Chapter Four

Working with Data: Binding and Templates

Most business applications deal extensively with data. As a developer, you have a constant need to expose data in a user interface.

You can do that in a brute-force fashion by writing a lot of code to move data between user interface elements and data containers. However, such code is repetitious, bug-prone, tedious to write, and difficult to maintain. Everybody hates it.

The alternative is a capability called *data binding.* Several UI technologies have implemented data binding in the past, with varying degrees of success.

XAML has the best implementation of data binding I've ever seen.[1] It's a key reason for XAML's flexibility and power. If you are going to learn to think in XAML, you'll need to know data binding in depth.

One of the main places data binding shows its power is through a XAML capability called *data templates*. They combine layout and composition with data binding to allow you to build data centric applications that would have been much harder, or even impossible, in other UI technologies.

In this chapter, we'll first look at some more data binding capabilities, and then explore how data binding is used in data templates.

A simple data object to use in the chapter

Up to this point, we have just assumed a Book object was available. Before going further, we need to pin that down.

When I refer to a Book object anywhere in this chapter, I'll assume that we are using an instance of a simple data class that looks like in code like the one below. We'll assume the class is defined in the {ProjectName}.Shared project. In the code downloads, that means it will be in Chapter4.Shared.

```
class Book
 {
     public string Title { get; set; }
     public string Author { get; set; }
     public string Genre { get; set; }
     public DateTime DatePublished { get; set; }
     public bool InPrint { get; set; }
     public int QuantityInStock { get; set; }

     //Needed to declare a date in XAML
     //Yeah, it's a hack for the book. Deal with it.
```

[1] If you have heard of JavaScript frameworks such as Angular, you may be interested to know that much of the design of those frameworks was inspired by earlier XAML platforms, such as WPF and Silverlight.

```
        public string StringDatePublished
        {
            get { return DatePublished.ToShortDateString(); }
            set { DatePublished = DateTime.Parse(value); }
        }

    }
```

Naturally, this is oversimplified for many application scenarios. We might use much more complicated data models, or even use viewmodels that are crafted to implement the Model-View-ViewModel pattern (MVVM). But for now, a simple data model class will do.

Please don't cringe because we have a single property for Author instead of an Authors collection, as good data modeling practices would suggest. We'll add that later.

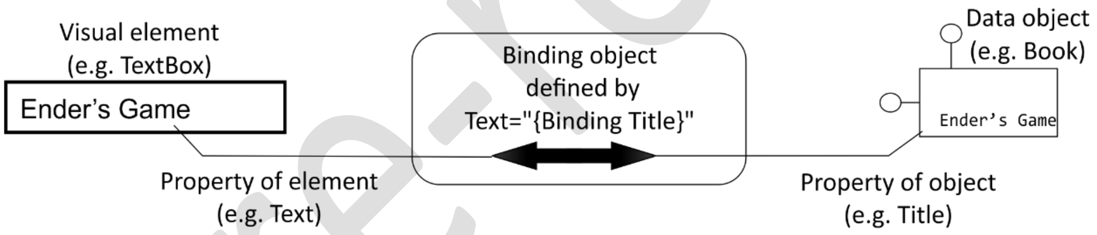
A data binding recap

We looked at some basic aspects of data binding in Chapter 1. For example, in Chapter 1, we had a line of XAML to take the title of a book from a Book object, and put it into the Text property of a TextBox.

```
<TextBox Text="{Binding Title}" Name="TitleTextBox" />
```

All the data binding examples in Chapter 1 were simple. They named a property from which to get a value (such as the Title property of a Book), and placed the value of that property in a target property of a visual element (such as the Text property of a TextBox).

The binding was expressed in XAML as a *binding expression*, and the result in the compiled code was a *binding object* that moved data between properties. Recall the diagram we saw earlier:



The binding expression supplies the property settings to create the binding object. The name of the target property, for example, becomes the setting for the Binding object's Path property. In fact, the binding

```
<TextBox Text="{Binding Title}" Name="TitleTextBox" />
```

is actually a short form of this binding:

```
<TextBox Text="{Binding Path=Title}" Name="TitleTextBox" />
```

Bindings like this assume the data object to be used is found via the DataContext property. Recall that it can be set for a container, and then be used by all elements in that container.

ElementName bindings

To finish our review of earlier coverage, in Chapter 2, we looked at another property of the Binding object, ElementName. If set, ElementName points to another element on the XAML page as the source data object instead of using the default DataContext. The "fuel tank" example at the end of that chapter included a binding to get the Height of a Grid from the Value of a Slider using ElementName. That part of the example looked like this:

```
<Grid Margin="5,20,5,5" VerticalAlignment="Bottom"
      Height="{Binding Value, ElementName=FuelLevelSlider}">
    <Rectangle Height="100" RadiusX="50"
               RadiusY="50" Fill="GreenYellow"
               VerticalAlignment="Bottom"/>
</Grid>
```

ElementName allows data binding to be used to connect pieces of an interface, as we discussed in Chapter 2. It allows all kinds of responsiveness, making parts of the interface change automatically based on changes in other parts.[2] We'll see another example of this later in the chapter when we use it to for navigating a list of data items.

More capabilities of data binding

In this chapter, we'll see additional capabilities of data binding, including:

- Controlling the direction data moves in a binding
- Binding to property paths to drill in on the data needed
- Converting data during binding – changing it from its original form into a form that is more helpful in the user interface

All of these are done by changing the binding expression that defines the binding. Most of them set properties of the new Binding object that is created from the expression. The two most important additional properties of the Binding object we'll be discussing are:

Mode – determine the direction the data moves – OneWay, which is from the source data object to the target only, or TwoWay, which allows the data to also move from the target element back to the data object

Converter – the object holding the logic to change the data from its original form into a form more convenient for the user interface

Let's start with the easy, straightforward one – the Mode property for controlling the direction data moves during binding.

[2] This point is often missed by developers when they start using XAML. They immediately start using data bindings with DataContext, because that's useful and essential to doing typical data editing views. They think of that kind of data binding as the only kind, and are less likely to incorporate ElementName bindings into their views. That's a shame.

Mode property

If you don't specify in your binding how the data moves, then it only moves one way, from the source data object to the target visual element.[3] All the examples we've seen so far work are one-way bindings.

This works fine for lots of bindings. Modern applications get data from many places and don't depend so heavily on data entry. Read-only consumption of data is common for searching and decision support.

But there are times when a user needs to enter data and have it stored. Controls such as TextBox and ToggleSwitch allow a user to enter or change information, and a two-way binding can then transmit that information to a data object.

The property of bindings that control the direction is Mode. Remember the binding from the Book object's Title property to the text in a TextBox? To make that binding two-way, add the setting for Mode to the binding's markup expression:

```
<TextBox Text="{Binding Path=Title, Mode=TwoWay}" Name="TitleTextBox" />
```

With this binding, user-entered text in the TextBox will be transferred into the Book object's Title property as soon as the user leaves the TextBox.

That doesn't store the new information permanently, of course. It's your responsibility as a developer to take that changed Book data object and shoot it off to some permanent data store, such as a relational database. Most developers do that sort of thing all the time, and there are lots of ways to do it, so I don't talk about how to do it in this book.

Data types and property paths

The bindings you've seen so far get simple information, such as text, from the data object. But data objects can have properties with all types of data.

Besides text, most data records have some numeric fields. XAML bindings work fine for numeric data types. If the Book object has a QuantityInStock property that is an integer, then this simple binding will get and set that property value from a TextBox:

```
<TextBox Text="{Binding Path= QuantityInStock, Mode=TwoWay}"  />
```

Naturally, a two-way binding to a numeric property will have a problem if the user types text into the TextBox instead of a number. The binding will fail, and the message in your Visual Studio output window will look like this:

```
Error: Converter failed to convert value of type
'Windows.Foundation.IReference`1<String>' to type 'Int32'; BindingExpression:
Path='QuantityInStock' DataItem='Chapter4.Shared.Book'; target element is
'Windows.UI.Xaml.Controls.TextBox' (Name='null'); target property is 'Text' (type
'String').
```

You'll run into the same kinds of issues with other type mismatches, such as trying to put decimal data in an integer property. But as long as the types match up, numeric types bind quite nicely.

[3] This is a difference from WPF, where some bindings are two-way by default.

Boolean data is common in most apps, and the simple bindings you've seen so far work fine in ToggleSwitch and CheckBox controls. Here's a typical example, assuming the Book object has a Boolean property named InPrint:

```
<ToggleSwitch OnContent="In print" OffContent="Out of print"
              IsOn="{Binding Path=InPrint, Mode=TwoWay}" />
```

But not everything can be so simple. Suppose the Book object has a property for the date the book was published, and it's a DateTime type. A user reviewing the information probably doesn't care about the exact date the book was published, but might be interested in knowing just the year it was published.

That year is not available as a direct property of the Book object, but it is available as a property of the DateTime value for the publication date. The Path property of the binding supports this case with something called a *property path*. If the Book object's property for publication date was named DatePublished, then we could bind a TextBlock to the year of publication with this binding:

```
<TextBlock Margin="6"
 Text="{Binding Path=DatePublished.Year}" />
```

A property path can descend through "properties of properties" as far as you need. I don't think I've used one with more then three levels, though.

Putting all this new binding stuff in an example

Let's wrap up where we are with these bindings. Remember that example at the beginning of Chapter 2, which looked like a simple data entry form for books? It had hard coded data for the book. Let's enhance it with more fields for the book, and include appropriate bindings so that it actually works as a data entry form. Also, I'll add some comments to make it easier to scan. Here's the XAML we end up with:

```
<Grid Background="LightBlue">
    <Grid.DataContext>
        <local:Book Title="Double Star" Genre="Science Fiction"
                    Author="Robert Heinlein" InPrint="True"
                    QuantityInStock="5"
                    StringDatePublished="06/01/1956" />
    </Grid.DataContext>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!--We need an auto-sized row for each data field,
    plus one to hold buttons-->
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <!--This rectangle furnishes a background for the data fields-->
    <Rectangle Fill="LightGoldenrodYellow" Margin="2"
```

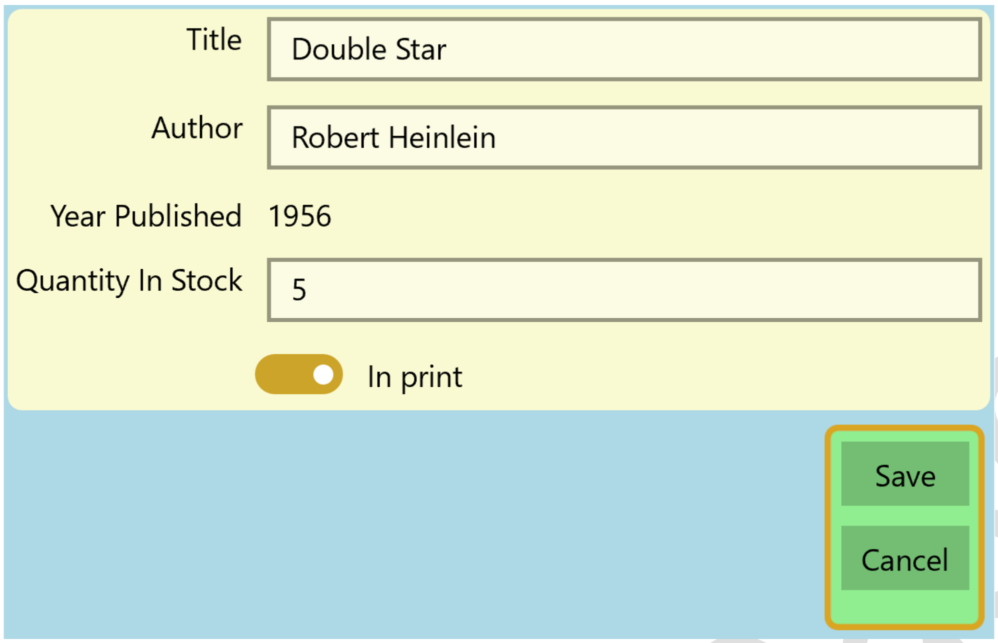
```
            Grid.RowSpan="5" Grid.ColumnSpan="2"
            RadiusX="7" RadiusY="7" />

        <!--TextBlocks hold the data field labels-->
        <TextBlock Text="Title" Margin="6"
            HorizontalAlignment="Right" />
        <TextBlock Text="Author" Margin="6" Grid.Row="1"
            HorizontalAlignment="Right" />
        <TextBlock Text="Year Published" Margin="6" Grid.Row="2"
            HorizontalAlignment="Right" />
        <TextBlock Text="Quantity In Stock" Margin="6" Grid.Row="3"
            HorizontalAlignment="Right" />

        <!--Various controls hold the data field values-->
        <TextBox Grid.Column="1" Margin="6"
         Text="{Binding Path=Title, Mode=TwoWay}" />
        <TextBox Grid.Column="1" Grid.Row="1" Margin="6"
         Text="{Binding Path=Author, Mode=TwoWay}" />
        <TextBlock Grid.Column="1" Grid.Row="2" Margin="6"
         Text="{Binding Path=DatePublished.Year}" />
        <TextBox Grid.Column="1" Grid.Row="3" Margin="6"
         Text="{Binding Path=QuantityInStock, Mode=TwoWay}" />
        <ToggleSwitch OnContent="In print" OffContent="Out of print"
                      Grid.Row="4" Grid.Column="1"
                      IsOn="{Binding Path=InPrint, Mode=TwoWay}" />

        <!--Border at the bottom holds a StackPanel of Buttons-->
        <Border BorderThickness="3" BorderBrush="Goldenrod"
        CornerRadius="5" Grid.Row="5" Grid.Column="1"
        Margin="5" HorizontalAlignment="Right">
            <StackPanel Background="LightGreen">
                <Button Margin="5" HorizontalAlignment="Stretch">Save</Button>
                <Button Margin="5" HorizontalAlignment="Stretch">Cancel</Button>
            </StackPanel>
        </Border>
    </Grid>
```

Notice the DataContext set at the top of the Grid, supplying some sample data. With that sample data, the resulting view will look like this:



Title	Double Star
Author	Robert Heinlein
Year Published	1956
Quantity In Stock	5
	In print

Now back to binding capabilities

A common use of a property path is to get the count of a collection. For example, if the person designing the database were following better data modelling practices, the Book object would have an Authors collection, which contained Author objects. This would allow the Book object to store multiple authors.

At times, an application might need to expose the number of authors of the book, and this is the binding that would do that:

```
<TextBlock Margin="6"
 Text="{Binding Path=Authors.Count}" />
```

You might be thinking this isn’t a very useful binding; does the user really want to see a “3” somewhere if the book has three authors? Probably not. But the values that come from data can be used for all kinds of things by creating the right binding.

For example, we could create a binding that made part of a view appear or disappear if a book had more than one author. That requires a more complex binding than we’ve seen so far, though. We have to take in the number of authors (the count of the authors collection) and transform it to a value for the visibility of an element. That’s done with a XAML construct called a value converter. Let’s do a simple example involving addresses first, and then we’ll get back to improving our Book example.

Transforming data during binding with value converters

Suppose we want our view to hold an address, and we would like the address to look like a mailing label. Now, a data class for a company with an address will typically have two lines for the street address, like this:

```
    class Company
     {
         //Another over-simplified data class for examples
         public string CompanyName { get; set; }
         public string AddressLine1 { get; set; }
```

```
        public string AddressLine2 { get; set; }
        public string City { get; set; }
        public string StateOrProvince { get; set; }
        public string PostalCode { get; set; }
    }
```

Our first attempt at XAML to show this data might be something like this:

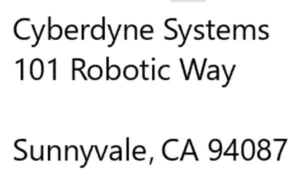
```
   <StackPanel>
        <StackPanel.DataContext>
            <local:Company CompanyName="Cyberdyne Systems"
                           AddressLine1="101 Robotic Way"
                           City="Sunnyvale" StateOrProvince="CA"
                           PostalCode="94087"/>
        </StackPanel.DataContext>


         <!--TextBlocks to hold the address lines-->
        <TextBlock Text="{Binding CompanyName}" />
        <TextBlock Text="{Binding AddressLine1}" />
        <TextBlock Text="{Binding AddressLine2}" />

        <!--This inner grid holds the city,state,postal code-->
        <!--Could have used horizontal StackPanel, but this shows an
        alternativej with more control-->
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
            <TextBlock Text="{Binding City}" />
            <TextBlock Text="," Margin="1,0,0,0" Grid.Column="1" />
            <TextBlock Margin="2,0" Grid.Column="2" Text="{Binding StateOrProvince}" />
            <TextBlock Margin="2,0" Grid.Column="3" Text="{Binding PostalCode}" />
        </Grid>

    </StackPanel>
```

The visual result would be like this:


Cyberdyne Systems
101 Robotic Way

Sunnyvale, CA 94087

But we really need to collapse that second line out when it's empty, while letting it show when it has additional address information. We can write a class called a *value converter* to do that. It will take in the string for the second address line, and return a Visibility value.[4]

[4] We covered the Visibility property of elements, and the two values of Visible and Collapsed, at the end of the previous chapter.

Creating a value converter is straight-forward. A value converter is a class that implements the IValueConverter interface[5], which is in Windows.UI.XAML.Data. The interface has two methods:

Convert – takes in the source data value and returns the value or object needed by the user interface

ConvertBack – takes in what the user enters in the UI and converts it to a value acceptable to the data object.

Most bindings you create will likely be one way, and value converters for such bindings only use the Convert method. ConvertBack is only needed for two-way bindings that need to transform the user information before storing it.[6]

Our address example only needs logic in the Convert method. Here is a value converter that takes in a string and returns a Visibility value based on it:

```
   class StringToVisibilityConverter : Windows.UI.Xaml.Data.IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string
language)
        {
            string CheckString = (string)value;
            if (String.IsNullOrEmpty(CheckString))
            {
                return Windows.UI.Xaml.Visibility.Collapsed;
            }
            else
            {
                return Windows.UI.Xaml.Visibility.Visible;
            }
        }

        public object ConvertBack(object value, Type targetType, object parameter, string
language)
        {
            return null;
        }
    }
```

This code can, of course, be simplified a bit by placing Using statements at the top for Windows.UI.Xaml.Data and Windows.UI.Xaml.[7]

This example shows the typical coding pattern in the Convert method of a value converter. The first argument in the method, named "value", contains the property value of the source property. In our example, it will be a string for the second address line. The abstract interface, though, has it in the

[5] In case you have some experience with WPF or Silverlight, IValueConverter is similar to the interface with the same name in those technologies, but it has a different argument signature. If you copy your value converters from those technologies, you'll need to fix that before using them in Uno Platform and UWP.

[6] For example, you might have a data field for phone number that has ten digits without punctuation. Older databases did that to save space. You may want to add the punctuation before displaying to the user, and allow then them to change the phone number. The logic to add the punctuation would be in the Convert method. The ConvertBack method would contain logic to strip out the punctuation for storage.

[7] I'm sure 98% of you already knew that. But if I included the using statements, the code sample would need namespace stuff. That just clutters it up and makes the code example less general.

argument list as type object. Typically, the first thing done in the Convert method is to cast this value to the appropriate data type.

Next, some sort of switching logic uses the input value to decide what to return to the visual element. In our case, it's a simple "if" clause. The code tests if the string is null or empty, and returns an appropriate visibility state. More complex value converters can have logic with nested if statements, or switch statements to decide what to return.

The decision logic can be as complex as you like, keeping in mind performance considerations. Various calculations may need to be made as part of that logic. Later in Part 2 of the book, we'll see a value converter with logic to create a shape to return based on the input.

The naming of this value converter shows the convention I like to use. The name includes the input value and what it's being converted to, making it easy to find the one you want in the Solution Explorer. Typical value converter names following this convention would be BooleanToVisibilityConverter[8], BalanceToBrushConverter, and BirthDateToAgeConverter.

Declaring and using value converter instances

The value converter is just a class, so you need to declare an instance of it to use one in your binding. This is usually done in a resources collection. To declare an instance of our StringToVisibilityConverter in a Page we are working on, the XAML would look like this, assuming we are working in a UserControl rather than a Page:

```
<UserControl.Resources>
     <local:StringToVisibilityConverter x:Key="CollapseEmptyStringConverter" />
 </UserControl.Resources>
```

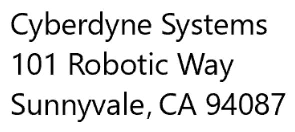
Now we can add a binding to control the visibility of the second address line. It will use that value converter instance just above. Here is the original line, from the XAML document above:

```
<TextBlock Text="{Binding AddressLine2}" />
```

It should be changed to look like this:

```
<TextBlock Text="{Binding AddressLine2}"
   Visibility="{Binding AddressLine2, Converter={StaticResource
CollapseEmptyStringConverter}}" />
```

The "mailing label" visual now looks like this:

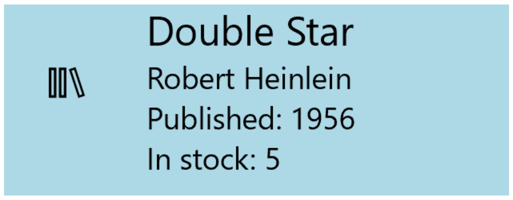

Cyberdyne Systems
101 Robotic Way
Sunnyvale, CA 94087

[8] WPF has a value converter named BooleanToVisibilityConverter as part of the standard API. That's where most of us got the idea to name value converters the way we do. That value converter did not make its way into the UWP libraries, but you only need a couple minutes to write one. It's a useful exercise, and you might as well do it, because I can pretty much guarantee you'll need one at some point.

Putting properties on value converters

The above value converter is about as simple as one can be. It returns one of two visibility results for an element, based on a string.

Sometimes value converters need more information to enhance their flexibility. Let's return to working with our Book object. Here is a simplified layout for book information, such as we might use in a list of books.[9]


Double Star
Robert Heinlein
Published: 1956
In stock: 5

Here's the XAML for that:

```
<Grid Background="LightBlue">
     <Grid.DataContext>
         <local:Book Title="Double Star" Genre="Science Fiction"
                     Author="Robert Heinlein" InPrint="True"
                     QuantityInStock="5"
                     StringDatePublished="06/01/1956" />
     </Grid.DataContext>

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

     <SymbolIcon Width="80" Height="80" Symbol="Library" VerticalAlignment="Top" />
     <StackPanel Grid.Column="1">
         <!--Various controls hold the data field values-->
         <TextBlock Text="{Binding Path=Title}" FontSize="20"/>
         <TextBlock Text="{Binding Path=Author}" />
         <TextBlock >
             <Run Text="Published:" />
             <Run Text="{Binding Path=DatePublished.Year}" />
         </TextBlock>
         <TextBlock >
             <Run Text="In stock:" />
             <Run Text="{Binding Path=QuantityInStock}" />
         </TextBlock>

     </StackPanel>
 </Grid>
```

On the left is a placeholder for a picture of the book cover. We'll get to an example that uses images of covers in Part 2.

Suppose we would like for this visual representation of a Book to vary based on whether the book is in print or out of print. One of the simplest ways to do that, and provide an intuitive user experience, is to vary the foreground color of something in the visual such as the book's title. A book that's in print could

[9] And we will get to lists of books later in the chapter.

have the normal black foreground, while an inactive book might have a gray foreground. We would then bind the Foreground property of the TextBlock holding the title to the InPrint Boolean field, using a value converter.

The value converter needs to take a peek at the Boolean value for InPrint, and output an appropriate Brush object for the TextBlock's Foreground. Here's a simple version of a value converter that does that. This time, I'll show the whole thing, including Using statements and namespaces:

```
using System;
using System.Collections.Generic;
using System.Text;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Media;
using Windows.UI;

namespace Chapter4.Shared
{
    class BooleanToBrushConverter : IValueConverter
    {

        public object Convert(object value, Type targetType, object parameter, string
language)
        {
            bool InPrintValue = (bool)value;
            if (InPrintValue)
            {
                return new SolidColorBrush(Colors.Black);
            }
            else
            {
                return new SolidColorBrush(Colors.Gray);
            }

        }

        public object ConvertBack(object value, Type targetType, object parameter, string
language)
        {
            return null;
        }
    }
}
```

This implementation has some problems. First, it creates a new SolidColorBrush every time it's called. That works, but is not good for performance.

A more interesting problem is that this value converter is only acceptable if the Brush for in print is always black, and the one for out of print is always gray. Of course, that's silly. You know some designer is going to come along and want to change the brushes used to indicate in print vs. out of print.

To fix that, the value converter needs properties for those brushes, so they can be configured. Those properties can have default values of black and gray, but those properties can be changed in XAML as needed. Here is the changed value converter with those two properties (and which incidentally fixes the "create a new brush every time" problem):

```
using System;
using System.Collections.Generic;
using System.Text;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Media;
using Windows.UI;

namespace Chapter4.Shared
{
    class BooleanToBrushConverter : IValueConverter
    {
        public Brush TrueBrush { get; set; }
        public Brush FalseBrush { get; set; }

        public BooleanToBrushConverter()
        {
            TrueBrush = new SolidColorBrush(Colors.Black);
            FalseBrush = new SolidColorBrush(Colors.Gray);
        }

        public object Convert(object value, Type targetType, object parameter, string
language)
        {
                        bool InPrintValue = (bool)value;
            if (InPrintValue)
            {
                return TrueBrush;
            }
            else
            {
                return FalseBrush;
            }

        }

        public object ConvertBack(object value, Type targetType, object parameter, string
language)
        {
            return null;
        }
    }
}
```

Remember that we declare instances of value converters in XAML to use them. When we do that for this value converter, we can optionally set the TrueBrush and FalseBrush properties to any brush we like.[10] Let's see it first using the default brushes:

```
    <UserControl.Resources>
        <local:BooleanToBrushConverter x:Key="ActiveInActiveConverter" />
    </UserControl.Resources>
```

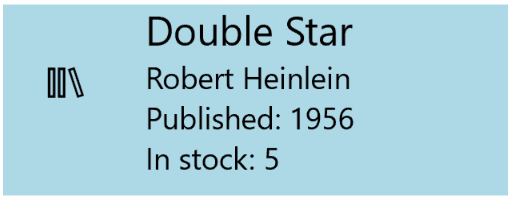
Then we can put a binding on the Foreground of the Title to use the InPrint property to set the Foreground. The TextBlock XAML for the title then looks like this:

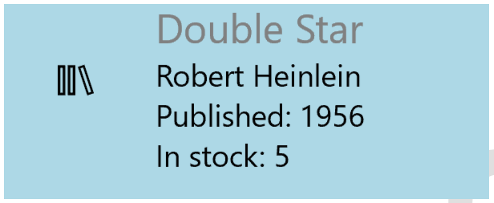
```
<TextBlock Text="{Binding Path=Title}" FontSize="20"
```

[10] Chapter 5 on graphical elements will get more into brushes and colors.

```
    Foreground="{Binding InPrint, Converter={StaticResource ActiveInActiveConverter}}"/>
```

Here’s the visual rendering, first for a book with InPrint=True and second for InPrint=False:

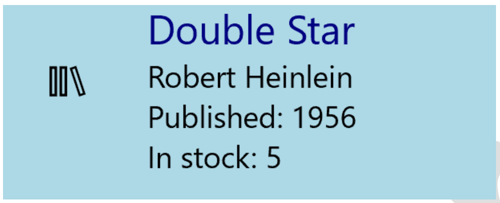

Double Star
Robert Heinlein
Published: 1956
In stock: 5

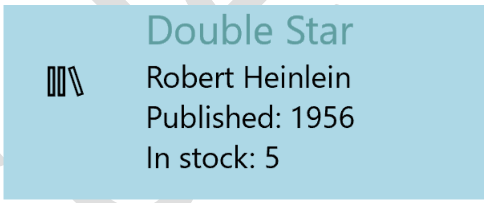

Double Star
Robert Heinlein
Published: 1956
In stock: 5

But since we have properties on the value converter, we can have the title using any colors we like for in print and out of print. Suppose we change the value converter to look like this:

```
<UserControl.Resources>
    <local:BooleanToBrushConverter x:Key="ActiveInActiveConverter"
                                   TrueBrush="Navy" FalseBrush="CadetBlue"/>
</UserControl.Resources>
```

With this changed value converter, we’ll get results like this for in-print and out-of-print:


Double Star
Robert Heinlein
Published: 1956
In stock: 5


Double Star
Robert Heinlein
Published: 1956
In stock: 5

I’m not saying these are ideal colors from a visual design perspective[11], but I hope you get the idea – you can have any colors or brushes you like in those properties, and easily evolve them over time.

Some use cases for value converters

Here are a few of the many scenarios in which we have used value converters on client projects to improve user experience.

- Transforming a date into a time span in the past or future. For example, transforming a birth date into an age.
- Transforming a status, such as “on time” or “overdue”, into a brush to use in the visual element for the status.
- Hiding and showing entire parts of a visual, based on the data. For example, in a payment screen, it might be desirable to change the fields for credit card vs. cash vs. check.

And there are many more. That list is just to give you a flavor of the flexibility you gain through value converters.

[11] Going into visual and layout design would be a book all on its own. If I ever get time, I’d like to write one for developers to understand the basics.

Value converters vs. calculated properties

In Part 2, we'll be covering some basics on a common coding pattern used in XAML. It's called Model-View-ViewModel, and is usually known by its acronym MVVM.[12]

This pattern includes an enhanced data model called the ViewModel. A normal data model is like our Book class we've been using. It has properties that are filled by data from some repository. The enhanced ViewModel in MVVM wraps a model such as that, but often adds additional properties that are used only in the user interface.

Many of these are calculated properties. Those properties can contain similar logic to what we could put in the Convert method of a value converter. Such properties allow simple bindings and relieve you from creating and managing a value converter class.

For example, a typical data model for a Patient would have a birth date for the patient. But a healthcare practitioner doesn't care what your birthday is. They want to know old you are.

With MVVM, the ViewModel, can have a calculated property for age. It takes the value of the birth date, which is a DateTime, and uses the current date to calculate the patient's age.

Some XAML applications use the MVVM pattern, and others don't. MVVM certainly has some benefits, but it adds complexity too, so I don't regard it as universally necessary. In Part 2, I'll tell you my own recommendations for deciding whether you want to use the pattern in your app.[13]

If your app does use the MVVM pattern, you'll probably still need to write value converters. Some of them are so widely useful that you want them to always be available. Others work intricately with the XAML API, and you don't want your ViewModel to have excess baggage in the form of references to the XAML libraries.

My own personal guideline is this: if I'm using MVVM, I only write value converters when I need to return some kind of data type that's in the Windows.UI namespaces. That means I'll use value converters for things like brushes, colors, visibility, and graphical elements. I'll normally use calculated properties on the ViewModel for anything else, though I'm not dogmatic about that. Sometimes I have a conversion that is so widely useful that I don't want to use it in every ViewModel that needs it. In that case, I'll still use a value converter instead of calculated properties on several ViewModels.

For the rest of our discussion in this chapter and the next few, I'll ignore the MVVM option and focus on value converters as the way we will convert information into a form the user needs or likes.

Less frequently needed data binding capabilities

The bulk of the data bindings you create will use the capabilities we've discussed. However, there are some additional data binding capabilities that are powerful but less frequently used. These include:

- Getting data from an arbitrary source object

[12] I think all discussions on MVVM are improved by pronouncing in like a motorcycle noise. "Emmmmm vee vee emmm"

[13] You will likely encounter fanatic MVVM purists in the XAML community. They will insist that you should always use MVVM, even for the simplest apps. I'm suspicious of such blanket assertions. When we discuss MVVM in more detail in Chapter x, I'll discuss my reservations about the "MVVM always and forever" school of thought.

- Creating data bindings in code
- Using parameters in value converters
- Including fallback values for binding failure and null data

These capabilities will be discussed briefly in Chapter 7, Useful Things Part 2. I'm putting them off for now because we need to get on to something much more important – data templates.

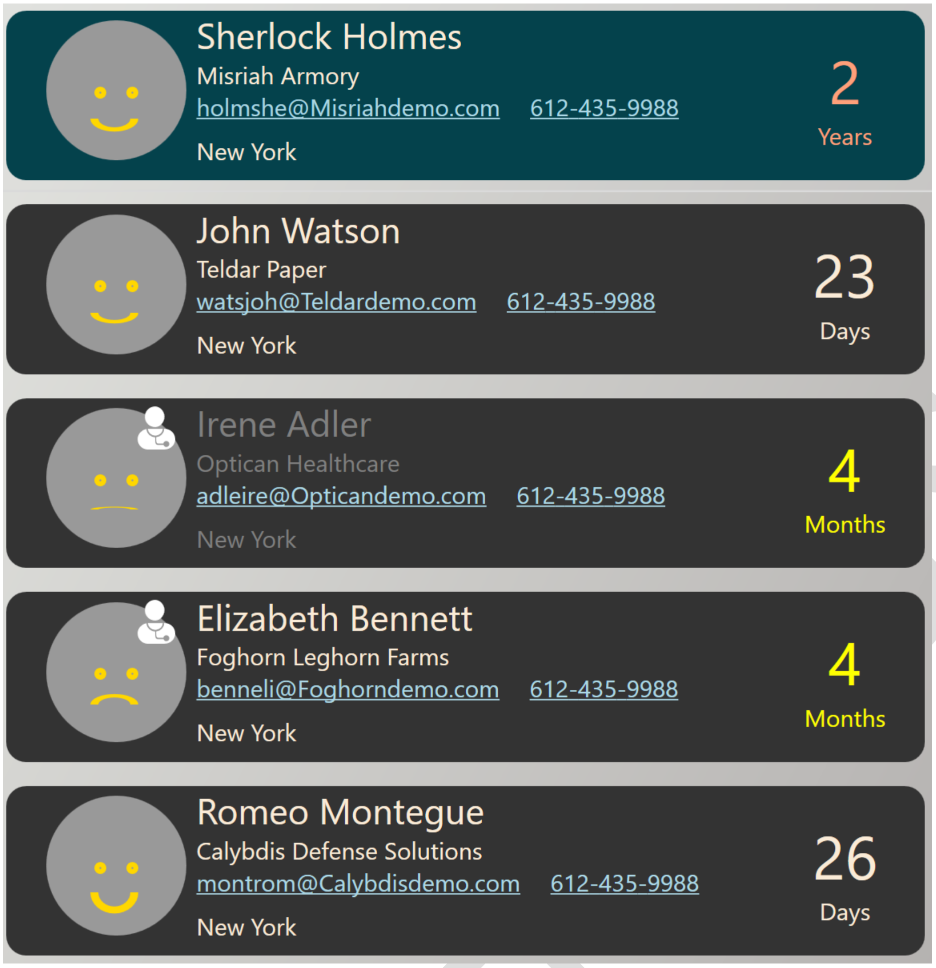
Data Templates

I vividly recall when I realized the power of data templates while writing my first XAML application. It changed the way I thought about designing user interfaces more than any other single XAML concept.

Data templates are most commonly used for lists. Most business application databases have lots of tables, and each one is a potential list in the application. For example, a user may need to see a list of customers, or a list of transactions for a customer.

Older UI technologies lacked good ways to show data lists to users. List boxes and data grids were usually used, but once you've seen the XAML equivalents, you'll look on those older versions as downright primitive. Each data item in the list can be shown with as much detail as desired, using the full set of layout and graphical capabilities of XAML. That includes everything we've seen so far, plus some graphical capabilities and additional controls that we'll discuss later.

Here is a screen shot of a list of data items shown in XAML using a data template. It's from one of my demo programs. Imagine that this is a healthcare app, and this is the list of contacts. Some of the contacts would be doctors, but others would not.


Sherlock Holmes
Misriah Armory
holmshe@Misriahdemo.com 612-435-9988
New York
2
Years

John Watson
Teldar Paper
watsjoh@Teldardemo.com 612-435-9988
New York
23
Days

Irene Adler
Optican Healthcare
adleire@Opticandemo.com 612-435-9988
New York
4
Months

Elizabeth Bennett
Foghorn Leghorn Farms
benneli@Foghorndemo.com 612-435-9988
New York
4
Months

Romeo Montegue
Calybdis Defense Solutions
montrom@Calybdisdemo.com 612-435-9988
New York
26
Days

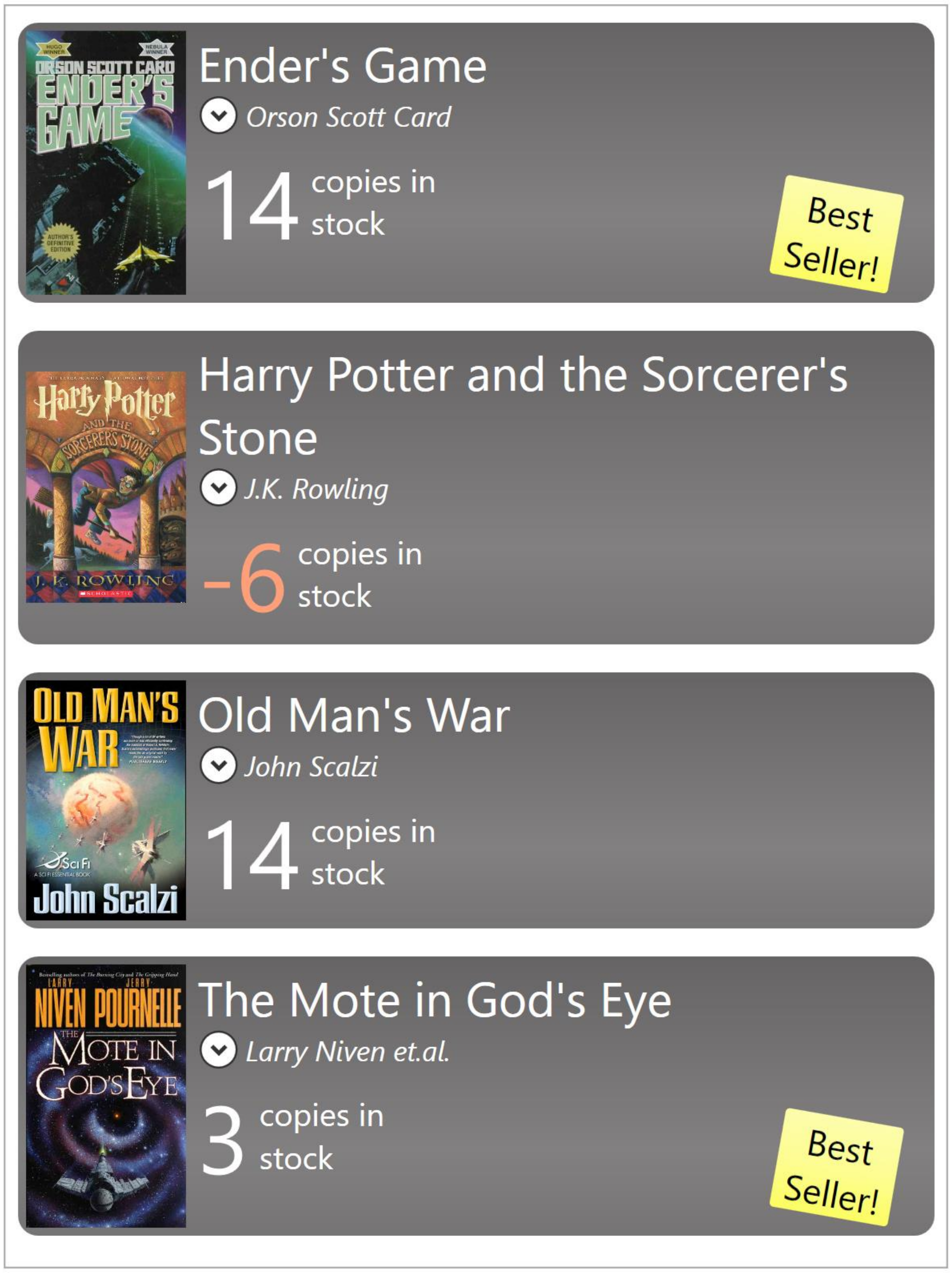
Notice the visual representation of that contact's satisfaction, in the form of a cartoon face. There is a physician symbol for contacts who are physicians. Finally, the time since last contact is on the right, color coded with yellow for times over a month, and light red for times over a year.

We have not yet discussed all the XAML capabilities needed to create this data template. I'll cover the rest of them in Part 2, mostly in Chapter 5 on graphical elements. I hope this example gives you some incentive to keep on learning what XAML can do.

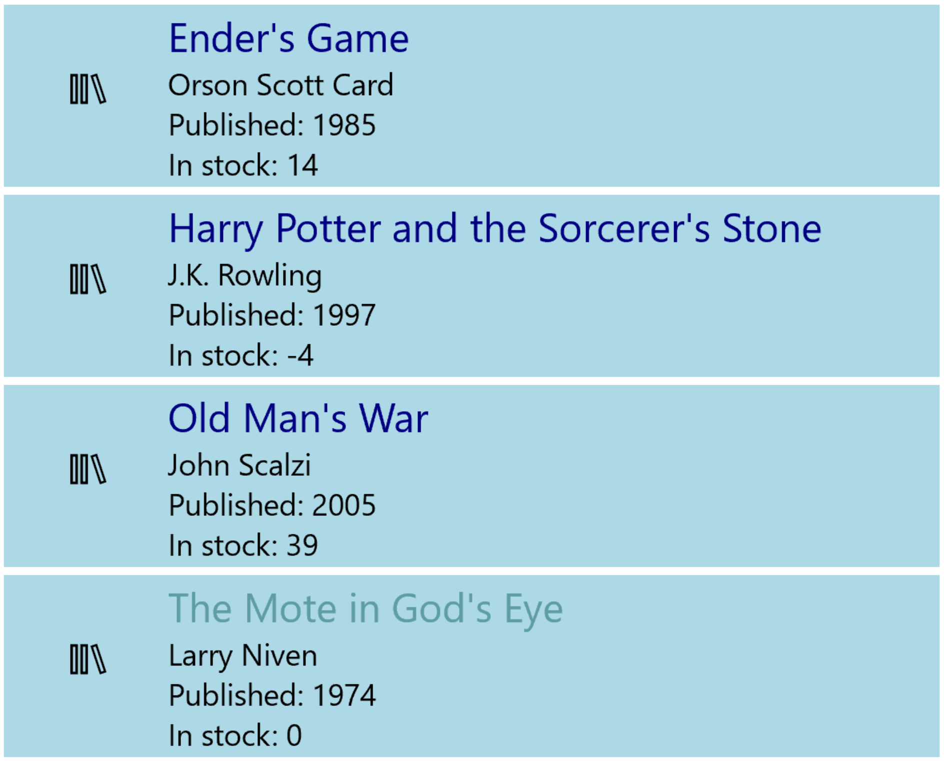
However, we have discussed some of what you need for such a list. Notice the gray text colors indicating inactive customers, and the transformation of the date of last contact into a time span. Both those are done with the value converters we just finished discussing.

Here's another example from another one of my demo programs – a list of books.[14] Again, this is advanced beyond what we've seen so far, but gives us a nice target to shoot for:

[14] This is taken from an exercise in my XAML class. The students are challenged to produce this result. The vast majority of students are able to do it. It's not as hard as it looks if you can think in XAML.



Since we can’t do all of this example yet, let’s get back to our old familiar Book object. A simpler list of books might look like this:



Ender's Game
Orson Scott Card
Published: 1985
In stock: 14

Harry Potter and the Sorcerer's Stone
J.K. Rowling
Published: 1997
In stock: -4

Old Man's War
John Scalzi
Published: 2005
In stock: 39

The Mote in God's Eye
Larry Niven
Published: 1974
In stock: 0

Does this layout look familiar? It should. I used the XAML for showing a book from earlier in the chapter. I just put it in a different place so that it applies to a list of books instead of a single book. Cool, huh? Let's talk about how that works.

List-oriented controls that use data templates

XAML has several controls to display a list of data items. They have different layout arrangements and interaction patterns. The four you will likely use the most are ListView, GridView, FlipView, and ComboBox.

All these controls descend from a base class called ItemsControl. While each one has some special capabilities, the basics of using them are the same for all three.

In this chapter, we'll focus on the ListView to show that core set of capabilities. In later chapters, we'll explore the other ItemsControls, and look at advanced usage of data templates.

What does an ItemsControl really do?

Thinking in XAML means realizing that most elements have rather narrow responsibilities. That doesn't seem to describe a control like a ListView. It does quite a bit when you put it in XAML and set a couple of properties. You immediately get a scrolling list of items with the ability to select one and do something with it.

But the ListView is hiding a lot of complexity. It does a lot because it contains other elements that do much of the heavy lifting underneath.

For example, a ListView does not really know how to stack or scroll the items it contains. That is delegated to a StackPanel[15] and a ScrollViewer respectively. In later chapters, we'll see how that dramatically increases the flexibility of a ListView.

For now, though, let's look at the real primary responsibility of a ListView. It's simply this: it takes a collection of data items, and uses that data template I just discussed to create a visual representation of each data item. Then it allows selection of one (or more) of those visual items.

Here's an oversimplified diagram showing how the ListView combines data and a template to get visual items. The template is an instance of the DataTemplate class, which is used in lots of places in XAML, and we will be seeing it several times in later chapters.

{insert the diagram}

ItemsSource and ItemTemplate properties

A ListView gets the raw data items through the ItemsSource property. This property is set to a collection of data objects.

In real applications, ItemsSource is commonly set in code. Some sort of asynchronous call to a data repository is made to request a list of items, and when the data returns, the results are used to set ItemsSource. Appendix XX has a simple end-to-end example of getting data from a public REST API and placing the data in a ListView by setting ItemsSource in code.

However, for instructional purposes, we can set ItemsSource to a list of data items defined in XAML. As I mentioned earlier, we are not going to talk in detail about getting data from repositories because that's done in lots of ways and experienced developers already know how to do it. Defining sample data items in XAML is a stand-in for getting data in code, but it dramatically simplifies my examples.

Let's walk through an example.

Remember that we are using a simplified data class for a Book object, shown earlier in the chapter:

```
class Book
 {
     public string Title { get; set; }
     public string Author { get; set; }
     public string Genre { get; set; }
     public DateTime DatePublished { get; set; }
     public bool InPrint { get; set; }
     public int QuantityInStock { get; set; }

     //Needed to declare a date in XAML
     //Yeah, it's a hack for the book. Deal with it.
     public string StringDatePublished
     {
         get { return DatePublished.ToShortDateString(); }
         set { DatePublished = DateTime.Parse(value); }
     }
```

[15] Actually a VirtualizingStackPanel to help performance by rendering items only as they are needed.

```
    }
```

To that, we need to add a collection class to hold a list of books. It inherits from a generic collection type and declares the type of data objects to hold. Such a typed collection class can be declared in XAML, so we can create sample data without needing a database.[16]

The most common generic base class used for collections in XAML is ObservableCollection, because it generates events that help list-oriented controls update themselves when data items are added or removed. A simple version of such a class looks like this:

```
    class BookCollection : ObservableCollection<Book>
    {
    }
```

Again, these classes are stand-ins for data models and viewmodels[17] that you would use in a real application. These are just simple classes with some properties to show how things work.

Next, let's use these classes in XAML to declare a list of books to use for our sample data:

```
        <local:BookCollection x:Key="SampleBooks">
            <local:Book Title="Ender's Game" Author="Orson Scott Card"
                        QuantityInStock="14" InPrint="True" Genre="Science Fiction"
                        StringDatePublished="01/15/1985">

            </local:Book>

            <local:Book  Title="Harry Potter and the Sorcerer's Stone"
                         Author="J.K. Rowling"
                         QuantityInStock="-4" InPrint="True" Genre="Fantasy"
                         StringDatePublished="06/26/1997">
            </local:Book>

            <local:Book Title="Old Man's War" Author="John Scalzi"
                        QuantityInStock="39" InPrint="True" Genre="Science Fiction"
                        StringDatePublished="06/01/2005">
            </local:Book>

            <local:Book Title="The Mote in God's Eye" Author="Larry Niven"
                        QuantityInStock="0" InPrint="False" Genre="Science Fiction"
                        StringDatePublished="06/01/1974">

            </local:Book>

        </local:BookCollection>
```

This XAML would be placed in an appropriate resources block. In the demo program for this chapter, it's in the resources area at the top of the UserControl named SimpleBookList.

[16] You'll see this technique used in all of my examples that require a list of data items. It simplifies distribution and deployment of the samples. Remember that it's just a convenience for instructional purposes – not a technique to use in real applications.
[17] Viewmodels are a more sophisticated data holder, used in the MVVM pattern that we'll be getting to later in the book.

Since this collection is a resource, with the resource name SampleBooks, we can refer to it from other places in that UserControl. I'll use it in the XAML example of a ListView, just below.

By the way, these classes show my personal convention for naming data classes. The class holding a data item usually has some kind of entity name, such as Person or Book. The collection class then appends "Collection" to the entity name, to get collection classes such as PersonCollection or BookCollection.[18]

Here is the XAML that declares a ListView that uses this data, using a DataTemplate declared inside the ListBox:

```
       <ListView ItemsSource="{StaticResource SampleBooks}" Name="BooksListView"
           HorizontalContentAlignment="Stretch">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <Grid Background="LightBlue" Margin="2" Padding="2">

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

                        <SymbolIcon Width="80" Height="80" Symbol="Library"
VerticalAlignment="Top" />
                        <StackPanel Grid.Column="1">
                            <!--Various controls hold the data field values-->
                            <TextBlock Text="{Binding Path=Title}" FontSize="20"
                       Foreground="{Binding InPrint, Converter={StaticResource
ActiveInActiveConverter}}"/>
                            <TextBlock Text="{Binding Path=Author}" />
                            <TextBlock >
                                <Run Text="Published:" />
                                <Run Text="{Binding Path=DatePublished.Year}" />
                                            </TextBlock>
                                            <TextBlock >
                                <Run Text="In stock:" />
                                <Run Text="{Binding Path=QuantityInStock}" />
                            </TextBlock>

                        </StackPanel>
                    </Grid>
                </DataTemplate>

            </ListView.ItemTemplate>
            <ListView.ItemContainerStyle>
                <Style TargetType="ListViewItem" >
                    <Setter Property="HorizontalContentAlignment" Value="Stretch" />
                </Style>
            </ListView.ItemContainerStyle>
        </ListView>
```

[18] I'm not dogmatic about using these conventions, but you will see them used throughout this book. Also, I use a variation on this convention for viewmodels, which I'll show later when I take them up.

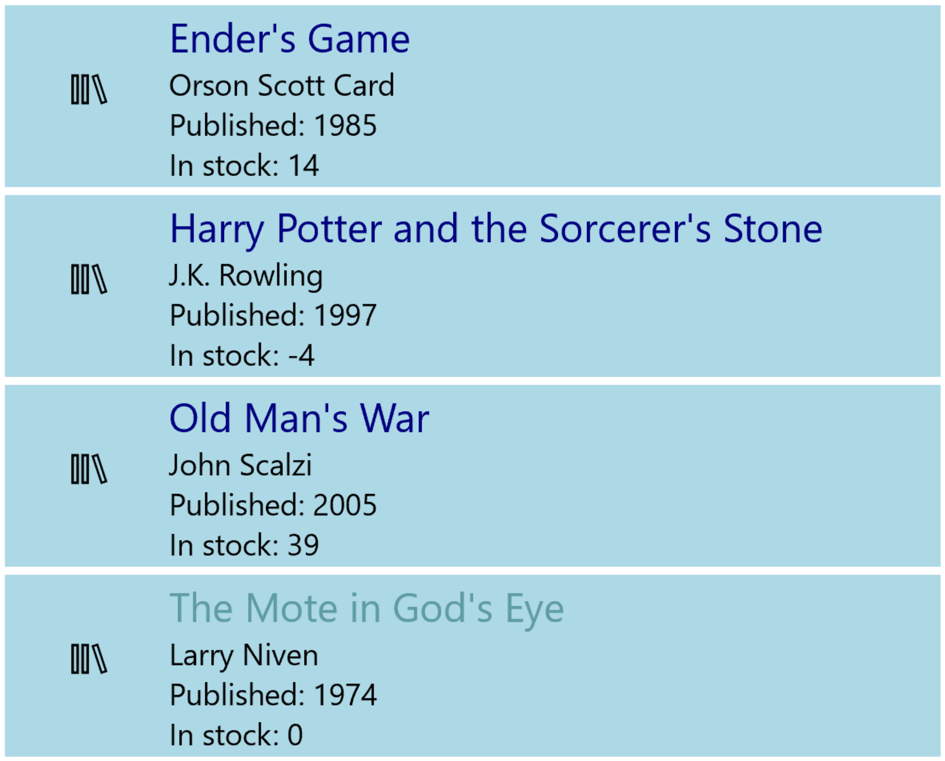
The last part of this XAML sets the ItemContainerStyle property, and I can't go into it in depth because we have not covered styles yet. They will be discussed in Chapter 6. For now, I'll just tell you that those lines of XAML in the ItemContainerStyle make all the items in the ListView stretch to the width of the ListView. If those lines are not included (try it!) then the book items will be of different widths, giving a ragged layout.

The main area to focus upon, which I've bolded above, is the setting of the ListView's ItemTemplate property. ItemTemplate needs an object of type DataTemplate.

As we discussed in the diagram above, DataTemplate serves as a pattern for formatting the visual representation of a data object. DataTemplate instances are heavily used in XAML; we're just seeing one simple example of a DataTemplate.

As you can see, the DataTemplate uses the data bindings we covered earlier in the chapter to hook properties of the data class to visual elements. The rest of it is just standard XAML – a Grid and a StackPanel for layout, and some TextBlock elements to expose the values of data properties. We don't have to walk through that XAML because we already did that in the example above that showed a single book. That was in the section discussing value converters, and the exact same value converters are used in this sample as well.

Here's a screenshot of that ListView rendered to the screen, which we also showed earlier in this section:



Ender's Game
Orson Scott Card
Published: 1985
In stock: 14

Harry Potter and the Sorcerer's Stone
J.K. Rowling
Published: 1997
In stock: -4

Old Man's War
John Scalzi
Published: 2005
In stock: 39

The Mote in God's Eye
Larry Niven
Published: 1974
In stock: 0

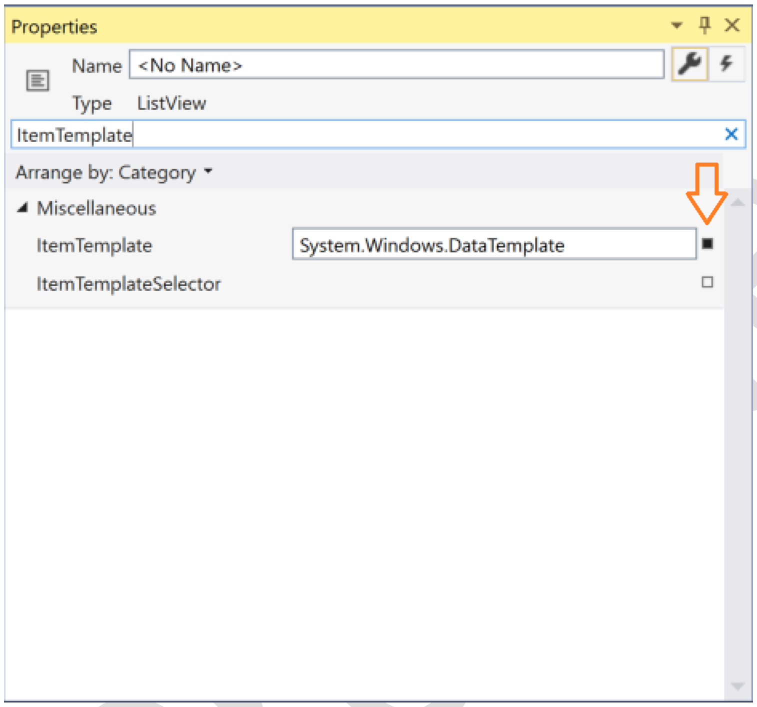
This ListView has a very simple DataTemplate, but it still shows some of the capabilities that are useful for helping your users understand the data presented to them. Different data fields have different sizes on the screen, for example. But there's lots more we can do.

This example is oversimplified, and anyone who knows much a data structures can probably spot the biggest flaw – the Book class only supports one author, and many books have multiple authors. However, the techniques for dealing with child collections in a DataTemplate need to wait until Part 2 on Going Deeper. We will be seeing our Book example again then.

Using a data template declared as a resource

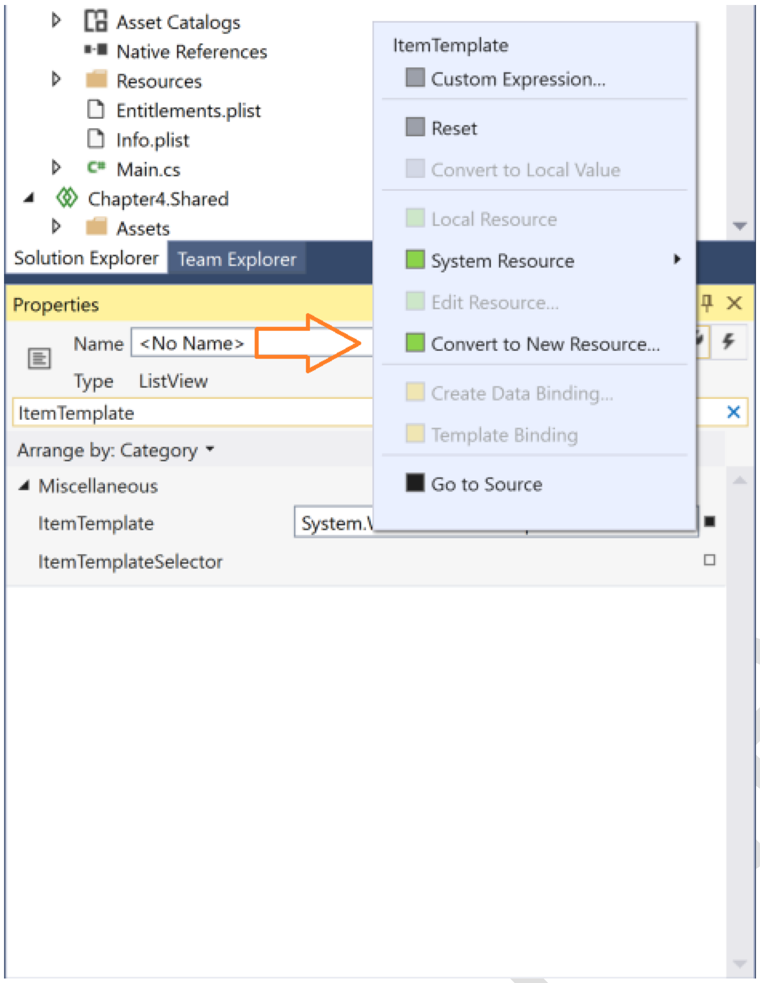
Declaring the data template inside the ListView is simple, but there are several reasons for an alternative. A data template can be defined as a resource instead. Then it can be used by multiple controls that need a data template for the same entity.

If you open the ItemTemplate property for the ListView in the Property Box, you'll see an option to open a menu with various options.

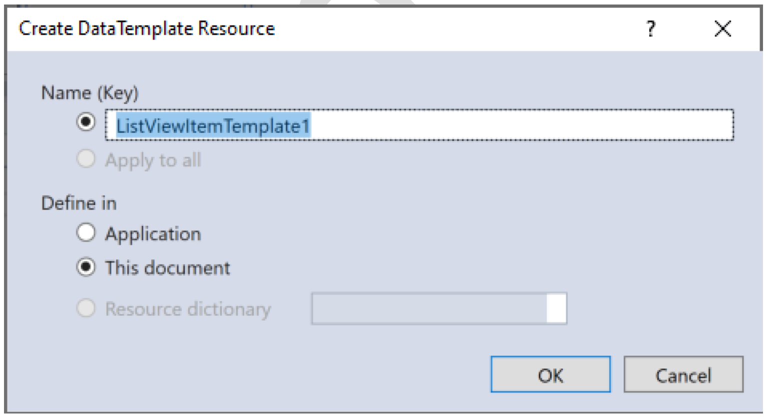


<No Name>
ListView
ItemTemplate
ItemTemplate System.Windows.DataTemplate
ItemTemplateSelector

When you press the glyph, the list of options includes one to make the property value (that is, the data template) into a resource. Here's where you find that:



Then you can select the option to Convert to New Resource. You'll get this dialog:



Name (Key)
[✓ ListViewItemTemplate1]
Apply to all

Define in
Application
[✓ This document]
Resource dictionary

You give the resource a name, which is really the x:Key for the resource we discussed earlier. You also select the level at which the resource should be declared. You can declare it in the same XAML document you're currently in, or at the level of the Application. Another more advanced option is to select a resource dictionary to store it, but we won't discuss those until Part 2.

Let’s assume you picked “This Document”, and gave the new resource the name “BookDataTemplate”. Then you’ll see the following XAML in the resources section higher in the document, and it will look like this:

```
        <DataTemplate x:Key="BookViewTemplate">
            <Grid Background="LightBlue" Margin="2" Padding="2">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <SymbolIcon Height="80" Symbol="Library" VerticalAlignment="Top"
Width="80"/>
                <StackPanel Grid.Column="1">
                    <TextBlock Foreground="{Binding InPrint, Converter={StaticResource
ActiveInActiveConverter}}"
                               FontSize="20" Text="{Binding Title}"/>
                    <TextBlock Text="{Binding Author}"/>
                    <TextBlock><Run Text="Published:"/><Run Text=" "/><Run Text="{Binding
DatePublished.Year}"/></TextBlock>
                    <TextBlock><Run Text="In stock:"/><Run Text=" "/><Run Text="{Binding
QuantityInStock}"/></TextBlock>
                </StackPanel>
            </Grid>
        </DataTemplate>
```

This is the same XAML in the earlier ListView above, in the ItemTemplate property, except that it has an x:Key now because it’s a resource.

The XAML for the ListView will change to set the ItemTemplate property to the new resource, so that XAML will now look like this:

```
        <ListView ItemTemplate="{StaticResource ListViewItemTemplate1}"
                  ItemsSource="{StaticResource SampleBooks}" Name="BooksListView"
                  HorizontalContentAlignment="Stretch">
            <ListView.ItemContainerStyle>
                <Style TargetType="ListViewItem" >
                    <Setter Property="HorizontalContentAlignment" Value="Stretch" />
                </Style>
            </ListView.ItemContainerStyle>
        </ListView>
```

Changing data templates on the fly

One of the most powerful things about data templates is the ability to switch them on the fly. To do that, you simply set the ItemTemplate property to a new value. That change can be made in code behind, or using a binding. We’ll do an advanced example in Part 2 using binding, but to show this capability in a simple way, let’s walk through changing the data template in code.

First, we’ll need a second data template. I’ll just copy the first one, called BookDataTemplate above, and change it up a bit. I’ll make the background color different, move the year of publication, and make it really big. That data template would be a new resource, with a different name such as BookViewWithPublicationYearTemplate. Its XAML would then look like this, and it can be placed just underneath the first data template declared as a resource.

```
     <DataTemplate x:Key="BookViewWithPublicationYearTemplate">
```

```
                <Grid Background="LightGoldenrodYellow" Margin="2" Padding="2">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto"/>
                        <ColumnDefinition Width="*"/>
                    </Grid.ColumnDefinitions>
                    <SymbolIcon Height="80" Symbol="Library" VerticalAlignment="Top"
Width="80"/>
                    <StackPanel Grid.Column="1">
                        <TextBlock Foreground="{Binding InPrint, Converter={StaticResource
ActiveInActiveConverter}}"
                                   FontSize="20" Text="{Binding Title}"/>
                        <TextBlock Text="{Binding Author}"/>
                        <!--<TextBlock><Run Text="Published:"/><Run Text=" "/><Run
Text="{Binding DatePublished.Year}"/></TextBlock>-->
                        <TextBlock><Run Text="In stock:"/><Run Text=" "/><Run Text="{Binding
QuantityInStock}"/></TextBlock>
                    </StackPanel>
                    <TextBlock Text="{Binding DatePublished.Year}"
HorizontalAlignment="Right"
                               VerticalAlignment="Bottom" Margin="5" FontSize="24"
Grid.Column="1" />
                </Grid>
            </DataTemplate>
```

Now let's put a ToggleSwitch on our view to toggle the data template between our two possibilities. We need a place to put it, so the Grid containing the ListView needs another row at the bottom. The top part of the XAML for the Grid will then look like this:

```
   <Grid>
        <Grid.RowDefinitions>
            <!--Row for the ListView-->
            <RowDefinition Height="*" />

            <!--Row for the ToggleSwitch-->
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
```

We'll need to handle the Toggled event to know when to change templates, so the XAML for the ToggleSwitch looks like this:

```
        <ToggleSwitch Header="Select template" OnContent="Original template"
                      OffContent="Changed template" Toggled="ToggleSwitch_Toggled"
                      Grid.Row="1" Name="TemplateToggleSwitch" />
```

The code behind to switch data templates looks like this:[19]

```
        private void ToggleSwitch_Toggled(object sender, RoutedEventArgs e)
        {
            if (TemplateToggleSwitch.IsOn)
            {
                DataTemplate FirstTemplate =
(DataTemplate)this.Resources["BookViewTemplate"];
```

[19] This code is as simple as I can make it just to show the concept. It could clearly be improved by checking to make sure the resource exists, etc., and you would want to do those niceties in any production application. I'm depending on readers to already know how to write professional code.

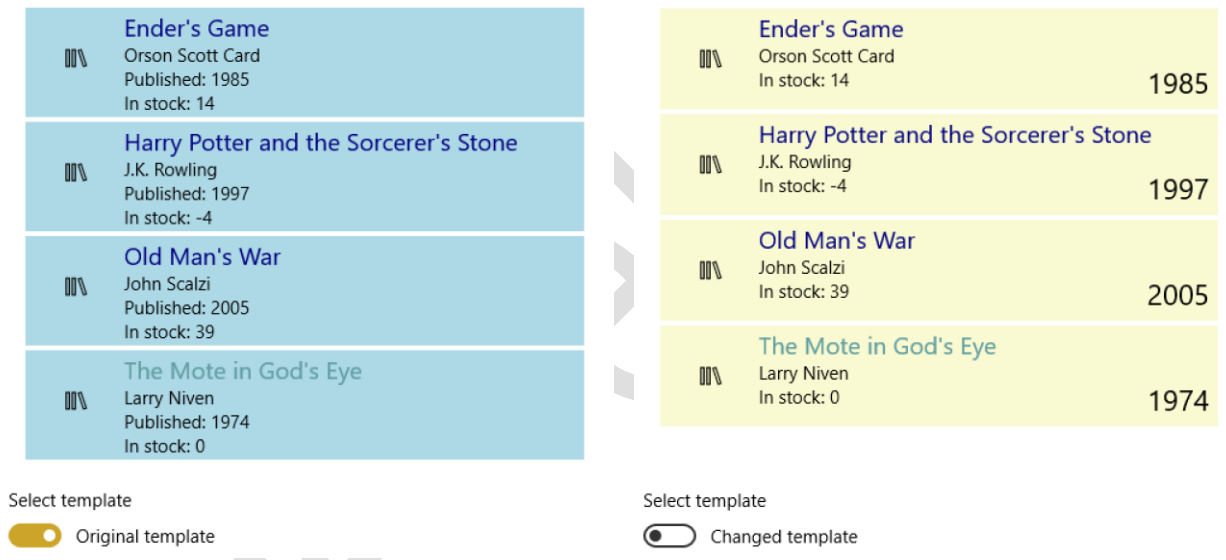
```
                BooksListView.ItemTemplate = FirstTemplate;

            }
            else
            {
                DataTemplate SecondTemplate =
(DataTemplate)this.Resources["BookViewWithPublicationYearTemplate"];
                BooksListView.ItemTemplate = SecondTemplate;

            }
        }
```

Flipping the ToggleSwitch will now change the data template for book objects in the ListView. Notice that no refresh operation is necessary. The XAML rendering engine realizes the ItemTemplate property has changed, and automatically starts a re-render of the ListView.

Here are side-by-side views of the two templates in action:



Ender's Game
Orson Scott Card
Published: 1985
In stock: 14

Harry Potter and the Sorcerer's Stone
J.K. Rowling
Published: 1997
In stock: -4

Old Man's War
John Scalzi
Published: 2005
In stock: 39

The Mote in God's Eye
Larry Niven
Published: 1974
In stock: 0

Select template
Original template

Ender's Game
Orson Scott Card
In stock: 14
1985

Harry Potter and the Sorcerer's Stone
J.K. Rowling
In stock: -4
1997

Old Man's War
John Scalzi
In stock: 39
2005

The Mote in God's Eye
Larry Niven
In stock: 0
1974

Select template
Changed template

As I mentioned, there are other, slicker ways to change out the data template. Part 2 will show an example of doing it from a drop down ComboBox, with a preview of each template in the drop down.

Using data binding in templates

All the data binding capabilities I covered earlier in the chapter are also available inside data templates. This example is using the value converter that changes the color of text, depending on the Boolean flag for whether the book is in print.

There are countless ways to use clever bindings to enhance data templates and make it intuitive for users to understand their information. We'll get into several more such ways after we cover graphical elements in the first chapter of Part 2.

Using a ListView for navigation in a list

We've focused up to this point on showing data in a list. However, it's common for a user to need to see more detail of a data item, or edit the information in the item.

Let's see how to let the user navigation among the Book items. First, we'll need a simple data-entry type form for a Book. It follows the pattern we saw in Chapter 2. To make it side-by-side with the ListView, the root Grid needs two columns. That means this XAML goes underneath the XAML that declared two rows, a couple pages back:

```
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <!--Column for the ListView-->
    <ColumnDefinition Width="*" />

    <!--Column for the editing view-->
    <ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>
```

Here's the XAML from Chapter 2 for editing a book, with the following changes:

- Grid.Column is set to 1, so that the editing view is to the right of the ListView
- The Grid has a minimum width, to keep it's width from bouncing around as different books are loaded into it
- The DataContext for the view is set to the current selected book in the ListView
- All fields have appropriate bindings to show data from an object instead of having the view's data hard-coded

(Changes are in bold.)

```
<Grid Background="LightBlue" Grid.Column="1" MinWidth="300"
      DataContext="{Binding ElementName=BooksListView, Path=SelectedItem}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Rectangle Fill="LightGoldenrodYellow" Margin="2"
   Grid.RowSpan="2" Grid.ColumnSpan="2"
   RadiusX="7" RadiusY="7" />
    <TextBlock Text="Title" Margin="6"
   HorizontalAlignment="Right" />
    <TextBlock Text="Author" Margin="6" Grid.Row="1"
   HorizontalAlignment="Right" />
    <TextBox Grid.Column="1" Margin="6"
 Text="{Binding Title, Mode=TwoWay}" />
    <TextBox Grid.Column="1" Grid.Row="1" Margin="6"
 Text="{Binding Author, Mode=TwoWay}" />

    <Border BorderThickness="3" BorderBrush="Goldenrod"
            CornerRadius="5" Grid.Row="2" Grid.Column="1"
            Margin="5" HorizontalAlignment="Right">
        <StackPanel Background="LightGreen">
            <Button Margin="5" HorizontalAlignment="Stretch">Save</Button>
```

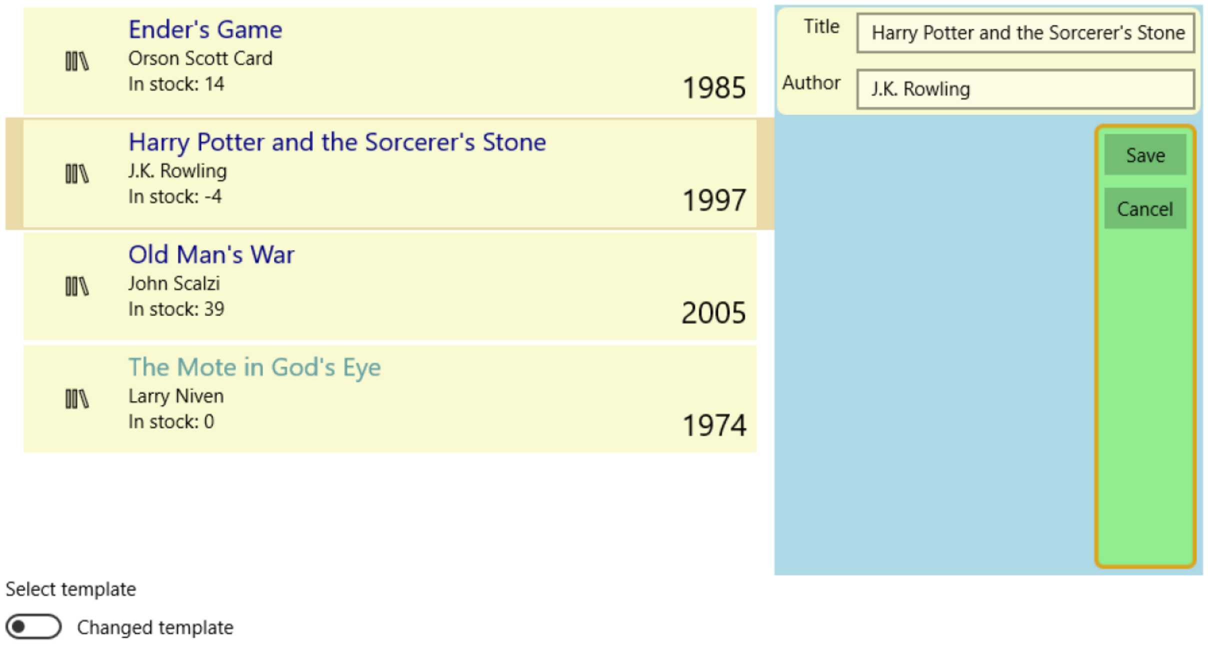
```
                    <Button Margin="5" HorizontalAlignment="Stretch">Cancel</Button>
                </StackPanel>
            </Border>
        </Grid>

    </Grid>
```

Notice the child Grid has a DataContext set with a binding, using the SelectedItem of the ListView holding Books. This is an important point to understand. Bindings are not just for getting data from a data object into and out of visual elements. Bindings can also be used to tie together visual elements.

In this case, changing the SelectedItem in the ListView by clicking on a Book automatically sets the DataContext of the data entry area. That makes the data entry area show the information for the book that is currently selected in the list. This is about as clean a navigation mechanism as it is possible to have. It's automatic, and requires absolutely no code behind for the navigation.[20]

Here is a screen cap of the final result. As with all the examples in this chapter, you can get a working project in the Chapter 4 code download.



Ender's Game
Orson Scott Card
In stock: 14
1985

Harry Potter and the Sorcerer's Stone
J.K. Rowling
In stock: -4
1997

Old Man's War
John Scalzi
In stock: 39
2005

The Mote in God's Eye
Larry Niven
In stock: 0
1974

Title: Harry Potter and the Sorcerer's Stone
Author: J.K. Rowling

There is one limitation that isn't obvious. If you change the book's title or author, that change does not propagate back to the ListView. That's because our simple data classes do not have any property change notification. If you do any client-side programming with any .NET Framework platform, you are likely to be familiar with that problem, and the INotifyPropertyChanged interface that fixes it. I'll start taking care

[20] There are various needed capabilities that I'm not covering in the interests of simplicity. For example, I'm not including any indication of how changed book information would be saved. That could be done automatically when the user moved to a new book, or there could be some kind of Save visual element they would press to save their information. Of course, it's completely academic for this example, because we don't have a database to store changes in.

of that in my sample data classes in Part 2, where we are going deeper and getting closer to production-level examples.

Part 1 conclusion

You’ve now seen the most essential capabilities of XAML for getting started on the Uno Platform. This is enough to create simple apps, but there’s so much more to learn about XAML.

If you are rather new to XAML, I’d encourage you to take a break at this point before Part 2, and get some hands-on practice using the concepts we’ve covered. If you are further down the XAML path, you may be ready to go straight to Part 2 – Going Deeper.