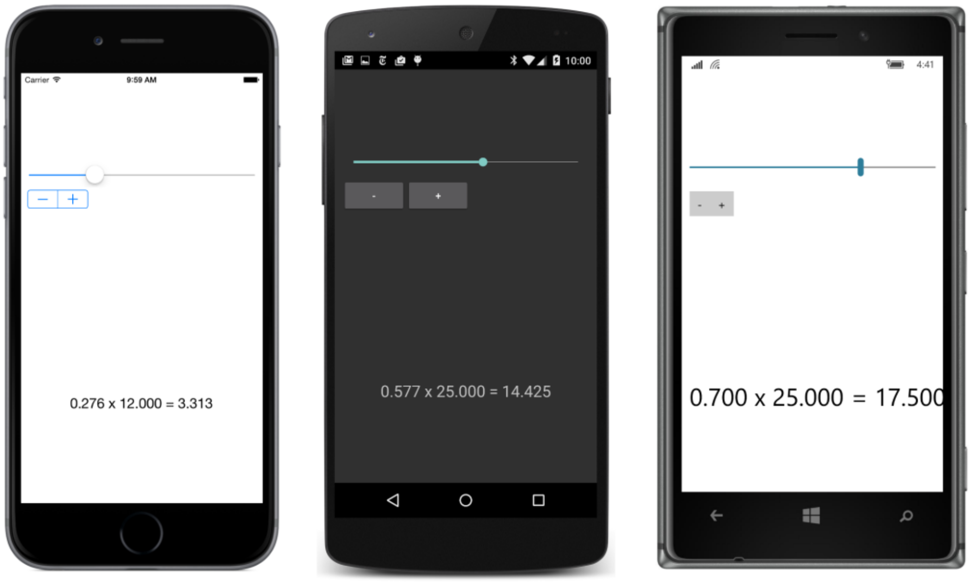
```
<local:SimpleMultiplierViewModel x:Key="viewModel"
                                 Multiplicand="0.5"
                                 Multiplier="0.5" />
```

The `Slider` elements will get these initial values as a result of the two-way binding.

The advantage to separating the user interface from the underlying business logic becomes evident when you want to change the user interface somewhat, perhaps by substituting a `Stepper` for the `Slider` for one or both numbers:

```
<StackLayout VerticalOptions="CenterAndExpand">
    <Slider Value="{Binding Multiplicand}" />
    <Stepper Value="{Binding Multiplier}" />
</StackLayout>
```

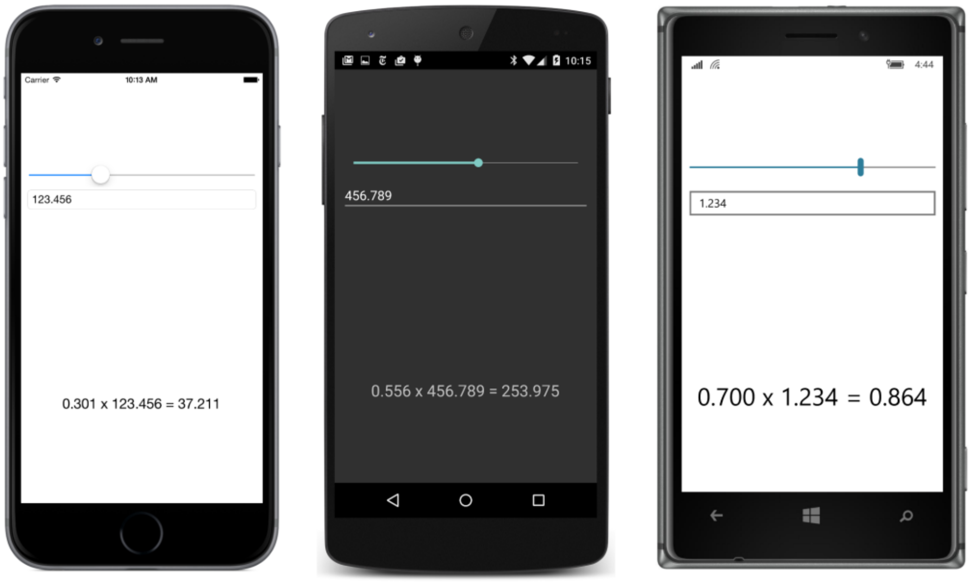
Aside from the different ranges of the two elements, the functionality is identical:



You could also substitute an `Entry`:

```
<StackLayout VerticalOptions="CenterAndExpand">
    <Slider Value="{Binding Multiplicand}" />
    <Entry Text="{Binding Multiplier}" />
</StackLayout>
```

The default binding mode for the `Text` property of the `Entry` is also `TwoWay`, so all you need to worry about is the conversion between the source property `double` and target property `string`. Fortunately, this conversion is automatically handled by the binding infrastructure:



If you type a series of characters that cannot be converted to a `double`, the binding will maintain the last valid value. If you want more sophisticated validation, you'll have to implement your own (such as with a trigger, which will be discussed in Chapter 23).

One interesting experiment is to type **1E-1**, which is scientific notation that is convertible to a `double`. You'll see it immediately change to "0.1" in the `Entry`. This is the effect of the `TwoWay` binding: The `Multiplier` property is set to 1E-1 from the `Entry` but the `ToString` method that the binding infrastructure calls when the value comes back to the `Entry` returns the text "0.1." Because that is different from the existing `Entry` text, the new text is set. To prevent that from happening, you can set the binding mode to `OneWayToSource`:

```
<StackLayout VerticalOptions="CenterAndExpand">
    <Slider Value="{Binding Multiplicand}" />
    <Entry Text="{Binding Multiplier, Mode=OneWayToSource}" />
</StackLayout>
```

Now the `Multiplier` property of the ViewModel is set from the `Text` property of the `Entry`, but not the other way around. If you don't need these two views to be updated from the ViewModel, you can set both of them to `OneWayToSource`. But generally you'll want MVVM bindings to be `TwoWay`.

Should you worry about infinite cycles in two-way bindings? Usually not, because `PropertyChanged` events are fired only when the property actually changes and not when it's merely set to the same value. Generally the source and target will stop updating each other after a bounce or two. However, it is possible to write a "pathological" value converter that doesn't provide for round-trip conversions, and that could indeed cause infinite update cycles in two-way bindings.

A Color ViewModel

Color always provides a good means of exploring the features of a graphical user interface, so you probably won't be surprised to learn that the **Xamarin.FormsBook.Toolkit** library contains a class called `ColorViewModel`.

The `ColorViewModel` class exposes a `Color` property but also `Red`, `Green`, `Blue`, `Alpha`, `Hue`, `Saturation`, and `Luminosity` properties, all of which are individually settable. This is not a feature that the Xamarin.Form `Color` structure provides. Once a `Color` value is created from a `Color` constructor or one of the methods in `Color` beginning with the words `Add`, `From`, `Multiply`, or `With`, it is immutable.

This `ColorViewModel` class is complicated by the interrelationship of its `Color` property and all the component properties. For example, suppose the `Color` property is set. The class should fire a `PropertyChanged` handler not only for `Color` but also for any component (such as `Red` or `Hue`) that also changes. Similarly, if the `Red` property changes, then the class should fire a `PropertyChanged` event for both `Red` and `Color`, and probably `Hue`, `Saturation`, and `Luminosity` as well.

The `ColorViewModel` class solves this problem by storing a backing field for the `Color` property only. All the `set` accessors for the individual components create a new `Color` by using the incoming value with a call to `Color.FromRgba` or `Color.FromHsla`. This new `Color` value is set to the `Color` property rather than the `color` field, which means that the new `Color` value is subjected to processing in the `set` accessor of the `Color` property:

```
public class ColorViewModel : INotifyPropertyChanged
{
    Color color;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Red
    {
        set
        {
            if (Round(color.R) != value)
                Color = Color.FromRgba(value, color.G, color.B, color.A);
        }
        get
        {
            return Round(color.R);
        }
    }

    public double Green
    {
        set
        {
            if (Round(color.G) != value)
                Color = Color.FromRgba(color.R, value, color.B, color.A);
        }
        get
        {
            return Round(color.G);
        }
    }

    public double Blue
    {
        set
        {
            if (Round(color.B) != value)
                Color = Color.FromRgba(color.R, color.G, value, color.A);
        }
        get
        {
            return Round(color.B);
        }
    }

    public double Alpha
    {
        set
        {
            if (Round(color.A) != value)
                Color = Color.FromRgba(color.R, color.G, color.B, value);
        }
        get
        {
            return Round(color.A);
        }
    }

    public double Hue
    {
        set
        {
            if (Round(color.Hue) != value)
                Color = Color.FromHsla(value, color.Saturation, color.Luminosity, color.A);
        }
        get
        {
            return Round(color.Hue);
        }
    }

    public double Saturation
    {
        set
        {
            if (Round(color.Saturation) != value)
                Color = Color.FromHsla(color.Hue, value, color.Luminosity, color.A);
        }
        get
        {
            return Round(color.Saturation);
        }
    }

    public double Luminosity
    {
        set
        {
            if (Round(color.Luminosity) != value)
                Color = Color.FromHsla(color.Hue, color.Saturation, value, color.A);
        }
        get
        {
            return Round(color.Luminosity);
        }
    }

    public Color Color
    {
        set
        {
            Color oldColor = color;

            if (color != value)
            {
                color = value;
                OnPropertyChanged("Color");
            }

            if (color.R != oldColor.R)
                OnPropertyChanged("Red");

            if (color.G != oldColor.G)
                OnPropertyChanged("Green");

            if (color.B != oldColor.B)
                OnPropertyChanged("Blue");

            if (color.A != oldColor.A)
                OnPropertyChanged("Alpha");

            if (color.Hue != oldColor.Hue)
                OnPropertyChanged("Hue");

            if (color.Saturation != oldColor.Saturation)
                OnPropertyChanged("Saturation");

            if (color.Luminosity != oldColor.Luminosity)
                OnPropertyChanged("Luminosity");
        }
        get
        {
            return color;
        }
    }

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

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

    double Round(double value)
    {
        return Device.OnPlatform(value, Math.Round(value, 3), value);
    }
}
```

The `set` accessor for the `Color` property is responsible for the firings of all `PropertyChanged` events based on changes to the properties.

Notice the device-dependent `Round` method at the bottom of the class and its use in the `set` and `get` accessors of the first seven properties. This was added when the **MultiColorSliders** sample in Chapter 23, "Triggers and behaviors," revealed a problem. Android seemed to be internally rounding the color components, causing inconsistencies between the properties being passed to the `Color.FromRgba` and `Color.FromHsla` methods and the properties of the resultant `Color` value, which lead to infinite `set` and `get` loops.

The **HslSliders** program instantiates the `ColorViewModel` between `Grid.BindingContext` tags so that it becomes the `BindingContext` for all the `Slider` and `Label` elements within the `Grid`:

```
<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="HslSliders.HslSlidersPage"
             SizeChanged="OnPageSizeChanged">

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

    <Grid x:Name="mainGrid">
        <Grid.BindingContext>
            <toolkit:ColorViewModel Color="Gray" />
        </Grid.BindingContext>

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

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

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

        <BoxView Color="{Binding Color}"
                 Grid.Row="0" Grid.Column="0" />

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

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider Value="{Binding Hue}" />
                <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
            </StackLayout>

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider Value="{Binding Saturation}" />
                <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
            </StackLayout>

            <StackLayout VerticalOptions="CenterAndExpand">
                <Slider Value="{Binding Luminosity}" />
                <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
            </StackLayout>
        </StackLayout>
    </Grid>
</ContentPage>
```

Notice that the `Color` property of `ColorViewModel` is initialized when `ColorViewModel` is instantiated. The two-way bindings of the sliders then pick up the resultant values of the `Hue`, `Saturation`, and `Luminosity` properties.

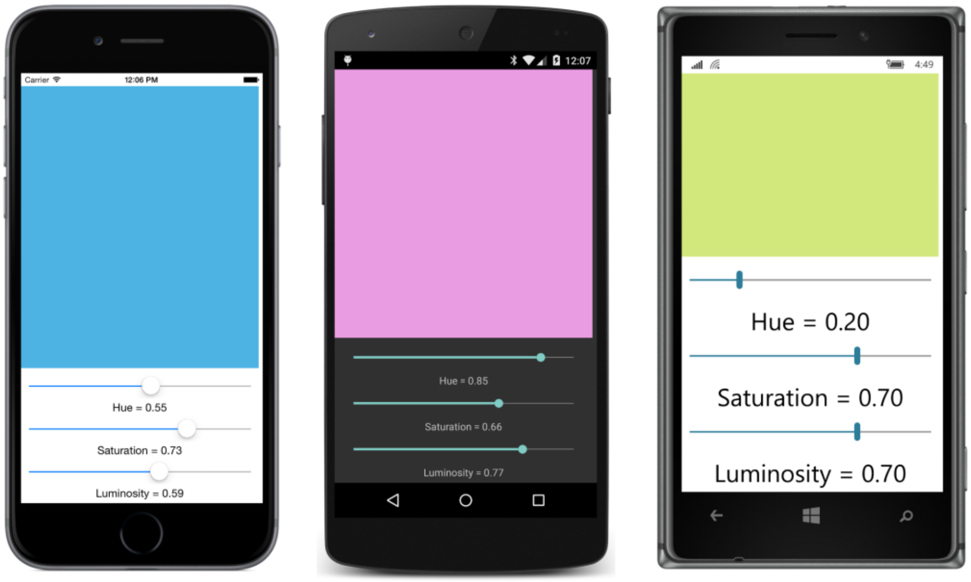
If you instead want to implement a display of hexadecimal values of `Red`, `Green`, and `Blue`, you can use the `DoubleToIntConverter` class demonstrated in connection with the **GridRgbSliders** program in the previous chapter.

The **HslSliders** program implements the same technique for switching between portrait and landscape modes as that **GridRgbSliders** program. The code-behind file handles the mechanics of this switch:

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

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

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

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

This code-behind file isn't quite as pretty as a file that merely calls `InitializeComponent`, but even in the context of MVVM, switching between portrait and landscape modes is a legitimate use of the code-behind file because it is solely devoted to the user interface rather than underlying business logic.

Here's the **HslSliders** program in action:



Streamlining the ViewModel

A typical implementation of `INotifyPropertyChanged` has a private backing field for every public property defined by the class, for example:

```
double number;
```

It also has an `OnPropertyChanged` method responsible for firing the `PropertyChanged` event:

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

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

A typical property definition looks like this:

```
public double Number
{
    set
    {
        if (number != value)
        {
            number = value;
            OnPropertyChanged("Number");

            // Do something with the new value.
        }
    }
    get
    {
        return number;
    }
}
```

A potential problem involves the text string you pass to the `OnPropertyChanged` method. If you misspell it, you won't get any type of error message, and yet bindings involving that property won't work. Also, the backing field appears three times within this single property. If you had several similar properties and defined them through copy-and-paste operations, it's possible to omit the renaming of one of the three appearances of the backing field, and that bug might be very difficult to track down.

You can solve the first problem with a feature introduced in C# 5.0. The `CallerMemberNameAttribute` class allows you to replace an optional method argument with the name of the calling method or property.

You can make use of this feature by redefining the `OnPropertyChanged` method. Make the argument optional by assigning `null` to it and preceding it with the `CallerMemberName` attribute in square brackets. You'll also need a `using` directive for `System.Runtime.CompilerServices`:

```
protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
    PropertyChangedEventHandler handler = PropertyChanged;

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

Now the `Number` property can call the `OnPropertyChanged` method without the argument that indicates the property name. That argument will be automatically set to the property name "Number" because that's where the call to `OnPropertyChanged` is originating:

```
public double Number
{
    set
    {
        if (number != value)
        {
            number = value;
            OnPropertyChanged();

            // Do something with the new value.
        }
    }
    get
    {
        return number;
    }
}
```

This approach avoids a misspelled text property name and also allows property names to be changed during program development without worrying about also changing the text strings. Indeed, one of the primary reasons that the CallerMemberName attribute was invented was to simplify classes that implement INotifyPropertyChanged.

However, this works only when OnPropertyChanged is called from the property whose value is changing. In the earlier ColorViewModel, explicit property names would still be required in all but one of the calls to OnPropertyChanged.

It's possible to go even further to simplify the set accessor logic: You'll need to define a generic method, probably named SetProperty or something similar. This SetProperty method is also defined with the CallerMemberName attribute:

```
bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
{
    if (Object.Equals(storage, value))
        return false;

    storage = value;
    OnPropertyChanged(propertyName);
    return true;
}

protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

The first argument to SetProperty is a reference to the backing field, and the second argument is the value being set to the property. SetProperty automates the checking and setting of the backing field. Notice that it explicitly includes the propertyName argument when calling OnPropertyChanged. (Otherwise the propertyName argument would become the string "SetProperty"!) The method returns true if the property was changed. You can use this return value to perform additional processing with the new value.

Now the Number property looks like this:

```
public double Number
{
    set
    {
        if (SetProperty(ref number, value))
        {
            // Do something with the new value.
        }
    }
    get
    {
        return number;
    }
}
```

Although SetProperty is a generic method, the C# compiler can deduce the type from the arguments. If you don't need to do anything with the new value in the property set accessor, you can even reduce the two accessors to single lines without obscuring the operations:

```
public double Number
{
    set { SetProperty(ref number, value); }
    get { return number; }
}
```

You might like this streamlining so much that you'll want to put the SetProperty and OnPropertyChanged methods in their own class and derive from that class when creating your own ViewModels. Such a class, called ViewModelBase, is already in the **Xamarin.FormsBook.Toolkit** library:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace Xamarin.FormsBook.Toolkit
{
    public class ViewModelBase : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected bool SetProperty<T>(ref T storage, T value,
                                      [CallerMemberName] string propertyName = null)
        {
            if (Object.Equals(storage, value))
                return false;

            storage = value;
            OnPropertyChanged(propertyName);
            return true;
        }

        protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

This class is used in the two remaining examples in this chapter.

The Command interface

Data bindings are very powerful. Data bindings connect properties of visual elements in the View with properties of data in the ViewModel, and allow the direct manipulation of data items through the user interface.

But not everything is a property. Sometimes ViewModels expose public *methods* that must be called from the View based on a user's interaction with a visual element. Without MVVM, you'd probably call such a method from a `Clicked` event handler of a `Button` or a `Tapped` event handler of a `TapGestureRecognizer`. When considering these needs, the whole concept of data bindings and MVVM might start to seem hopelessly flawed. How can the code-behind file of a page class be stripped down to an `InitializeComponent` call if it must still make method calls from the View to the ViewModel?

Don't give up on MVVM so quickly! Xamarin.Forms supports a feature that allows data bindings to make method calls in the ViewModel directly from `Button` and `TapGestureRecognizer` and a few other elements. This is a protocol called the *command interface* or the *commanding interface*.

The command interface is supported by eight classes:

- `Button`
- `MenuItem` (covered in Chapter 19, "Collection views"), and hence also `ToolbarItem`
- `SearchBar`
- `TextCell`, and hence also `ImageCell` (also to be covered in Chapter 19)
- `ListView` (also to be covered in Chapter 19)
- `TapGestureRecognizer`

It is also possible to implement commanding in your own custom classes.

The command interface is likely to be a little confusing at first. Let's focus on `Button`.

`Button` defines two ways for code to be notified when the element is clicked. The first is the `Clicked` event. But you can also use the button's command interface as an alternative to (or in addition to) the `Clicked` event. This interface consists of two public properties that `Button` defines:

- `Command` of type `System.Windows.Input.ICommand`.
- `CommandParameter` of type `Object`.

To support commanding, a ViewModel must define a public property of type `ICommand` that is then connected to the `Command` property of the `Button` through a normal data binding.

Like INotifyPropertyChanged, the ICommand interface is not a part of Xamarin.Forms. It's defined in the System.Windows.Input namespace and implemented in the **System.ObjectModel** assembly, which is one of the .NET assemblies linked to a Xamarin.Forms application. ICommand is the *only* type in the System.Windows.Input namespace that Xamarin.Forms supports. Indeed it's the only type in *any* System.Windows namespace supported by Xamarin.Forms.

Is it a coincidence that INotifyPropertyChanged and ICommand are both defined in .NET assemblies rather than Xamarin.Forms? No. These interfaces are often used in ViewModels, and some developers might already have ViewModels developed for one or more of Microsoft's XAML-based environments. It's easiest for developers to incorporate these existing ViewModels into Xamarin.Forms if INotifyPropertyChanged and ICommand are defined in standard .NET namespaces and assemblies rather than in Xamarin.Forms.

The ICommand interface defines two methods and one event:

```
public interface ICommand
{
    void Execute(object arg);

    bool CanExecute(object arg);

    event EventHandler CanExecuteChanged;
}
```

To implement commanding, the ViewModel defines one or more properties of type ICommand, meaning that the property is a type that implements these two methods and the event. A property in the ViewModel that implements ICommand can then be bound to the Command property of a Button. When the Button is clicked, the Button fires its normal Clicked event as usual, but it also calls the Execute method of the object bound to its Command property. The argument to the Execute method is the object set to the CommandParameter property of the Button.

That's the basic technique. However, it could be that certain conditions in the ViewModel prohibit a Button click at the current time. In that case, the Button should be disabled. This is the purpose of the CanExecute method and the CanExecuteChanged event in ICommand. The Button calls CanExecute when its Command property is first set. If CanExecute returns false, the Button disables itself and doesn't generate Execute calls. The Button also installs a handler for the CanExecuteChanged event. Thereafter, whenever the ViewModel fires the CanExecuteChanged event, the button calls CanExecute again to determine whether the button should be enabled.

A ViewModel that supports the command interface defines one or more properties of type ICommand and internally sets this property to a class that implements the ICommand interface. What is this class, and how does it work?

If you were implementing the commanding protocol in one of Microsoft's XAML-based environments, you would be writing your own class that implements ICommand, or perhaps using one that you found on the web, or one that was included with some MVVM tools. Sometimes such classes are named CommandDelegate or something similar.

You can use that same class in the ViewModels of your Xamarin.Forms applications. However, for your convenience, Xamarin.Forms includes two classes that implement `ICommand` that you can use instead. These two classes are named simply `Command` and `Command<T>`, where `T` is the type of the arguments to `Execute` and `CanExecute`.

If you are indeed sharing a ViewModel between Microsoft environments and Xamarin.Forms, you can't use the `Command` classes defined by Xamarin.Forms. However, you'll be using something similar to these `Command` classes, so the following discussion will certainly be applicable regardless.

The `Command` class includes the two methods and event of the `ICommand` interface and also defines a `ChangeCanExecute` method. This method causes the `Command` object to fire the `CanExecute-Changed` event, and that facility turns out to be very handy.

Within the ViewModel, you'll probably create an object of type `Command` or `Command<T>` for every public property in the ViewModel of type `ICommand`. The `Command` or `Command<T>` constructor requires a callback method in the form of an `Action` object that is called when the `Button` calls the `Execute` method of the `ICommand` interface. The `CanExecute` method is optional but takes the form of a `Func` object that returns `bool`.

In many cases, the properties of type `ICommand` are set in the ViewModel's constructor and do not change thereafter. For that reason, these `ICommand` properties do not generally need to fire `PropertyChanged` events.

Simple method executions

Let's look at a simple example. A program called **PowersOfThree** lets you use two buttons to explore various powers of 3. One button increases the exponent and the other button decreases the exponent.

The `PowersViewModel` class derives from the `ViewModelBase` class in the **Xamarin.Forms-Book.Toolkit** library, but the ViewModel itself is in the **PowersOfThree** application project. It is not restricted to powers of 3, but the constructor requires an argument that the class uses as a base value for the power calculation, and which it exposes as the `BaseValue` property. Because this property has a private `set` accessor and doesn't change after the constructor concludes, the property does not fire a `PropertyChanged` event.

Two other properties, named `Exponent` and `Power`, do fire `PropertyChanged` events, but both properties also have private `set` accessors. The `Exponent` property is increased and decreased only from external button clicks.

To implement the response to `Button` taps, the `PowersViewModel` class defines two properties of type `ICommand`, named `IncreaseExponentCommand` and `DecreaseExponentCommand`. Again, both properties have private `set` accessors. As you can see, the constructor sets these two properties by instantiating `Command` objects that reference little private methods immediately following the constructor. These two little methods are called when the `Execute` method of `Command` is called. The View-Model uses the `Command` class rather than `Command<T>` because the program doesn't make use of any argument to the `Execute` methods:

```
class PowersViewModel : ViewModelBase
{
    double exponent, power;

    public PowersViewModel(double baseValue)
    {
        // Initialize properties.
        BaseValue = baseValue;
        Exponent = 0;

        // Initialize ICommand properties.
        IncreaseExponentCommand = new Command(ExecuteIncreaseExponent);
        DecreaseExponentCommand = new Command(ExecuteDecreaseExponent);
    }

    void ExecuteIncreaseExponent()
    {
        Exponent += 1;
    }

    void ExecuteDecreaseExponent()
    {
        Exponent -= 1;
    }

    public double BaseValue { private set; get; }

    public double Exponent
    {
        private set
        {
            if (SetProperty(ref exponent, value))
            {
                Power = Math.Pow(BaseValue, exponent);
            }
        }
        get
        {
            return exponent;
        }
    }

    public double Power
    {
        private set { SetProperty(ref power, value); }
        get { return power; }
    }

    public ICommand IncreaseExponentCommand { private set; get; }

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

The ExecuteIncreaseExponent and ExecuteDecreaseExponent methods both make a change to the Exponent property (which fires a PropertyChanged event), and the Exponent property recalculates the Power property, which also fires a PropertyChanged event.

Very often a ViewModel will instantiate its Command objects by passing lambda functions to the Command constructor. This approach allows these methods to be defined right in the ViewModel constructor, like so:

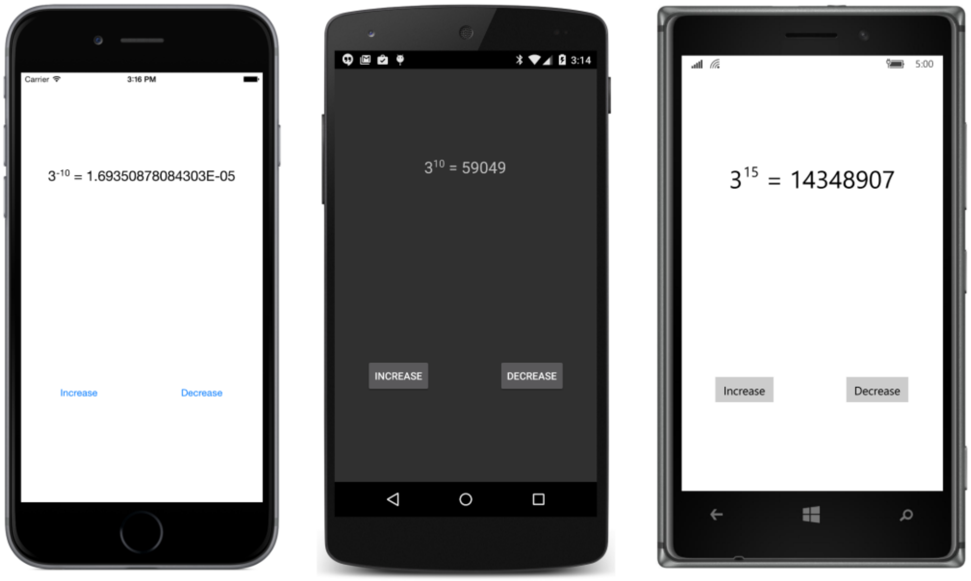
```
IncreaseExponentCommand = new Command(() =>
    {
        Exponent += 1;
    });

DecreaseExponentCommand = new Command(() =>
    {
        Exponent -= 1;
    });
```

The PowersOfThreePage XAML file binds the Text properties of three Label elements to the BaseValue, Exponent, and Power properties of the PowersViewModel class, and binds the Command properties of the two Button elements to the IncreaseExponentCommand and DecreaseExponentCommand properties of the ViewModel.

Notice how an argument of 3 is passed to the constructor of PowersViewModel as it is instantiated in the Resources dictionary. Passing arguments to ViewModel constructors is the primary reason for the existence of the x:Arguments tag:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:PowersOfThree"
             x:Class="PowersOfThree.PowersOfThreePage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <local:PowersViewModel x:Key="viewModel">
                <x:Arguments>
                    <x:Double>3</x:Double>
                </x:Arguments>
            </local:PowersViewModel>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout BindingContext="{StaticResource viewModel}">
        <StackLayout Orientation="Horizontal"
                     Spacing="0"
                     HorizontalOptions="Center"
                     VerticalOptions="CenterAndExpand">
            <Label FontSize="Large"
                   Text="{Binding BaseValue, StringFormat='{0}'}" />

            <Label FontSize="Small"
                   Text="{Binding Exponent, StringFormat='{0}'}" />
            <Label FontSize="Large"
                   Text="{Binding Power, StringFormat=' = {0}'}" />
        </StackLayout>

        <StackLayout Orientation="Horizontal"
                     VerticalOptions="CenterAndExpand">

            <Button Text="Increase"
                    Command="{Binding IncreaseExponentCommand}"
                    HorizontalOptions="CenterAndExpand" />

            <Button Text="Decrease"
                    Command="{Binding DecreaseExponentCommand}"
                    HorizontalOptions="CenterAndExpand" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

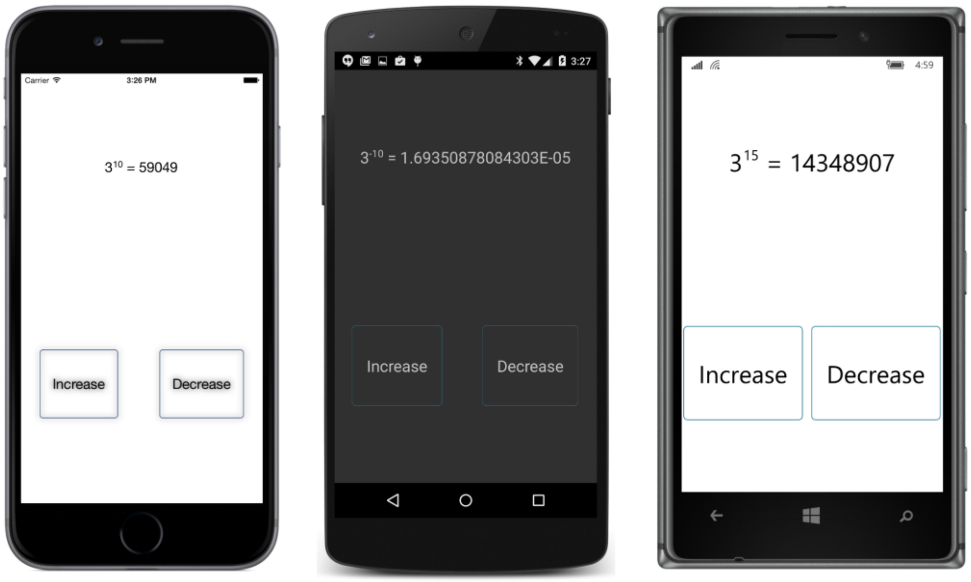
Here's what it looks like after several presses of one button or the other:



Once again, the wisdom of separating the user interface from the underlying business logic is revealed when the time comes to change the View. For example, suppose you want to replace the buttons with an element with a `TapGestureRecognizer`. Fortunately, `TapGestureRecognizer` has a `Command` property:

```
<StackLayout Orientation="Horizontal"
                VerticalOptions="CenterAndExpand">

    <Frame OutlineColor="Accent"
           BackgroundColor="Transparent"
           Padding="20, 40"
           HorizontalOptions="CenterAndExpand">
        <Frame.GestureRecognizers>
            <TapGestureRecognizer Command="{Binding IncreaseExponentCommand}" />
        </Frame.GestureRecognizers>

        <Label Text="Increase"
               FontSize="Large" />
    </Frame>

    <Frame OutlineColor="Accent"
           BackgroundColor="Transparent"
           Padding="20, 40"
           HorizontalOptions="CenterAndExpand">
        <Frame.GestureRecognizers>
            <TapGestureRecognizer Command="{Binding DecreaseExponentCommand}" />
        </Frame.GestureRecognizers>

        <Label Text="Decrease"
               FontSize="Large" />
    </Frame>
</StackLayout>
```

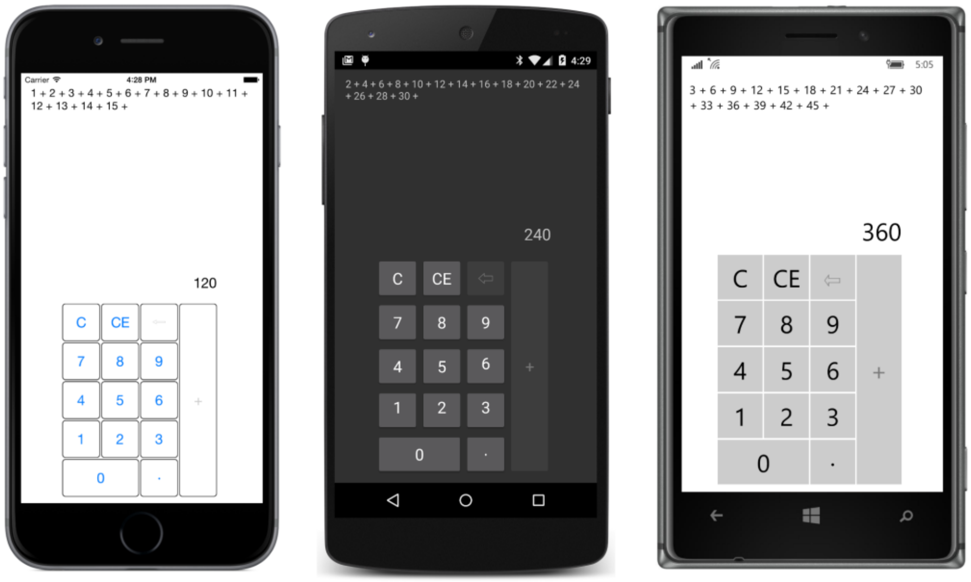
Without touching the ViewModel or even renaming an event handler so that it applies to a tap rather than a button, the program works the same, but with a different look:



A calculator, almost

Now it's time to make a more sophisticated ViewModel with `ICommand` objects that have both `Execute` and `CanExecute` methods. The next program is almost like a calculator except that it only adds a series of numbers together. The ViewModel is named `AdderViewModel`, and the program is called **AddingMachine**.

Let's look at the screenshots first:



At the top of the page you can see a history of the series of numbers that have already been entered and added. This is a `Label` in a `ScrollView`, so it can get rather long.

The sum of those numbers is displayed in the `Entry` view above the keypad. Normally, that `Entry` view contains the number that you're typing in, but after you hit the big plus sign at the right of the keypad, the `Entry` displays the accumulated sum and the plus sign button becomes disabled. You need to begin typing another number for the accumulated sum to disappear and for the button with the plus sign to be enabled. Similarly, the backspace button is enabled as soon as you begin to type.

These are not the only keys that can be disabled. The decimal point is disabled when the number you're typing already has a decimal point, and all the number keys become disabled when the number contains 16 characters. This is to avoid the number in the `Entry` from becoming too long to display.

The disabling of these buttons is the result of implementing the `CanExecute` method in the `ICommand` interface.

The `AdderViewModel` class is in the **Xamarin.FormsBook.Toolkit** library and derives from `ViewModelBase`. Here is the part of the class with all the public properties and their backing fields:

```
public class AdderViewModel : ViewModelBase
{
    string currentEntry = "0";
    string historyString = "";
    …
    public string CurrentEntry
    {
        private set { SetProperty(ref currentEntry, value); }
        get { return currentEntry; }
    }

    public string HistoryString
    {
        private set { SetProperty(ref historyString, value); }
        get { return historyString; }
    }

    public ICommand ClearCommand { private set; get; }

    public ICommand ClearEntryCommand { private set; get; }

    public ICommand BackspaceCommand { private set; get; }

    public ICommand NumericCommand { private set; get; }

    public ICommand DecimalPointCommand { private set; get; }

    public ICommand AddCommand { private set; get; }
    …
}
```

All the properties have private `set` accessors. The two properties of type `string` are only set internally based on the key taps, and the properties of type `ICommand` are set in the `AdderViewModel` constructor (which you'll see shortly).

These eight public properties are the only part of `AdderViewModel` that the XAML file in the **AddingMachine** project needs to know about. Here is that XAML file. It contains a two-row and two-column main `Grid` for switching between portrait and landscape mode, and a `Label`, `Entry`, and 15 `Button` elements, all of which are bound to one of the eight public properties of the `AdderViewModel`. Notice that the `Command` properties of all 10 digit buttons are bound to the `NumericCommand` property and that the buttons are differentiated by the `CommandParameter` property. The setting of this `CommandParameter` property is passed as an argument to the `Execute` and `CanExecute` methods:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="AddingMachine.AddingMachinePage"
             SizeChanged="OnPageSizeChanged">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                    iOS="10, 20, 10, 10"
                    Android="10"
                    WinPhone="10" />
    </ContentPage.Padding>
    <Grid x:Name="mainGrid">
        <!-- Initialized for Portrait mode. -->
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

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

        <!-- History display. -->
        <ScrollView Grid.Row="0" Grid.Column="0"
                    Padding="5, 0">
            <Label Text="{Binding HistoryString}" />
        </ScrollView>

        <!-- Keypad. -->
        <Grid x:Name="keypadGrid"
              Grid.Row="1" Grid.Column="0"
              RowSpacing="2"
              ColumnSpacing="2"
              WidthRequest="240"
              HeightRequest="360"
              VerticalOptions="Center"
              HorizontalOptions="Center">
            <Grid.Resources>
                <ResourceDictionary>
                    <Style TargetType="Button">
                        <Setter Property="FontSize" Value="Large" />
                        <Setter Property="BorderWidth" Value="1" />
                    </Style>
                </ResourceDictionary>
            </Grid.Resources>

            <Label Text="{Binding CurrentEntry}"
                   Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4"
                   FontSize="Large"
                   LineBreakMode="HeadTruncation"
                   VerticalOptions="Center"
                   HorizontalTextAlignment="End" />

            <Button Text="C"
                    Grid.Row="1" Grid.Column="0"
                    Command="{Binding ClearCommand}" />

            <Button Text="CE"
                    Grid.Row="1" Grid.Column="1"
                    Command="{Binding ClearEntryCommand}" />

            <Button Text="&#x21E6;"
                    Grid.Row="1" Grid.Column="2"
                    Command="{Binding BackspaceCommand}" />
            <Button Text="+"
                    Grid.Row="1" Grid.Column="3" Grid.RowSpan="5"
                    Command="{Binding AddCommand}" />

            <Button Text="7"
                    Grid.Row="2" Grid.Column="0"
                    Command="{Binding NumericCommand}"
                    CommandParameter="7" />

            <Button Text="8"
                    Grid.Row="2" Grid.Column="1"
                    Command="{Binding NumericCommand}"
                    CommandParameter="8" />

            <Button Text="9"
                    Grid.Row="2" Grid.Column="2"
                    Command="{Binding NumericCommand}"
                    CommandParameter="9" />

            <Button Text="4"
                    Grid.Row="3" Grid.Column="0"
                    Command="{Binding NumericCommand}"
                    CommandParameter="4" />

            <Button Text="5"
                    Grid.Row="3" Grid.Column="1"
                    Command="{Binding NumericCommand}"
                    CommandParameter="5" />

            <Button Text="6"
                    Grid.Row="3" Grid.Column="2"
                    Command="{Binding NumericCommand}"
                    CommandParameter="6" />

            <Button Text="1"
                    Grid.Row="4" Grid.Column="0"
                    Command="{Binding NumericCommand}"
                    CommandParameter="1" />

            <Button Text="2"
                    Grid.Row="4" Grid.Column="1"
                    Command="{Binding NumericCommand}"
                    CommandParameter="2" />

            <Button Text="3"
                    Grid.Row="4" Grid.Column="2"
                    Command="{Binding NumericCommand}"
                    CommandParameter="3" />

            <Button Text="0"
                    Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2"
                    Command="{Binding NumericCommand}"
                    CommandParameter="0" />
```

```
            <Button Text="&#x00B7;"
                    Grid.Row="5" Grid.Column="2"
                    Command="{Binding DecimalPointCommand}" />
        </Grid>
    </Grid>
</ContentPage>
```

What you won't find in the XAML file is a reference to AdderViewModel. For reasons you'll see shortly, AdderViewModel is instantiated in code.

The core of the adding-machine logic is in the Execute and CanExecute methods of the six ICommand properties. These properties are all initialized in the AdderViewModel constructor shown below, and the Execute and CanExecute methods are all lambda functions.

When only one lambda function appears in the Command constructor, that's the Execute method (as the parameter name indicates), and the Button is always enabled. This is the case for ClearCommand and ClearEntryCommand.

All the other Command constructors have two lambda functions. The first is the Execute method, and the second is the CanExecute method. The CanExecute method returns true if the Button should be enabled and false otherwise.

All the ICommand properties are set with the nongeneric form of the Command class except for NumericCommand, which requires an argument to the Execute and CanExecute methods to identify which key has been tapped:

```
public class AdderViewModel : ViewModelBase
{
    …
    bool isSumDisplayed = false;
    double accumulatedSum = 0;

    public AdderViewModel()
    {
        ClearCommand = new Command(
            execute: () =>
            {
                HistoryString = "";
                accumulatedSum = 0;
                CurrentEntry = "0";
                isSumDisplayed = false;
                RefreshCanExecutes();
            });

        ClearEntryCommand = new Command(
            execute: () =>
            {
                CurrentEntry = "0";
                isSumDisplayed = false;
                RefreshCanExecutes();
            });
        BackspaceCommand = new Command(
            execute: () =>
            {
                CurrentEntry = CurrentEntry.Substring(0, CurrentEntry.Length - 1);

                if (CurrentEntry.Length == 0)
                {
                    CurrentEntry = "0";
                }

                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return !isSumDisplayed && (CurrentEntry.Length > 1 || CurrentEntry[0] != '0');
            });

        NumericCommand = new Command<string>(
            execute: (string parameter) =>
            {
                if (isSumDisplayed || CurrentEntry == "0")
                    CurrentEntry = parameter;
                else
                    CurrentEntry += parameter;

                isSumDisplayed = false;
                RefreshCanExecutes();
            },
            canExecute: (string parameter) =>
            {
                return isSumDisplayed || CurrentEntry.Length < 16;
            });

        DecimalPointCommand = new Command(
            execute: () =>
            {
                if (isSumDisplayed)
                    CurrentEntry = "0.";
                else
                    CurrentEntry += ".";

                isSumDisplayed = false;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return isSumDisplayed || !CurrentEntry.Contains(".");
            });

        AddCommand = new Command(
            execute: () =>
            {
                double value = Double.Parse(CurrentEntry);
                HistoryString += value.ToString() + " + ";
                accumulatedSum += value;
                CurrentEntry = accumulatedSum.ToString();
                isSumDisplayed = true;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return !isSumDisplayed;
            });
    }

    void RefreshCanExecutes()
    {
        ((Command)BackspaceCommand).ChangeCanExecute();
        ((Command)NumericCommand).ChangeCanExecute();
        ((Command)DecimalPointCommand).ChangeCanExecute();
        ((Command)AddCommand).ChangeCanExecute();
    }
    …
}
```

All the `Execute` methods conclude by calling a method named `RefreshCanExecute` following the constructor. This method calls the `ChangeCanExecute` method of each of the four `Command` objects that implement `CanExecute` methods. That method call causes the `Command` object to fire a `ChangeCanExecute` event. Each `Button` responds to that event by making another call to the `CanExecute` method to determine if the `Button` should be enabled or not.

It is not necessary for every `Execute` method to conclude with a call to all four `ChangeCanExecute` methods. For example, the `ChangeCanExecute` method for the `DecimalPointCommand` need not be called when the `Execute` method for `NumericCommand` executes. However, it turned out to be easier—both in terms of logic and code consolidation—to simply call them all after every key tap.

You might be more comfortable implementing these `Execute` and `CanExecute` methods as regular methods rather than lambda functions. Or you might be more comfortable having just one `Command` object that handles all the keys. Each key could have an identifying `CommandParameter` string and you could distinguish between them with a `switch` and `case` statement.

There are lots of ways to implement the commanding logic, but it should be clear that the use of commanding tends to structure the code in a flexible and ideal way.

Once the adding logic is in place, why not add a couple of more buttons for subtraction, multiplication, and division?

Well, it's not quite so easy to enhance the logic to accept multiple operations rather than just one operation. If the program supports multiple operations, then when the user types one of the operation keys, that operation needs to be saved to await the next number. Only after the next number is completed (signaled by the press of another operation key or the equals key) is that saved operation applied.

An easier approach would be to write a Reverse Polish Notation (RPN) calculator, where the operation *follows* the entry of the second number. The simplicity of RPN logic is one big reason why RPN calculators appeal to programmers so much!

ViewModels and the application lifecycle

In a real calculator program on a mobile device, one important feature involves saving the entire state of the calculator when the program is terminated, and restoring it when the program starts up again.

And once again, the concept of the ViewModel seems to break down.

Sure, it's possible to write some application code that accesses the public properties of the ViewModel and saves them, but the state of the calculator depends on private fields as well. The `isSumDisplayed` and `accumulatedSum` fields of `AdderViewModel` are essential for restoring the calculator's state.

It's obvious that code external to the `AdderViewModel` can't save and restore the `AdderViewModel` state without the ViewModel exposing more public properties. There's only one class that knows what's necessary to represent the entire internal state of a ViewModel, and that's the ViewModel itself.

The solution is for the ViewModel to define public methods that save and restore its internal state. But because a ViewModel should strive to be platform independent, these methods shouldn't use anything specific to a particular platform. For example, they shouldn't access the Xamarin.Forms `Application` object and then add items to (or retrieve items from) the `Properties` dictionary of that `Application` object. That is much too specific to Xamarin.Forms.

However, working with a generic `IDictionary` object in methods named `SaveState` and `RestoreState` is possible in *any* .NET environment, and that's how `AdderViewModel` implements these methods:

```
public class AdderViewModel : ViewModelBase
{
    …
    public void SaveState(IDictionary<string, object> dictionary)
    {
        dictionary["CurrentEntry"] = CurrentEntry;
        dictionary["HistoryString"] = HistoryString;
        dictionary["isSumDisplayed"] = isSumDisplayed;
        dictionary["accumulatedSum"] = accumulatedSum;
    }

    public void RestoreState(IDictionary<string, object> dictionary)
    {
        CurrentEntry = GetDictionaryEntry(dictionary, "CurrentEntry", "0");
        HistoryString = GetDictionaryEntry(dictionary, "HistoryString", "");
        isSumDisplayed = GetDictionaryEntry(dictionary, "isSumDisplayed", false);
        accumulatedSum = GetDictionaryEntry(dictionary, "accumulatedSum", 0.0);

        RefreshCanExecutes();
    }

    public T GetDictionaryEntry<T>(IDictionary<string, object> dictionary,
                                   string key, T defaultValue)
    {
        if (dictionary.ContainsKey(key))
            return (T)dictionary[key];

        return defaultValue;
    }
}
```

The code in **AddingMachine** involved in saving and restoring this state is mostly implemented in the `App` class. The `App` class instantiates the `AdderViewModel` and calls `RestoreState` using the `Properties` dictionary of the current `Application` class. That `AdderViewModel` is then passed as an argument to the `AddingMachinePage` constructor:

```
public class App : Application
{
    AdderViewModel adderViewModel;

    public App()
    {
        // Instantiate and initialize ViewModel for page.
        adderViewModel = new AdderViewModel();
        adderViewModel.RestoreState(Current.Properties);
        MainPage = new AddingMachinePage(adderViewModel);
    }

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

    protected override void OnSleep()
    {
        // Handle when your app sleeps.
        adderViewModel.SaveState(Current.Properties);
    }

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

The `App` class is also responsible for calling `SaveState` on `AdderViewModel` during processing of the `OnSleep` method.

The `AddingMachinePage` constructor merely needs to set the instance of `AdderViewModel` to the page's BindingContext property. The code-behind file also manages the switch between portrait and landscape layouts:

```
public partial class AddingMachinePage : ContentPage
{
    public AddingMachinePage(AdderViewModel viewModel)
    {
        InitializeComponent();

        // Set ViewModel as BindingContext.
        BindingContext = viewModel;
    }

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

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

            Grid.SetRow(keypadGrid, 0);
            Grid.SetColumn(keypadGrid, 1);
        }
    }
}
```

The **AddingMachine** program demonstrates one way to handle the ViewModel, but it's not the only way. Alternatively, it's possible for App to instantiate the AdderViewModel but define a property of type AdderViewModel that the constructor of AddingMachinePage can access.

Or, if you want the page to have full control over the ViewModel, you can do that as well. Adding-MachinePage can define its own OnSleep method that is called from the OnSleep method in the App class, and the page class can also handle the instantiation of AdderViewModel and the calling of the RestoreState and SaveState methods. However, this approach might become somewhat clumsy for multipage applications.

In a multipage application, you might have separate ViewModels for each page, perhaps deriving from a ViewModel with properties applicable to the entire application. In such a case, you'll want to avoid properties with the same name using the same dictionary keys for saving each ViewModel's state. You can use more extensive dictionary keys that include the class name, for example, "Adder-ViewModel.CurrentEntry".

Although the power and advantages of data binding and ViewModels should be apparent by now, these features really blossom when used with the Xamarin.Forms `ListView`. That's up in the next chapter.