There are many projects in Head First C# where you build Windows Store apps that require Windows 8. In this appendix, you'll use WPF to build them as desktop apps instead.

appendix ii: Windows Presentation Foundation

WPF Learner's Guide to Head First C#

Not running Windows 8? Not a problem.

We wrote many chapters in the third edition of *Head First C#* using the latest technology available from Microsoft, which **requires Windows 8 and Visual Studio 2013**. But what if you're using this book at work, and you can't install the latest version? That's where **Windows Presentation Foundation** (or **WPF**) comes in. It's an older technology, so it works with Visual Studio 2010 and 2008 running on Windows editions as mature as 2003. But it's also a core C# technology, so even if you're running Windows 8 it's a good idea to **get some experience with WPF**. In this appendix, we'll **guide you through building** *most* of the Windows Store projects in the book using WPF.

Why you should learn WPF

Windows Presentation Foundation, or WPF, is a technology that's used to build user interfaces for programs written in .NET. WPF programs typically run on the Windows desktop and display their user interfaces in windows. WPF is one of the most popular technologies for developing Windows software, and familiarity with WPF is considered by many employers to be a required skill for professional C# and .NET developers.

WPF programs use XAML (Extensible Application Markup Language) to lay out their UIs. This is great news for *Head First C#* readers who have been reading about Windows Store apps. Most of the Windows Store projects in the book can be built for WPF **with few or no modifications to the XAML code**.

Some things, like app bars and page navigation, are specific to Windows Store apps. In this appendix, we show you WPF alternatives wherever possible.

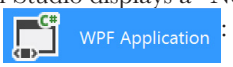
Every C# developer should work with WPF.

Almost every programming language can be used in lots of different environments and operating systems, and C# is no exception. If your goal is to improve as a C# developer, you should go out of your way to work with as many different technologies as possible. And WPF in particular is especially important for C# developers, because there are many programs that use WPF in companies, and this will continue for a long time. If your goal is to use C# in a professional environment, WPF is technology you'll want to list on your resumé.

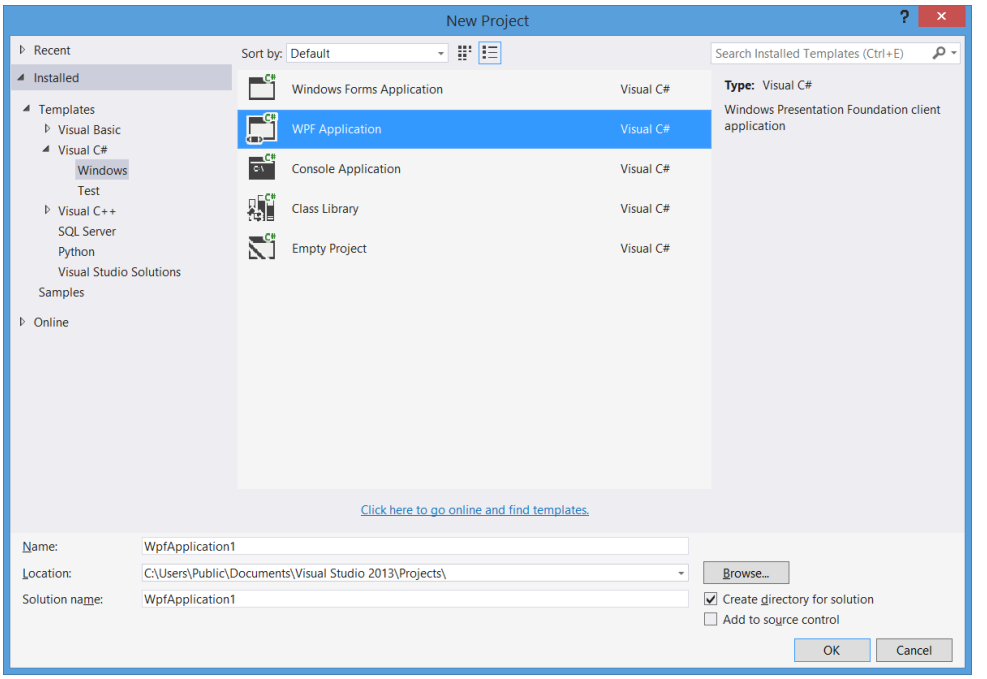
Learning WPF is also great for a hobby programmer who's using Windows 8 and can build all of the code in *Head First C#*. One of the most effective learning tools you have as a developer is seeing **the same problem solved in different ways**. This appendix will guide you through building many of the projects in *Head First C#* using WPF. Seeing those projects built in WPF and Windows 8 will give you valuable perspective, and that's one of the things that **helps turn good programmers into great developers**.

You can download the code for all of the projects in this appendix. Go to the Head First Labs website for more information: http://www.headfirstlabs.com/hfcsharp

Build WPF projects in Visual Studio

Creating a new WPF application in Visual Studio works just like creating other kinds of desktop applications. If you're using Visual Studio Express 2013, make sure you're using Visual Studio 2013 Express for *Desktop* (the edition for Windows 8 will not create WPF projects). You can also create programs using Visual Studio 2013 Professional, Premium, or Ultimate. When you create a new project, Visual Studio displays a "New Project" dialog. Make sure you select **Visual C#**, and then choose 

:



You can also create C# WPF applications using all editions of Visual Studio 2010, Visual C# 2010 Express, and Visual Studio 2008. Note that if you use the Express editions of Visual Studio 2010 or 2008, the project files are initially created in a temporary folder and are not saved to the location specified in the New Project dialog until you use Save or Save All to save your files.

WPF can also be used to build XAML browser applications that run inside Internet Explorer and other browsers. We won't be covering it in this appendix, but you can learn more about it here: http://msdn.microsoft.com/en-us/library/aa970060.aspx

Microsoft has yet another technology that also uses XAML. It's called Silverlight, and you can read about it here: http://www.microsoft.com/silverlight/

Did you find an error in this appendix? Please submit it using the Errata page for Head First C# (3rd edition) so we can fix it as quickly as possible!

http://www.oreilly.com/catalog/errata.csp?isbn=0636920027812

How to use this appendix

This appendix contains complete replacements for pages in *Head First C# (3rd edition)*. We've divided this appendix up into individual guides for each chapter, starting with an overview page that has specific instructions for how to work through that chapter: what pages to replace in the chapter, what to read in it, and any specific instructions to help you get the best learning experience.

If you're using an old version of Visual Studio, you'll be able to do these projects... but things will be a little harder for you.

The team at Microsoft did a really good job of improving the user interface of Visual Studio 2013, especially when it comes to editing XAML. One important feature of *Head First C#* is its use of the Visual Studio IDE as a tool for teaching, learning, and exploration. This is why we strongly recommend that you use the latest version of Visual Studio if possible.

However, we do understand that some readers cannot install Visual Studio 2013. (For example, a lot of our readers are using a computer provided by an employer, and do not have administrative privileges to install new software.) We still want you to be able to use our book, even if you're stuck using an old version of Visual Studio! We'll do our best to give you as much guidance as we can. But we also need to strike a balance here, because we're being careful not to compromise the learning for the majority of our readers who are using the latest version of Visual Studio.

If you're using Visual Studio 2010 or earlier, and you find yourself stuck because the IDE's user interface doesn't look right or menu options aren't where you expect them to be, **we recommend that you enter the XAML and C# code by hand**—or even better, copy it and paste it into Visual Studio. Once the XAML is correct, it's often easier to track down the feature in the IDE that generated it.

We've made all of the source code in the book available for download, and we encourage you to copy and paste it into your programs anytime you get stuck. Go to the book's website(http://www.headfirstlabs.com/hfcsharp) for more details and links to the source code.

You can download the source code directly from http://hfcsharp.codeplex.com/ — but for the replacement chapters in this appendix, make sure that you sure you download the code from the WPF folder. If you try to use the Windows Store code in a WPF project, you'll get frustrating errors.

One more thing. This appendix has replacements for pages that you'll find in the printed or PDF version this book, and you can find those pages using their page numbers. However, if you're using a Kindle or another eBook reader, you might not be able to use the page numbers. Instead, just use the section heading to look up the section to replace. For example, this appendix has replacements for pages 72 and 73 section called *Build an app from the ground up*, which you can find in your eBook reader's Table of Contents underneath Chapter 2. (Exercises like the one on page 83 and the solution on page 85 might not show up in your reader's Table of Contents, but you'll get to the exercises as you go through each chapter.) *This will be much easier for you if you download the PDF of this appendix from the book's website.*

Chapter 1

Build a game, and get a feel for the IDE.

The first project in the book walks you through building a complete—and fun!—video game. The goal of the project is to help you get used to creating user interfaces and writing C# code using the Visual Studio IDE.

We recommend that you read through page 11 in the main part of the book, and then flip to the next page in this appendix. We designed pages 12–47 in this appendix so that they can be 100% replacements for the corresponding pages in the book. Once you've finished building the WPF version of *Save the Humans*, you can go on to Chapter 2 in the book.

The screenshots in this chapter are from Visual Studio 2013 for Windows Desktop, the latest version of Visual Studio available at this time. If you're using Visual Studio 2010, some of the menu options and windows in the IDE will be different. We'll give you guidance to help you find the right menu options.

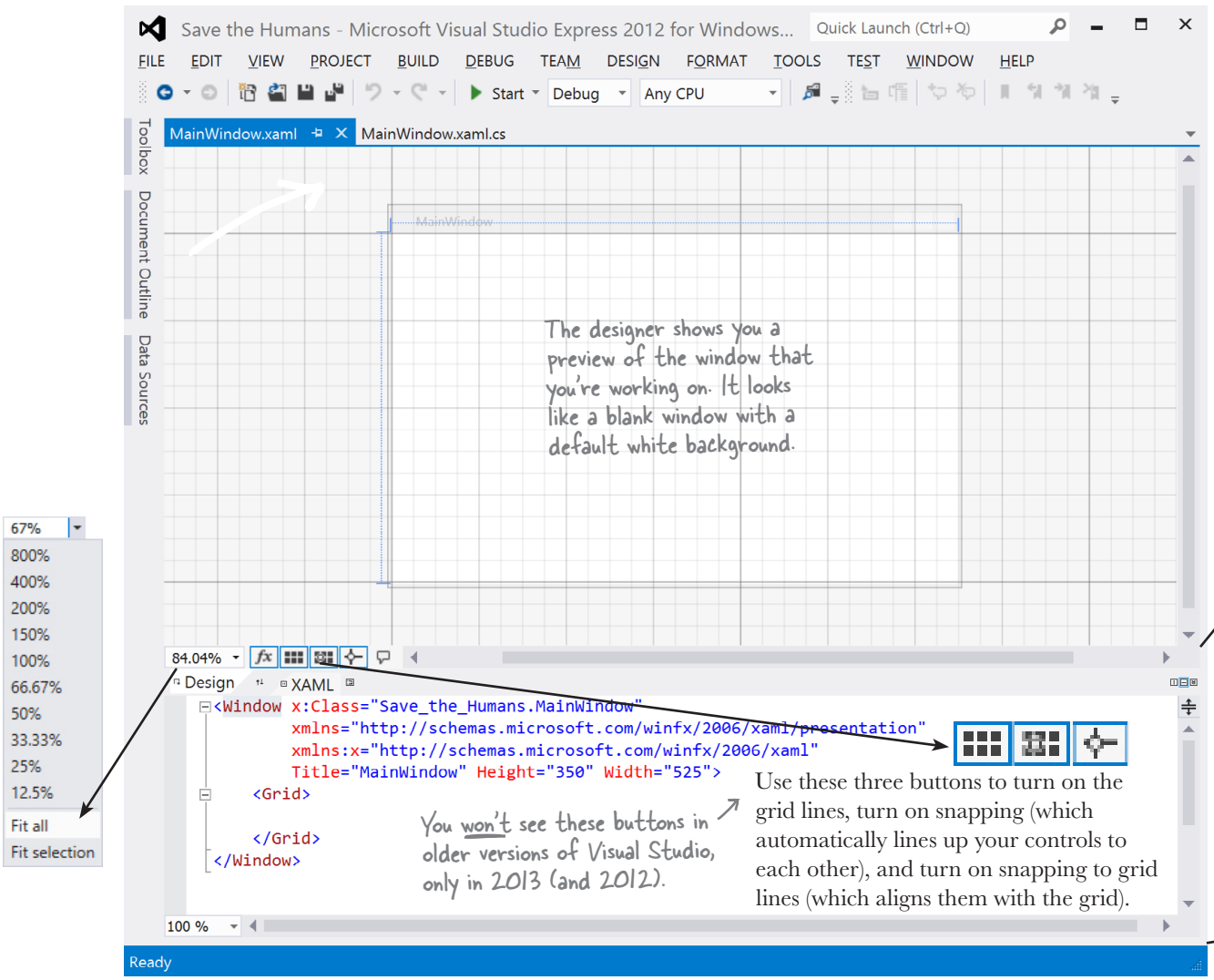
We worked really hard to keep the page flipping to a minimum, because by reducing distractions we make it easier for you to learn important C# concepts. After you read the first 11 pages of Chapter 1, you won't have to flip back to the main part of the book at all for the rest of the chapter. Then there are just five pages that you need in this appendix for Chapter 2. After that, the book concentrates on building desktop applications, which you can build with any version of Windows. You won't need this appendix again until you get to Chapter 10.

Start with a blank application

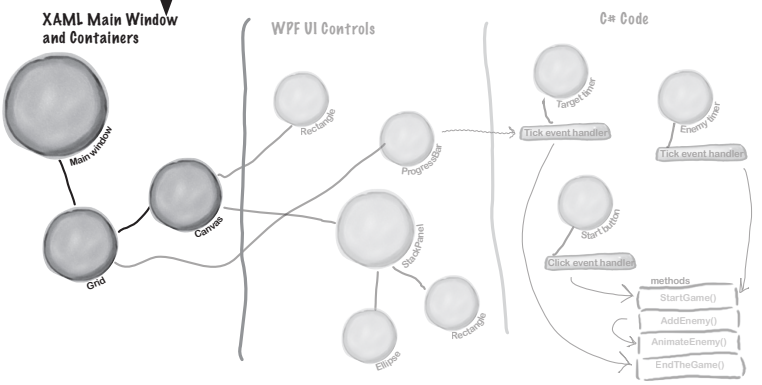
If your code filenames don't end in ".cs" you may have accidentally created a JavaScript, Visual Basic, or Visual C++ program. You can fix this by closing the solution and starting over. If you want to keep the project name "Save the Humans," then you'll need to delete the previous project folder.

Every great app starts with a new project. Choose New Project from the File menu. Make sure you have Visual C#→Windows selected and choose **WPF Application** as the project type. Type "Save the Humans" as the project name.

① Your starting point is the **Designer window**. Double-click on *MainWindow.xaml* in the Solution Explorer to bring it up (if it's not already displayed). Find the zoom drop-down in the lower-left corner of the designer and choose "Fit all" to zoom it out.



You are here!



The bottom half of the Designer window shows you the XAML code. It turns out your "blank" window isn't blank at all—it contains a **XAML grid**. The grid works a lot like a table in an HTML page or Word document. We'll use it to lay out our windows in a way that lets them grow or shrink to different screen sizes and shapes.

You can see the XAML code for the blank window that the IDE generated for you. Keep your eyes on it—we'll add some columns and rows in a minute.

```
84.04%
Design   XAML
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
100 %
```

These are the opening and closing tags for a grid that contains controls. When you add rows, columns, and controls to the grid, the code for them will go between these opening and closing tags.

This part of the project has steps numbered ① to ③.

Flip the page to keep going!

THIS PROJECT CLOSELY FOLLOWS CHAPTER 1.

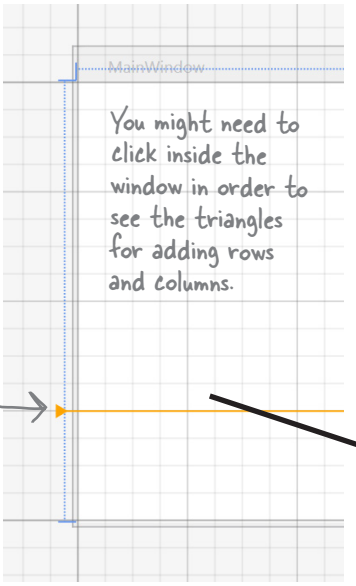
We want to give you a solid learning foundation, so we've designed this project so that it can replace pages 12-48 of *Head First C#*. Other projects in this appendix will give you all the information that you need to adapt the material in the book. So even when we don't give you one-to-one page replacements, we'll make sure you get all the information you need to do the projects.

Over the next few pages you'll explore a lot of different features in the Visual Studio IDE, because we'll be using the IDE as a powerful tool for learning and teaching. You'll use the IDE throughout the book to explore C#. That's a really effective way to get it into your brain!

② Your app will be a grid with two rows and three columns, with one big cell in the middle that will contain the play area. Start defining rows by hovering over the border of the window until a line and triangle appear:

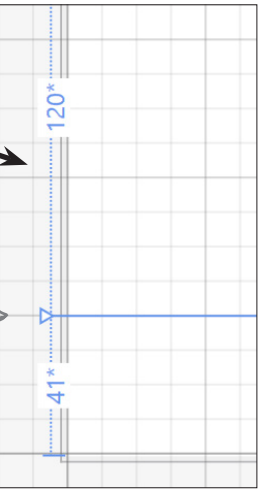
Hover over the border of the window until an orange triangle and line appear...

...then click to create a bottom row in the grid.



WPF apps often need to adapt to different window sizes displayed at different screen resolutions.

After the row is added, the line will change to blue and you'll see the heights of both rows in the border. The height of each row will be a number followed by a star. Don't worry about the numbers for now.



Laying out the window using a grid's columns and rows allows your program to automatically adjust to the window size.

there are no Dumb Questions

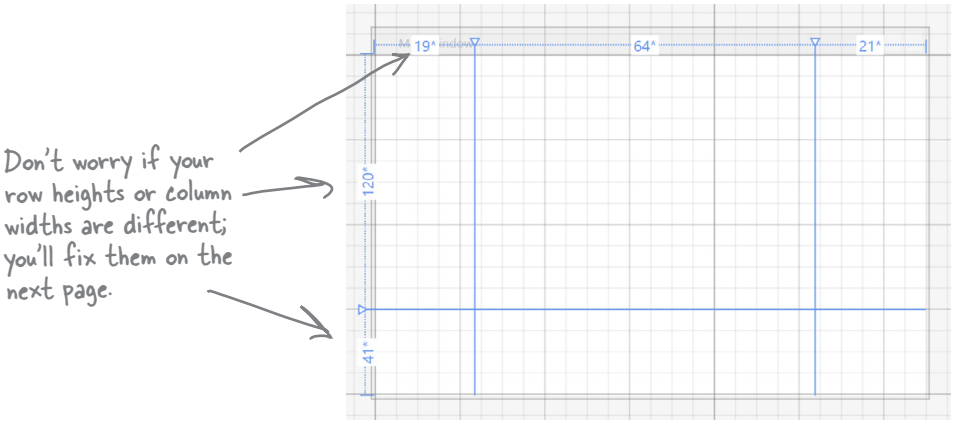
Q: But it looks like I already have many rows and columns in the grid. What are those gray lines?

A: The gray lines are just Visual Studio giving you a grid of guidelines to help you lay your controls out evenly in the window. You can turn them on and off with the button. None of the lines you see in the designer show up when you run the app outside of Visual Studio. But when you clicked and created a new row, you actually altered the XAML, which will change the way the app behaves when it's compiled and executed.

Q: Wait a minute. I wanted to learn about C#. Why am I spending all this time learning about XAML?

A: Because WPF apps built in C# almost always start with a user interface that's designed in XAML. That's also why Visual Studio has such a good XAML editor—to give you the tools you need to build stunning user interfaces. Throughout the book, you'll learn how to build other types of programs with C#: Windows Store apps, which use XAML, and desktop applications and console applications, which don't. Seeing all of these different technologies will give you a deeper understanding of programming with C#.

③ Do the same thing along the top border of the window—except this time create two columns, a small one on the left-hand side and another small one on the right-hand side. Don't worry about the row heights or column widths—they'll vary depending on where you click. We'll fix them in a minute.



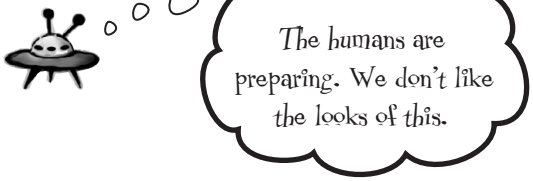
When you're done, look in the XAML window and go back to the same grid from the previous page. Now the column widths and row heights match the numbers on the top and side of your window.

```
84.04%
Design  XAML
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="19*"/>
            <ColumnDefinition Width="64*"/>
            <ColumnDefinition Width="21*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="120*"/>
            <RowDefinition Height="41*"/>
        </Grid.RowDefinitions>

    </Grid>
</Window>
100 %
```

Here's the width of the left column you created in step 3—the width matches the width that you saw in the designer. That's because the IDE generated this XAML code for you.

Your grid rows and columns are now added!

XAML grids are **container controls**, which means they hold other controls. Grids consist of rows and columns that define cells, and each cell can hold other XAML controls that show buttons, text, and shapes. A grid is a great way to lay out a window, because you can set its rows and columns to resize themselves based on the size of the screen.



Set up the grid for your window

Your program needs to be able to work on different sized windows, and using a grid is a great way to do that. You can set the rows and columns of a grid to a specific pixel height. But you can also use the **Star** setting, which keeps them the same size proportionally—to one another and also to the window—no matter how big the window or resolution of the display.

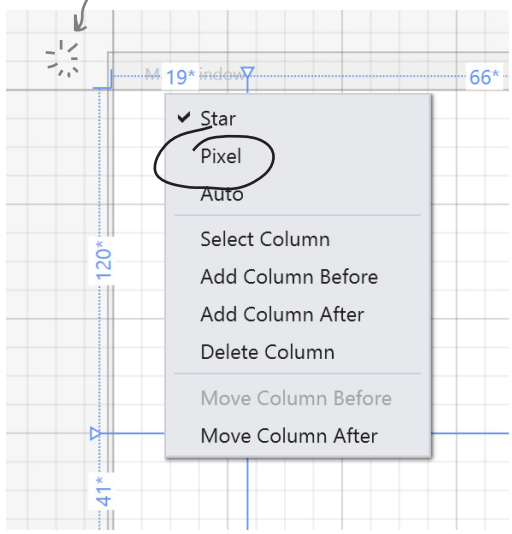
1. **SET THE WIDTH OF THE *LEFT* COLUMN.**
   Hover over the number above the *leftmost* column until a drop-down menu appears. Choose Pixel to change the star to a lock, and then click on the number to change it to `140`. Your column's number should now look like this:

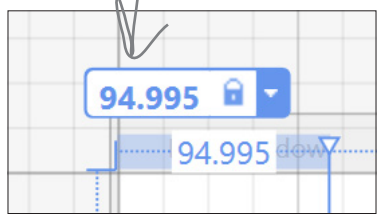
2. **REPEAT FOR THE RIGHT COLUMN AND THE BOTTOM ROW.**
   Make the right column 160 pixels and the bottom row 150 by choosing Pixel and typing `160` or `150` into the box.

Set your columns or rows to Pixel to give them a fixed width or height. The Star setting lets a row or column grow or shrink proportionally to the rest of the grid. Use this setting in the designer to alter the Width or Height property in the XAML. If you remove the Width or Height property, it's the same as setting the property to 1*.

If you don't see the numbers like 120* and 19* along the border of your window, click outside the window in the designer.



When you switch the column to pixels, the number changes from a proportional width to the actual pixel width.



It's OK if you're not a pro at app design...*yet*.

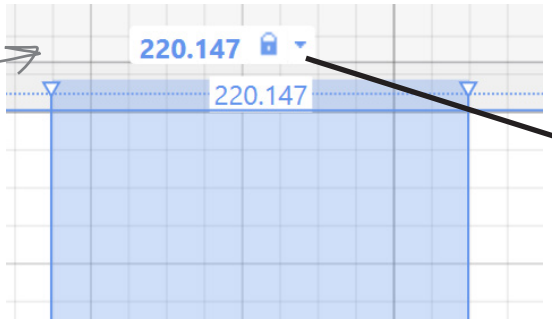
We'll talk a lot more about what goes into designing a good app later on. For now, we'll walk you through building this game. By the end of the book, you'll understand exactly what all of these things do!

MAKE THE CENTER COLUMN THE DEFAULT SIZE.

3 Make sure that the center column width is set to 1*. If it isn't, click on the number above the center column and enter 1. Don't use the drop-down (leave it star) so it looks like the picture below. Then make sure to look back at the other columns to make sure the IDE didn't resize them. If it did, just change them back to the widths you set in steps 1 and 2.

XAML and C# are case sensitive! Make sure your uppercase and lowercase letters match example code.

If you accidentally changed the center column's width to Pixels, you can change it back to 1*.



When you enter 1* into the box, the IDE sets the column to its default width. It might adjust the other columns. If it does, just reset them back to 160 pixels.

LOOK AT YOUR XAML CODE!

4 Click on the grid to make sure it's selected, then look in the XAML window to see the code that you built.

```
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="140"/>
            <ColumnDefinition/>
            <ColumnDefinition Width="160"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition Height="150"/>
        </Grid.RowDefinitions>
    </Grid>
</Window>
```

The <Grid> line at the top means everything that comes after it is part of the grid.

This is how a column is defined for a XAML grid. You added three columns and two rows, so there are three ColumnDefinition tags and two RowDefinition tags.

You used the designer to set the height of the bottom row to 150 pixels.

In a minute, you'll be adding controls to your grid, which will show up here, after the row and column definitions.

You used the column and row drop-downs to set the Width and Height properties.

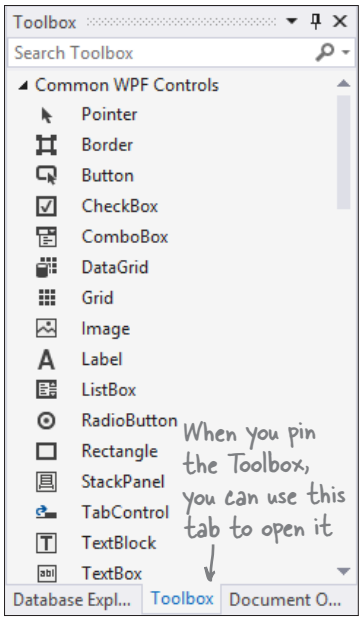
If you're using Visual Studio 2010, the IDE looks different. When you hover over a column size, you'll see this box to select pixel or star: 

It's possible to edit the column sizes in the designer using the older versions of the IDE, but it's not nearly as easy to do. We recommend that if you're using an older version of the IDE, you create the columns and rows, and then edit the XAML row and column definitions by hand.

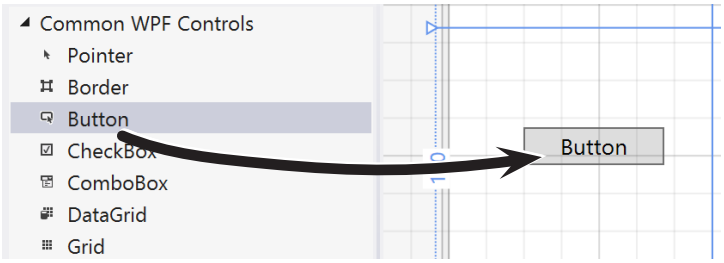
Add controls to your grid

If you don't see the toolbox in the IDE, you can open it using the View menu. Use the pushpin to keep it from collapsing.

Ever notice how programs are full of buttons, text, pictures, progress bars, sliders, drop-downs, and menus? Those are called **controls**, and it's time to add some of them to your app—*inside* the cells defined by your grid's rows and columns.



1 Expand the **Common WPF Controls** section of the toolbox and drag a Button into the **bottom-left cell** of the grid.



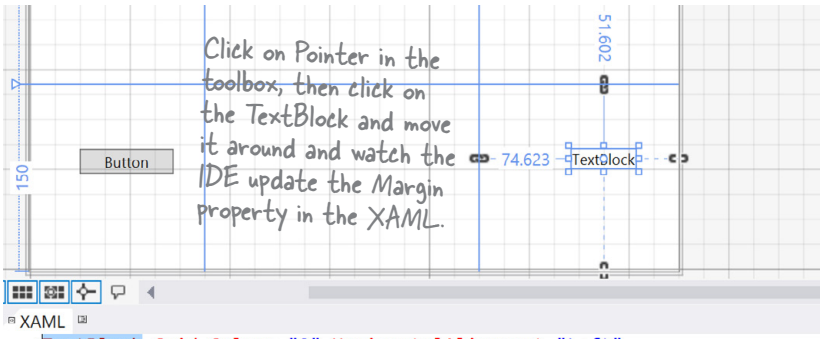
Then look at the bottom of the Designer window and have a look at the **XAML tag** that the IDE generated for you. You'll see something like this—your margin numbers will be different depending on where in the cell you dragged it, and the properties might be in a different order.

The XAML for the button starts here, with the opening tag.

These are properties. Each property has a name, followed by an equals sign, followed by its value.

```
<Button Content="Button" HorizontalAlignment="Left"
        Margin="40,52,0,0" Grid.Row="1" VerticalAlignment="Top" Width="75" />
```

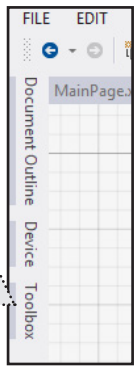
2 Drag a TextBlock into the **lower-right cell** of the grid. Your XAML will look something like this. See if you can figure out how it determines which row and column the controls are placed in.



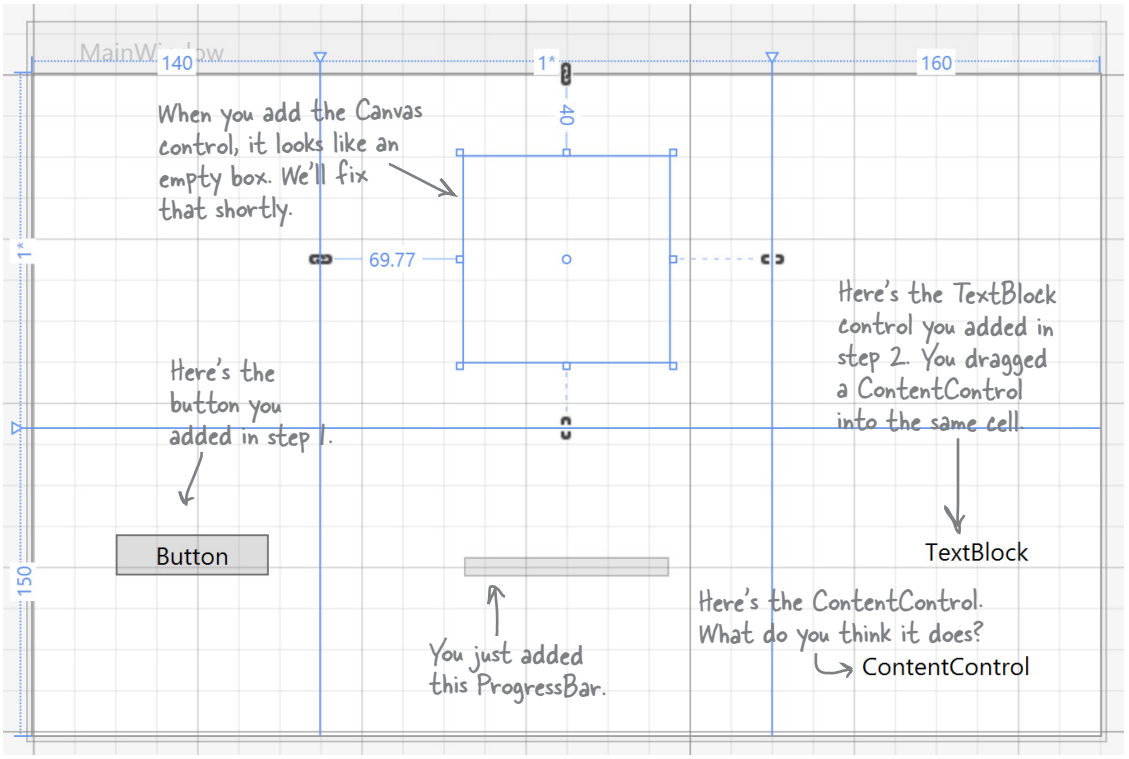

```
<TextBlock Grid.Column="2" HorizontalAlignment="Left"
           Margin="74.623,51.602,0,0" Grid.Row="1" TextWrapping="Wrap"
           Text="TextBlock" VerticalAlignment="Top"/>
```

We added line breaks to make the XAML easier to read. You can add line breaks, too. Give it a try!

If you don't see the toolbox, try clicking on the word "Toolbox" that shows up in the upper-left corner of the IDE. If it's not there, select Toolbox from the View menu to make it appear.



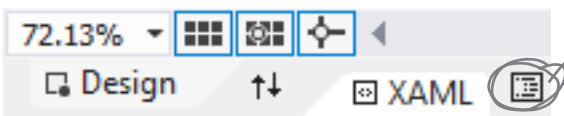
3 Next, expand the **All WPF Controls** section of the toolbox. Drag a ProgressBar into the bottom-center cell, a ContentControl into the bottom-right cell (make sure it's **below** the TextBlock you already put in that cell), and a Canvas into the top center cell. Your window should now have controls on it (don't worry if they're placed differently than the picture below; we'll fix that in a minute):



4 You've got the Canvas control currently selected, since you just added it. (If not, use the pointer to select it again.) Look in the XAML window:

```
<Canvas Grid.Column="1" HorizontalAlignment="Left" Height="100"...
```

It's showing you the XAML tag for the Canvas control. It starts with `<Canvas` and ends with `/>`, and between them it has properties like `Grid.Column="1"` (to put the Canvas in the center column) and `Height="100"` (to set its height in pixels). Try clicking *in both the grid and the XAML window* to select different controls.



Try clicking this button. It brings up the Document Outline window. Can you figure out how to use it? You'll learn more about it in a few pages.

When you drag a control out of the toolbox and onto your window, the IDE automatically generates XAML to put it where you dragged it.

Use properties to change how the controls look

The Visual Studio IDE gives you fine control over your controls. The **Properties window** in the IDE lets you change the look and even the behavior of the controls on your window.

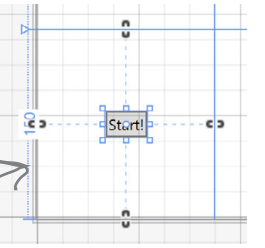
When you're editing text, use the Escape key to finish. This works for other things in the IDE, too.

1 Change the text of the button.

Right-click on the button control that you dragged onto the grid and choose **Edit Text** from the menu. Change the text to: `Start!` and see what you did to the button's XAML:

```
<Button Content="Start!" HorizontalAlignment="Left" VerticalAlignment="Top" . . .
```

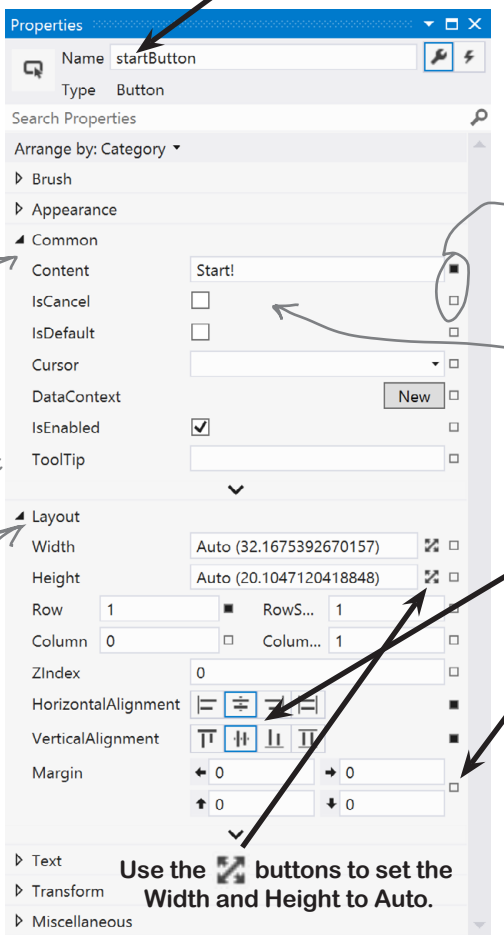
When you edit the text in the button, the IDE updates the Content property in the XAML.



2 Use the Properties window to modify the button.

Make sure the button is selected in the IDE, and then look at the Properties window in the lower-right corner of the IDE. Use it to change the name of the control to `startButton` and center the control in the cell. Once you've got the button looking right, **right-click on it and choose View Source** to jump straight to the `<Button>` tag in the XAML window.

Use the Name box to change the name of the control to startButton.



You might need to expand the Common and Layout sections.

These little squares tell you if the property has been set. A filled square means it's been set; an empty square means it's been left with a default value.

When you used "Edit Text" on the right-click menu to change the button's text, the IDE updated the Content property.

Use the ⫟ and ⫠ buttons to set the HorizontalAlignment and VerticalAlignment properties to "Center" and center the button in the cell.

Older versions of the IDE use the word "Center" instead of icons like this.

When you dragged the button onto the window, the IDE used the Margin property to place it in an exact position in the cell. Click on the square ■ and choose Reset from the menu to reset the margins to 0.

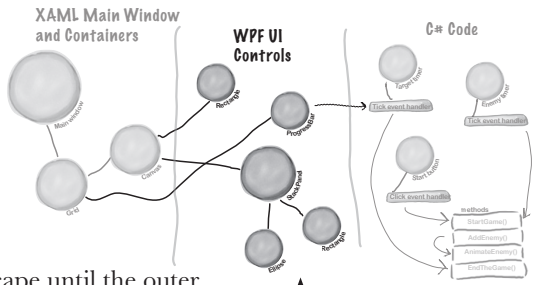
Use the ⤢ buttons to set the Width and Height to Auto.

```
<Button x:Name="startButton"
        Content="Start!"
        Grid.Row="1"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"/>
```

Go back to the XAML window in the IDE and have a look at the XAML that you updated!

The properties may be in a different order. That's OK!

You can use Edit→Undo (or Ctrl-Z) to undo the last change. Do it several times to undo the last few changes. If you selected the wrong thing, you can choose Select None from the Edit menu to deselect. You can also hit Escape to deselect the control. If it's living inside a container like a StackPanel or Grid, hitting Escape will select the container, so you may need to hit it a few times.



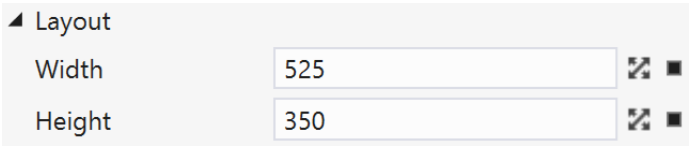
You are here!

3 Change the size and title of the window.

Select any of the controls. Then hit Escape, and keep hitting Escape until the outer <Window> tag is displayed in the XAML editor:

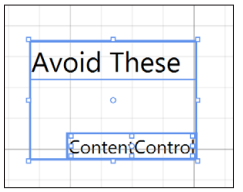
```
<Window x:Class="Save_the_Humans.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

Click <Window in the XAML editor. The <Window> tag has properties for Height and Width. Look for their corresponding values in the Properties window in the IDE:



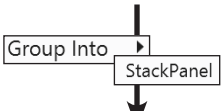
Set the width to 1000 and height to 700, and the window immediately resizes itself to the new size. You can use the "Fit all" option in the Zoom drop-down to show the whole window in the designer. Notice how the center column and top row resized themselves to fit the new window, while the other rows and columns kept their pixel sizes. Then expand the Common section in the Properties window and **set the Title property to `Save the Humans`**. You'll see the window title get updated.

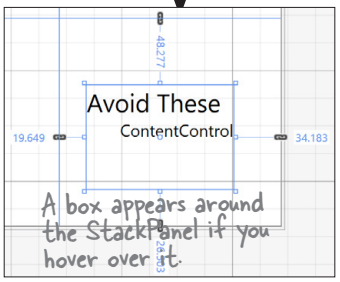
Your TextBlock and ContentControl are in the lower-right cell of the grid.



4 Update the TextBlock to change its text and its font size.

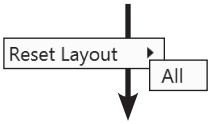
Use the Edit Text right-mouse menu option to change the TextBlock so it says `Avoid These` (hit Escape to finish editing the text). Then expand the Text section of the Properties window and change the font size to `18 px`. This may cause the text to wrap and expand to two lines. If it does, drag the TextBlock to make it wider.

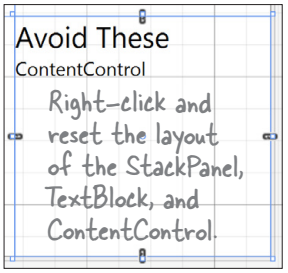




5 Use a StackPanel to group the TextBlock and ContentControl.

Make sure that the TextBlock is near the top of the cell, and the ContentControl is near the bottom. **Click and drag to select both the TextBlock and ContentControl, and then right-click**. Choose Group Into from the pop-up menu, then choose StackPanel. This adds a new control to your form: a **StackPanel control**. You can select the StackPanel by clicking between the two controls.





The StackPanel is a lot like the Grid and Canvas: its job is to hold other controls (it's called a "container"), so it's not visible on the form. But since you dragged the TextBlock to the top of the cell and the ContentControl to the bottom, the IDE created the StackPanel so it fills up most of the cell. Click in the middle of the StackPanel to select it, then right-click and choose Layout and Reset All to quickly reset its properties, which will set its vertical and horizontal alignment to Stretch. Right-click on the TextBox and ContentControl to reset their properties as well. While you have the ContentControl selected, set its vertical and horizontal alignments to Center.

Controls make the game work

Controls aren't just for decorative touches like titles and captions. They're central to the way your game works. Let's add the controls that players will interact with when they play your game. Here's what you'll build next:

The user interface for editing colors in earlier versions of Visual Studio is not as advanced, but you should still be able to set the colors so they look correct. The Document Outline window is also a little more primitive, but it still works. However, **there is *not* an easy way to visually create a template in Visual Studio 2010.** The easiest way to do this in the old version of the IDE is to copy the entire `<Window.Resources>` section (up through the closing `</Window.Resources>` tag) from the downloadable source code and paste it into your XAML just above the opening `<Grid>` tag. **Make sure you download the code from the WPF folder!** Then you can select the ContentControl and use the Properties window to set the Template property to EnemyTemplate. Your enemies will already look like evil aliens, so make sure you still read pages 44 and 45.

You'll create a play area with a gradient background...

...and you'll work on the bottom row.

You'll make the ProgressBar as wide as its column...

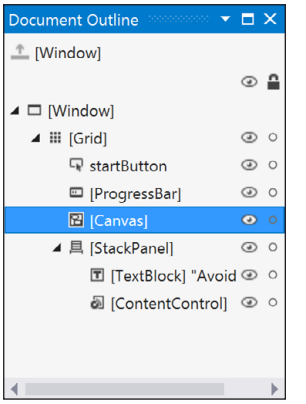
...and you'll use a template to make your enemy look like this.

1 Update the ProgressBar.

Right-click on the ProgressBar in the bottom-center cell of the grid, choose the **Layout** menu option, and then choose **Reset All** to reset all the properties to their default values. Use the Height box in the Layout section of the Properties window to set the `Height` to **20**. The IDE stripped all of the properties from the XAML, and then added the new `Height`:

```
<ProgressBar Grid.Column="1" Grid.Row="2" Height="20"/>
```

You can also get to the Document Outline by choosing the View→Other Windows menu.



Document Outline

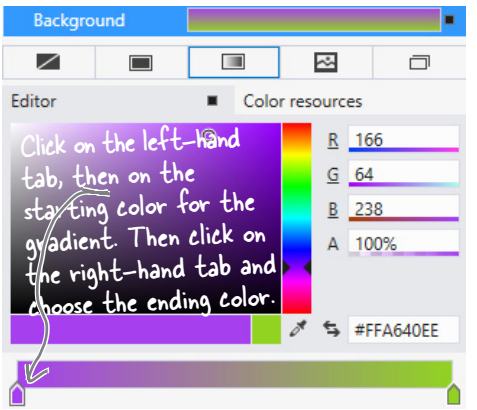
You can also open the Document Outline by clicking the tab on the side of the IDE.

2 Turn the Canvas control into the gameplay area.

Remember that Canvas control that you dragged into the center square? It's hard to see it right now because a Canvas control is invisible when you first drag it out of the toolbox, but there's an easy way to find it. Click the very small button above the XAML window to bring up the **Document Outline**. Click on [Canvas] to select the Canvas control.

Make sure the Canvas control is selected, then **use the Name box** in the Properties window to set the name to `playArea`.

Once you change the name, it'll show up as playArea instead of [Canvas] in the Document Outline window.



After you've named the Canvas control, you can close the Document Outline window. Then use the and buttons in the Properties window to set its vertical and horizontal alignments to Stretch, reset the margins, and click both buttons to set the `Width` and `Height` to Auto. Then set its Column to 0, and its ColumnSpan (next to Column) to 3.

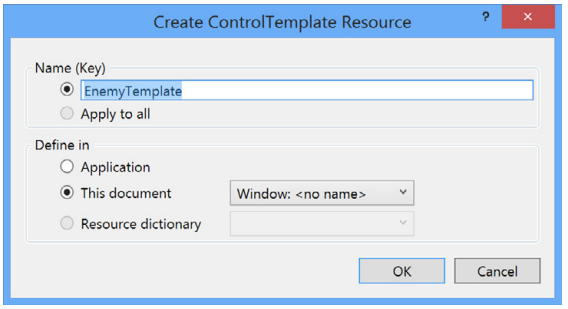
Finally, open the **Brush** section of the Properties window and use the button to give it a **gradient**. Choose the starting and ending colors for the gradient by clicking each of the tabs at the bottom of the color editor and then clicking a color.

3 Create the enemy template.

Your game will have a lot of enemies bouncing around the screen, and you're going to want them all to look the same. Luckily, XAML gives us **templates**, which are an easy way to make a bunch of controls look alike.

Next, right-click on the ContentControl in the Document Outline window. Choose **Edit Template**, then choose **Create Empty...** from the menu. Name it `EnemyTemplate`. The IDE will add the template to the XAML.

You're "flying blind" for this next bit—the designer won't display anything for the template until you add a control and set its height and width so it shows up. Don't worry; you can always undo and try again if something goes wrong.



You can also use the Document Outline window to select the grid if it gets deselected.

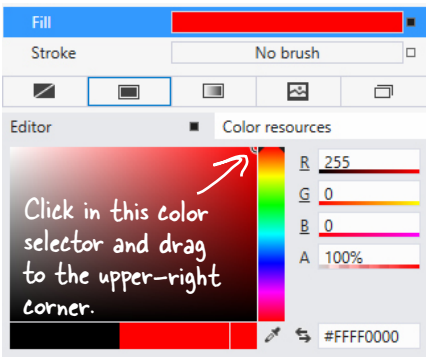
Your newly created template is currently selected in the IDE. Collapse the Document Outline window so it doesn't overlap the Toolbox. *Your template is* ***still invisible****, but you'll change that in the next step. If you accidentally click out of the control template,* ***you can always get back to it*** *by opening the Document Outline, right-clicking on the Content Control, and choosing Edit Template→Edit Current.*

4 Edit the enemy template.

Add a red circle to the template:

Make sure you don't click anywhere else in the designer until you see the ellipse. That will keep the template selected.

- Double-click on Ellipse in the Toolbox to add an ellipse.
- Set the ellipse's `Height` and `Width` properties to **`100`**, which will cause the ellipse to be displayed in the cell.
- Reset the `Margin`, `HorizontalAlignment`, and `VerticalAlignment` properties by clicking their squares and choosing Reset.
- Go to the Brush section of the Properties window and click on ▬ to select a solid-color brush.
- Color your ellipse red by clicking in the color selector and dragging to the upper-right corner.



The XAML for your ContentControl now looks like this:

```
<ContentControl Content="ContentControl" Template="{DynamicResource EnemyTemplate}"
                VerticalAlignment="Center" HorizontalAlignment="Center"/>
```

Scroll around your window's XAML window and see if you can find where EnemyTemplate is defined. It should be right below the AppName resource.

5 Use the Document Outline to modify the StackPanel, TextBlock, and Grid controls.

Go back to the Document Outline (if you see EnemyTemplate (ContentControl Template) at the top of the Document Outline window, just click ⬆ to get back to the Window outline). Select the StackPanel control, make sure its vertical and horizontal alignments are set to center, and clear the margins. Then do the same for the TextBlock, and use the Properties window to set the **Foreground** property to **white** using the color selector.

Click here and use the color selector to make the TextBlock white.

Finally, select the Grid, then open the Brush section of properties and click ▬ to give it a black Background.

You're almost done laying out the form! Flip the page for the last steps...

6 Add the human to the Canvas.

You've got two options for adding the human. The first option is to follow the next three paragraphs. The second, quicker option is to just type the four lines of XAML into the IDE. It's your choice!

Select the Canvas control, and then open the All XAML Controls section of the toolbox and double-click on Ellipse to add an Ellipse control to the Canvas. Select the Canvas control again and double-click on Rectangle. The Rectangle will be added right on top of the Ellipse, so drag the Rectangle below it.

Hold down the Shift key and click on the Ellipse so both controls are selected. Right-click on the Ellipse, choose **Group Into**, and then **StackPanel**. Select the Ellipse, use the solid brush property to change its color to white, and set its `Width` and `Height` properties to 10. Then select the Rectangle, make it white as well, and change its `Width` to 10 and its `Height` to 25.

If you used the designer to create your human, make sure its source matches this XAML.

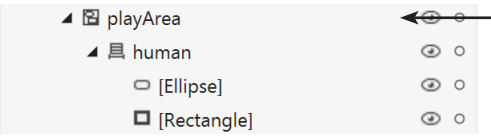
Use the Document Outline window to select the Stack Panel (make sure you see Type StackPanel at the top of the Properties window). Reset its margins, **then click both ⛶ buttons to set the `Width` and `Height` to Auto**. Then use the Name box at the top of the window to set its name to `human`. Here's the XAML you generated:

```
<StackPanel x:Name="human" Orientation="Vertical">
    <Ellipse Fill="White" Height="10" Width="10"/>
    <Rectangle Fill="White" Height="25" Width="10"/>
</StackPanel>
```

If you choose to type this into the XAML window of the IDE, make sure you do it directly above the </Canvas> tag. That's how you indicate that the human is contained in the Canvas.

You might also see a Stroke property on the Ellipse and Rectangle set to `"Black"`. (If you don't see one, try adding it. What happens?)

Go back to the Document Outline window to see how your new controls appear:



You gave the Canvas control the name `playArea` in step 2, so it shows up in the Document Outline window. Try hovering over the controls in it.

If `human` isn't indented underneath `playArea`, click and drag `human` onto it.

7 Add the Game Over text.

When your player's game is over, the game will need to display a Game Over message. You'll do it by adding a TextBlock, setting its font, and giving it a name:

- Select the Canvas, and then drag a TextBlock out of the toolbox and onto it.
- Use the Name box in the Properties window to change its name to `gameOverText`.
- Use the Text section of the Properties window to change the font to Arial, change the size to 100 px, and make it Bold and Italic.
- Click on the TextBlock and drag it to the middle of the Canvas.
- Edit the text so it says `Game Over`.

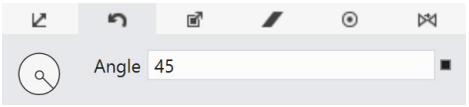
When you drag a control around a Canvas, its Left and Top properties are changed to set its position. If you change the Left and Top properties, you move the control.

8 **Add the target portal that the player will drag the human onto.**

There's one last control to add to the Canvas: the target portal that your player will drag the human into. (It doesn't matter where in the Canvas you drag it.)

Select the Canvas control, and then drag a Rectangle control onto it. Use the button in the Brushes section of the Properties window to give it a gradient. Set its `Height` and `Width` properties to **50**.

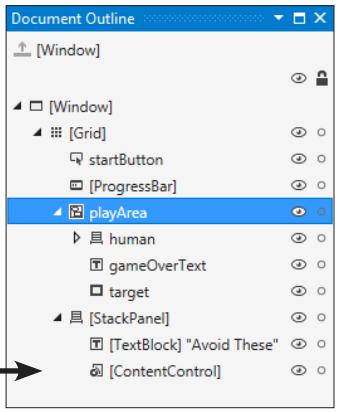
Turn your rectangle into a diamond by rotating it 45 degrees. Open the Transform section of the Properties window to rotate the Rectangle 45 degrees by clicking and setting the angle to **45**.



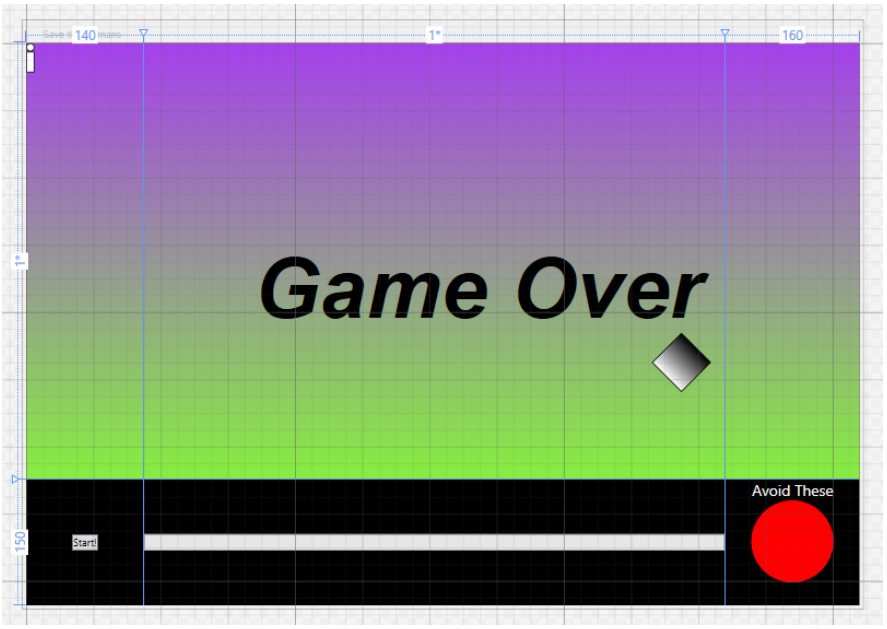
Finally, use the Name box in the Properties window to give it the name `target`.

9 **Take a minute and double-check a few things.**

Open the Document Outline window and make sure that the `human` StackPanel, `gameOverText` TextBlock, and `target` Rectangle are indented underneath the `playArea` Canvas control, which is indented under the second [Grid]. Select the `playArea` Canvas control and make sure its Height and Width are set to Auto. These are all things that **could cause bugs** in your game that will be difficult to track down. Your Document Outline window should look like this:



Congratulations—you've finished building the window for your app!



We collapsed human to make it obvious that it's indented underneath playArea, along with gameOverText and target. It's okay if the controls are in a different order (you can even drag them up an down!), as long as the indenting is correct—that's how you know which controls are inside other container controls.

WHO DOES WHAT?

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.

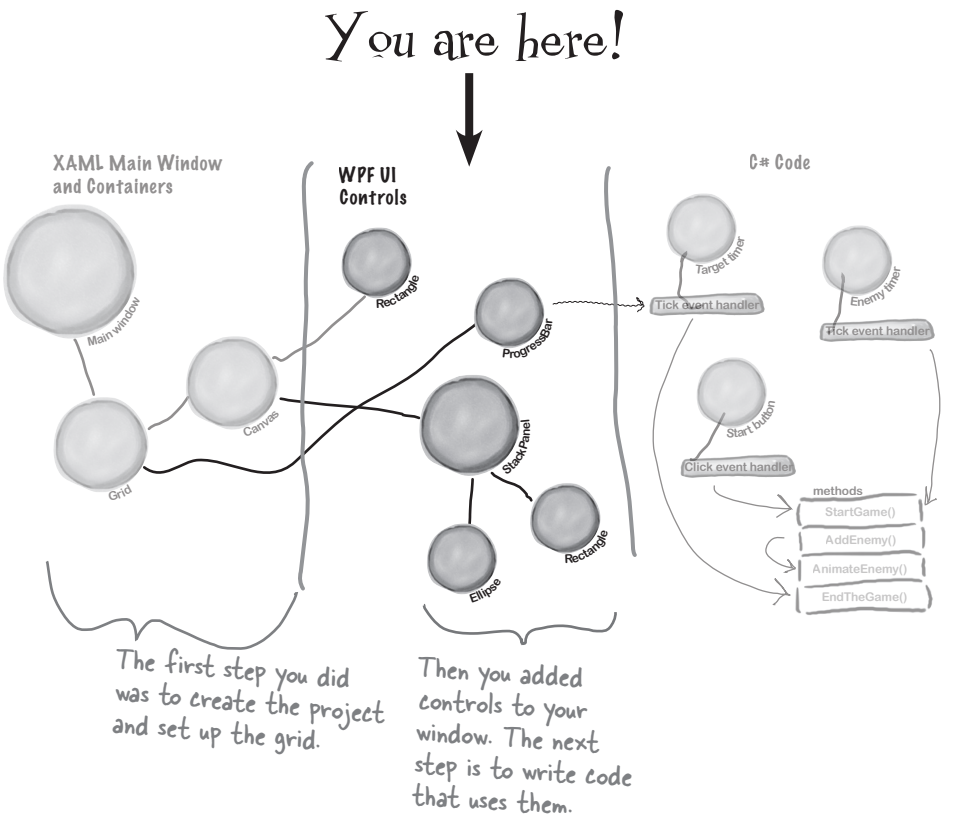
XAML property	Where to find it in the Properties window in the IDE	What it does
	At the top	Determines how tall the control should be
`Content`		
	Brush	Sets the angle that the control is turned
`Height`	Appearance	
	Common	You use this in your C# code to manipulate a specific control
`Rotation`		
	Layout	The color of the control
`Fill`		
	Transform	Use this when you want to change text displayed inside your control
`x:Name`		

Solution on page 35

Here's a hint: you can use the Search box in the Properties window to find properties—but some of these properties aren't on every type of control.

You've set the stage for the game

Your window is now all set for coding. You set up the grid that will serve as the basis of your window, and you added controls that will make up the elements of the game.



Visual Studio gave you useful tools for laying out your window, but all it really did was help you create XAML code. You're the one in charge!

What you'll do next

Now comes the fun part: adding the code that makes your game work. You'll do it in three stages: first you'll animate your enemies, then you'll let your player interact with the game, and finally you'll add polish to make the game look better.

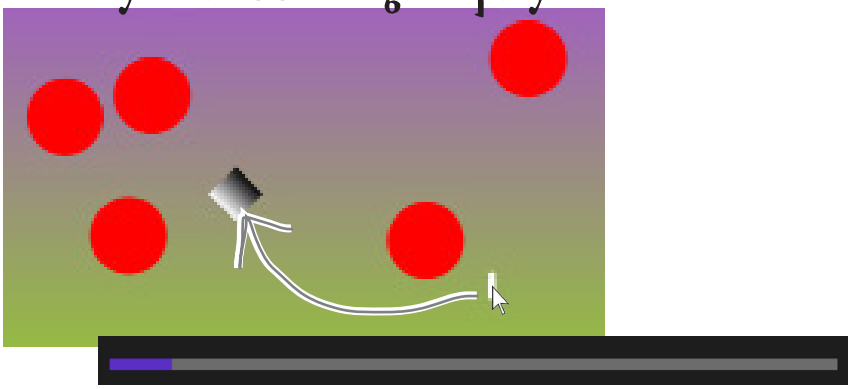
First you'll animate the enemies...

The first thing you'll do is add C# code that causes enemies to shoot out across the play area every time you click the Start button.

A lot of programmers build their code in small increments, making sure one piece works before moving on to the next one. That's how you'll build the rest of this program. You'll start by creating a method called `AddEnemy()` that adds an animated enemy to the Canvas control. First you'll hook it up to the Start button so you can fill your window up with bouncing enemies. That will lay the groundwork to build out the rest of the game.

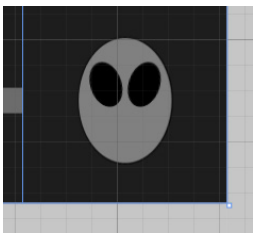
...then you'll add the gameplay...

To make the game work, you'll need the progress bar to count down, the human to move, and the game to end when the enemy gets him or time runs out.



You used a template to make the enemies look like red circles. Now you'll update the template to make them look like evil alien heads.

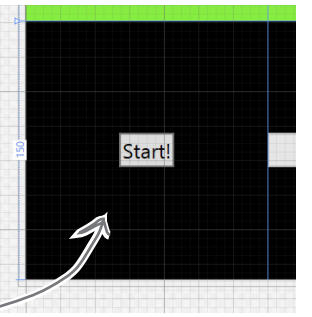
...and finally, you'll make it look good.



Add a method that does something

It's time to start writing some C# code, and the first thing you'll do is add a **method**—and the IDE can give you a great starting point by generating code.

When you're editing a window in the IDE, double-clicking on any of the toolbox controls causes the IDE to automatically add code to your project. Make sure you've got the window designer showing in the IDE, and then double-click on the Start button. The IDE will add code to your project that gets run anytime a user clicks on the button. You should see some code pop up that looks like this:



When you double-clicked the button control, the IDE created this method. It will run when a user clicks the "Start!" button in the running application.

```
private void startButton_Click(object sender, RoutedEventArgs e)
{

}
```

```
Click="startButton_Click"
```

The IDE also added this to the XAML. See if you can find it. You'll learn more about what this is in Chapter 2.

Use the IDE to create your own method

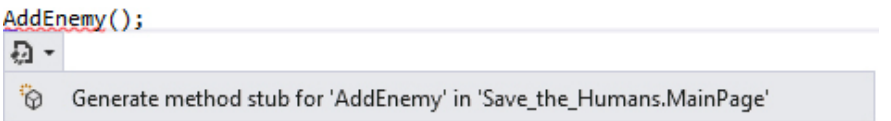
Click between the { } brackets and type this, including the parentheses and semicolon:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    AddEnemy();
}
```

The red squiggly line is the IDE telling you there's a problem, and the blue box is the IDE telling you that it might have a solution.

Notice the red squiggly line underneath the text you just typed? That's the IDE telling you that something's wrong. If you click on the squiggly line, a blue box appears, which is the IDE's way of telling you that it might be able to help you fix the error.

Hover over the blue box and click the  icon that pops up. You'll see a box asking you to generate a method stub. What do you think will happen if you click it? Go ahead and click it to find out!



there are no Dumb Questions

Q: What's a method?

A: A **method** is just a *named block of code*. We'll talk a lot more about methods in Chapter 2.

Q: And the IDE generated it for me?

A: Yes...for now. A method is one of the basic building blocks of programs—you'll write a lot of them, and you'll get used to writing them by hand.

Fill in the code for your method

It's time to make your program ***do something***, and you've got a good starting point. The IDE generated a **method stub** for you: the starting point for a method that you can fill in with code.

C# code must be added exactly as you see it here.

It's really easy to throw off your code. When you're adding C# code to your program, the capitalization has to be exactly right, and make sure you get all of the parentheses, commas, and semicolons. If you miss one, your program won't work!

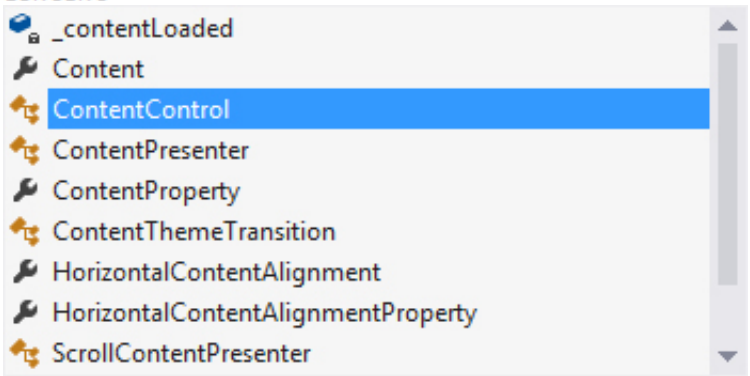
1. Delete the contents of the method stub that the IDE generated for you.

```
private void AddEnemy()
{
    throw new NotImplementedException();
}
```

Select this and delete it. You'll learn about exceptions in Chapter 12.

2. Start adding code. Type the word "Content" into the method body. The IDE will pop up a window called an **IntelliSense Window** with suggestions. Choose `ContentControl` from the list.

```
private void AddEnemy()
{
    Content
}
```



3. Finish adding the first line of code. You'll get another IntelliSense window after you type `new`.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
}
```

This line creates a new ContentControl object. You'll learn about objects and the new keyword in Chapter 3, and reference variables like enemy in Chapter 4.

4 Before you fill in the `AddEnemy()` method, you'll need to add a line of code near the top of the file. Find the line that says **`public partial class MainWindow : Window`** and add this line after the bracket (**`{`**):

```
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
    Random random = new Random();
```

This is called a field. You'll learn more about how it works in Chapter 4.

5 Finish adding the method. You'll see some squiggly red underlines. The ones under `AnimateEnemy()` will go away when you generate its method stub.

Do you see a squiggly underline under playArea? Go back to the XAML editor and make sure you set the name of the Canvas control to playArea.

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);
}
```

This line adds your new enemy control to a collection called Children. You'll learn about collections in Chapter 8.

If you need to switch between the XAML and C# code, use the tabs at the top of the window.

6 Use the blue box and the button to generate a method stub for `AnimateEnemy()`, just like you did for `AddEnemy()`. This time it added four **parameters** called `enemy`, `p1`, `p2`, and `p3`. Edit the top line of the method to change the last three parameters. Change the property `p1` to **`from`**, the property `p2` to **`to`**, and the property `p3` to **`propertyToAnimate`**. Then change any `int` types to **`double`**.

```
private void AnimateEnemy(ContentControl enemy, int p1, double p2, string p3)
{
    throw new NotImplementedException();
}
```

You'll learn about methods and parameters in Chapter 2.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
```

The IDE may generate the method stub with "int" types. Change them to "double". You'll learn about types in Chapter 4.

Flip the page to see your program run!

Finish the method and run your program

Your program is almost ready to run! All you need to do is finish your `AnimateEnemy()` method. Don't panic if things don't quite work yet. You may have missed a comma or some parentheses—when you're programming, you need to be really careful about those things!

Still seeing red? The IDE helps you track down problems.

If you still have some of those red squiggly lines, don't worry! You probably just need to track down a typo or two. If you're still seeing squiggly red underlines, it just means you didn't type in some of the code correctly. We've tested this chapter with a lot of different people, and we didn't leave anything out. All the code you need to get your program working is in these pages.

1 Add a using statement to the top of the file.
Scroll all the way to the top of the file. The IDE generated several lines that start with `using`. Add one more to the bottom of the list:

Statements like these let you use code from .NET libraries that come with C#. You'll learn more about them in Chapter 2.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

using System.Windows.Media.Animation;
```

You'll need this line to make the next bit of code work. You can use the IntelliSense window to get it right—and don't forget the semicolon at the end.

This using statement lets you use animation code from the .NET Framework in your program to move the enemies on your screen.

2 Add code that creates an enemy bouncing animation.
You generated the method stub for the `AnimateEnemy()` method on the previous page. Now you'll add its code. It makes an enemy start bouncing across the screen.

You'll learn about object initializers like this in Chapter 4.

And you'll learn about animation in Chapter 16.

```
private void AnimateEnemy(ContentControl enemy, double from, double to, string propertyToAnimate)
{
    Storyboard storyboard = new Storyboard() { AutoReverse = true, RepeatBehavior = RepeatBehavior.Forever };
    DoubleAnimation animation = new DoubleAnimation()
    {
        From = from,
        To = to,
        Duration = new Duration(TimeSpan.FromSeconds(random.Next(4, 6))),
    };
    Storyboard.SetTarget(animation, enemy);
    Storyboard.SetTargetProperty(animation, new PropertyPath(propertyToAnimate));
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

This code makes the enemy you created move across playArea. If you change 4 and 6, you can make the enemies move slower or faster.

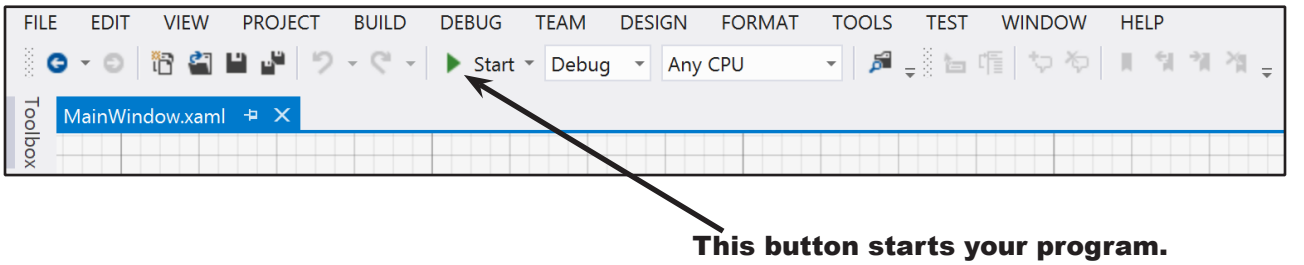
3 Look over your code.
You shouldn't see any errors, and your Error List window should be empty. If not, double-click on the error in the Error List. The IDE will jump your cursor to the right place to help you track down the problem.

If you can't see the Error List window, choose Error List from the View menu to show it. You'll learn more about using the error window and debugging your code in Chapter 2.

Here's a hint: if you move too many windows around your IDE, you can always reset by choosing Reset Window Layout from the Window menu.

4 Start your program.

Find the ▶ button at the top of the IDE. This starts your program running.



5 Now your program is running!

When you start your program, the main window will be displayed. Click the "Start!" button a few times. Each time you click it, a circle is launched across your canvas.

You built something cool! And it didn't take long, just like we promised. But there's more to do to get it right.

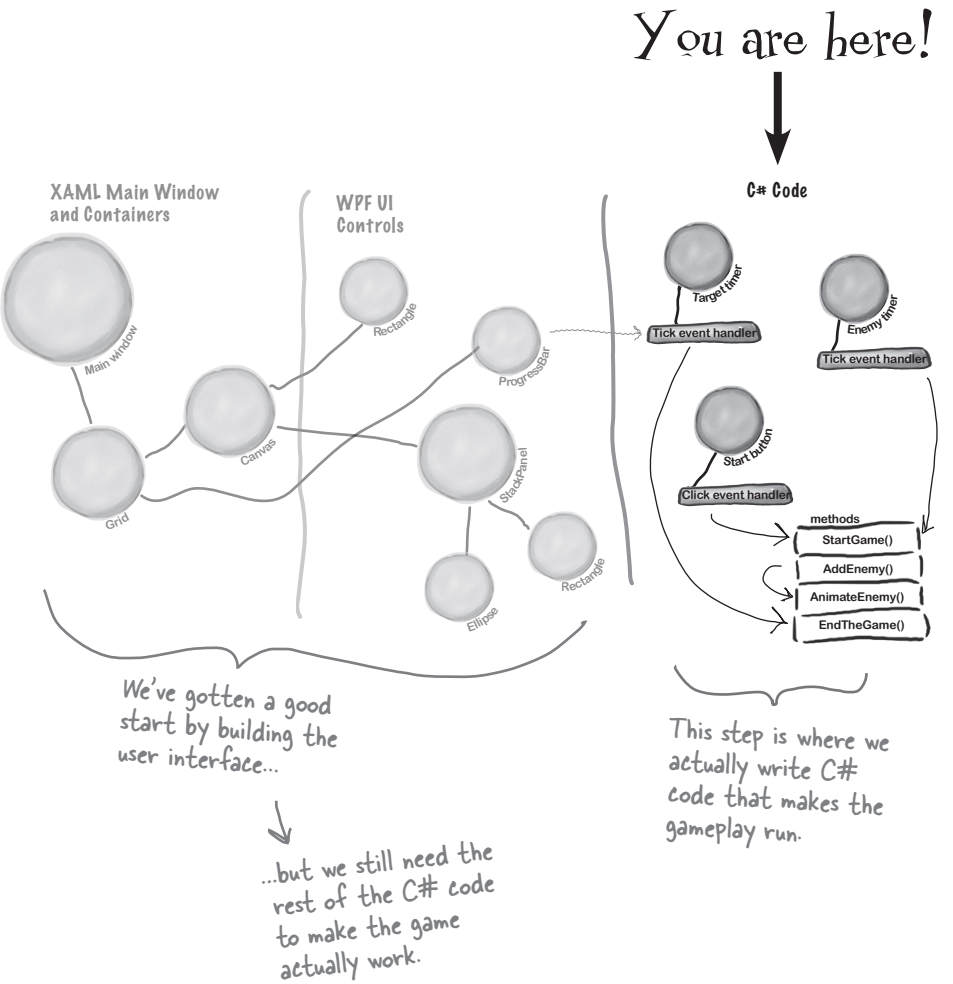
If the enemies aren't bouncing, or if they leave the play area, double-check the code. You may be missing parentheses or keywords.

6 Stop your program.

Press Alt-Tab to switch back to the IDE. The ▶ button in the toolbar has been replaced with ⏸ ■ ↻ to break, stop, and restart your program. Click the square to stop the program running.

Here's what you've done so far

Congratulations! You've built a program that actually does something. It's not quite a playable game, but it's definitely a start. Let's look back and see what you built.

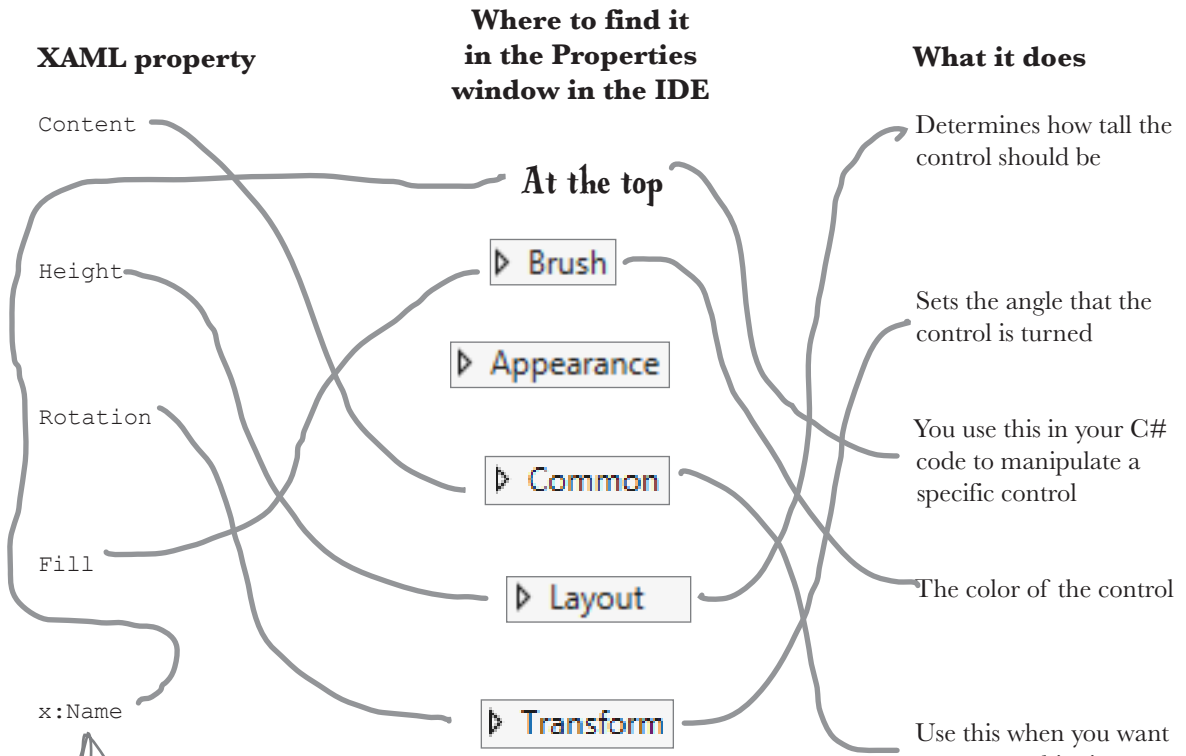


Visual Studio can generate code for you, but you need to know what you want to build BEFORE you start building it. It won't do that for you!

Here's the solution for the "Who Does What" exercise on page 28. We'll give you the answers to the pencil-and-paper puzzles and exercises, but they won't always be on the next page.

WHO DOES WHAT? solution

Now that you've built a user interface, you should have a sense of what some of the controls do, and you've used a lot of different properties to customize them. See if you can work out which property does what, and where in the Properties window in the IDE you find it.



Remember how you set the Name of the Canvas control to "playArea"? That set its "x:Name" property in the XAML, which will come in handy in a minute when you write C# code to work with the Canvas.

Add timers to manage the gameplay

The MainWindow.Xaml.cs file you've been editing contains the code for a class called MainWindow. You'll learn about classes in Chapter 3.

Let's build on that great start by adding working gameplay elements. This game adds more and more enemies, and the progress bar slowly fills up while the player drags the human to the target. You'll use **timers** to manage both of those things.

1 ADD ANOTHER LINE TO THE TOP OF YOUR C# CODE.

You'll need to add one more `using` line right below the one you added a few pages ago:

```
using System.Windows.Media.Animation;
using System.Windows.Threading;
```

This using statement lets you use DispatcherTimers.

Then go up to the top of the file where you added that `Random` line. Add three more lines:

```
public partial class MainWindow : Window
{
    Random random = new Random();
    DispatcherTimer enemyTimer = new DispatcherTimer();
    DispatcherTimer targetTimer = new DispatcherTimer();
    bool humanCaptured = false;
```

Add these three lines below the one you added before. These are fields, and you'll learn about them in Chapter 4.

2 ADD A METHOD FOR ONE OF YOUR TIMERS.

Find this code that the IDE generated:

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

Put your cursor right after the semicolon, hit Enter two times, and type `enemyTimer.` (including the period). As soon as you type the dot, an IntelliSense window will pop up. Choose `Tick` from the IntelliSense window and type the following text. As soon as you enter `+=` the IDE pops up a box:

```
enemyTimer.Tick +=
```

```
enemyTimer_Tick;    (Press TAB to insert)
```

Press the Tab key. The IDE will pop up another box:

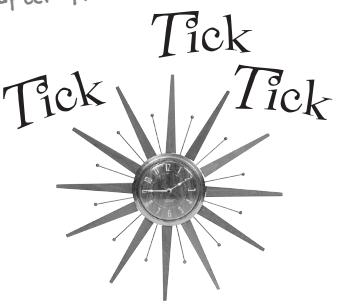
```
enemyTimer.Tick +=enemyTimer_Tick;
```

```
Press TAB to generate handler 'enemyTimer_Tick' in this class
```

Press Tab one more time. Here's the code the IDE generated for you:

```
public MainWindow()
{
    InitializeComponent();

    enemyTimer.Tick += enemyTimer_Tick;
}

void enemyTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

The IDE generated a method for you called an event handler. You'll learn about event handlers in Chapter 15.



Timers "tick" every time interval by calling methods over and over again. You'll use one timer to add enemies every few seconds, and the other to end the game when time expires.

It's normal to add parentheses () when writing about a method.

3 FINISH THE MainWindow() METHOD.

You'll add another `Tick` event handler for the other timer, and you'll add two more lines of code. Here's what your finished `MainWindow()` method and the two methods the IDE generated for you should look like:

```
public MainWindow()
{
    InitializeComponent();

    enemyTimer.Tick += enemyTimer_Tick;
    enemyTimer.Interval = TimeSpan.FromSeconds(2);

    targetTimer.Tick += targetTimer_Tick;
    targetTimer.Interval = TimeSpan.FromSeconds(.1);
}

void targetTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}

void enemyTimer_Tick(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

Try changing these numbers once your game is finished. How does that change the gameplay?

The IDE generated these lines as placeholders when you pressed Tab to add the Tick event handlers. You'll replace them with code that gets run every time the timers tick.

BRAIN POWER

Right now your Start button adds bouncing enemies to the play area. What do you think you'll need to do to make it start the game instead?

4 ADD THE EndTheGame() METHOD.

Go to the new `targetTimer_Tick()` method, delete the line that the IDE generated, and add the following code. Type `EndTheGame()` and generate a method stub for it, just like before:

```
void targetTimer_Tick(object sender, object e)
{
    progressBar.Value += 1;
    if (progressBar.Value >= progressBar.Maximum)
        EndTheGame();
}
```

Did the IDE keep trying to capitalize the P in progressBar? That's because there was no lowercase-P progressBar, and the closest match it could find was the type of the control.

If you closed the Designer tab that had the XAML code, double-click on MainWindow.xaml in the Solution Explorer window to bring it up.

Notice how `progressBar` has an error? That's OK. We did this on purpose (and we're not even sorry about it!) to show you what it looks like when you try to use a control that doesn't have a name, or has a typo in the name. Go back to the XAML code (it's in the other tab in the IDE), find the ProgressBar control that you added to the bottom row, and change its name to `progressBar`.

Next, go back to the code window and generate a method stub for `EndTheGame()`, just like you did a few pages ago for `AddEnemy()`. Here's the code for the new method:

```
private void EndTheGame()
{
    if (!playArea.Children.Contains(gameOverText))
    {
        enemyTimer.Stop();
        targetTimer.Stop();
        humanCaptured = false;
        startButton.Visibility = Visibility.Visible;
        playArea.Children.Add(gameOverText);
    }
}
```

If gameOverText comes up as an error, it means you didn't set the name of the "Game Over" TextBlock. Go back and do it now.

This method ends the game by stopping the timers, making the Start button visible again, and adding the GAME OVER text to the play area.

Make the Start button work

Remember how you made the Start button fire circles into the Canvas? Now you'll fix it so it actually starts the game.

1 **Make the Start button start the game.**
Find the code you added earlier to make the Start button add an enemy. Change it so it looks like this:

```
private void startButton_Click(object sender, RoutedEventArgs e)
{
    StartGame();
}
```

When you change this line, you make the Start button start the game instead of just adding an enemy to the playArea Canvas.

2 **Add the StartGame() method.**
Generate a method stub for the `StartGame()` method. Here's the code to fill into the stub method that the IDE added:

```
private void StartGame()
{
    human.IsHitTestVisible = true;
    humanCaptured = false;
    progressBar.Value = 0;
    startButton.Visibility = Visibility.Collapsed;
    playArea.Children.Clear();
    playArea.Children.Add(target);
    playArea.Children.Add(human);
    enemyTimer.Start();
    targetTimer.Start();
}
```

You'll learn about IsHitTestVisible in Chapter 15.

Did you forget to set the names of the target Rectangle or the human StackPanel? You can look a few pages back to make sure you set the right names for all the controls.

3 **Make the enemy timer add the enemy.**
Find the `enemyTimer_Tick()` method that the IDE added for you and replace its contents with this:

```
void enemyTimer_Tick(object sender, object e)
{
    AddEnemy();
}
```

Once you're used to working with code, you'll be good at spotting those missing parentheses, semicolons, etc.

Are you seeing errors in the Error List window that don't make sense? One misplaced comma or semicolon can cause two, three, four, or more errors to show up. Don't waste your time trying to track down every typo! Just go to the Head First Labs web page—we made it really easy for you to copy and paste all the code in this program.

There's also a link to the Head First C# forum, which you can check for tips to get this game working!

http://www.headfirstlabs.com/hfcsharp/

Ready Bake Code

We're giving you a lot of code to type in.

By the end of the book, you'll know what all this code does—in fact, you'll be able to write code just like it on your own.

For now, your job is to make sure you enter each line accurately and to follow the instructions exactly. This will get you used to entering code and will help give you a feel for the ins and outs of the IDE.

If you get stuck, you can download working versions of *MainWindow.xaml* and *MainWindow.Xaml.cs* or copy and paste XAML or C# code for each individual method: *http://www.headfirstlabs.com/hfcsharp.*

One more thing... if you download code for this project (or anything else in this appendix), *make sure you get it from the WPF folder!* If you try to use Windows Store code with your WPF project, it won't work.

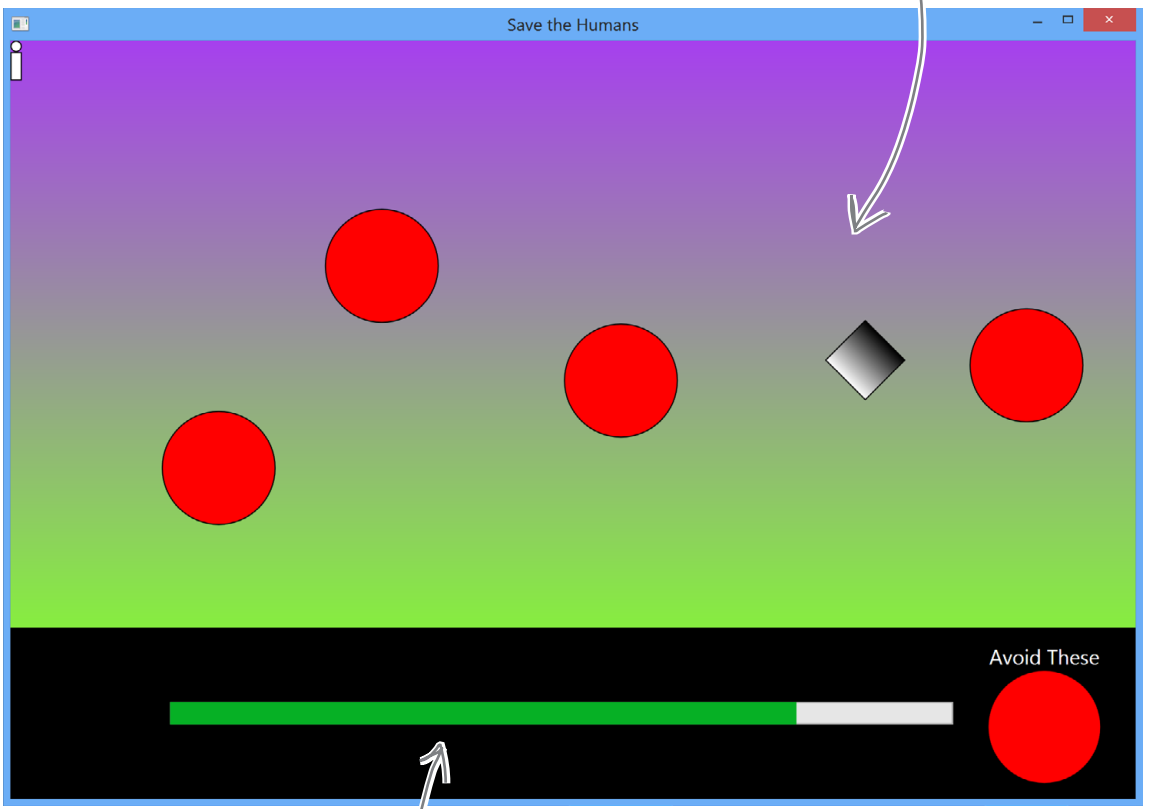
Run the program to see your progress

Your game is coming along. Run it again to see how it's shaping up.

Alert! Our spies have reported that the humans are building up their defenses!

When you press the "Start!" button, it disappears, clears the enemies, and starts the progress bar filling up.

The play area slowly starts to fill up with bouncing enemies.



When the progress bar at the bottom fills up, the game ends and the Game Over text is displayed.

The target timer should fill up slowly, and the enemies should appear every two seconds. If the timing is off, make sure you added all the lines to the MainWindow() method.

BRAIN POWER

What do you think you'll need to do to get the rest of your game working?

Flip the page to find out!

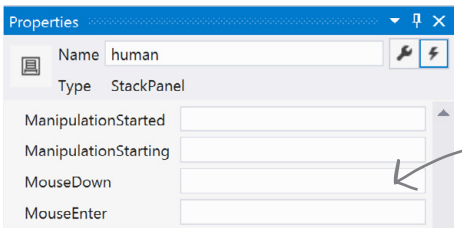
Add code to make your controls interact with the player

You've got a human that the player needs to drag to the target, and a target that has to sense when the human's been dragged to it. It's time to add code to make those things work.

Make sure you switch back to the IDE and stop the app before you make more changes to the code.

You'll learn more about the event handlers in the Properties window in Chapter 4.

1 Go to the XAML designer and use the Document Outline window to select `human` (remember, it's the StackPanel that contains a Circle and a Rectangle). Then go to the Properties window and press the ⚡ button to switch it to show event handlers. Find the MouseDown row and double-click in the empty box.



Now go back and check out what the IDE added to your XAML for the StackPanel:

```
<StackPanel x:Name="human" Orientation="Vertical" MouseDown="human_MouseDown">
```

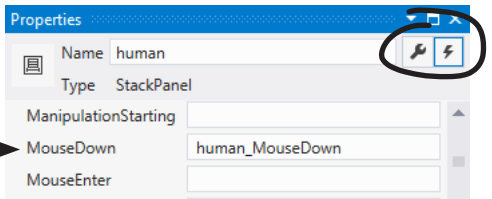
It also generated a method stub for you. Right-click on `human_MouseDown` in the XAML and choose "Navigate to Event Handler" to jump straight to the C# code:

```
private void human_MouseDown(object sender, MouseButtonEventArgs e)
{

}
```

2 Fill in the C# code:

```
private void human_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (enemyTimer.IsEnabled)
    {
        humanCaptured = true;
        human.IsHitTestVisible = false;
    }
}
```

You can use these buttons to switch between showing properties and event handlers in the Properties window.



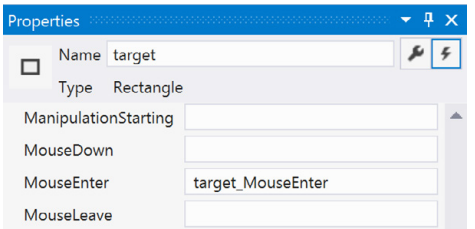
If you go back to the designer and click on the StackPanel again, you'll see that the IDE filled in the name of the new event handler method. You'll be adding more event handler methods the same way.

Make sure you add the right event handler! You added a MouseDown event handler to the human, but now you're adding a MouseEnter event handler to the target.

When the Properties window is in the mode where it displays event handlers, double-clicking on an empty event handler box causes the IDE to add a method stub for it.

3 Use the Document Outline window to select the Rectangle named `target`, and then use the event handlers view of the Properties window to add a `MouseEnter` event handler. Here's the code for the method:

```
private void target_MouseEnter(object sender, MouseEventArgs e)
{
    if (targetTimer.IsEnabled && humanCaptured)
    {
        progressBar.Value = 0;
        Canvas.SetLeft(target, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(target, random.Next(100, (int)playArea.ActualHeight - 100));
        Canvas.SetLeft(human, random.Next(100, (int)playArea.ActualWidth - 100));
        Canvas.SetTop(human, random.Next(100, (int)playArea.ActualHeight - 100));
        humanCaptured = false;
        human.IsHitTestVisible = true;
    }
}
```



You'll need to switch your Properties window back to show properties instead of event handlers.

4 Now you'll add two more event handlers, this time to the `playArea` Canvas control. You'll need to find the `[Grid]` in the Document Outline, select it, and set its name to `grid`. Then you can add these methods to handle the MouseMove and MouseLeave event handlers for the Canvas:

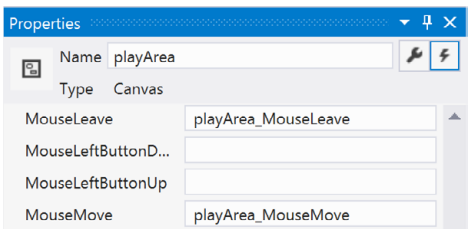
```
private void playArea_MouseMove(object sender, MouseEventArgs e)
{
    if (humanCaptured)
    {
        Point pointerPosition = e.GetPosition(null);
        Point relativePosition = grid.TransformToVisual(playArea).Transform(pointerPosition);
        if ((Math.Abs(relativePosition.X - Canvas.GetLeft(human)) > human.ActualWidth * 3)
            || (Math.Abs(relativePosition.Y - Canvas.GetTop(human)) > human.ActualHeight * 3))
        {
            humanCaptured = false;
            human.IsHitTestVisible = true;
        }
        else
        {
            Canvas.SetLeft(human, relativePosition.X - human.ActualWidth / 2);
            Canvas.SetTop(human, relativePosition.Y - human.ActualHeight / 2);
        }
    }
}

private void playArea_MouseLeave(object sender, MouseEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

That's a lot of parentheses! Be really careful and get them right.

These two vertical bars are a logical operator. You'll learn about them in Chapter 2.

You can make the game more or less sensitive by changing these 3s to a lower or higher number.

Make sure you put the right code in the correct event handler! Don't accidentally swap them.



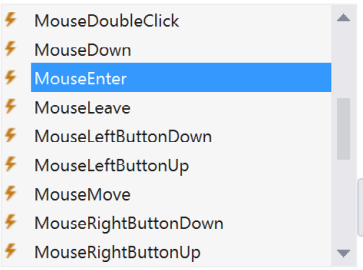
Dragging humans onto enemies ends the game

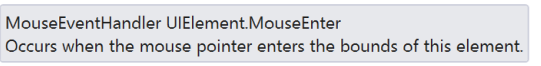
When the player drags the human into an enemy, the game should end. Let's add the code to do that. Go to your `AddEnemy()` method and add one more line of code to the end. Use the IntelliSense window to fill in `enemy.PointerEntered` from the list:

```
private void AddEnemy()
{
    ContentControl enemy = new ContentControl();
    enemy.Template = Resources["EnemyTemplate"] as ControlTemplate;
    AnimateEnemy(enemy, 0, playArea.ActualWidth - 100, "(Canvas.Left)");
    AnimateEnemy(enemy, random.Next((int)playArea.ActualHeight - 100),
        random.Next((int)playArea.ActualHeight - 100), "(Canvas.Top)");
    playArea.Children.Add(enemy);

    enemy.Mouse
}
```

Here's the last line of your AddEnemy() method. Put your cursor at the end of the line and hit Enter to add the new line of code.

Start typing this line of code. As soon as you enter the dot, an IntelliSense window will pop up. Keep typing "Enter" to jump down to the right entry in the list.





Choose `MouseEnter` from the list. (If you choose the wrong one, don't worry—just backspace over it to delete everything past the dot. Then enter the dot again to bring up the IntelliSense window.)

Next, add an event handler, just like you did before. Type += and then press Tab:

```
enemy.MouseEnter +=
```

enemy_MouseEnter; (Press TAB to insert)

You'll learn all about how event handlers like this work in Chapter 15.

Then press Tab again to generate the stub for your event handler:

```
enemy.MouseEnter +=enemy_MouseEnter;
```

Press TAB to generate handler 'enemy_MouseEnter' in this class

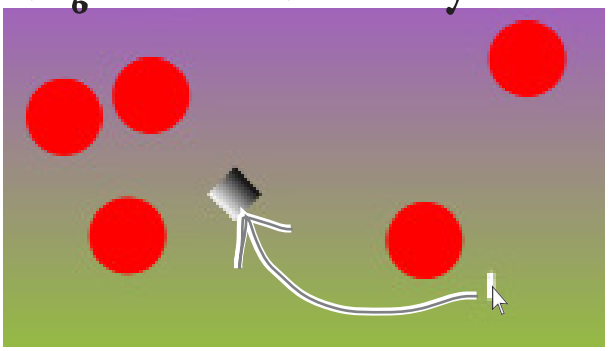
Now you can go to the new method that the IDE generated for you and fill in the code:

```
void enemy_MouseEnter(object sender, MouseEventArgs e)
{
    if (humanCaptured)
        EndTheGame();
}
```

Your game is now playable

Run your game—it's almost done! When you click the Start button, your play area is cleared of any enemies, and only the human and target remain. You have to get the human to the target before the progress bar fills up. Simple at first, but it gets harder as the screen fills with dangerous alien enemies!

Drag the human to safety!

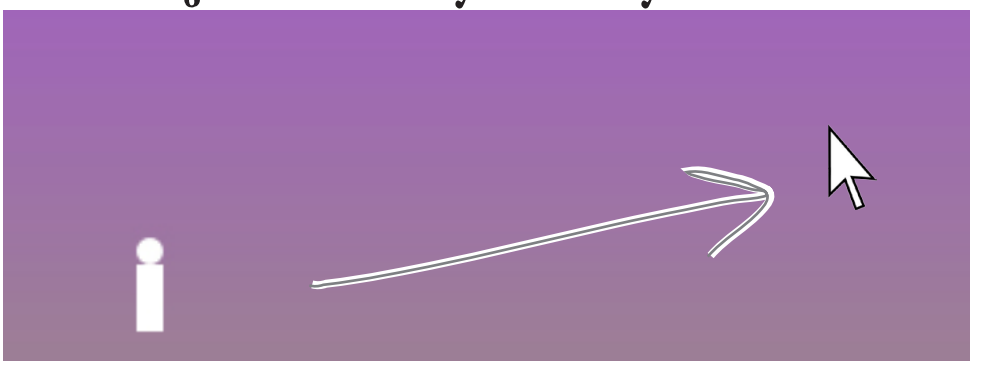


The aliens spend their time patrolling for moving humans, so the game ends only if you drag a human onto an enemy. Once you release the human, he's temporarily safe from aliens.

Look through the code and find where you set the IsHitTestVisible property on the human. When it's on, the human intercepts the PointerEntered event because the human's StackPanel control is sitting between the enemy and the pointer.

Get him to the target before time's up...

...but drag too fast, and you'll lose your human!



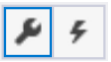
Make your enemies look like aliens

Red circles aren't exactly menacing. Luckily, you used a template. All you need to do is update it.

Watch it!

Seeing events instead of properties?

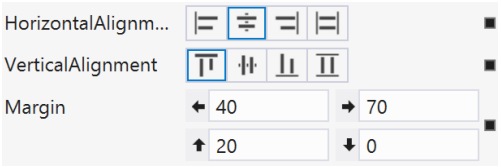
You can toggle the Properties window between displaying properties or events for the selected control by clicking the wrench or lightning bolt icons.



1. Go to the Document Outline, right-click on the ContentControl, choose Edit Template, and then Edit Current to edit the template. You'll see the template in the XAML window. Edit the XAML code for the ellipse to set the width to `75` and the fill to `Gray`. Then add `Stroke="Black"` to add a black outline. Here's what it should look like (you can delete any additional properties that may have inadvertently been added while you worked on it):

```
<Ellipse Fill="Gray" Stroke="Black" Height="100" Width="75"/>
```

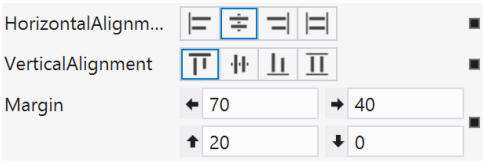
2. Drag another Ellipse control out of the toolbox on top of the existing ellipse. Change its **Fill** to black, set its width to `25`, and its height to `35`. Set the alignment and margins like this:



You can also "eyeball" it (excuse the pun) by using the mouse or arrow keys to drag the ellipse into place. Try using Copy and Paste in the Edit menu to copy the ellipse and paste another one on top of it.

3. Use the  button in the Transforms section of the Properties window to add a Skew transform:

4. Drag one more Ellipse control out of the toolbox on top of the existing ellipse. Change its fill to `Black`, set its width to `25`, and set its height to `35`. Set the alignment and margins like this:



and add a skew like this:

Now your enemies look a lot more like human-eating aliens.

Here's the final XAML for the updated enemy ControlTemplate you created:

```
<ControlTemplate x:Key="EnemyTemplate" TargetType="{x:Type ContentControl}">
    <Grid>
        <Ellipse Fill="Gray" Stroke="Black" Height="100" Width="75"/>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 VerticalAlignment="Top" HorizontalAlignment="Center"
                 Margin="40,20,70,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform/>
                    <SkewTransform AngleX="10"/>
                    <RotateTransform/>
                    <TranslateTransform/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
        <Ellipse Fill="Black" Stroke="Black" Height="35"  Width="25"
                 VerticalAlignment="Top" HorizontalAlignment="Center"
                 Margin="70,20,40,0" RenderTransformOrigin="0.5,0.5">
            <Ellipse.RenderTransform>
                <TransformGroup>
                    <ScaleTransform/>
                    <SkewTransform AngleX="-10"/>
                    <RotateTransform/>
                    <TranslateTransform/>
                </TransformGroup>
            </Ellipse.RenderTransform>
        </Ellipse>
    </Grid>
</ControlTemplate>
```

See if you can get creative and change the way the human, target, play area, and enemies look.

And don't forget to step back and really appreciate what you built. Good job!

THERE'S JUST ONE MORE THING YOU NEED TO DO...

PLAY YOUR GAME!

Chapter 2

Start diving into code with WPF projects.

The second chapter gets you started writing C# code, and most of the chapter is focused around building Windows Store apps.

We recommend that you do the following:

- Read Chapter 2 in the main part of the book through page 68.
- We provide a replacement for page 69 in this appendix. After that, you can read pages 70, 71, and 72 in the book.
- Then there are replacements for pages 73 and 74, where you build a program from scratch. You can follow the rest of the project in the book.
- The book will work just fine for you through page 82.
- There's an exercise on page 83, and its solution is on page 85. We provide replacements for those pages in this PDF.

Once you finish that exercise, the chapter no longer requires any Windows Store apps or Windows 8. You'll be able to continue on in the book through Chapter 9, and you can do the first and second labs.

Use the debugger to see your variables change

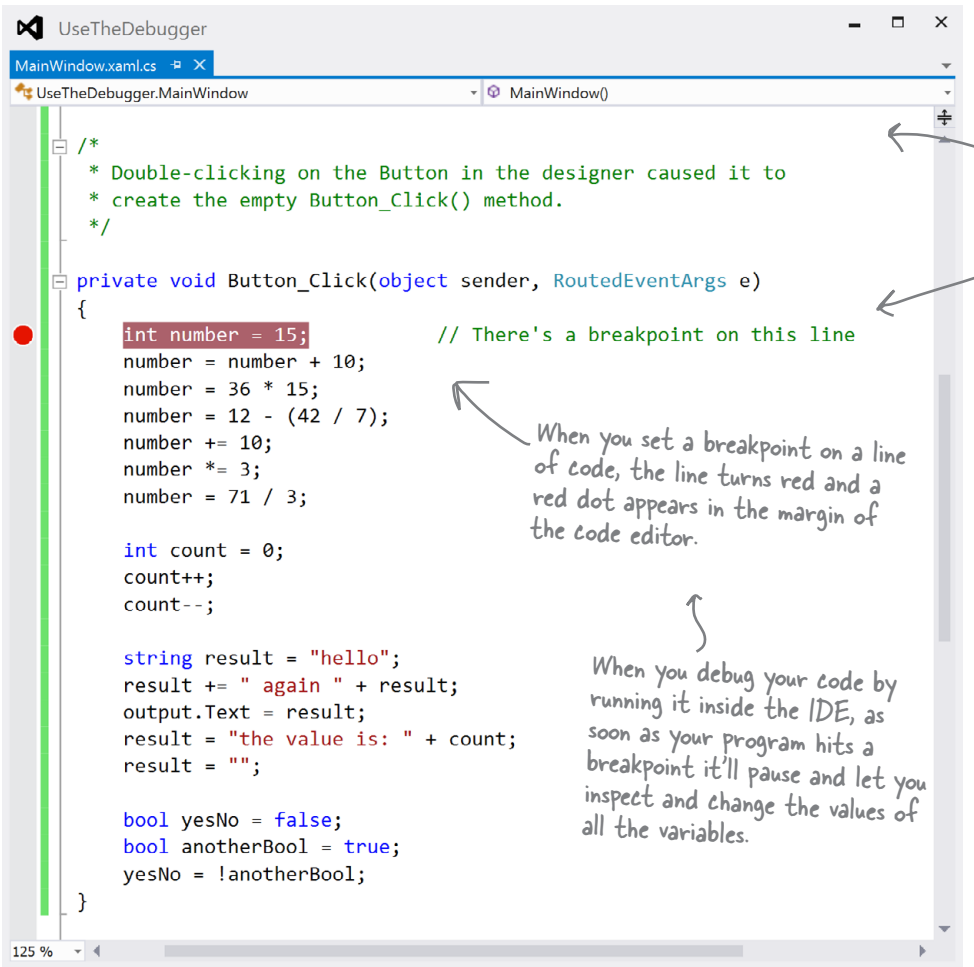
The debugger is a great tool for understanding how your programs work. You can use it to see the code on the previous page in action.

1 **CREATE A NEW WPF APPLICATION PROJECT.**

Drag a TextBlock onto your page and give it the name `output`. Then add a button and double-click it to add a method called `Button_Click()`. The IDE will automatically open that method in the code editor. Enter all the code on the previous page into the method.

2 **INSERT A BREAKPOINT ON THE FIRST LINE OF CODE.**

Right-click on the first line of code (`int number = 15;`) and choose Insert Breakpoint from the Breakpoint menu. (You can also click on it and choose Debug→Toggle Breakpoint or press F9.)



Comments (which either start with two or more slashes or are surrounded by /* and */ marks) show up in the IDE as green text. You don't have to worry about what you type in between those marks, because comments are always ignored by the compiler.

When you set a breakpoint on a line of code, the line turns red and a red dot appears in the margin of the code editor.

When you debug your code by running it inside the IDE, as soon as your program hits a breakpoint it'll pause and let you inspect and change the values of all the variables.

Creating a new WPF Application project will tell the IDE to create a new project with a blank window. You might want to name it something like UseTheDebugger (to match the header of this page). You'll be building a whole lot of programs throughout the book, and you may want to go back to them later.

Flip back to page 70 in the book and keep going!

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Make sure you choose a sensible name for this project, because you'll refer back to it later in the book.

Build an app from the ground up

When you see these sneakers, it means that it's time for you to come up with code on your own.

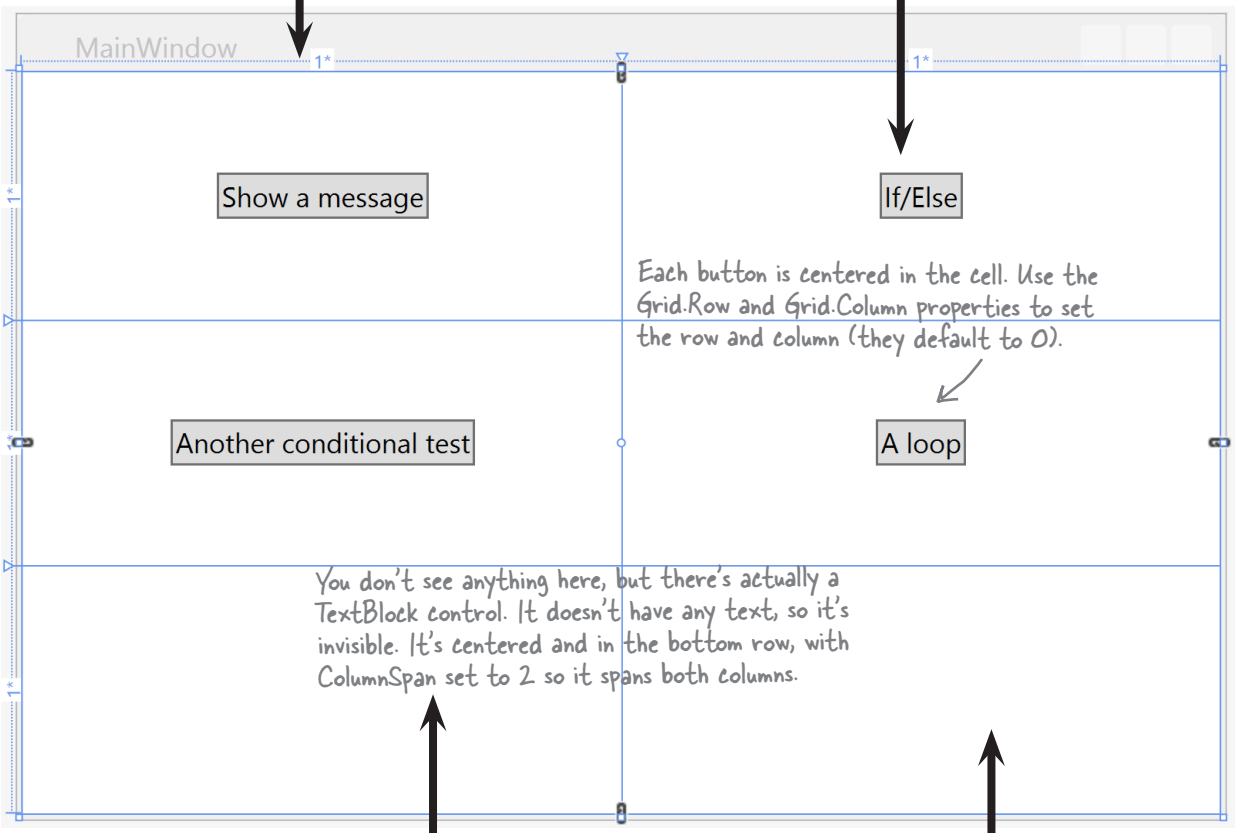
Exercise

The real work of any program is in its statements. You've already seen how statements fit into a window. Now let's really dig into a program so you can understand every line of code. Start by **creating a new Visual C# WPF Application project**. Open the main window and use the IDE to modify it by adding three rows and two columns to the grid, and then adding four button controls and a TextBlock to the cells.

Build this window

The window has a grid with three rows and two columns. Each row definition has its height set to `1*`, which gives it a `<RowDefinition/>` without any properties. The column heights work the same way.

The window has four button controls, one in each row. Use the `Content` property to set their text to Show a message, If/else, Another conditional test, and A loop.



Each button is centered in the cell. Use the Grid.Row and Grid.Column properties to set the row and column (they default to 0).

You don't see anything here, but there's actually a TextBlock control. It doesn't have any text, so it's invisible. It's centered and in the bottom row, with ColumnSpan set to 2 so it spans both columns.

The bottom cell has a TextBlock control named `myLabel`. Use its `Style` property to set the style to `BodyTextStyle`.

Use the `x:Name` property to name the buttons `button1`, `button2`, `button3`, and `button4`. Once they're named, double-click on each of them to add an event handler method.

If you need to use the Edit Style right-mouse menu to set this but you're having trouble selecting the control, you can right-click on the TextBlock control in the Document Outline and choose Edit Style from there.

Here's our solution to the exercise. Does your solution look similar? Are the line breaks different, or the properties in a different order? If so, that's OK!

A lot of programmers don't use the IDE to create their XAML—they build it by hand. If we asked you to type in the XAML by hand instead of using the IDE, would you be able to do it?

BuildAnApp

MainWindow.xaml

```
<Window x:Class="BuildAnApp.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Button x:Name="button1" Content="Show a message" HorizontalAlignment="Center"
                VerticalAlignment="Center"  Click="button1_Click"/>

        <Button x:Name="button2" Content="If/Else" HorizontalAlignment="Center"
                VerticalAlignment="Center" Grid.Column="1" Click="button2_Click"/>

        <Button x:Name="button3" Content="Another conditional test" HorizontalAlignment="Center"
                VerticalAlignment="Center" Grid.Row="1" Click="button3_Click"/>

        <Button x:Name="button4" Content="A loop" HorizontalAlignment="Center"
                VerticalAlignment="Center" Grid.Column="1" Grid.Row="1" Click="button4_Click"/>

        <TextBlock x:Name="myLabel" HorizontalAlignment="Center"
                   VerticalAlignment="Center" Grid.Row="2" Grid.ColumnSpan="2"/>

    </Grid>
</Window>
```

Here's the <Window> and <Grid> tags that the IDE generated for you when you created the WPF application.

Here are the row and column definitions: three rows and two columns.

When you double-clicked on each button, the IDE generated a method with the name of the button followed by _Click.

This button is in the second column and second row, so these properties are set to 1.

120 %

XAML Design

Try removing the HorizontalAlignment or VerticalAlignment property from one of the buttons. It expands to fill the entire cell horizontally or vertically if the alignment isn't set.

Why do you think the left column and top row are given the number 0, not 1? Why is it OK to leave out the `Grid.Row` and `Grid.Column` properties for the top-left cell?

We'll give you a lot of exercises like this throughout the book. We'll give you the answer in a couple of pages. If you get stuck, don't be afraid to peek at the answer—it's not cheating!

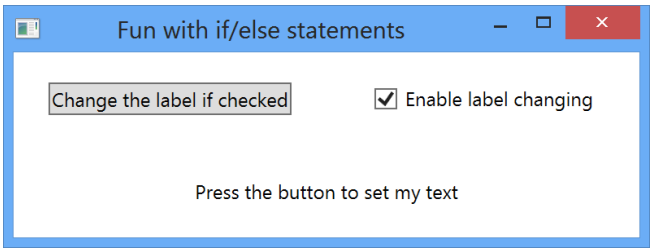
You'll be creating a lot of applications throughout this book, and you'll need to give each one a different name. We recommend naming this one "PracticeUsingIfElse". It helps to put programs from a chapter in the same folder.

Exercise

Time to get some practice using if/else statements. Can you build this program?

Build this window.
It's got a grid with two rows and two columns, it's 150 pixels tall and 450 pixels wide, and it's got the window title *Fun with if/else statements*.

If you create two rows and set one row's height to 1* in the IDE, it seems to disappear because it's collapsed to a tiny size. Just set the other row to 1* and it'll show up again.



Add a button and a checkbox.
You can find the checkbox control in the toolbox, just below the button control. Set the Button's name to `changeText` and the checkbox's name to `enableCheckbox`. Use the Edit Text right-click menu option to set the text for both controls (hit Escape to finish editing the text). Right-click on each control and choose Reset Layout→All, then make sure both of them have their VerticalAlignment and HorizontalAlignment set to `Center`.

Add a TextBlock.
It's almost identical to the one you added to the bottom of the window in the last project. This time, name it `labelToChange` and set its `Grid.Row` property to `"1"`.

Set the TextBlock to this message if the user clicks the button but the box IS NOT checked.
Here's the conditional test to see if the checkbox is checked:

Text changing is disabled

```
enableCheckbox.IsChecked == true
```

If that test is **NOT** true, then your program should execute two statements:

Hint: you'll put this code in the else block.

```
labelToChange.Text = "Text changing is disabled";
labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
```

If the user clicks the button and the box IS checked, change the TextBlock so it either shows Left on the left-hand side or Right on the right-hand side.

If the label's `Text` property is currently equal to `"Right"` then the program should change the text to `"Left"` and set its `HorizontalAlignment` property to `HorizontalAlignment.Left`. Otherwise, set its text to `"Right"`and its `HorizontalAlignment` property to `HorizontalAlignment.Right`. This should cause the program to flip the label back and forth when the user presses the button—but only if the checkbox is checked.

Time to get some practice using if/else statements. Can you build this program?

We added line breaks as usual to make it easier to read on the window.

Here's the XAML code for the grid:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
        HorizontalAlignment="Center"  VerticalAlignment="Center"
        IsChecked="true" Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
            Text="Press the button to set my text"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.ColumnSpan="2"/>

</Grid>
```

If you double-clicked the button in the designer before you set its name, it may have created a Click event handler method called Button_Click_1() instead of changeText_Click().

And here's the C# code for the button's event handler method:

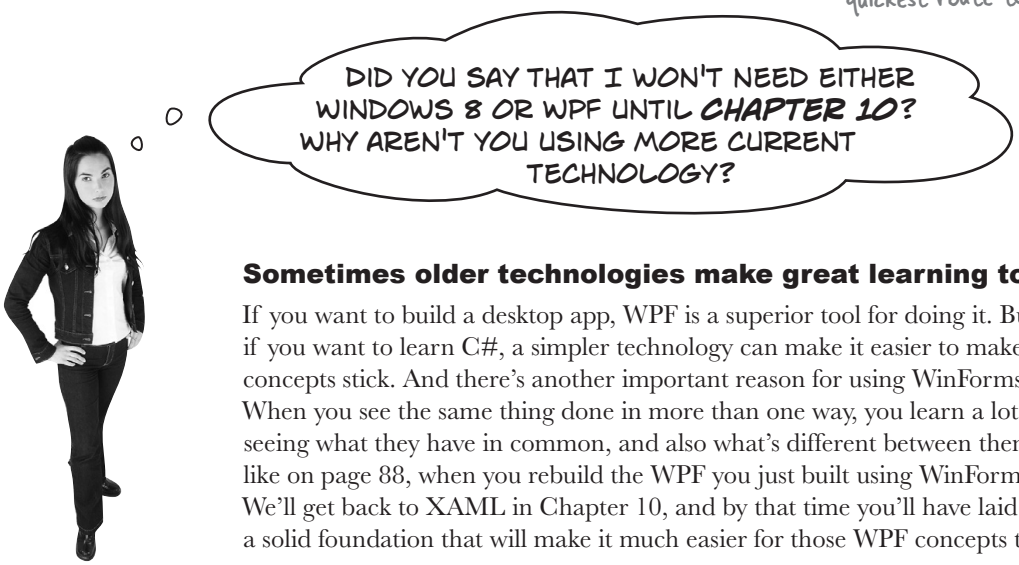
```
private void changeText_Click(object sender, RoutedEventArgs e)
{
    if (enableCheckbox.IsChecked == true)
    {
        if (labelToChange.Text == "Right")
        {
            labelToChange.Text = "Left";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Left;
        }
        else
        {
            labelToChange.Text = "Right";
            labelToChange.HorizontalAlignment = HorizontalAlignment.Right;
        }
    }
    else
    {
        labelToChange.Text = "Text changing is disabled";
        labelToChange.HorizontalAlignment = HorizontalAlignment.Center;
    }
}
```

You won't use XAML for the next part of the book.

The rest of Chapter 2 doesn't require Windows 8 and can be done with Visual Studio 2010, or using a Windows operating system as early as Windows 2003. You won't need to replace any pages in the book until you get to Chapter 10. That's because the next part of the book uses Windows Forms Application (or WinForms) projects. These C# projects use an older technology for building desktop apps. You'll use WinForms as a teaching and learning tool, just like you've been using the IDE to learn and explore C# and XAML.

Have a look at page 87, which explains why switching to WinForms is a good tool for getting C# concepts into your brain.

This applies to WPF, too! Building these WinForms projects will help get core C# concepts into your brain faster, and that's the quickest route to learning WPF.



Sometimes older technologies make great learning tools.

If you want to build a desktop app, WPF is a superior tool for doing it. But if you want to learn C#, a simpler technology can make it easier to make concepts stick. And there's another important reason for using WinForms. When you see the same thing done in more than one way, you learn a lot from seeing what they have in common, and also what's different between them—like on page 88, when you rebuild the WPF you just built using WinForms. We'll get back to XAML in Chapter 10, and by that time you'll have laid down a solid foundation that will make it much easier for those WPF concepts to stick.

Watch it!

Some chapters use C# features introduced in .NET 4.0 that are not supported by Visual Studio 2008.

If you're using Visual Studio 2008, you may run into a few problems once you reach the end of Chapter 3. That's because the latest version of the .NET Framework available in 2008 was 3.5. And that's a problem, because the book uses features of C# that were only introduced in .NET 4.0. In Chapter 3 we'll teach you about ***object initializers****, and in Chapter 8 you'll learn about* ***collection initializers*** *and* ***covariance****—and if you're using Visual Studio 2008, the code for those examples* ***won't compile*** *because in 2008 those things hadn't been added to C# yet! If you absolutely can't install a newer version of Visual Studio, you'll still be able to do almost all the exercises, but you won't be able to use these features of C#.*

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 10

IN THIS CHAPTER, YOU'LL DIVE INTO WPF DEVELOPMENT BY REDESIGNING SOME FAMILIAR PROGRAMS AS WPF APPS.

You can port your WinForms apps to WPF.

If you've completed chapters 3–9 and finished all the exercises and labs so far, then you've **written a lot of code**. In this chapter, you'll revisit some of that code and use it as a springboard for learning WPF.

Here's how we recommend that you work through Chapter 10:

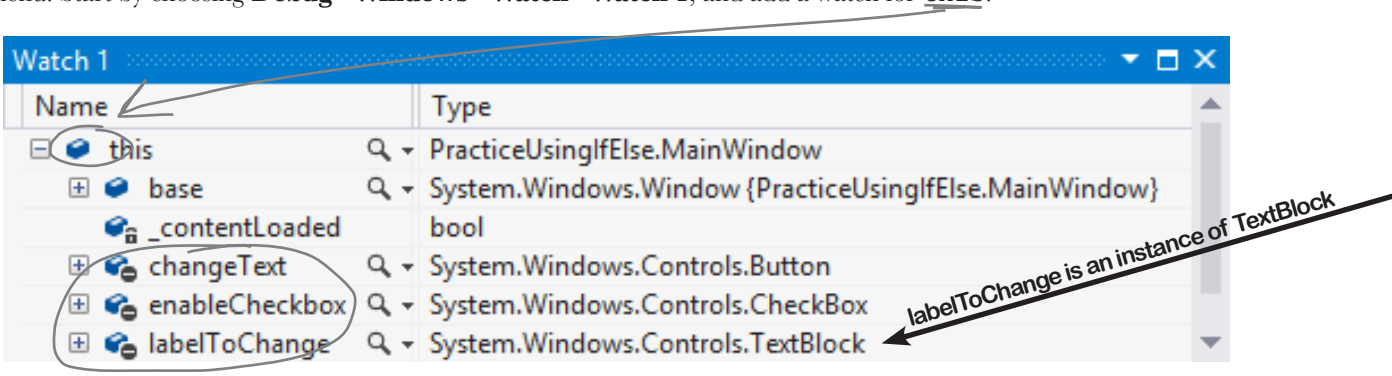
- We recommend that you follow the chapter in the main part of the book through page 497. This includes doing everything on page 489, the "Sharpen your Pencil" exercises, and the "Do this!" exploration project on page 497.
- This appendix has replacement pages for pages 498–505, so use those instead.
- Page 506 **applies only to Windows Store projects**, so you can read it but it won't help you with WPF.
- After that, use pages 509–511 from this appendix.
- Finally, read pages 514 and 515 in the book. Once you've read them, you can replace the rest of the chapter (pages 516–533) with pages in this appendix.

WPF applications use XAML to create UI objects

Do this!

When you use XAML to build the user interface for a WPF application, you're building out an object graph. And just like with WinForms, you can explore it with IDE's Watch window. **Open the "fun with if-else statements" program from Chapter 2**. Then open *MainWindow.xaml.cs*, place a breakpoint in the constructor on the call to `InitializeComponent()`, and **use the IDE to explore the app's UI objects**.

1 Start debugging, then press F10 to step over the method. Open a Watch window using the Debug menu. Start by choosing **Debug→Windows→Watch→Watch 1**, and add a watch for `this`:

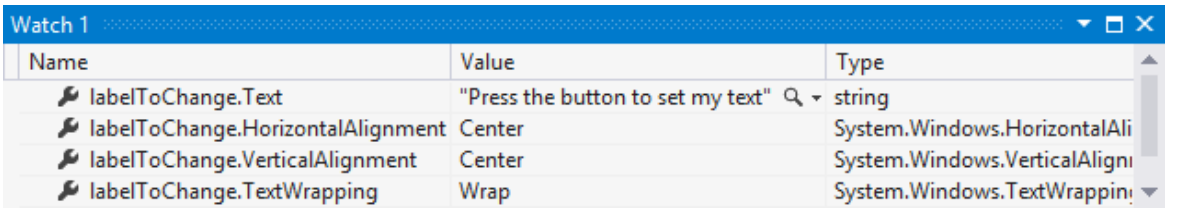


2 Now have another look at the XAML that defines the page:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <Button x:Name="changeText" Content="Change the label if checked"
            HorizontalAlignment="Center" Click="changeText_Click"/>

    <CheckBox x:Name="enableCheckbox" Content="Enable label changing"
              HorizontalAlignment="Center" IsChecked="true"
              Grid.Column="1"/>

    <TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
               Text="Press the button to set my text"
               HorizontalAlignment="Center" VerticalAlignment="Center"
               Grid.ColumnSpan="2"/>
</Grid>
```

The XAML that defines the controls on a page is turned into a Page object with fields and properties that contain references to UI controls.

3 Add some of the `labelToChange` properties to the Watch window:



The app automatically sets the properties based on your XAML:

```
<TextBlock x:Name="labelToChange" Grid.Row="1" TextWrapping="Wrap"
           Text="Press the button to set my text"
           HorizontalAlignment="Center" VerticalAlignment="Center"
           Grid.ColumnSpan="2"/>
```

But try putting `labelToChange.Grid` or `labelToChange.ColumnSpan` into the Watch window. The control is a `Windows.UI.Controls.TextBlock` object, and that object doesn't have those properties. Can you guess what's going on with those XAML properties?

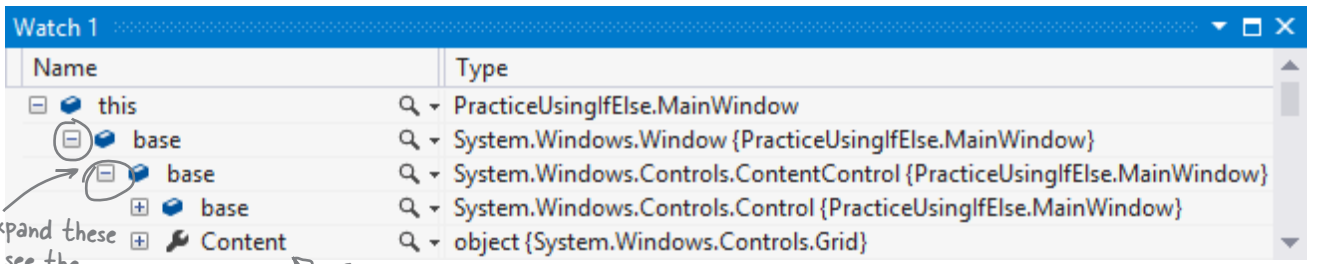
4 Stop your program, open *MainWindow.xaml.cs*, and find the class declaration for `MainWindow`. Take a look at the declaration—it's a subclass of `Window`. Hover over `Window` so the IDE shows you its full class name:

Hover over Window to see its class.

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
```

class System.Windows.Window
Provides the ability to create, configure, show, and manage the lifetime of windows and dialog boxes.

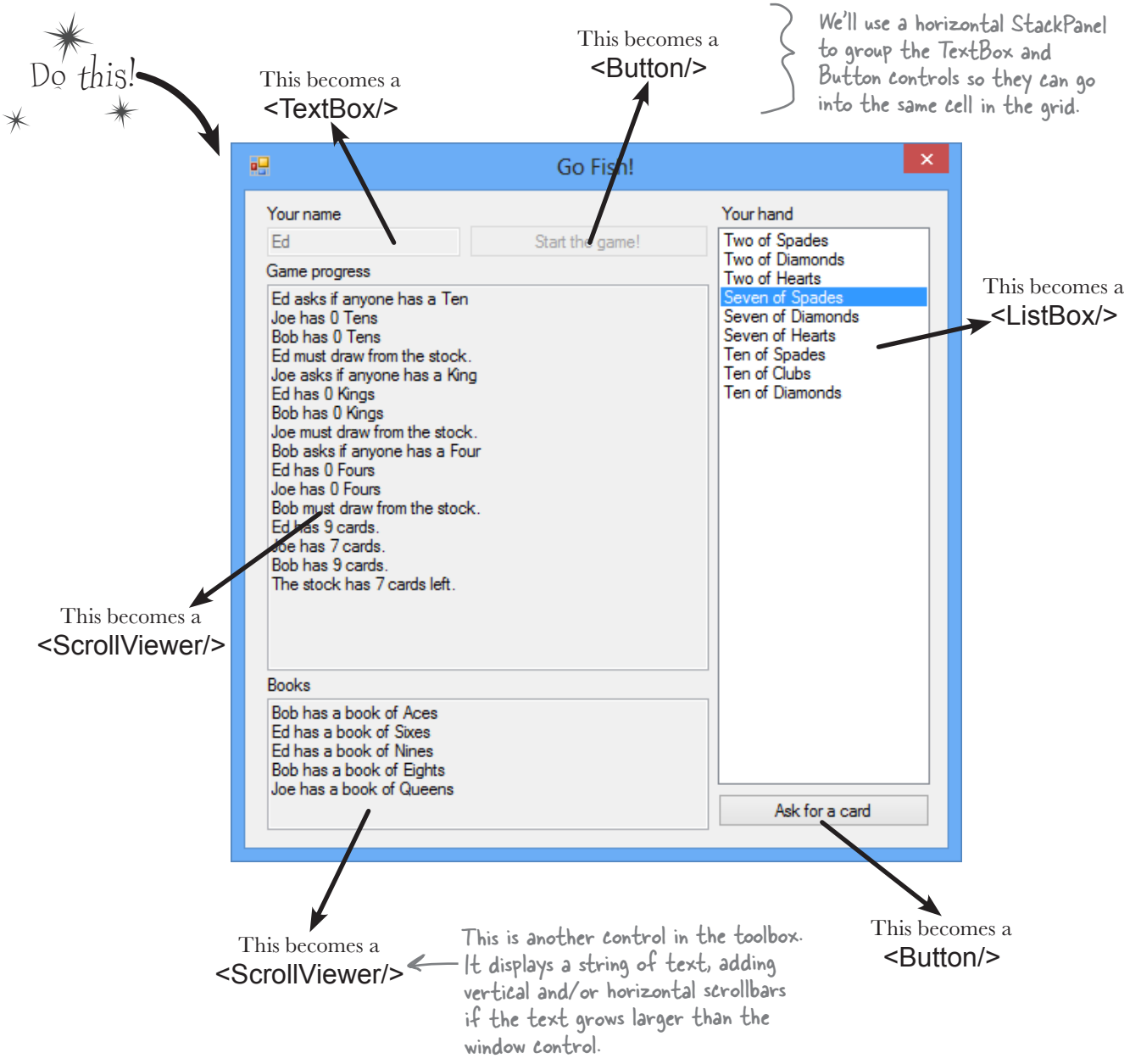
Now start your program again and press F10 to step over the call to `InitializeComponent()`. Go back to the Watch window and expand `this` >> `base` >> `base` to traverse back up the inheritance hierarchy.



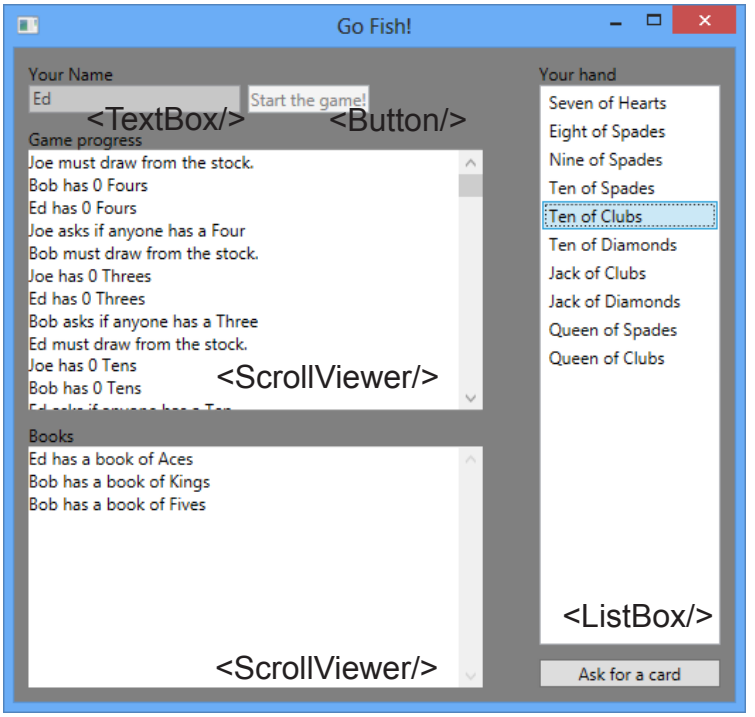
Take a little time and explore the objects that your XAML generated. We'll dig into some of these objects later on in the book. For now, just poke around and get a sense of how many objects are behind your app.

Redesign the Go Fish! form as a WPF application

The Go Fish! game that you built in Chapter 8 would make a great WPF application. Open Visual Studio and **create a new WPF Application project** (just like you did for Save the Humans). Over the next few pages, you'll redesign it in XAML, with a main window that adjusts its content as it's resized. Instead of using Windows Forms controls on a form, you'll use WPF XAML controls.

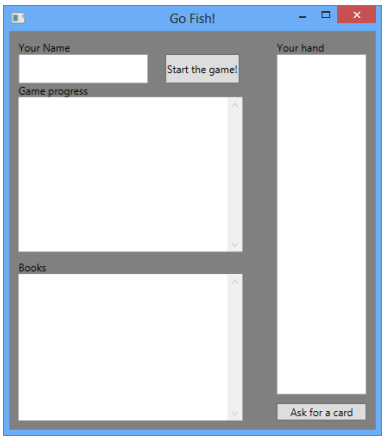


Here's how those controls will look on the app's main window:



Most of the code to manage the gameplay will remain the same, but the UI code will change.

The controls will be contained in a grid, with rows and columns that expand or contract based on the size of the window. This will allow the game to shrink or grow if the user resizes the window:



The game will be playable no matter what the window dimensions are.

Page layout starts with controls

WPF apps and WinForms have one thing in common: they both rely on controls to lay out your page. The Go Fish! page has two buttons, a ListBox to show the hand, a TextBox for the user to enter the name, and four TextBlock labels. It also has two ScrollViewer controls with a white background to display the game progress and books.

The XAML for the main window starts with an opening `<Window>` tag. The title property sets the title of the window to "Go Fish!" Setting the Height and Width property changes the window size—and you'll see the size change in the designer as soon as you change those properties. Use the Background property to give it a gray background.

Here's the updated `<Window>` opening tag. We named our project `GoFish`—if you use a different name, the first line will have that name in its `x:Class` property.

```
<Window x:Class="GoFish.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Go Fish!" Height="500" Width="525" Background="Gray">
```

The window title and starting width and height are set using properties in the <Window> tag.

```
<Grid Margin="10" >
```

We'll use a StackPanel to put the TextBox for the player's name and the Start button in one cell:

```
1 <TextBlock Text="Your Name" />

  <StackPanel Orientation="Horizontal" Grid.Row="1">
      <TextBox x:Name="playerName" FontSize="12" Width="150" />

2  <Button x:Name="startButton" Margin="5,0"
   Content="Start the game!"/>
</StackPanel>
```

This Margin property sets the left and right margins for the button to 5, and the top and bottom margins to 0. We could also have set it to 5,0,0,0 to set just the left margin and left the right margin zero.

Each label on the page ("Your name," "Game progress," etc.) is a TextBlock. Use the Margin property to add a 10-pixel margin above the label:

```
<TextBlock Text="Game progress" Grid.Row="2"
           Margin="0,10,0,0"/>
```

A ScrollViewer control displays the game progress, with scrollbars that appear if the text is too big for the window:

3

```
<ScrollViewer Grid.Row="3" FontSize="12"
    Background="White" Foreground="Black" />
```

Here's another TextBlock and ScrollViewer to display the books. The default vertical and horizontal alignment for the ScrollViewer is `Stretch`, and that's going to be really useful. We'll set up the rows and columns so the ScrollViewer controls expand to fit any screen size.

4

```
<TextBlock Text="Books"
       Margin="0,10,0,0" Grid.Row="4"/>

<ScrollViewer FontSize="12" Background="White" Foreground="Black"
          Grid.Row="5" Grid.RowSpan="2" />
```

We used a small 40-pixel column to add space, so the ListBox and Button controls need to go in the third column. The ListBox spans rows 2–6, so we gave it `Grid.Row="1"` and `Grid.RowSpan="5"`—this will also let the ListBox grow to fill the page.

Remember, rows and columns start at zero, so a control in the third column has Grid.Column="2".

5

```
<TextBlock Text="Your hand" Grid.Row="0" Grid.Column="2" />

<ListBox x:Name="cards" Background="White" FontSize="12"
          Height="Auto" Margin="0,0,0,10"
          Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"/>
```

The "Ask for a card" button has its horizontal and vertical alignment set to `Stretch` so that it fills up the cell. The 20-pixel margin at the bottom of the ListBox adds a small gap.

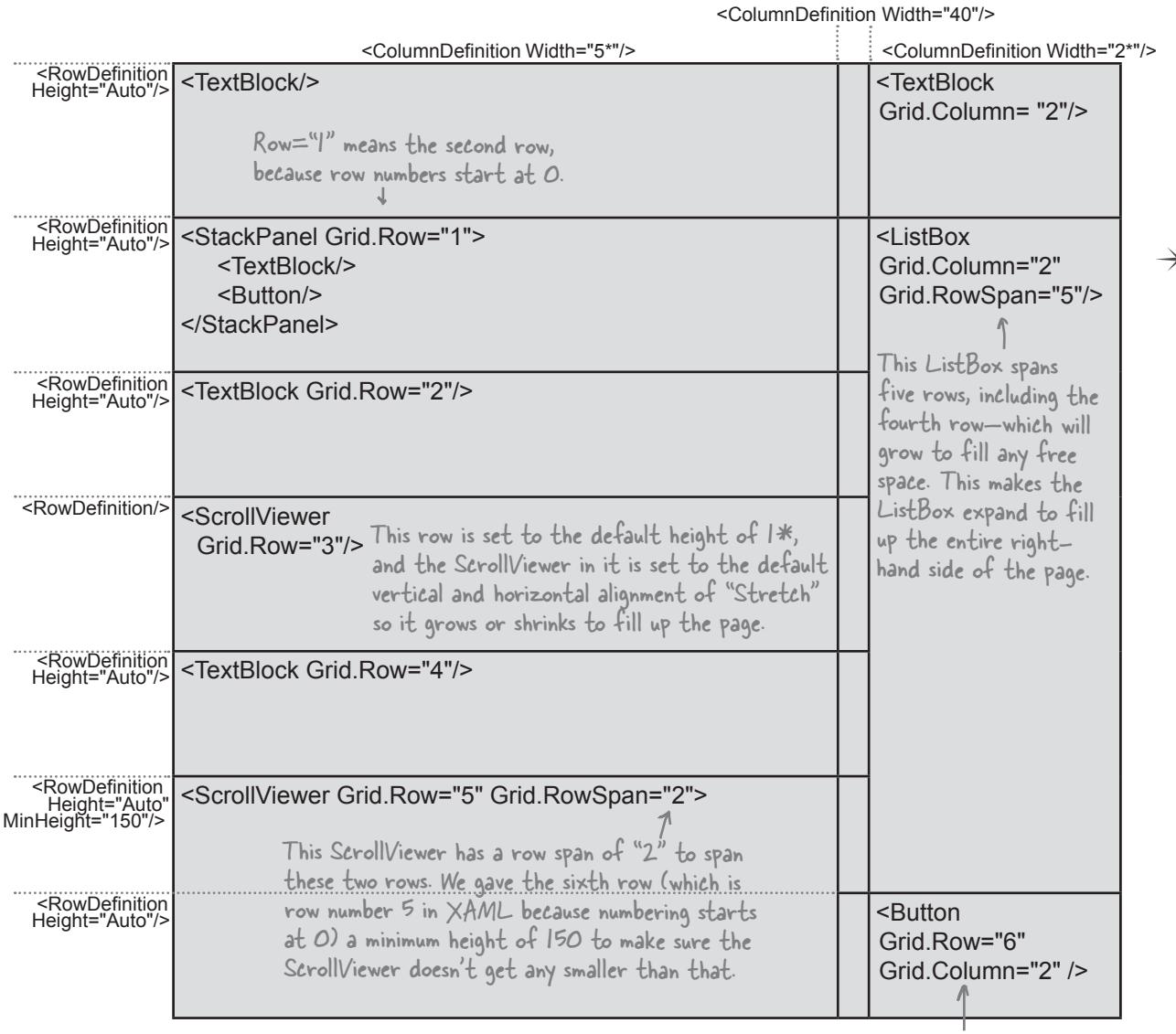
6

```
<Button x:Name="askForACard" Content="Ask for a card"
    HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
    Grid.Row="6" Grid.Column="2"/>
```

We'll finish this grid on the next page →

Rows and columns can resize to match the page size

Grids are very effective tools for laying out windows because they help you design pages that can be displayed on many different devices. Heights or widths that end in * **adjust automatically** to different screen geometries. The Go Fish! window has three columns. The first and third have widths of 5* and 2*, so they will **grow or shrink proportionally** and always keep a 5:2 ratio. The second column has a fixed width of 40 pixels to keep them separated. Here's how the rows and columns for the window are laid out (including the controls that live inside them):



XAML row and column numbering start at 0, so this button's row is 6 and its column is 2 (to skip the middle column). Its vertical and horizontal alignment are set to Stretch so the button takes up the entire cell. The row has a height of Auto, so its height is based on the contents (the button plus its margin).

Here's how the row and column definitions make the window layout work:

```
    <Grid.ColumnDefinitions>
         <ColumnDefinition Width="5*"/>
         <ColumnDefinition Width="40"/>
         <ColumnDefinition Width="2*"/>
     </Grid.ColumnDefinitions>
     <Grid.RowDefinitions>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto" MinHeight="150" />
         <RowDefinition Height="Auto"/>
     </Grid.RowDefinitions>

    You can add the row and column
    definitions above or below the controls in
    the grid. We added them below this time.
  </Grid>

</Window>
```

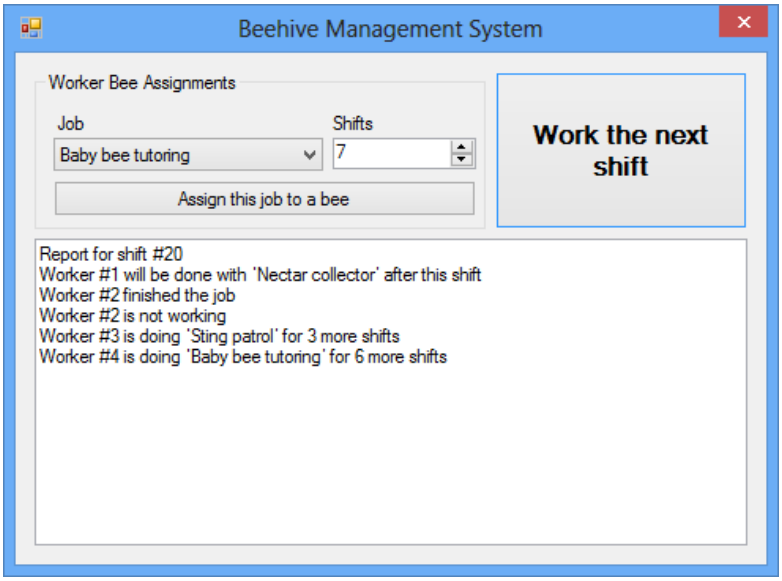
The first column will always be 2.5 times as wide as the third (a 5:2 ratio), with a 40-pixel column to add space between them. The ScrollViewer and ListBox controls that display data have HorizontalAlignment set to "Stretch" to fill up the columns.

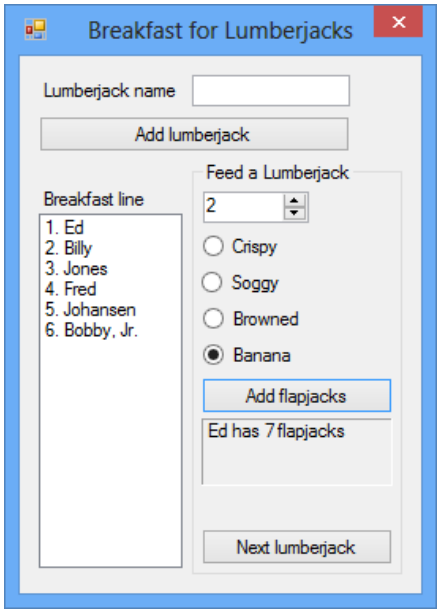
The fourth row has the default height of 1* to make it grow or shrink to fill up any space that isn't taken up by the other rows. The ListBox and first ScrollViewer span this row, **so they will grow and shrink, too.**

Almost all the row heights are set to `Auto`. There's only one row that will grow or shrink, and any control that spans this row will also grow or shrink.

Here's the closing tag for the grid, followed by the closing tab for the window. You'll bring this all together at the end of the chapter when you finish porting the Go Fish! game to a WPF app.

Use XAML to redesign each of these Windows desktop forms as WPF applications. Create a new WPF Application project for each of them, and modify each page by updating or replacing the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.





Use a Border control to draw a border around ScrollViewers.

If you look in the Properties window or look at the IntelliSense window, you'll see that the ScrollViewer control has BorderBrush and BorderThickness properties. This is a little misleading, because these properties don't actually do anything. ScrollViewer is a subclass of ContentControl, and it inherits those properties from ContentControl but doesn't actually do anything with them.

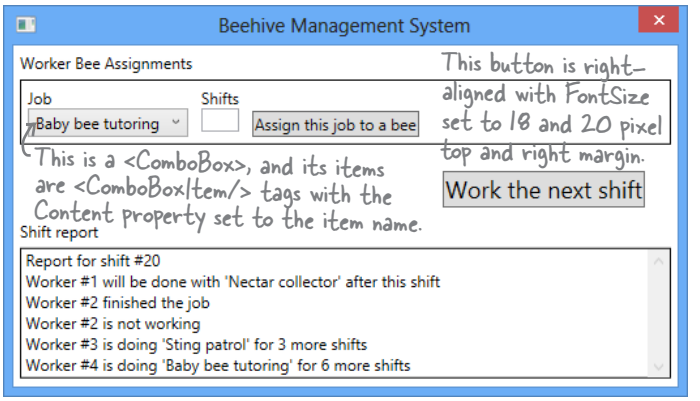
Luckily, there's an easy way to draw a border around a ScrollViewer, or any other control, by using a Border control. Here's XAML code that you can use in the Breakfast for Lumberjacks window:

Use the BorderThickness and BorderBrush properties to set the thickness and color of the border. You can also add a background, round the corners, and make other visual changes.

```
<Border Grid.Row="6" BorderThickness="1"
        Margin="0,5,0,0" BorderBrush="Gray">
    <ScrollViewer Content="Ed has 7 flapjacks"
                  BorderThickness="2" BorderBrush="White" MinHeight="50"/>
</Border>
```

The Border control can contain one other control. If you want to put more than one control inside it, use a StackPanel, Grid, Canvas, or other container.

Use StackPanels to design this window. Its height is set to `300`, its width is `525`, and its ResizeMode property is set to `NoResize`. It uses two `<Border>` controls, one to draw a border around the top StackPanel and one to draw a border around the ScrollViewer.



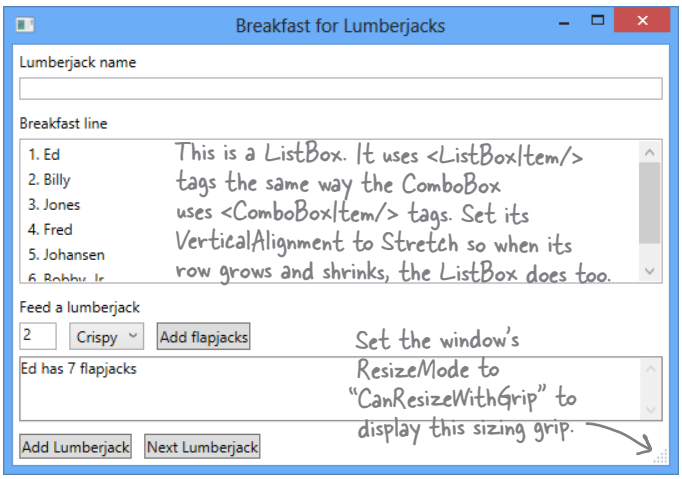

```
<StackPanel Margin="5">
<TextBlock/>
<StackPanel Orientation="Horizontal">
  <StackPanel>
    <TextBlock/>
    <ComboBox>
     <ComboBoxItem/>
     <ComboBoxItem/>
      ... 4 more ...
    </ComboBox>
  </StackPanel>
  <StackPanel>
    <TextBlock/>
    <TextBox/>
  </StackPanel>
  <Button/>
<Button/>
<TextBlock/>
<ScrollViewer/>
</StackPanel>
```

Set the ComboBox control's SelectedIndex property to 0 so it displays the first item.

Use the Content property to add text to this ScrollViewer.
 will add line breaks. Give it a 2-pixel white border using BorderThickness and BorderBrush, and a height of 250.

Use a Grid to design this form. It has seven rows with `height` set to `Auto` so they expand to fit their contents, and one with the default height (which is the same as `1*`) so that row expands with the grid. Use StackPanels to put multiple controls in the same row. Each TextBlock has a 5-pixel margin below it, and the bottom two TextBlocks each have a 10-pixel margin above them. Use the `<Window>` properties

Use these <Window> properties to set the initial and minimum size for the window, then resize the window to make sure they work: Height="400" MinHeight="350" Width="525" MinWidth="300"




```
<Grid Grid.Row="1" Margin="5">
  <TextBlock/>
  <TextBox/>
  <TextBlock/>
  <ListBox VerticalAlignment="Stretch">
    <ListBoxitem/>
    <ListBoxitem/>
    ... 4 more ...
  </ListBox>
  <TextBlock>
  <StackPanel Orientation="Horizontal">
    <TextBox/>
    <ComboBox> ... 4 items ... </ComboBox>
    <Button/>
  </StackPanel>
  <ScrollViewer/>
  <StackPanel Orientation="Horizontal">
    <Button/>
    <Button/>
  </StackPanel>
```

Set this row to the default height 1* and make all the other row heights "Auto" so this row grows and shrinks when the window is resized.

Get your pages to look just like these screenshots by adding **dummy data** to the controls that would normally be filled in using the methods and properties in your classes.

Use XAML to redesign each of these Windows desktop forms as WPF applications. Create a new WPF Application project for each of them, and modify each page by updating or replacing the grid and adding controls. You don't need to get them working. Just create the XAML so they match the screenshots.

```
<Window x:Class="BeehiveManagementSystem.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Beehive Management System"
        Height="300" Width="525"
        ResizeMode="NoResize">

    <StackPanel Margin="5">
        <TextBlock Text="Worker Bee Assignments" Margin="0,0,0,5" />
        <Border BorderThickness="1" BorderBrush="Black">
            <StackPanel Orientation="Horizontal" Margin="5">
                <StackPanel Margin="0,0,10,0">
                    <TextBlock Text="Job"/>
                    <ComboBox SelectedIndex="0" >
                        <ComboBoxItem Content="Baby bee tutoring"/>
                        <ComboBoxItem Content="Egg care"/>
                        <ComboBoxItem Content="Hive maintenance"/>
                        <ComboBoxItem Content="Honey manufacturing"/>
                        <ComboBoxItem Content="Nectar collector"/>
                        <ComboBoxItem Content="Sting patrol"/>
                    </ComboBox>
                </StackPanel>
                <StackPanel>
                    <TextBlock Text="Shifts" />
                    <TextBox/>
                </StackPanel>
                <Button Content="Assign this job to a bee"
                        VerticalAlignment="Bottom" Margin="10,0,0,0" />
            </StackPanel>
        </Border>

        <Button Content="Work the next shift" Margin="0,20,20,0"
                FontSize="18"
                HorizontalAlignment="Right" />

        <TextBlock Text="Shift report" Margin="0,10,0,5"/>
        <Border BorderBrush="Black" BorderThickness="1" Height="100">
            <ScrollViewer
                  Content="
Report for shift #20&#13;
Worker #1 will be done with 'Nectar collector' after this shift&#13;
Worker #2 finished the job&#13;
Worker #2 is not working&#13;
Worker #3 is doing 'Sting patrol' for 3 more shifts&#13;
Worker #4 is doing 'Baby bee tutoring' for 6 more shifts
                        "/>
        </Border>
    </StackPanel>
</Window>
```

Here's the margin we gave you. Specifying just one number (5) sets the top, left, bottom, and right margins to the same value.

Does your XAML code look different from ours? There are many ways to display very similar (or even identical) pages in XAML.

And don't forget that XAML is very flexible about tag order. You can put many of these tags in a different order and still create the same object graph for your window.

This Border control draws a border around the ScrollViewer.

Here's the dummy data we used to populate the shift report. The Content property ignores line breaks—we added them to make the solution easier to read.

```
<Window x:Class="BreakfastForLumberjacks.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Breakfast for Lumberjacks"
        Width="525" Height="400"
        MinWidth="300" MinHeight="350"
        ResizeMode="CanResizeWithGrip" >

    <Grid Grid.Row="1" Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition />
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <TextBlock Text="Lumberjack name" Margin="0,0,0,5" />
        <TextBox Grid.Row="1"/>

        <TextBlock Grid.Row="2" Text="Breakfast line" Margin="0,10,0,5" />
        <ListBox Grid.Row="3" VerticalAlignment="Stretch">
            <ListBoxItem Content="1. Ed"/>
            <ListBoxItem Content="2. Billy"/>
            <ListBoxItem Content="3. Jones"/>
            <ListBoxItem Content="4. Fred"/>
            <ListBoxItem Content="5. Johansen"/>
            <ListBoxItem Content="6. Bobby, Jr."/>
        </ListBox>

        <TextBlock Grid.Row="4" Text="Feed a lumberjack" Margin="0,10,0,5" />
        <StackPanel Grid.Row="5" Orientation="Horizontal">
            <TextBox Text="2" Margin="0,0,10,0" Width="30"/>
            <ComboBox SelectedIndex="0" Margin="0,0,10,0">
                <ComboBoxItem Content="Crispy"/>
                <ComboBoxItem Content="Soggy"/>
                <ComboBoxItem Content="Browned"/>
                <ComboBoxItem Content="Banana"/>
            </ComboBox>
            <Button Content="Add flapjacks" />
        </StackPanel>

        <Border BorderThickness="1"  BorderBrush="Gray" Grid.Row="6" Margin="0,5,0,0">
            <ScrollViewer Content="Ed has 7 flapjacks"
                          BorderThickness="2" BorderBrush="White"
                              MinHeight="50"/>
        </Border>

        <StackPanel Grid.Row="7" Orientation="Horizontal" Margin="0,10,0,0">
            <Button Content="Add Lumberjack" Margin="0,0,10,0" />
            <Button Content="Next Lumberjack" />
        </StackPanel>

    </Grid>
</Window>
```

Here are the Window properties that set the initial window size to 525x400, and set a minimum size of 300x350.

You can set the ResizeMode property to NoResize to prevent all resizing, CanMinimize to allow only minimizing, CanResize to allow all resizing, or CanResizeWithGrip to display a sizing grip in the lower right-hand corner of the window.

Just to be 100% clear, we asked you to add these dummy items **as part of the exercise**, to make the form look like it's being used. You're about to learn how to bind controls like this ListBox to properties in your classes.

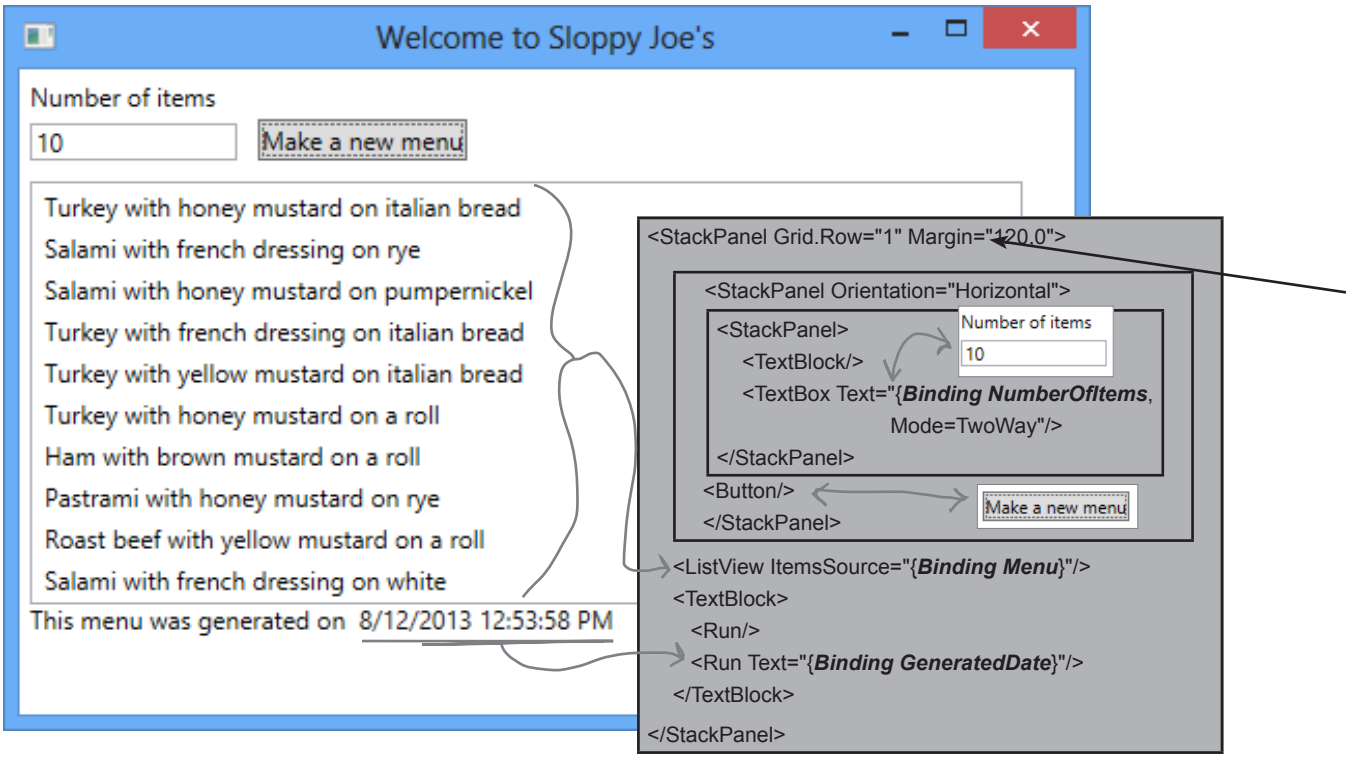
More dummy content...

Use data binding to build Sloppy Joe a better menu

Remember Sloppy Joe from Chapter 4? Well, he's heard that you're becoming an XAML pro, and he wants a WPF app for his sandwich menu. Let's build him one.

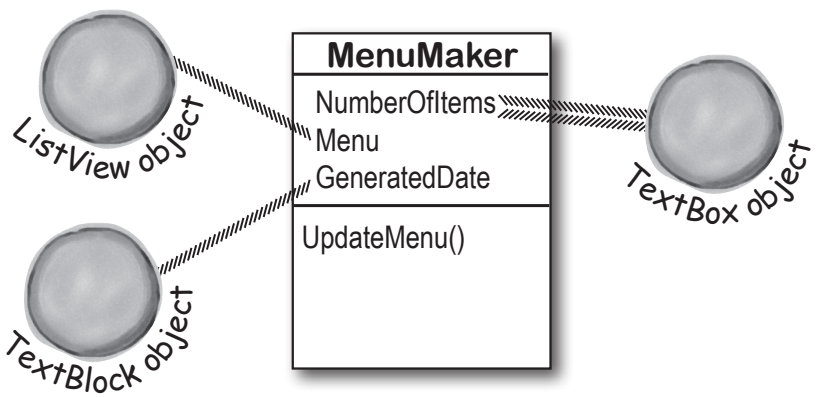
Here's the window we're going to build.

It uses one-way data binding to populate a ListView and a Run inside a TextBlock, and it uses two-way data binding for a TextBox, using one of its `<Run>` tags to do the actual binding.



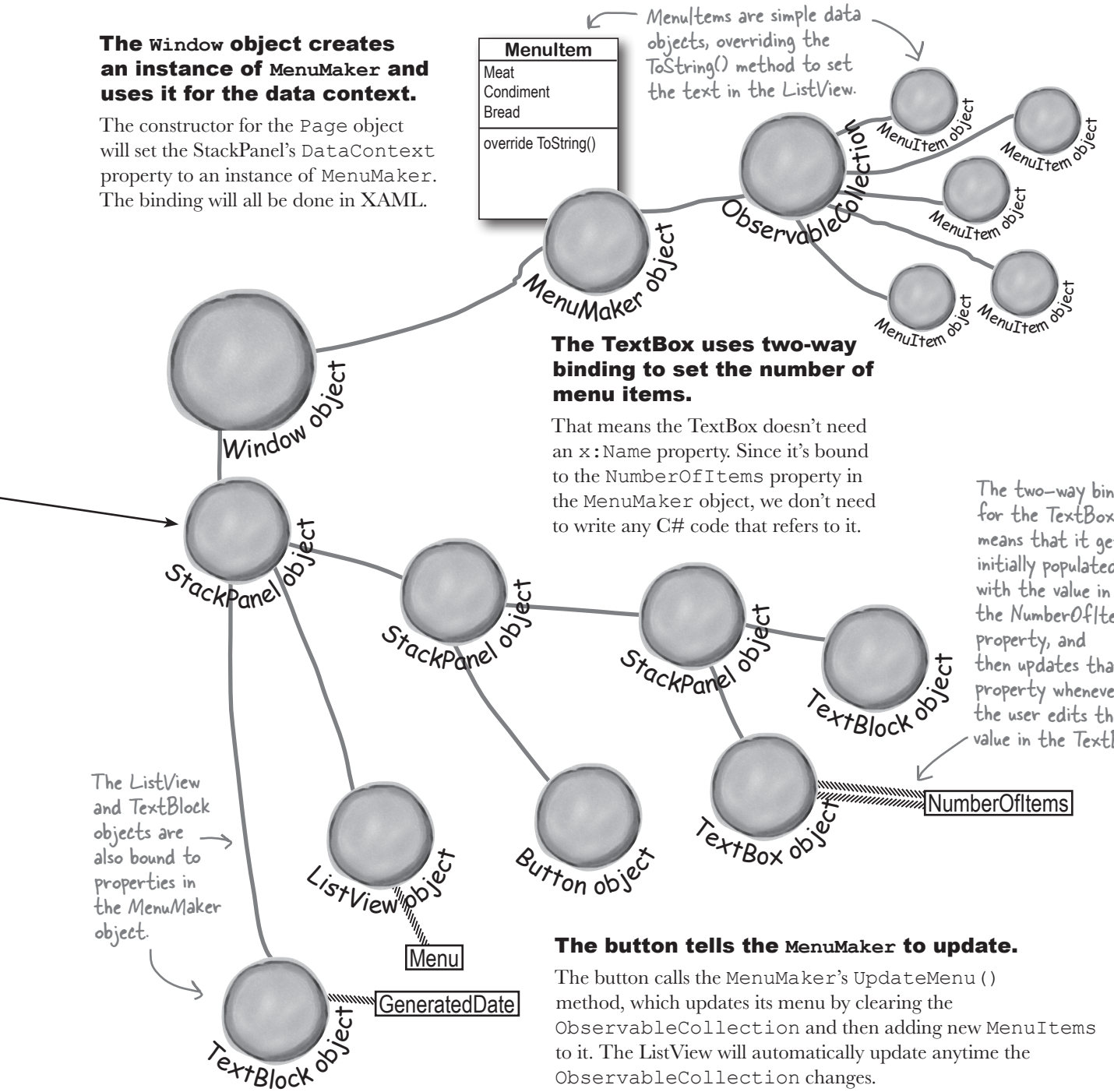
We'll need an object with properties to bind to.

The `Window` object will have an instance of the `MenuMaker` class, which has three public properties: an int called `NumberOfItems`, an `ObservableCollection` of menu items called `Menu`, and a `DateTime` called `GeneratedDate`.



The `Window` object creates an instance of `MenuMaker` and uses it for the data context.

The constructor for the `Page` object will set the StackPanel's `DataContext` property to an instance of `MenuMaker`. The binding will all be done in XAML.



The TextBox uses two-way binding to set the number of menu items.

That means the TextBox doesn't need an `x:Name` property. Since it's bound to the `NumberOfItems` property in the `MenuMaker` object, we don't need to write any C# code that refers to it.

The button tells the `MenuMaker` to update.

The button calls the `MenuMaker`'s `UpdateMenu()` method, which updates its menu by clearing the `ObservableCollection` and then adding new `MenuItems` to it. The ListView will automatically update anytime the `ObservableCollection` changes.

Here's a coding challenge. Based on what you've read so far, how much of the new and improved Sloppy Joe app can you build before you flip the page and see the code for it?

1 **Create the project.**

Create a **new WPF Application project**. You'll keep the default window size. Set the window title to *Welcome to Sloppy Joe's*.

2 **Add the new and improved `MenuMaker` class.**

You've come a long way since Chapter 4. Let's build a well-encapsulated class that lets you set the number of items with a property. You'll create an `ObservableCollection` of `MenuItem` in its constructor, which is updated every time the `UpdateMenu()` is called. That method will also update a `DateTime` property called `GeneratedDate` with a timestamp for the current menu. Add this `MenuMaker` class to your project:

Just right-click on the project name in the Solution Explorer and add a new class, just like you did with other projects.

```
using System.Collections.ObjectModel;

class MenuMaker {
    private Random random = new Random();
    private List<String> meats = new List<String>()
                { "Roast beef", "Salami", "Turkey", "Ham", "Pastrami" };
    private List<String> condiments = new List<String>() { "yellow mustard",
             "brown mustard", "honey mustard", "mayo", "relish", "french dressing" };
    private List<String> breads = new List<String>() { "rye", "white", "wheat",
            "pumpernickel", "italian bread", "a roll" };

    public ObservableCollection<MenuItem> Menu { get; private set; }
    public DateTime GeneratedDate { get; set; }
    public int NumberOfItems { get; set; }

    public MenuMaker() {
        Menu = new ObservableCollection<MenuItem>();
        NumberOfItems = 10;
        UpdateMenu();
    }

    private MenuItem CreateMenuItem() {
        string randomMeat = meats[random.Next(meats.Count)];
        string randomCondiment = condiments[random.Next(condiments.Count)];
        string randomBread = breads[random.Next(breads.Count)];
        return new MenuItem(randomMeat, randomCondiment, randomBread);
    }

    public void UpdateMenu() {
        Menu.Clear();
        for (int i = 0; i < NumberOfItems; i++) {
            Menu.Add(CreateMenuItem());
        }
        GeneratedDate = DateTime.Now;
    }
}
```

You'll need this using line because ObservableCollection<T> is in this namespace.

You'll use data binding to display data from these properties on your page. You'll also use two-way binding to update NumberOfItems.

The new CreateMenuItem() method returns MenuItem objects, not just strings. That will make it easier to change the way items are displayed if we want.

Take a closer look at how this works. It never actually creates a new MenuItem collection. It updates the current one by clearing it and adding new items.

What happens if the NumberOfItems is set to a negative number?

Use DateTime to work with dates

You've already seen the DateTime type that lets you store a date. You can also use it to create and modify dates and times. It has a static property called Now that returns the current time. It also has methods like AddSeconds() for adding and converting seconds, milliseconds, days, etc., and properties like Hour and DayOfWeek to break down the date. How timely!

3 Add the `MenuItem` class.

You've already seen how you can build more flexible programs if you use classes instead of strings to store data. Here's a simple class to hold a menu item—add it to your project, too:

```
class MenuItem {
    public string Meat { get; set; }
    public string Condiment { get; set; }
    public string Bread { get; set; }

    public MenuItem(string meat, string condiment, string bread) {
        Meat = meat;
        Condiment = condiment;
        Bread = bread;
    }

    public override string ToString() {
        return Meat + " with " + Condiment + " on " + Bread;
    }
}
```

The three strings that make up the item are passed into the constructor and held in read-only automatic properties.

Override the ToString() method so the MenuItem knows how to display itself.

4 Build the XAML page.

Here's the screenshot. Can you build it using StackPanels? The TextBox has a width of 100. The bottom TextBlock has the style `BodyTextStyle`, and it has two `<Run>` tags (the second one just holds the date).

Don't add dummy data this time. We'll let data binding do that for us.

Can you build this page on your own just from the screenshot before you see the XAML?

5 Add object names and data binding to the XAML.

Here's the XAML that gets added to *MainWindow.xaml*. We used a StackPanel to lay it out, so you can replace the opening `<Grid>` and closing `</Grid>` tags with the XAML below. We named the button `newMenu`. Since we used data binding of the ListView, TextBlock, and TextBox, we didn't need to give them names. *(Here's a shortcut. We didn't even really need to name the button; we did it just to get the IDE to automatically add an event handler named newMenu_Click when we double-clicked it in the IDE. Try it out!)*

```
<StackPanel Margin="5" x:Name="pageLayoutStackPanel">
    <StackPanel Orientation="Horizontal" Margin="0,0,0,10">
        <StackPanel Margin="0,0,10,0">
            <TextBlock Text="Number of items" Margin="0,0,0,5" />
            <TextBox Width="100" HorizontalAlignment="Left"
                Text="{Binding NumberOfItems, Mode=TwoWay}" />
        </StackPanel>
        <Button x:Name="newMenu" VerticalAlignment="Bottom"
                Click="newMenu_Click" Content="Make a new menu"/>
    </StackPanel>
    <ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0" />
    <TextBlock>
        <Run Text="This menu was generated on " />
        <Run Text="{Binding GeneratedDate}"/>
    </TextBlock>
</StackPanel>
```

Here's that ListView control. Try swapping it out for ListBox to see how it changes your window.

We need two-way data binding to both get and set the number of items with the TextBox.

This is where <Run> tags come in handy. You can have a single TextBlock but bind only part of its text.

6 Add the code-behind for the page to MainWindow.xaml.cs.

The page constructor creates the menu collection and the `MenuMaker` instance and sets the data contexts for the controls that use data binding. It also needs a `MenuMaker` field called `menuMaker`.

```
MenuMaker menuMaker = new MenuMaker();

public MainWindow() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

Your main window's class in MainWindow.xaml.cs gets a MenuMaker field, which is used as the data context for the StackPanel that contains all the bound controls.

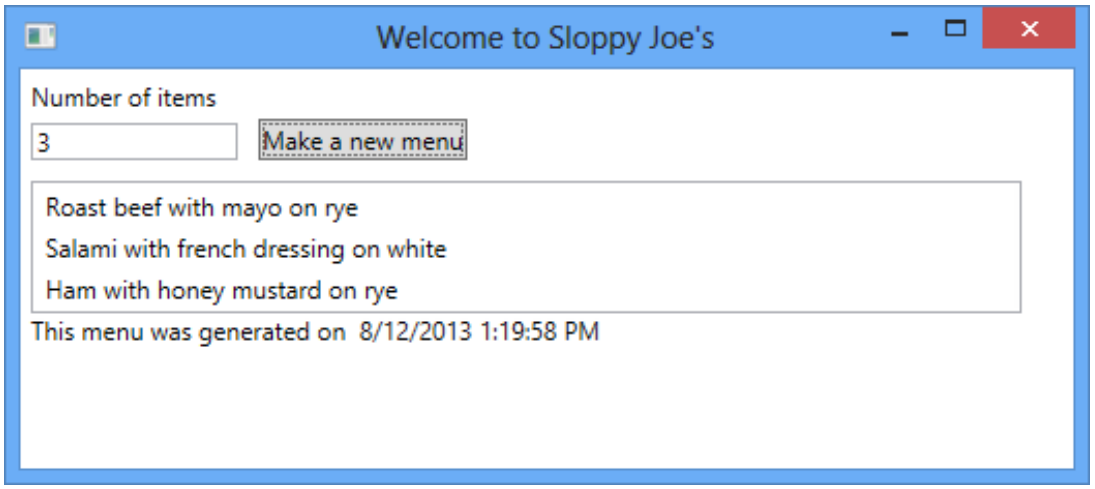
You just need to set the data context for the outer StackPanel. It will pass that data context on to all the controls contained inside it.

Finally, double-click on the button to generate a method stub for its `Click` event handler. Here's the code for it—it just updates the menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    menuMaker.UpdateMenu();
}
```

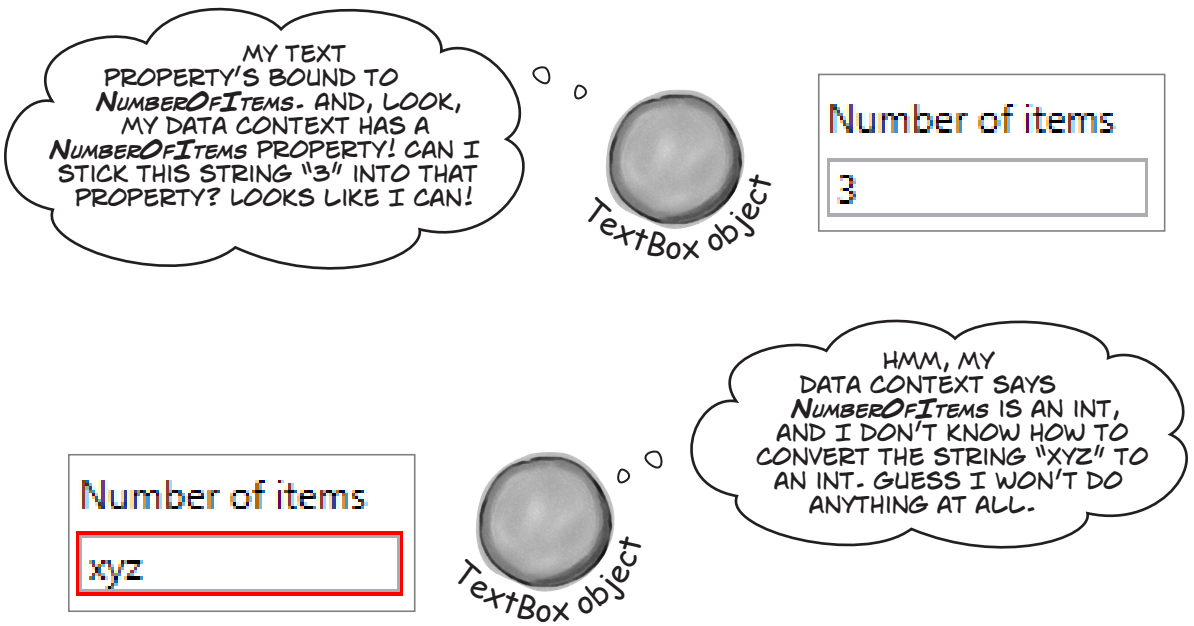
There's an easy way to rename an event handler so that it updates XAML and C# code at the same time. Flip to leftover #8 in Appendix I to learn more about the refactoring tools in the IDE.

Now run your program! Try changing the TextBox to different values. Set it to 3, and it generates a menu with three items:



Now you can play with binding to see just how flexible it is. Try entering "xyz" or no data at all into the TextBox. Nothing happens! When you enter data into the TextBox, you're giving it a string. The TextBox is pretty smart about what it does with that string. It knows that its binding path is `NumberOfItems`, so it looks in its data context to see if there are any properties with that name, and then does its best to convert the string to whatever that property's type is.

Keep your eye on the generated date. It's not updating, even though the menu updates. Hmm, maybe there's still something we need to do.



Use static resources to declare your objects in XAML

When you build a page with XAML, you're creating an object graph with objects like StackPanel, Grid, TextBlock, and Button. And you've seen that there's no magic or mystery to any of that—when you add a `<TextBox>` tag to your XAML, then your page object will have a TextBox field with a reference to an instance of TextBox. And if you give it a name using the `x:Name` property, your code-behind C# code can use that name to access the TextBox.

You can do exactly the same thing to create instances of *almost any* class and store them as fields in your page by adding a **static resource** to your XAML. And data binding works particularly well with static resources, especially when you combine it with the visual designer in the IDE. Let's go back to your program for Sloppy Joe and move the `MenuMaker` to a static resource.

1 DELETE THE MENUMAKER FIELD FROM THE CODE-BEHIND.

You're going to be setting up the `MenuMaker` class and the data context in the XAML, so delete these lines from your C# code:

```
MenuMaker menuMaker = new MenuMaker();

public MainWindow() {
    this.InitializeComponent();

    pageLayoutStackPanel.DataContext = menuMaker;
}
```

When you use XAML to add a static resource to a Window, you can access it using its FindResource() method.

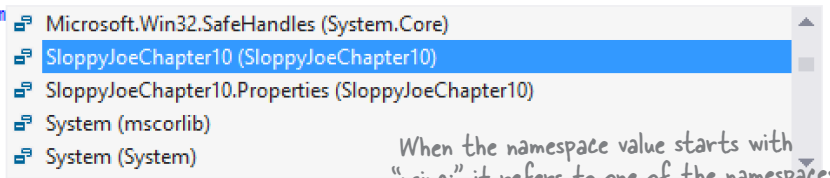
2 ADD YOUR PROJECT'S NAMESPACE TO THE XAML.

Look at the top of the XAML code for your window, and you'll see that the opening tag has a set of `xmlns` properties. Each of these properties defines a namespace:

```
<Window x:Class="SloppyJoeChapter10.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Welcome to Sloppy Joe's" Height="350" Width="525">
```

Start adding a new xmlns property:

```
<Window x:Class="SloppyJoeChapter10.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local=""
        Title="Welcom
```



When the namespace value starts with "using:" it refers to one of the namespaces in the project. It can also start with "http://" to refer to a standard XAML namespace.

Here's what you'll end up with:

This is an XML namespace property. It consists of "xmlns:" followed by an identifier, in this case "local".

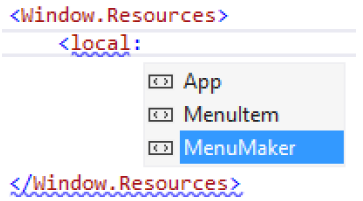
```
xmlns:local="using:SloppyJoeChapter10"
```

You'll use this identifier to create objects in your project's namespace.

Since we named our app SloppyJoeChapter10, the IDE created this namespace for us. Find the namespace that corresponds to your app, because that's where your `MenuMaker` lives.

3 ADD THE STATIC RESOURCE TO YOUR XAML AND SET THE DATA CONTEXT.

Add a `<Window.Resources>` tag to the top of the XAML (just under the opening tag), and add a closing `</Window.Resources>` tag for it. Then type **`<local:`** between them to pop up an IntelliSense window:



You can add static resources only if their classes have parameterless constructors. This makes sense! If the constructor has a parameter, how would the XAML page know what arguments to pass to it?

The window shows all the classes in the namespace that you can use. Choose `MenuMaker`. Then give it the **resource key** `menuMaker` using the **`x:Key`** XAML property:

```
<local:MenuMaker x:Key="menuMaker"/>
```

Now your page has a static `MenuMaker` resource with the key `menuMaker`.

4 SET THE DATA CONTEXT FOR YOUR STACKPANEL AND ALL OF ITS CHILDREN.

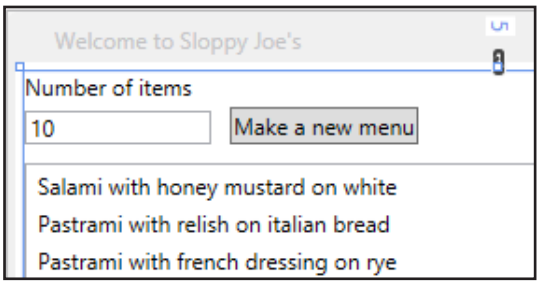
Then go to the outermost StackPanel and set its `DataContext` property:

```
<StackPanel Margin="5"
      DataContext="{StaticResource ResourceKey=menuMaker}">
```

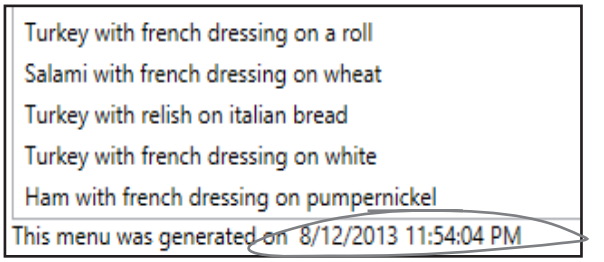
Finally, modify the button's Click event handler to find the static resource and method to update the menu:

```
private void newMenu_Click(object sender, RoutedEventArgs e) {
    MenuMaker menuMaker = FindResource("menuMaker") as MenuMaker;
    menuMaker.UpdateMenu();
}
```

Your program will still work, just like before. But did you notice what happened in the IDE when you added the data context to the XAML? As soon as you added it, the IDE created an instance of MenuMaker and used its properties to populate all the controls that were bound to it. You got a menu generated immediately, right there in the designer—before you even ran your program. Neat!



The menu shows up in the designer immediately, even before you run your program.



Hmm, something's not quite right. It updates the menu items when the button is clicked, but the date doesn't change. What's going on?

Use a data template to display objects

When you show items in a list, you're showing contents of ListViewItem (which you use for ListViews), ListBoxItem, or ComboBoxItem controls, which get bound to objects in an `ObservableCollection`. Each ListViewItem in the Sloppy Joe menu generator is bound to a `MenuItem` object in its `Menu` collection. The ListViewItem objects call the `MenuMaker` objects' `ToString()` methods by default, but you can use a **data template** that uses data binding to display data from the bound object's properties.

Modify the `<ListView>` tag to add a basic data template. It uses the basic `{Binding}` to call the item's `ToString()`.

Leave the ListView tag intact, but replace /> with > and add a closing </ListView> tag at the bottom. Then add the ListView.ItemTemplate tag to contain the data template.

This is a really basic data template, and it looks just like the default one used to display the ListView Items.

```
<ListView ItemsSource="{Binding Menu}" Margin="0,0,20,0">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding}"/>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Adding a {Binding} without a path just calls the ToString() method of the bound object.

Change your data template to add some color to your menu.

Replace the <DataTemplate>, but leave the rest of the ListView intact.

You can bind individual Run tags. You can change each tag's color, font, and other properties, too.

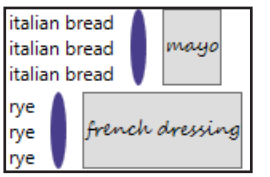
```
<DataTemplate>
    <TextBlock>
        <Run Text="{Binding Meat}" Foreground="Blue"/><Run Text=" on "/>
        <Run Text="{Binding Bread}" FontWeight="Light"/><Run Text=" with "/>
        <Run Text="{Binding Condiment}" Foreground="Red" FontWeight="ExtraBold"/>
    </TextBlock>
</DataTemplate>
```

Salami on pumpernickel with mayo
Pastrami on rye with relish
Salami on white with brown mustard

Go crazy! The data template can contain any controls you want.

```
<DataTemplate>
    <StackPanel Orientation="Horizontal">
        <StackPanel>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
            <TextBlock Text="{Binding Bread}"/>
        </StackPanel>
        <Ellipse Fill="DarkSlateBlue" Height="Auto" Width="10" Margin="10,0"/>
        <Button Content="{Binding Condiment}" FontFamily="Segoe Script"/>
    </StackPanel>
</DataTemplate>
```

The DataTemplate object's Content property can hold only one object, so if you want multiple controls in your data template, you'll need a container like StackPanel.



there are no Dumb Questions

Q: So I can use a StackPanel or a Grid to lay out my page. I can use XAML static resources, or I can use fields in code-behind. I can set properties on controls, or I can use data binding. Why are there so many ways to do the same things?

A: Because C# and XAML are extremely flexible tools for building apps. That flexibility makes it possible to design very detailed pages that work on many different devices and displays. This gives you a very large toolbox that you can use to get your pages *just right*. So don't look at it as a confusing set of choices; look at it as many different options that you can choose from.

Q: I'm still not clear on how static resources work. What happens when I add a tag inside `<Window.Resources>`?

A: When you add that tag, it updates the Window object and adds static resources. In this case, it **created an instance** of MenuMaker and added it to the Window object's resources. The Window object contains a dictionary called Resources, and if you use the debugger to explore the Window object after you add the tag you can find that it contains an instance of MenuMaker. When you declared the resource, you used `x:Key` to assign the resource a key. That allowed you to **use that key to look up your MenuMaker object** in the window's static resources with the FindResource() method.

Q: I used `x:Key` to set my MenuMaker resource's key. But earlier in the chapter, I used `x:Name` to give names to my controls. What's the difference? Why did I have to use FindResources() to look up the MenuMaker object—couldn't I give it a name instead?

A: When you add a control to a WPF window, it actually adds a field to the Window object that's created by the XAML. When you use the `x:Name` property, you give it a name that you can use in your code. If you don't give it a name, the control object is still created as part of the Window object's graph. However, if you give it a name, then the XAML object is **given a field with that name** with a reference to that control. You can see this in your code by putting a breakpoint in the button's event handler and adding `newMenu` to the Watch window. You'll see that it refers to a System.Windows.Controls.Button object whose Content property is set to "Make a new menu."

Resources are treated differently: they're **added to a dictionary in the Window object**. The FindResource() method uses the key specified in the `x:Key` markup. Set the same breakpoint and try adding `this.Resources["menuMaker"]` to the Watch window. This time, you'll see a reference to your MenuMaker object, because you're looking it up in the Resources dictionary.

Q: Does my binding path have to be a string property?

A: No, you can bind a property of any type. If it can be converted between the source and property types, then the binding will work. If not, the data will be ignored. And remember, not all properties on your controls are text, either. Let's say you've got a bool in your data context called `EnableMyObject`. You can bind it to any Boolean property, like `IsEnabled`. This will enable or disable the control based on the value of the `EnableMyObject` property:

```
IsEnabled="{Binding EnableMyObject}"
```

Of course, if you bind it to a text property it'll just print `True` or `False` (which, if you think about it, makes perfect sense).

Q: Why did the IDE display the data in my form when I added the static resource and set the data context in XAML, but not when I did it in C#?

A: Because the IDE understands your XAML, which has all the information that it needs to create the objects to render your page. As soon as you added the `MenuMaker` resource to your XAML code, the IDE created an instance of `MenuMaker`. But it couldn't do that from the `new` statement in its constructor, because there could be many other statements in the constructor, and they would need to be run. The IDE runs the code-behind C# code only when the program is executed. But if you add a static resource to the page, the IDE will create it, just like it creates instances of TextBlock, StackPanel, and the other controls on your page. It sets the controls' properties to show them in the designer, so when you set up the data context and binding paths, those got set as well, and your menu items showed up in the IDE's designer.

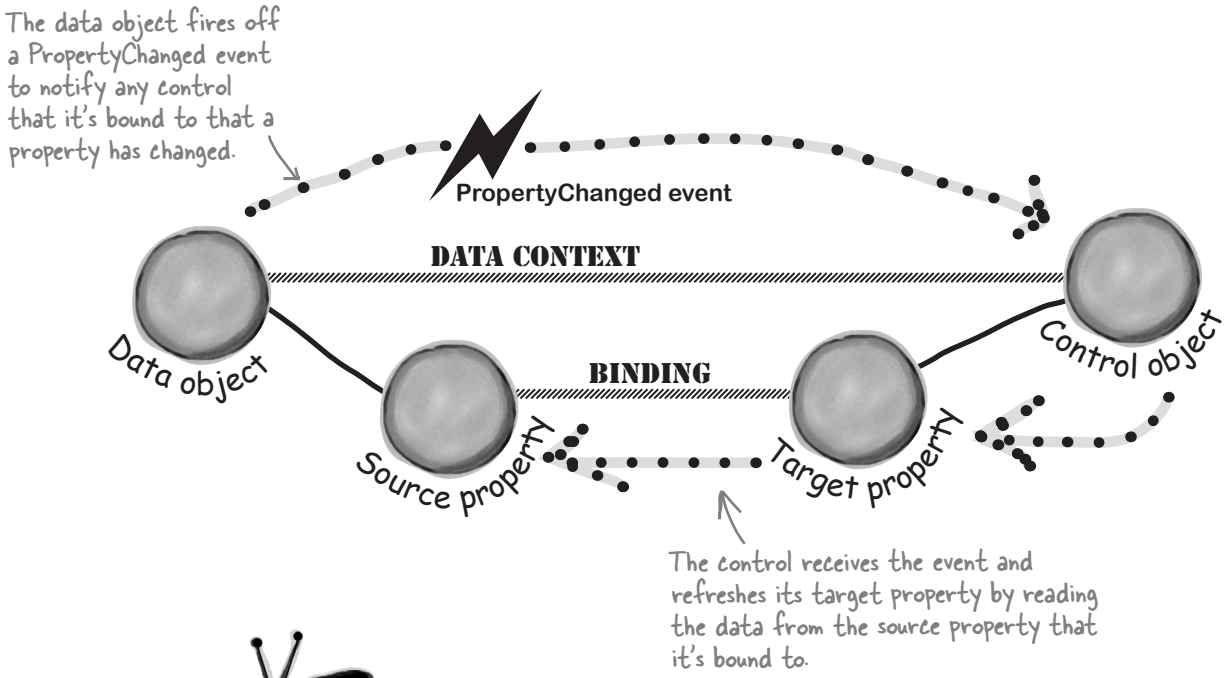
The static resources in your page are instantiated when the page is first loaded and can be used at any time by the objects in the application.

The name "static resource" is a little misleading. Static resources are definitely created for each instance; they're not static fields!

INotifyPropertyChanged lets bound objects send updates

When the `MenuMaker` class updates its menu, the ListView that's bound to it gets updated. But the `MenuMaker` updates the `GeneratedDate` property at the same time. Why doesn't the TextBlock that's bound to it get updated, too? The reason is that every time an `ObservableCollection` changes, it **fires off an event** to tell any bound control that its data has changed. This is just like how a `Button` control raises a `Click` event when it's clicked, or a `Timer` raises a `Tick` event when its interval elapses. Whenever you add, remove, or delete items from an `ObservableCollection`, it raises an event.

You can make your data objects notify their target properties and bound controls that data has changed, too. All you need to do is **implement the `INotifyPropertyChanged` interface**, which contains a single event called `PropertyChanged`. Just fire off that event whenever a property changes, and watch your bound controls update themselves automatically.



Watch it!

Collections work *almost* the same way as data objects.

The ObservableCollection<T> object doesn't actually implement INotifyPropertyChanged. Instead, it implements a closely related interface called **INotifyCollectionChanged** *that fires off a CollectionChanged event instead of a PropertyChanged event. The control knows to look for this event because ObservableCollection implements the INotifyCollectionChanged interface. Setting a ListView's DataContext to an INotifyCollectionChanged object will cause it to respond to these events.*

Modify MenuMaker to notify you when the GeneratedDate property changes

`INotifyPropertyChanged` is in the `System.ComponentModel` namespace, so start by adding this `using` statement to the top of the `MenuMaker` class file:

```
using System.ComponentModel;
```

Update the `MenuMaker` class to implement `INotifyPropertyChanged`, and then use the IDE to automatically implement the interface:

```
class MenuMaker : INotifyPropertyChanged
{
```

Implement interface 'INotifyPropertyChanged'
Explicitly implement interface 'INotifyPropertyChanged'

This will be a little different from what you saw in chapters 7 and 8. It won't add any methods or properties. Instead, it will add an event:

```
public event PropertyChangedEventHandler PropertyChanged;
```

Next, add this `OnPropertyChanged()` method, which you'll use to raise the `PropertyChanged` event.

```
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This is a standard .NET pattern for raising events.

Now all you need to do to notify a bound control that a property is changed is to call `OnPropertyChanged()` with the name of the property that's changing. We want the TextBlock that's bound to `GeneratedDate` to refresh its data every time the menu is updated, so all we need to do is add one line to the end of `UpdateMenu()`:

```
public void UpdateMenu() {
    Menu.Clear();
    for (int i = 0; i < NumberOfItems; i++) {
        Menu.Add(CreateMenuItem());
    }
    GeneratedDate = DateTime.Now;

    OnPropertyChanged("GeneratedDate");
}
```

Now the date should change when you generate a menu.

This is the first time you're raising events.

You've been writing event handler methods since Chapter 1, but this is the first time you're firing an event. You'll learn all about how this works and what's going on in Chapter 15. For now, all you need to know is that an interface can include an event, and that your `OnPropertyChanged()` method is following a standard C# pattern for raising events to other objects.

Don't forget to implement `INotifyPropertyChanged`.

Data binding works only when the controls implement that interface. If you leave `: INotifyPropertyChanged` out of the class declaration, your bound controls won't get updated—even if the data object fires PropertyChanged events.

Finish porting the Go Fish! game to a WPF application. You'll need to modify the XAML from earlier in this chapter to add data binding, copy all the classes and enums from the Go Fish! game in Chapter 8 (or download them from our website), and update the `Player` and `Game` classes.

1 Add the existing class files and change their namespace to match your app.

Add these files to your project from the Chapter 8 Go Fish! code: *Values.cs*, *Suits.cs*, *Card.cs*, *Deck.cs*, *CardComparer_bySuit.cs*, *CardComparer_byValue.cs*, *Game.cs*, and *Player.cs*. You can use the Add Existing Item option in the Solution Explorer, but you'll need to **change the namespace** in each of them to match your new projects (just like you did with multipart projects earlier in the book).

Try building your project. You should get errors in *Game.cs* and *Player.cs* that look like this:

```
1  The type or namespace name 'Forms' does not exist in the namespace 'System.Windows' (are you missing an assembly reference?)
2  The type or namespace name 'TextBox' could not be found (are you missing a using directive or an assembly reference?)
3  The type or namespace name 'TextBox' could not be found (are you missing a using directive or an assembly reference?)
```

2 Remove all references to WinForms classes and objects; add `using` lines to `Game`.

You're not in the WinForms world anymore, so delete `using System.Windows.Forms;` from the top of *Game.cs* and *Player.cs*. You'll also need to remove all mentions of TextBox. You'll need to modify the `Game` class to use `INotifyPropertyChanged` and `ObservableCollection<T>`, so add these `using` lines to the top of *Game.cs*:

```
using System.ComponentModel;
using System.Collections.ObjectModel;
```

3 Add an instance of `Game` as a static resource and set up the data context.

Modify your XAML to add an instance of `Game` as a static resource and use it as the data context for the grid that contains the Go Fish! page you built earlier in the chapter. Here's the XAML for the static resource: **`<local:Game x:Key="game"/>`** — and you're going to need a new constructor because you can include only resources that have parameterless constructors:

```
public Game() {
    PlayerName = "Ed";
    Hand = new ObservableCollection<string>();
    ResetGame();
}
```

Make sure you add the `<Window.Resources>` section to the top of your XAML, and you'll also need to add the `xmlns:local` tag, exactly like you did on pages 522 and 523.

4 Add public properties to the `Game` class for data binding.

Here are the properties you'll be binding to properties of the controls in the page:

```
public bool GameInProgress { get; private set; }
public bool GameNotStarted { get { return !GameInProgress; } }
public string PlayerName { get; set; }
public ObservableCollection<string> Hand { get; private set; }
public string Books { get { return DescribeBooks(); } }
public string GameProgress { get; private set; }
```

5 Use binding to enable or disable the TextBox, ListBox, and Buttons.

You want the "Your Name" TextBox and the "Start the game!" Button to be enabled only when the game is not started, and you want the "Your hand" ListBox and "Ask for a card" Button to be enabled only when the game is in progress. You'll add code to the `Game` class to set the `GameInProgress` property. Have a look at the `GameNotStarted` property. Figure out how it works, and then add the following property bindings to the TextBox, ListBox, and two Buttons:

You'll need two of each of these.

```
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
IsEnabled="{Binding GameInProgress}"   IsEnabled="{Binding GameNotStarted}"
```

6 Modify the `Player` class so it tells the `Game` to display the game's progress.

The WinForms version of the `Player` class takes a TextBox as a parameter for its constructor. Change that to take a reference to the `Game` class and store it in a private field. (Look at the `StartGame()` method below to see how this new constructor is used when adding players.) Find the lines that use the TextBox reference and replace them with calls to the `Game` object's `AddProgress()` method.

7 Modify the `Game` class.

Change the `PlayOneRound()` method so that it's void instead of returning a Boolean, and have it use the `AddProgress()` method instead of the TextBox to display progress. If a player won, display that progress, reset the game, and return. Otherwise, refresh the `Hand` collection and describe the hands.

You'll also need to add/update these four methods and figure out what they do and how they work.

```
public void StartGame() {
    ClearProgress();
    GameInProgress = true;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    Random random = new Random();
    players = new List<Player>();
    players.Add(new Player(PlayerName, random, this));
    players.Add(new Player("Bob", random, this));
    players.Add(new Player("Joe", random, this));
    Deal();
    players[0].SortHand();
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
    OnPropertyChanged("Books");
}

  public void ClearProgress() {
      GameProgress = String.Empty;
      OnPropertyChanged("GameProgress");
  }
```

```
public void AddProgress(string progress)
{
    GameProgress = progress +
           Environment.NewLine +
           GameProgress;
    OnPropertyChanged("GameProgress");
}
```

You'll **also** need to implement the `INotifyPropertyChanged` interface and add the same `OnPropertyChanged()` method that you used in the `MenuMaker` class. The updated methods use it, and your modified `PullOutBooks()` method will also use it.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}
```

Here's all the code-behind that you had to write.

```
Game game;

public MainWindow() {
    InitializeComponent();
    game = this.FindResource("game") as Game;
}
private void startButton_Click(object sender, RoutedEventArgs e) {
    game.StartGame();
}
private void askForACard_Click(object sender, RoutedEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
private void cards_MouseDoubleClick(object sender, MouseButtonEventArgs e) {
    if (cards.SelectedIndex >= 0)
        game.PlayOneRound(cards.SelectedIndex);
}
```

These are the changes needed for the `Player` class:

```
class Player {
    private string name;
    public string Name { get { return name; } }
    private Random random;
    private Deck cards;
    private Game game;
    public Player(String name, Random random, Game game) {
        this.name = name;
        this.random = random;
        this.game = game;
        this.cards = new Deck(new Card[] { });
        game.AddProgress(name + " has just joined the game");
    }
    public Deck DoYouHaveAny(Values value)
    {
        Deck cardsIHave = cards.PullOutValues(value);
        game.AddProgress(Name + " has " + cardsIHave.Count + " " + Card.Plural(value));
        return cardsIHave;
    }
    public void AskForACard(List<Player> players, int myIndex, Deck stock, Values value) {
        game.AddProgress(Name + " asks if anyone has a " + value);
        int totalCardsGiven = 0;
        for (int i = 0; i < players.Count; i++) {
            if (i != myIndex) {
                Player player = players[i];
                Deck CardsGiven = player.DoYouHaveAny(value);
                totalCardsGiven += CardsGiven.Count;
                while (CardsGiven.Count > 0)
                    cards.Add(CardsGiven.Deal());
            }
        }
        if (totalCardsGiven == 0) {
            game.AddProgress(Name + " must draw from the stock.");
            cards.Add(stock.Deal());
        }
    }
    // ... the rest of the Player class is the same ...
```

These are the changes needed for the XAML:

```
<Grid Margin="10" DataContext="{StaticResource ResourceKey=game}">

    <TextBlock Text="Your Name" />

    <StackPanel Orientation="Horizontal" Grid.Row="1">
        <TextBox x:Name="playerName" FontSize="12" Width="150"
                 Text="{Binding PlayerName, Mode=TwoWay}"
                 IsEnabled="{Binding GameNotStarted}" />
        <Button x:Name="startButton" Margin="5,0" IsEnabled="{Binding GameNotStarted}"
                Content="Start the game!" Click="startButton_Click"/>
    </StackPanel>

    <TextBlock Text="Game progress" Grid.Row="2" Margin="0,10,0,0"/>

    <ScrollViewer Grid.Row="3" FontSize="12" Background="White" Foreground="Black"
                  Content="{Binding GameProgress}" />

    <TextBlock Text="Books" Margin="0,10,0,0" Grid.Row="4"/>

    <ScrollViewer FontSize="12" Background="White" Foreground="Black"
                  Grid.Row="5" Grid.RowSpan="2"
                  Content="{Binding Books}" />

    <TextBlock Text="Your hand" Grid.Row="0" Grid.Column="2" />

    <ListBox x:Name="cards" Background="White" FontSize="12"
             Height="Auto" Margin="0,0,0,10"
             Grid.Row="1" Grid.RowSpan="5" Grid.Column="2"
             ItemsSource="{Binding Hand}" IsEnabled="{Binding GameInProgress}"
             MouseDoubleClick="cards_MouseDoubleClick" />

    <Button x:Name="askForACard" Content="Ask for a card"
        HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
        Grid.Row="6" Grid.Column="2"
        Click="askForACard_Click" IsEnabled="{Binding GameInProgress}" />

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="5*"/>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="2*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto" MinHeight="150" />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

</Grid>
```

The data context for the grid is the Game class, since all of the binding is to properties on that class.

The TextBox has a two-way binding to PlayerName.

Here's the Click event handler for the Start button.

The Game Progress and Books ScrollViewers bind to the Progress and Books properties.

The IsEnabled property enables or disables the control. It's a Boolean property, so you can bind it to a Boolean property to turn the control on or off based on that property.

Here's everything that changed in the `Game` class, including the code we gave you with the instructions.

```
using System.ComponentModel;
using System.Collections.ObjectModel;

class Game : INotifyPropertyChanged {
    private List<Player> players;
    private Dictionary<Values, Player> books;
    private Deck stock;

    public bool GameInProgress { get; private set; }
    public bool GameNotStarted { get { return !GameInProgress; } }
    public string PlayerName { get; set; }
    public ObservableCollection<string> Hand { get; private set; }
    public string Books { get { return DescribeBooks(); } }
    public string GameProgress { get; private set; }

    public Game() {
        PlayerName = "Ed";
        Hand = new ObservableCollection<string>();
        ResetGame();
    }

    public void AddProgress(string progress) {
        GameProgress = progress + Environment.NewLine + GameProgress;
        OnPropertyChanged("GameProgress");
    }

    public void ClearProgress() {
        GameProgress = String.Empty;
        OnPropertyChanged("GameProgress");
    }

    public void StartGame() {
        ClearProgress();
        GameInProgress = true;
        OnPropertyChanged("GameInProgress");
        OnPropertyChanged("GameNotStarted");
        Random random = new Random();
        players = new List<Player>();
        players.Add(new Player(PlayerName, random, this));
        players.Add(new Player("Bob", random, this));
        players.Add(new Player("Joe", random, this));
        Deal();
        players[0].SortHand();
        Hand.Clear();
        foreach (String cardName in GetPlayerCardNames())
            Hand.Add(cardName);
        if (!GameInProgress)
            AddProgress(DescribePlayerHands());
        OnPropertyChanged("Books");
    }
```

You need these lines for INotifyPropertyChanged and ObservableCollection.

These properties are used by the XAML data binding.

Here's the new Game constructor. We create only one collection and just clear it when the game is reset. If we created a new object, the form would lose its reference to it, and the updates would stop.

These methods make the game progress data binding work. New lines are added to the top so the old activity scrolls off the bottom of the ScrollViewer.

Here's the StartGame() method we gave you. It clears the progress, creates the players, deals the cards, and then updates the progress and books.

Every program you've written in the book so far can be adapted or rewritten as a WPF application using XAML. But there are so many ways to write them, and that's especially true when you're using XAML! That's why we gave you so much of the code for this exercise.

This used to return a Boolean value so the form could update its progress. Now it just needs to call AddProgress, and data binding will take care of the updating for us.

```
public void PlayOneRound(int selectedPlayerCard) {
    Values cardToAskFor = players[0].Peek(selectedPlayerCard).Value;
    for (int i = 0; i < players.Count; i++) {
        if (i == 0)
            players[0].AskForACard(players, 0, stock, cardToAskFor);
        else
            players[i].AskForACard(players, i, stock);
        if (PullOutBooks(players[i])) {
            AddProgress(players[i].Name + " drew a new hand");
            int card = 1;
            while (card <= 5 && stock.Count > 0) {
                players[i].TakeCard(stock.Deal());
                card++;
            }
        }
        OnPropertyChanged("Books");
        players[0].SortHand();
        if (stock.Count == 0) {
            AddProgress("The stock is out of cards. Game over!");
            AddProgress("The winner is... " + GetWinnerName());
            ResetGame();
            return;
        }
    }
    Hand.Clear();
    foreach (String cardName in GetPlayerCardNames())
        Hand.Add(cardName);
    if (!GameInProgress)
        AddProgress(DescribePlayerHands());
}
```

The books changed, and the form needs to know about the change so it can refresh its ScrollViewer.

Here are the modifications to the PlayOneRound() method that update the progress when the game is over, or update the hand and the books if it's not.

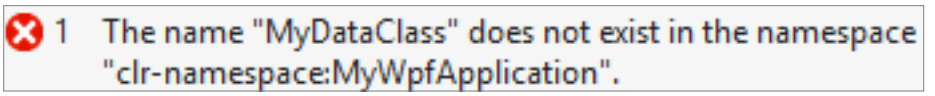
This is the ResetGame() method from the instructions. It clears the books, stock, and hand.

```
public void ResetGame() {
    GameInProgress = false;
    OnPropertyChanged("GameInProgress");
    OnPropertyChanged("GameNotStarted");
    books = new Dictionary<Values, Player>();
    stock = new Deck();
    Hand.Clear();
}

public event PropertyChangedEventHandler PropertyChanged;
private void OnPropertyChanged(string propertyName) {
    PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
    if (propertyChangedEvent != null) {
        propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
    }
}

// ... the rest of the Game class is the same ...
```

This is the standard PropertyChanged event pattern from earlier in the chapter.

Are you getting a strange XAML error about a class not existing in the namespace? Make sure that ALL your C# code compiles and that every control's event handler method is declared in the code-behind.

Sometimes you'll get an error like this when you declare a static resource, even though you definitely have a class called MyDataClass in the namespace MyWpfApplication:



This is often caused by either an error in the code-behind or a missing event handler for a XAML control. This can be a little misleading, because the IDE is telling you that there's an error on the tag that declares the static resource, when the error is actually somewhere else in the code.

You can reproduce this yourself: create a new WPF project called MyWpfApplication, add a data class called MyDataClass, add it as a static resource to your page's <Window.Resources>, and add a button to your page. Then add `Click="Button_Click"` *to the XAML to add an event handler for the button, but* ***don't add the Button_Click() method****. When you try to rebuild your code, you should see the error above. You can make it go away by adding the Button_Click() method to the code-behind.*

Sometimes the error message becomes a little clearer if you right-click on the project in the Solution Explorer, click "Unload Project" to unload it, and then right-click it again and choose "Reload Project" to load it again. This may cause the IDE to show you a different error message that might be more helpful.

Chapter 11

EVEN THOUGH A LOT OF THIS CHAPTER WORKS ONLY WITH WINDOWS STORE APPS, YOU CAN **STILL GET THE CORE LEARNING** WITH WPF.

Windows Store was built for asynchronous programming, but WPF can still use it... but not all the tools are there.

Read through pages 536 and 537 in the main part of the book—see how Brian is shocked (shocked!) to find that his familiar file classes from Chapter 9 aren't there? Well, WPF apps don't have that problem. That's a good thing, because it means you can keep using the file classes and serialization that you're used to. But it also means that your WPF apps can't take advantage of the new asynchronous file and dialog classes that come with the .NET Framework for Windows Store.

In this appendix, we'll give you two replacement projects to show you how to use the `async` and `await` keywords and data contract serialization with WPF apps. Here's how we recommend that you work through Chapter 11:

- Pages 538 and 539 have replacements in this appendix. Use the replacements in place of the book pages.
- Pages 540–545 are specific to Windows Store apps. Skip them.
- Read pages 546 and 547 to learn about data contract serialization.
- Skip pages 548, 549, and 550; they apply only to Windows Store apps.
- Read page 551 in the book. Then follow the "Do this!" project on the replacement pages 552–556 in this appendix.
- The rest of the chapter has you build a Windows Store replacement for Brian's excuse manager. The goal of this project is to learn about the file tools in the Windows.Storage namespace for Windows Store apps. We don't have a WPF alternative for this project, because those classes are specific to Windows Store apps.

C# programs can use await to be more responsive

What happens when you call `MessageBox.Show()` from a WinForms program? Everything stops, and your program freezes until the dialog disappears. That's literally the most unresponsive that a program can be! Windows Store apps should always be responsive, even when they're waiting for feedback from a user. But some things—like waiting for a dialog, or reading or writing all the bytes in a file—take a long time. When a method sits there and makes the rest of the program wait for it to complete, programmers call that **blocking**, and it's one of the biggest causes of program unresponsiveness.

Windows Store apps use **the `await` operator and the `async` modifier** to keep from becoming unresponsive during operations that block. You can see how it works by looking at an example of how a WPF could call a define task that blocks, but can be called asynchronously:

Declare the method using the async modifier to indicate that it can be called asynchronously.

```
private async Task LongTaskAsync()
{
    await Task.Delay(5000);
}
```

The Task class is in the System.Threading.Tasks namespace. Its Delay() method blocks for a specified number of milliseconds. That method is really similar to the Thread.Sleep() method that you used in Chapter 2, but it's defined with the async modifier so it can be called asynchronously with await.

The **`await`** operator causes the method that's running this code to stop and wait until the `ShowAsync()` method completes—and that method will block until the user chooses one of the commands. In the meantime, the rest of the program ***will keep responding to other events***. As soon as the `LongTaskAsync()` method returns, the method that called it will pick up where it left off (although it may wait until after any other events that started up in the meantime have finished).

If your method uses the `await` operator, then it **must be declared with the `async` modifier**:

```
private async void countButton_Click(object sender, RoutedEventArgs e) {
    // ... some code ...
        await LongTaskAsync();
    // ... some more code:
}
```

Notice how this is a Click event handler. Since it uses await, it also needs to be declared with the async modifier.

When a method is declared with `async`, you have some options with how you call it. If you call the method as usual, then as soon as it hits the `await` statement it returns, which keeps the blocking call from freezing your app.

You can see exactly how this works by **creating a new WPF application** with the following main window XAML:

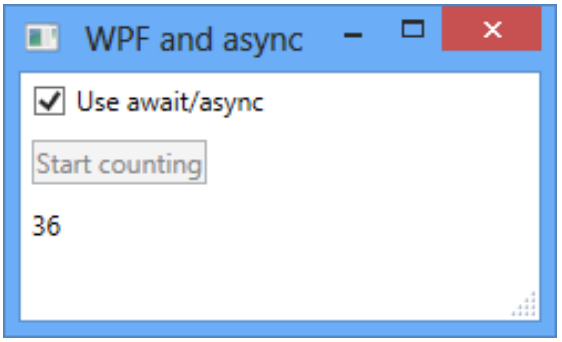
Do this!

```
<Window x:Class="WpfAndAsync.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF and async" Height="150" Width="200" ResizeMode="CanResizeWithGrip">
    <Grid>
        <StackPanel>
            <CheckBox x:Name="useAwaitAsync" IsChecked="True" Content="Use await/async" Margin="5"/>
            <Button x:Name="countButton" Content="Start counting"
                    HorizontalAlignment="Left" Click="countButton_Click" Margin="5"/>
            <TextBlock x:Name="progress" HorizontalAlignment="Left" Margin="5" />
        </StackPanel>
    </Grid>
</Window>
```

We named our project WpfAndAsync. If you named your project something else, you'll need to change this line to match its namespace:
`x:Class="WpfAndAsync.MainWindow"`

Here's the code-behind:

```
using System.Threading;
using System.Windows.Threading;

public partial class MainWindow : Window {
    DispatcherTimer timer = new DispatcherTimer();

    public MainWindow() {
        InitializeComponent();

        timer.Tick += timer_Tick;
        timer.Interval = TimeSpan.FromSeconds(.1);
    }

    int i = 0;
    void timer_Tick(object sender, EventArgs e) {
        progress.Text = (i++).ToString();
    }

    private async void countButton_Click(object sender, RoutedEventArgs e) {
        countButton.IsEnabled = false;
        timer.Start();
        if (useAwaitAsync.IsChecked == true)
            await LongTaskAsync();
        else
            LongTask();
        countButton.IsEnabled = true;
    }

    private void LongTask() {
        Thread.Sleep(5000);
        timer.Stop();
    }

    private async Task LongTaskAsync() {
        await Task.Delay(5000);
        timer.Stop();
    }
}
```



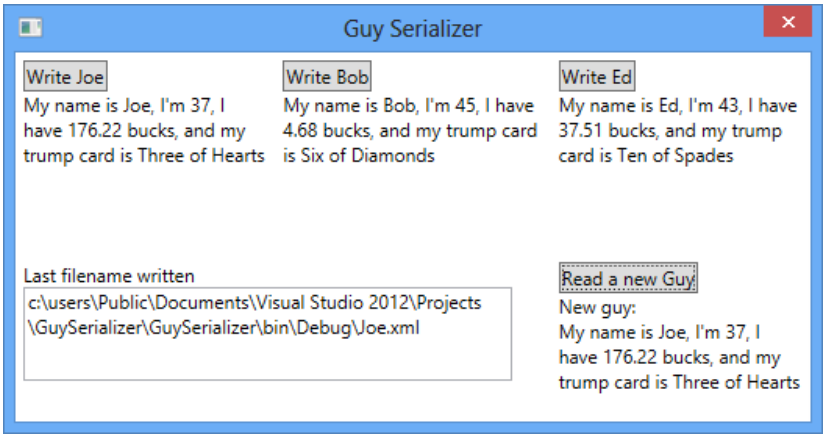
The button's event handler uses the CheckBox's IsChecked property. If the box is checked, the event handler calls `await LongTaskAsync()`, which is asynchronous. The method is called with `await`, so the event handler method pauses and lets the rest of the program continue to run. Try adding other buttons to the window that change properties or print output to the console. You'll be able to use them while the timer ticks.

If the CheckBox is not checked, IsChecked is false and the button's event handler calls `LongTask()`, which blocks. This causes the event handler method to block, which makes the entire program become unresponsive, and if you add other buttons they won't respond either.

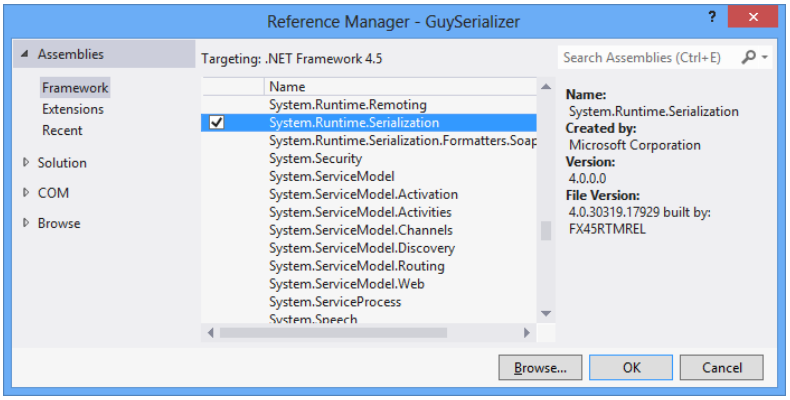
Make sure the box is checked, and then click the button. You'll see the numbers increase, and the form is responsive: the button disables itself, and you can move and resize the form. Then uncheck the box and click the button—now the form freezes.

Stream some Guy objects to a file

Here's a project to help you experiment with data contract serialization. **Create a new WPF application**. Then **add both classes** with the data contracts from page 551 in the book (you'll need `using System.Runtime.Serialization` in each of them). And add the familiar `Suits` and `Values` enums, too (for the `Card` class). Here's the window you'll build next:



1. Before you start coding, you'll need to **right-click on References in the Solution Explorer and choose Add Reference** from the menu. Click on Framework, scroll down to System.Runtime.Serialization, check it, and click OK:



This will allow your WPF application to use the System.Runtime.Serialization namespace.

You can also add an empty GuyManager class to get rid of the IDE error on the `<local:GuyManager>` tag when you add the XAML in step 2. You'll fill in the GuyManager in step 3 when you flip the page.

2 Here's the XAML for the page.

We named this project GuySerializer. If your project has a different namespace, make sure you change these lines to match it.

```
<Window x:Class="GuySerializer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:GuySerializer"
        Title="Guy Serializer" Height="275" Width="525" ResizeMode="NoResize">

    <Window.Resources>
        <local:GuyManager x:Key="guyManager"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource guyManager}" Margin="5">
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="4*"/>
            <RowDefinition Height="3*"/>
        </Grid.RowDefinitions>

        <StackPanel>
            <Button x:Name="WriteJoe" Content="Write Joe"
                    HorizontalAlignment="Left" Click="WriteJoe_Click"/>
            <TextBlock Text="{Binding Joe}" Margin="0,0,10,20" TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Column="1">
            <Button x:Name="WriteBob" Content="Write Bob"
                    HorizontalAlignment="Left" Click="WriteBob_Click"/>
            <TextBlock Text="{Binding Bob}" Margin="0,0,0,20"  TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Column="2" Margin="10,0,0,0">
            <Button x:Name="WriteEd" Content="Write Ed"
                    HorizontalAlignment="Left" Click="WriteEd_Click"/>
            <TextBlock Text="{Binding Ed}" Margin="0,0,0,20"  TextWrapping="Wrap"/>
        </StackPanel>

        <StackPanel Grid.Row="1" Grid.ColumnSpan="2" Margin="0,0,20,0">
            <TextBlock>Last filename written</TextBlock>
            <TextBox Text="{Binding GuyFile, Mode=TwoWay}"
                    TextWrapping="Wrap" Height="60" Margin="0,0,0,20"/>
        </StackPanel>

        <StackPanel Grid.Row="1" Grid.Column="2"  Margin="10,0,0,0">
            <Button x:Name="ReadNewGuy" Content="Read a new Guy"
                    HorizontalAlignment="Left" Click="ReadNewGuy_Click" />
            <StackPanel>
                <TextBlock Text="New guy:"/>
                <TextBlock TextWrapping="Wrap" Text="{Binding NewGuy}"/>
            </StackPanel>
        </StackPanel>
    </Grid>
</Window>
```

The grid's data context is the GuyManager static resource.

The page has three columns and two rows.

Each column in the top row has a StackPanel with a TextBlock and a Button.

This TextBlock is bound to the Ed property in GuyManager.

The first cell in the bottom row spans two columns. It has several controls bound to properties. Why do you think we used a TextBox for the path?

We're not done yet—flip the page!

This program uses TextBoxes that are bound to read-only properties that have only get accessors. If you try to bind to a property that has a public get accessor with a private set accessor, you'll get an error. Luckily, a backing field will work just fine.

3 Add the `GuyManager` class.

```
using System.ComponentModel;
using System.IO;
using System.Runtime.Serialization;

class GuyManager : INotifyPropertyChanged
{
    private Guy joe = new Guy("Joe", 37, 176.22M);
    public Guy Joe
    {
        get { return joe; }
    }

    private Guy bob = new Guy("Bob", 45, 4.68M);
    public Guy Bob
    {
        get { return bob; }
    }

    private Guy ed = new Guy("Ed", 43, 37.51M);
    public Guy Ed
    {
        get { return ed; }
    }

    public Guy NewGuy { get; set; }

    public string GuyFile { get; set; }


    public void ReadGuy()
    {
        if (String.IsNullOrEmpty(GuyFile))
            return;

        using (Stream inputStream = File.OpenRead(GuyFile))
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            NewGuy = serializer.ReadObject(inputStream) as Guy;
        }
        OnPropertyChanged("NewGuy");
    }
```

There are three read-only Guy properties with private backing fields. The XAML has a TextBlock bound to each of them.

A fourth TextBlock is bound to this Guy property, which is set by the ReadGuy() method.

The ReadGuy() method uses familiar System.IO methods to open a stream and read from it. But instead of using a BinaryFormatter, it uses a DataContractSerializer to serialize data from an XML file.

```
    public void WriteGuy(Guy guyToWrite)
    {
        GuyFile = Path.GetFullPath(guyToWrite.Name + ".xml");

        if (File.Exists(GuyFile))
            File.Delete(GuyFile);
        using (Stream outputStream = File.OpenWrite(GuyFile))
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(Guy));
            serializer.WriteObject(outputStream, guyToWrite);
        }

        OnPropertyChanged("GuyFile");
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler propertyChangedEvent = PropertyChanged;
        if (propertyChangedEvent != null)
        {
            propertyChangedEvent(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

This uses the GetFullPath() method in the Path class (in System.IO) to get the full path of the filename to write.

If the file exists, it's deleted, then recreated using a file stream. It's serialized using the data contract serializer.

Here's the same code you used earlier to implement INotifyPropertyChanged and fire off PropertyChanged events.

4 Here's the code-behind for *MainWindow.xaml.cs*:

```
public partial class MainWindow : Window
{
    GuyManager guyManager;

    public MainWindow() {
        InitializeComponent();

        guyManager = FindResource("guyManager") as GuyManager;
    }

    private void WriteJoe_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Joe);
    }
    private void WriteBob_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Bob);
    }
    private void WriteEd_Click(object sender, RoutedEventArgs e) {
        guyManager.WriteGuy(guyManager.Ed);
    }
    private void ReadNewGuy_Click(object sender, RoutedEventArgs e) {
        guyManager.ReadGuy();
    }
}
```

Take your Guy Serializer for a test drive

Use the Guy Serializer to experiment with data contract serialization:

- Write each `Guy` object to the files—they'll be written to the bin\Debug folder in your projects folder. Click the ReadGuy button to read the guy that was just written. It uses the path in the TextBox to read the file, so try updating that path to read a different guy. Try reading a file that doesn't exist. What happens?
- Open up the Simple Text Editor you built earlier in the chapter. You added XML files as options for the open and save file pickers, so you can use it to edit Guy files. Open one of the Guy files, change it, save it, and read it back into your Guy Serializer. What happens if you add invalid XML? What if you change the card suit or value so it doesn't match a valid enum value?
- Try adding or removing the `DataMember` names (`[DataMember(Name="...")]`). What does that do to the XML? What happens when you update the contract and then try to load a previously saved XML file? Can you fix the XML file to make it work?
- Try changing the namespace of the `Card` data contract. What happens to the XML?

there are no Dumb Questions

Q: Sometimes I make a change in my XAML or my code, and the IDE's designer gives me a message that I need to rebuild. What's going on?

A: The XAML designer in the IDE is really clever. It's able to show you an updated page in real time as you make changes to your XAML code. You already know that when the XAML uses static resources, that adds object references to the `Page` class. Well, those objects need to get instantiated in order for them to be displayed in the designer. If you make a change to the class that's being used for a static resource, the designer doesn't get updated until you rebuild that class. That makes sense—the IDE rebuilds your project only when you ask it to, and until you do that it doesn't actually have the compiled code in memory that it needs to instantiate the static resources.

You can use the IDE to see exactly how this works. Open your Guy Serializer and edit the `Guy.ToString()` method to add some extra words to the return value. Then go back to the main page designer. It's still showing the old output. Now choose Rebuild from the Build menu. The designer will update itself as soon as the code finishes rebuilding. Try making another change, but don't rebuild yet. Instead, add another TextBlock that's bound to a `Guy` object. The IDE will use the old version of the object until you rebuild.

Q: I'm confused about namespaces. How is the namespace in the program different from the one in an XML file?

A: Let's take a step back and understand why namespaces are necessary. C#, XML files, the Windows filesystem, and web pages all use different (but often related) naming systems to give each class, XML document, file, or web page its own unique name. So why is this important? Well, let's say back in Chapter 9, you created a class called `KnownFolders` to help Brian keep track of excuse folders. Uh-oh! Now you find out that the .NET Framework for Windows Store already has a `KnownFolders` class. No worries. The .NET `KnownFolders` class is in the Windows.Storage namespace, so it can exist happily alongside your class with the same name, and that's called **disambiguation**.

Data contracts also need to disambiguate. You've seen several different versions of a `Guy` class throughout this book. What if you wanted to have two different contracts to serialize different versions of `Guy`? You can put them in different namespaces to disambiguate them. And it makes sense that these namespaces would be separate from the ones for your classes, because you can't really confuse classes and contracts.

One more thing. Your WPF applications can use the same OpenFileDialog and SaveFileDialog classes that you used in your WinForms projects. Here's an MSDN page that has more information and code samples:

http://msdn.microsoft.com/en-us/library/aa969773.aspx

Chapter 12

Exception handling works the same in WPF as it does in WinForms and Windows Store.

If you flip through the replacement pages for Chapter 12, you'll notice that there's no XAML. That's because the material on exception handling that we cover in *Head First C#* is basically the same whether you're working on a WPF application, a WinForms program, a Windows Store app, or even a console application.

Here's how you should use this appendix for Chapter 12:

- Read through page 575 in the book, including the "Sharpen your Pencil" exercise.
- Use the appendix replacement pages for 576 and 577.
- Read pages 578 and 579 in the book.
- Follow pages 580–590 in this appendix, and skip 591 in the main part of the book.
- Finish the rest of the chapter in the book.
- Then do all of Chapter 13 in the book, too!

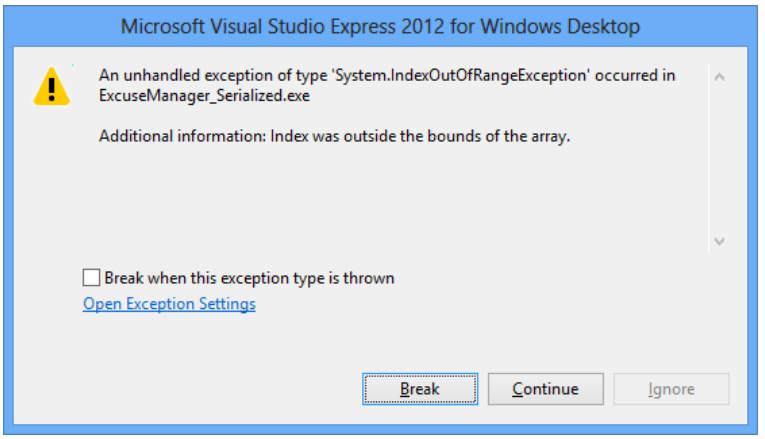
Once you're done with this chapter, you can go straight through Chapter 13 in the book. It doesn't depend on Windows 8 or Windows Store apps at all.

Brian's code did something unexpected

This appendix depends on the Excuse Manager WinForms app that you built in Chapter 9. If your code doesn't match the code in the appendix, you can download it from http://headfirstlabs.com/hfcsharp.

When Brian wrote his Excuse Manager, he never expected the user to try to pull a random excuse out of an empty directory.

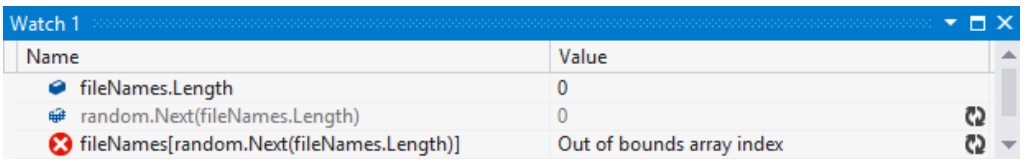
1. The problem happened when Brian pointed his Excuse Manager program at an empty folder on his laptop and clicked the Random Excuse button. Let's take a look at it and see if we can figure out what went wrong. Here's the unhandled exception window that popped up when he ran the program in the IDE:



2. OK, that's a good starting point. It's telling us that there's some value that doesn't fall inside some range. Clicking the Break button drops the IDE back into the debugger, with the execution halted on a specific line of code:

```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

3. Let's use the Watch window to track down the problem. Add a watch for `fileNames.Length`. Looks like that returns 0. Try adding a watch for `random.Next(fileNames.Length)`. That returns 0, too. So add a watch for `fileNames[random.Next(fileNames.Length)]`. This time the Value column in the Watch window has the same error message that you saw in step 1: "Out of bounds array index."



You can call methods and use indexers in the Watch window. When one of those things throws an exception, you'll see that exception in the Watch window, too.

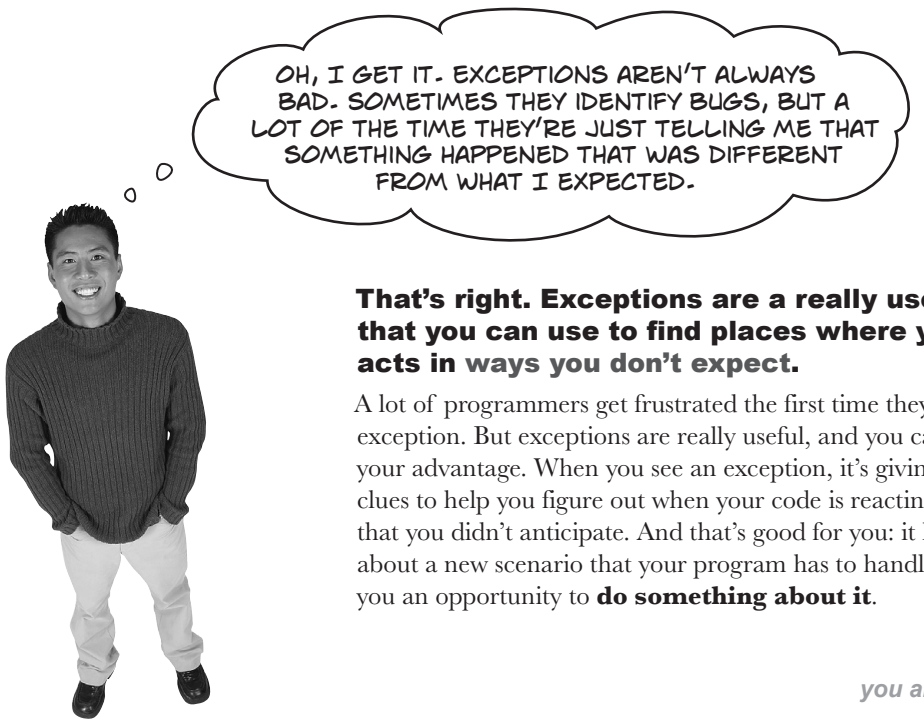
4 So what happened? It turns out that `Directory.GetFiles()` returns an empty array when you point it at an empty folder. So `fileNames.Length` is zero, and passing 0 to `Random.Next()` will always return 0 as well. Try to get the 0th element of an empty array and your program will throw a `System.IndexOutOfRangeException`, with the message "Index was outside the bounds of the array."

Now that we know what the problem is, we can fix it. All we need to do is check to see if the selected folder has excuses in it before we try to load a random excuse from it:

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    if (Directory.GetFiles(selectedFolder).Length == 0)
        MessageBox.Show("There are no excuse files in the selected folder.");
    else if (CheckChanged())
    {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

By checking for excuse files in the folder **before** we create the Excuse object, we can prevent the exception from being thrown—and display a helpful dialog, too.

What do you think about that solution? Does it make the most sense to put it in the form, or would it be better to find a way to encapsulate it inside the Excuse class?



That's right. Exceptions are a really useful tool that you can use to find places where your code acts in ways you don't expect.

A lot of programmers get frustrated the first time they see an exception. But exceptions are really useful, and you can use them to your advantage. When you see an exception, it's giving you a lot of clues to help you figure out when your code is reacting to a situation that you didn't anticipate. And that's good for you: it lets you know about a new scenario that your program has to handle, and it gives you an opportunity to **do something about it**.

Use the IDE's debugger to ferret out exactly what went wrong in the Excuse Manager

Let's use the debugger to take a closer look at the problem that we ran into in the Excuse Manager. You've probably been using the debugger a lot over the last few chapters, but we'll go through it step by step anyway to make sure we don't leave out any details.

Debug this

1 ADD A BREAKPOINT TO THE RANDOM BUTTON'S EVENT HANDLER.

You've got a starting point—the exception happens when the Random Excuse button is clicked after an empty folder is selected. So open up the button's event handler and use Debug→Toggle Breakpoint (F9) to add a breakpoint to the first line of the method. Start debugging, **choose an empty folder**, and then click the Random button to make your program break at the breakpoint:

```
private void randomExcuse_Click(object sender, EventArgs e)
{
    if (Directory.GetFiles(selectedFolder).Length == 0)
        MessageBox.Show("There are no excuse files in the selected folder.");
    else if (CheckChanged())
    {
        currentExcuse = new Excuse(random, selectedFolder);
        UpdateForm(false);
    }
}
```

2 STEP INTO THE EXCUSE CONSTRUCTOR.

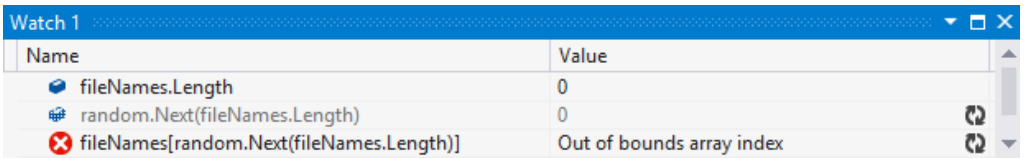
We want to reproduce the problem, but we already added code to get past it. No problem. Right-click on the line `currentExcuse = new Excuse(random, selectedFolder);` and choose **Set Next Statement** (Ctrl+Shift+F10). Then use **Step Into** (F11) to step into the constructor:

```
public Excuse(Random random, string folder)
{
    string[] fileNames = Directory.GetFiles(folder, "*.excuse");
    OpenFile(fileNames[random.Next(fileNames.Length)]);
}
```

You used the debugger to skip past the workaround that you added to avoid the exception, so now the Excuse constructor is about to throw the exception again.

3 STEP THROUGH THE PROGRAM UNTIL IT THROWS THE EXCEPTION.

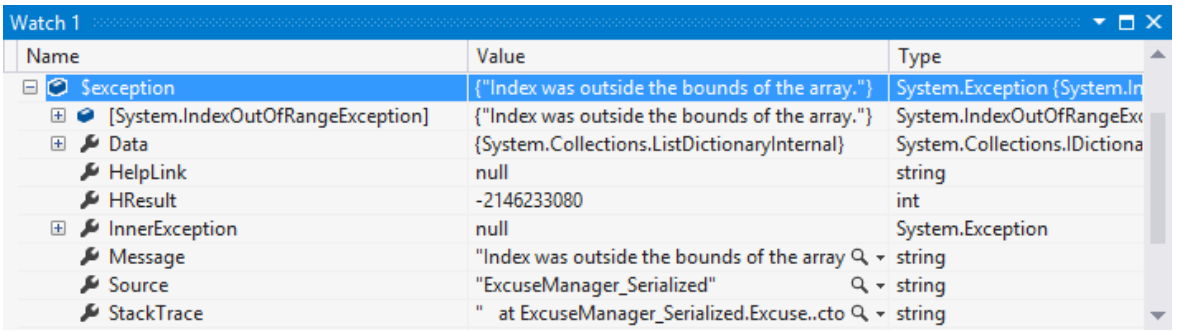
You've already seen how handy the Watch window is. Now we'll use it to reproduce the exception. Choose Step Over (F10) twice to get your program to throw the exception. Then use the IDE to select `fileNames.Length`, right-click on it, and choose Add Watch to add a watch. Then do it again for `random.Next(fileNames.Length)` and `fileNames[random.Next(fileNames.Length)]`:



The Watch window has another very useful feature. It lets you **change the value** of variables and fields that it's displaying, and it even lets you **execute methods and create new objects**. When you do, it displays its reevaluate icon that you can click to tell it to execute that method again.

4 ADD A WATCH FOR THE EXCEPTION OBJECT.

Debugging is a little like *performing a forensic crime scene investigation on your program*. You don't necessarily know what you're looking for until you find it, so you need to use your debugger "CSI kit" to follow clues and track down the culprit. One important tool is adding `$exception` to the Watch window, because it shows you the contents of the Exception object that's been thrown:



When you get an exception, you can go back and reproduce it in the debugger and use the `Exception` object to help you fix your code.

there are no Dumb Questions

Q: How do I know where to put a breakpoint?

A: That's a really good question, and there's no one right answer. When your code throws an exception, it's always a good idea to start with the statement that threw it. But usually, the problem actually happened earlier in the program, and the exception is just fallout from it. For example, the statement that throws a divide-by-zero error could be dividing values that were generated 10 statements earlier but just haven't been used yet. So there's no one good answer to where you should put a breakpoint, because every situation is different. But as long as you've got a good idea of how your code works, you should be able to figure out a good starting point.

Q: Can I run any method in the Watch window?

A: Yes. Any statement that's valid in your program will work inside the Watch window, even things that make absolutely no sense to run inside a Watch window. Here's an example. Bring up a program, start it running, break it, and then **add this to the Watch window**: `System.Threading.Thread.Sleep(2000)`. That method causes your program to delay for two seconds.There's no reason you'd ever do that in real life, but it's interesting to see what happens: the IDE will block and you'll get a wait cursor for two seconds while the method evaluates. Then, since `Sleep()` has no return value, the Watch window will display the value `Expression has been evaluated and has no value` to let you know that it didn't return anything. But it did evaluate it. Not only that, but it displays IntelliSense pop-ups to help you type code into the window. That's useful because it shows the available properties and methods for objects currently in memory.

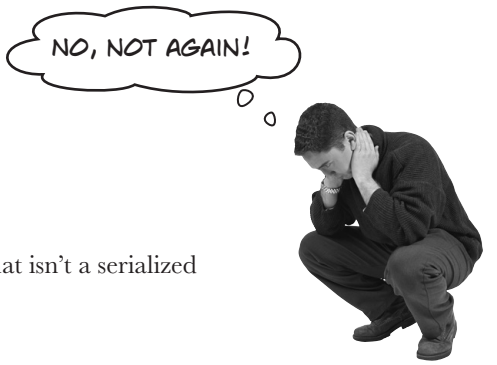
Q: Wait, so isn't it possible for me to run something in the Watch window that'll change the way my program runs?

A: Yes! Not permanently, but it can definitely affect your program's output. But even better, just ***hovering*** over fields inside the debugger can cause your program to change its behavior, because hovering over a property ***executes its*** `get` ***accessor***. If you have a property that has a `get` accessor that executes a method, then hovering over that property will cause that method to execute. And if that method sets a value in your program, then that value will stay set if you run the program again. And that can cause some pretty unpredictable results inside the debugger. Programmers have a name for results that seem to be unpredictable and random: they're called **heisenbugs** (which is a joke that makes sense to physicists and cats trapped in boxes).

When you run your program inside the IDE, an unhandled exception will cause it to break as if it had run into a breakpoint.

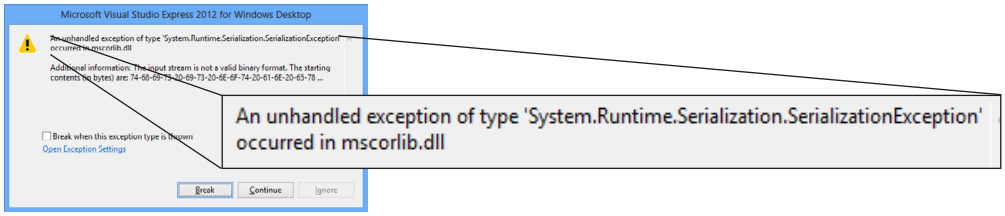
Uh-oh—the code's still got problems...

Brian was happily using his Excuse Manager when he accidentally chose a folder full of files that weren't created by the Excuse Manager. Let's see what happens when he tries to load one of them....

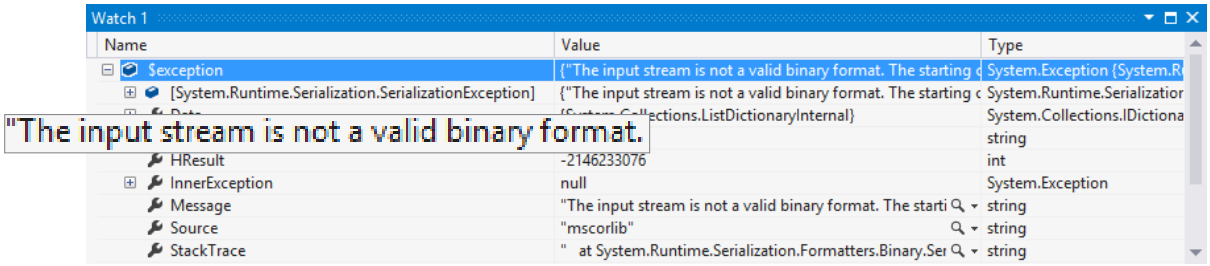


1. You can re-create Brian's problem. Take a random file that isn't a serialized excuse and give it the .excuse file extension.

2. Pop open the Excuse Manager in the IDE and open up the file you created. It throws an exception! Look at the message, then click the Break button to start investigating.



3. Open up the Locals window and expand `$exception` (you can also enter it into the Watch window). Take a close look at its members to see if you can figure out what went wrong.



DO YOU SEE WHY THE PROGRAM THREW THE EXCEPTION?

DOES IT MAKE SENSE FOR THE PROGRAM TO CRASH IF IT ENCOUNTERS AN INVALID EXCUSE XML FILE?

CAN YOU THINK OF ANYTHING YOU CAN DO ABOUT THIS?

Actually, there is something you can do about it.

Yes, it's true that users screw up all the time. That's a fact of life. But that doesn't mean you can't do anything about it. There's a name for programs that deal with bad data, malformed input, and other unexpected situations gracefully: they're called **robust** programs. And C# gives you some really powerful exception handling tools to help you make your programs more robust. Because while you *can't* control what your users do, you *can* make sure that your program doesn't crash when they do it.

ro-bust, adj.
sturdy in construction; able to withstand or overcome adverse conditions. *After the Tacoma Narrows Bridge disaster, the civil engineering team looked for a more **robust** design for the bridge that would replace it.*

The BinaryFormatter class will also throw a SeralizationException if you give it a file that doesn't contain exactly the right serialized object. It's even more finicky than DataContractSerializer!

Watch it!

Serializers will throw an exception if there's anything at all wrong with a serialized file.

It's easy to get the Excuse Manager to throw a `SerializationException`—just feed it any file that's not a serialized Excuse object. When you try to deserialize an object from a file, `DataContractSerializer` expects the file to contain a serialized object that matches the contract of the class that it's trying to read. If the file contains anything else, almost anything at all, then the `ReadObject()` method will throw a `SerializationException`.

Handle exceptions with try and catch

In C#, you can basically say, "**Try** this code, and if an exception occurs, **catch** it with this *other* bit of code." The part of the code you're trying is the **try block**, and the part where you deal with exceptions is called the **catch block**. In the `catch` block, you can do things like print a friendly error message instead of letting your program come to a screeching halt:

You'll also need to add these lines to the top of Excuse.cs:

```
using System.Runtime.Serialization;

using System.Windows.Forms;
```

Put the code that might throw an exception inside the try block. If no exception happens, it'll get run exactly as usual, and the statements in the catch block will be ignored. But if a statement in the try block throws an exception, the rest of the try block **won't get executed**.

```
private void OpenFile(string excusePath) {
    try
    {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException)
    {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
}
```

This is the **try block**. You start exception handling with try. In this case, we'll put the existing code in it.

You'll recognize the code here because we surrounded the entire method with this try block.

The catch keyword means that the block immediately following it contains an **exception handler**.

When an exception is thrown, the program immediately jumps to the **catch** statement and starts executing the **catch block**.

What happens if you leave out this last line of code? Can you figure out why we included it in the catch block?

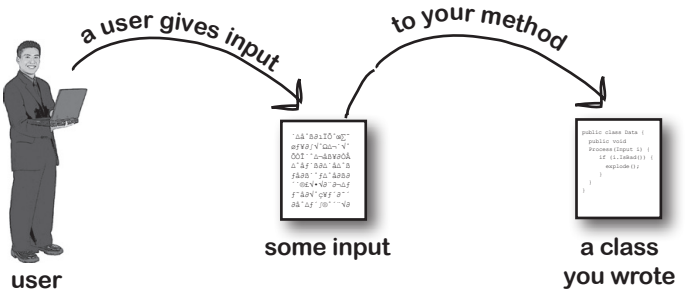
This is the simplest kind of exception handling: stop the program, write out the exception message, and keep running.

If throwing an exception makes your code automatically jump to the `catch` block, what happens to the objects and data you were working with before the exception happened?

What happens when a method you want to call is risky?

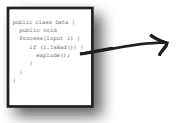
Users are unpredictable. They feed all sorts of weird data into your program and click on things in ways you never expected. And that's just fine, because you can handle unexpected input with good exception handling.

1 Let's say your user is using your code and gives it some input that it didn't expect.



2 That method does something risky, something that might not work at runtime.

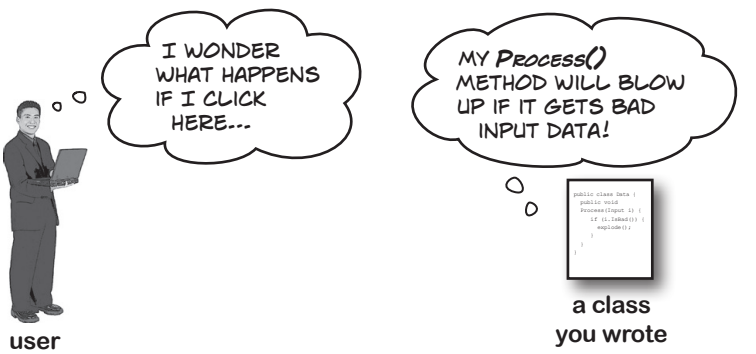
"Runtime" just means "while your program is running." Some people refer to exceptions as "runtime errors."



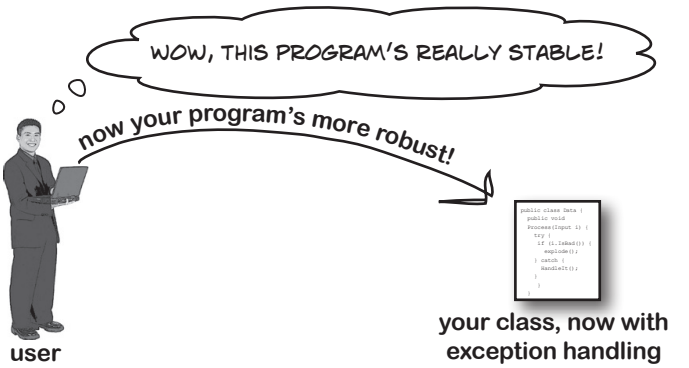

```
public void
    Process(Input i) {
    if (i.IsBad()) {
       Explode();
    }
}
```

3 You need to *know* that the method you're calling is risky.

If you can come up with a way to do a less risky thing that avoids throwing the exception, that's the best possible outcome! But some risks just can't be avoided, and that's when you want to do this.



4 You then write code that can handle the failure if it *does* happen. You need to be prepared, just in case.



there are no Dumb Questions

Q: So when do I use `try` and `catch`?

A: Anytime you're writing risky code, or code that could throw an exception. The trick is figuring out which code is risky, and which code is safer.

You've already seen that code that uses input provided by a user can be risky. Users give you incorrect files, words instead of numbers, and names instead of dates, and they pretty much click everywhere you could possibly imagine. A good program will take all that input and work in a calm, predictable way. It might not give the users a result they can use, but it will let them know that it found the problem and hopefully suggest a solution.

Q: How can a program suggest a solution to a problem it doesn't even know about in advance?

A: That's what the `catch` block is for. A `catch` block is executed only when code in the `try` block throws an exception. It's your chance to make sure the user knows that something went wrong, and to let the user know that it's a situation that might be corrected.

If the Excuse Manager simply crashes when there's bad input, that's not particularly useful. But if it tries to read the input and displays garbage in the form, that's also not useful—in fact, some people might say that it's worse. But if you have the program display an error message telling the user that it couldn't read the file, then the user has an idea of what went wrong, and information that he can use to fix the problem.

Q: So the debugger should really only be used to troubleshoot exceptions then?

A: No. As you've already seen many times throughout the book, the debugger's a really useful tool that you can use to examine any code you've written. Sometimes it's useful to step through your code and check the values of certain fields and variables—like when you've got a really complex method and you want to make sure it's working properly.

But as you may have guessed from the name "debugger," its most common use is to track down and remove bugs. Sometimes those bugs are exceptions that get thrown. But a lot of the time, you'll be using the debugger to try to find other kinds of problems, like code that gives a result that you don't expect.

Q: I'm not sure I totally got what you did with the Watch window.

A: When you're debugging a program, you usually want to pay attention to how a few variables and fields change. That's where the Watch window comes in. If you add watches for a few variables, the Watch window updates their values every time you step into, out of, or over code. That lets you monitor exactly what happens to them after every statement, which can be really useful when you're trying to track down a problem.

The Watch window also lets you type in any statement you want, and even call methods, and the IDE will evaluate it and display the results. If the statement updates any of the fields and variables in your program, then it does that, too. That lets you change values while your program is running, which can be another really useful tool for reproducing exceptions and other bugs.

Any changes you make in the Watch window just affect the data in memory, and last only as long as the program is running. Restart your program, and values that you changed will be undone.

The catch block is executed only when code in the try block throws an exception. It gives you a chance to make sure your user has the information to fix the problem.

Use the debugger to follow the try/catch flow

An important part of exception handling is that when a statement in your `try` block throws an exception, the rest of the code in the block gets **short-circuited**. The program's execution immediately jumps to the first line in the `catch` block. ***But don't take our word for it...***

1. Add the `try/catch` from a few pages ago to your Excuse Manager app's `ReadExcuseAsync()` method. Then place a breakpoint on the opening bracket `{` in the `try` block.

2. Start debugging your app and open up a file that's **not a valid excuse file** (but still has the *.excuse* extension). When the debugger breaks on your breakpoint, click the Step Over button (or F10) five times to get to the statement that calls `ReadObject()` to deserialize the `Excuse` object. Here's what your debugger screen should look like:

Put the breakpoint on the opening bracket of the try block.

Step over the statements until your yellow "next statement" bar shows that the next statement to get executed will read the Excuse object from the stream.

```
private void OpenFile(string excusePath)
{
    try
    {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException)
    {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
}
```

3 Step over the next statement. As soon as the debugger executes the `Deserialize()` statement, the exception is thrown and the program **short-circuits** right past the rest of the method and **jumps straight to the `catch` block**.

The debugger will highlight the catch statement with its yellow "next statement" block, but it shows the rest of the block in gray to show you that it's about to execute the whole thing.

```
private void OpenFile(string excusePath)
{
    try
    {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException)
    {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
}
```

4 Start the program again by pressing the Continue button (or F5). It'll begin running the program again, starting with whatever's highlighted by the yellow "next statement" block—in this case, the `catch` block. It will just display the dialog and then act as if nothing happened. The ugly crash has now been handled.

Here's a career tip: a lot of C# programming job interviews include a question about how you deal with exceptions in a constructor.

Watch it!

Keep risky code out of the constructor!

You've noticed by now that a constructor doesn't have a return value, not even `void`. That's because a constructor doesn't actually return anything. Its only purpose is to initialize an object—which is a problem for exception handling inside the constructor. When an exception is thrown inside the constructor, then the statement that tried to instantiate the class ***won't end up with an instance of the object****.*

If you have code that should ALWAYS run, use a finally block

When your program throws an exception, a couple of things can happen. If the exception ***isn't*** handled, your program will stop processing and crash. If the exception ***is*** handled, your code jumps to the `catch` block. But what about the rest of the code in your `try` block? What if you were closing a stream, or cleaning up important resources? That code needs to run, even if an exception occurs, or you're going to make a mess of your program's state. That's where the **`finally`** block comes in really handy. It comes after the `try` and `catch` blocks. The **`finally` block always runs**, whether or not an exception was thrown.

```
private void OpenFile(string excusePath) {
    try {
        this.ExcusePath = excusePath;
        BinaryFormatter formatter = new BinaryFormatter();
        Excuse tempExcuse;
        using (Stream input = File.OpenRead(excusePath))
        {
            tempExcuse = (Excuse)formatter.Deserialize(input);
        }
        Description = tempExcuse.Description;
        Results = tempExcuse.Results;
        LastUsed = tempExcuse.LastUsed;
    }
    catch (SerializationException) {
        MessageBox.Show("Unable to read " + excusePath);
        LastUsed = DateTime.Now;
    }
    finally
    {
        // Any code here will get executed no matter what
    }
}
```

If there is no exception thrown during the try block, the code in the finally block will execute after the try block completes. If there's an exception handled by a catch block, then it will short-circuit as usual, and then run the finally block after the catch block.

Always catch specific exceptions like `SerializationException`. You typically follow a `catch` statement with a specific kind of exception telling it what to catch. It's valid C# code to just have **`catch (Exception)`** and you can even leave the exception type out and just use **`catch`**. When you do that, it **catches all exceptions**, no matter what type of exception is thrown. But it's a ***really bad practice to have a catch-all exception handler*** like that. Your code should always catch as specific an exception as possible.

Reminder: Once you finish Chapter 12, you can go straight through Chapter 13 in the book. It doesn't depend on Windows 8 or Windows Store apps at all.

Chapter 14

LINQ works with *any* kind of C# program.

When you read Chapter 14 in the main part of the book, you'll see that it's structured differently from other chapters. It has a series of increasingly complex LINQ queries, and small console apps to demonstrate each of them. Throughout the chapter, you'll also see exercises to build a Windows Store app that combines all the queries into a single user interface. Over the next few pages of this appendix, we'll show you how to build a WPF application that executes those same queries. Here's how we recommend you use this appendix with Chapter 14:

- Read through page 657 in the book.
- Even though pages in the chapter through 665 are about building a Windows Store app, read them—especially the parts about **anonymous types**. It will help to get a sense of how the Comic, ComicQuery, and ComicQueryManager classes work.
- Pages 666 and 667 describe more LINQ queries. You can skim pages 668 and 669, because those are more Windows Store-related pages.
- Read pages 670–680, but don't do the exercise on page 679.
- You can skip the rest of the chapter, because it's related to Windows Store apps. Instead, follow the replacement pages 680–683.

Build a WPF comic query application

When you read through Chapter 14 in the book, you saw that we built a Windows Store app to execute the LINQ queries throughout the chapter. Since we followed the principle of separation of concerns, the classes for managing data and issuing queries were separated from the code that created the user interface. That let us **reuse the same data and query management classes** to build another app using the Visual Studio Split App template. Now we'll be able to take advantage of the same separation of concerns and build a WPF application using the same data and query classes.

Do this!

1 CREATE A NEW WPF APPLICATION AND ADD EXISTING CLASSES AND IMAGES FROM THE COMIC APP.

Before you start this project, you'll need to download source code to the JimmysComics app from Chapter 14. See the Head First Labs website (http://headfirstlabs.com/hfcsharp) for a link to the source code.

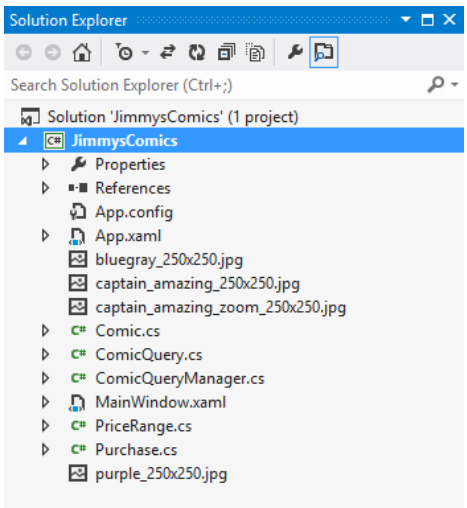
Once you've got the source code, you'll build a new WPF application called JimmysComics. Then right-click on the project name in the Solution Explorer and choose "Add Existing Item" to add the following items from the Windows Store app we built in the book (you can download the source from the book's website):

- Purchase.cs
- Comic.cs
- ComicQuery.cs
- ComicQueryManager.cs
- PriceRange.cs.

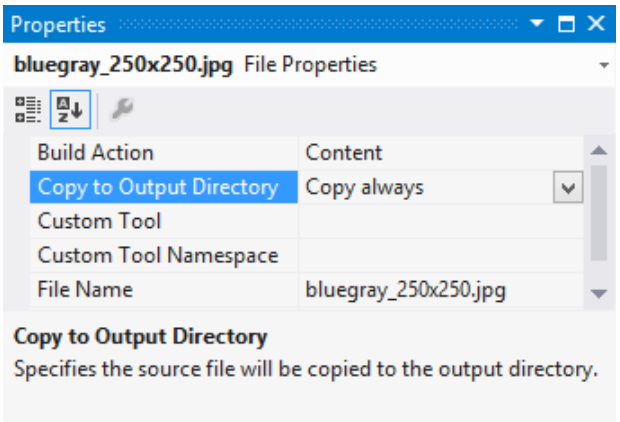
If you give your project a different name, make sure you change the namespace for the C# files you added to match your project's namespace.

- The following files are in the Assets folder: bluegray_250x250.jpg, bluegray_250x250.jpg, captain_amazing_250x250.jpg, captain_amazing_zoom_250x250.jpg — add them to the root level of your WPF application so they're alongside your XAML and C# files.

Your Solution Explorer should look like this:



You'll also need to select each image file in the Solution Explorer and **use the Properties window** to set "Build Action" to `Content` and "Copy to Output Directory" to `Copy always`. Here's what it looks like—make sure you do this for each of the .jpg files that you added:



2 MAKE TWO MODIFICATIONS TO COMICQUERYMANAGER.CS.

There are two small changes you'll need to make to ComicQueryManager.cs. WPF applications cannot use the Windows.UI namespace because it's only part of the .NET Framework for Windows Store. You'll need to change the using statements at the top to replace "Windows.UI" with "System.Windows":

```
using System.Collections.ObjectModel;
using System.Windows.Media.Imaging;
```

And WPF applications load images slightly differently from Windows Store apps, so you'll need to change the CreateImageFromAssets() method in ComicQueryManager. Here's the new method:

```
private static BitmapImage CreateImageFromAssets(string imageFilename)
{
    try
    {
        Uri uri = new Uri(imageFilename, UriKind.RelativeOrAbsolute);
        return new BitmapImage(uri);
    }
    catch (System.IO.IOException)
    {
        return new BitmapImage();
    }
}
```

You copied the .jpg files into your project's top-level folder. This new CreateImageFromAssets() method will load those files.

3 ADD CODE-BEHIND FOR THE MAIN WINDOW.

Here's all the code-behind you'll need for MainWindow.xaml.cs.

```
public partial class MainWindow : Window
{
    ComicQueryManager comicQueryManager;

    public MainWindow()
    {
        InitializeComponent();

        comicQueryManager = FindResource("comicQueryManager") as ComicQueryManager;
        comicQueryManager.UpdateQueryResults(comicQueryManager.AvailableQueries[0]);
    }

    private void ListView_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        if (e.AddedItems.Count >= 1 && e.AddedItems[0] is ComicQuery)
        {
            comicQueryManager.CurrentQueryResults.Clear();
            comicQueryManager.UpdateQueryResults(e.AddedItems[0] as ComicQuery);
        }
    }
}
```

The ListView control fires its SelectionChanged event whenever the user selects or deselects items. The items that were selected can be found in the e.AddedItems collection.

4 ADD THE XAML FOR THE MAIN WINDOW.

Here's the XAML for the main window. Remember, if you used a different project name, make sure you change `JimmysComics` to match your project's namespace.

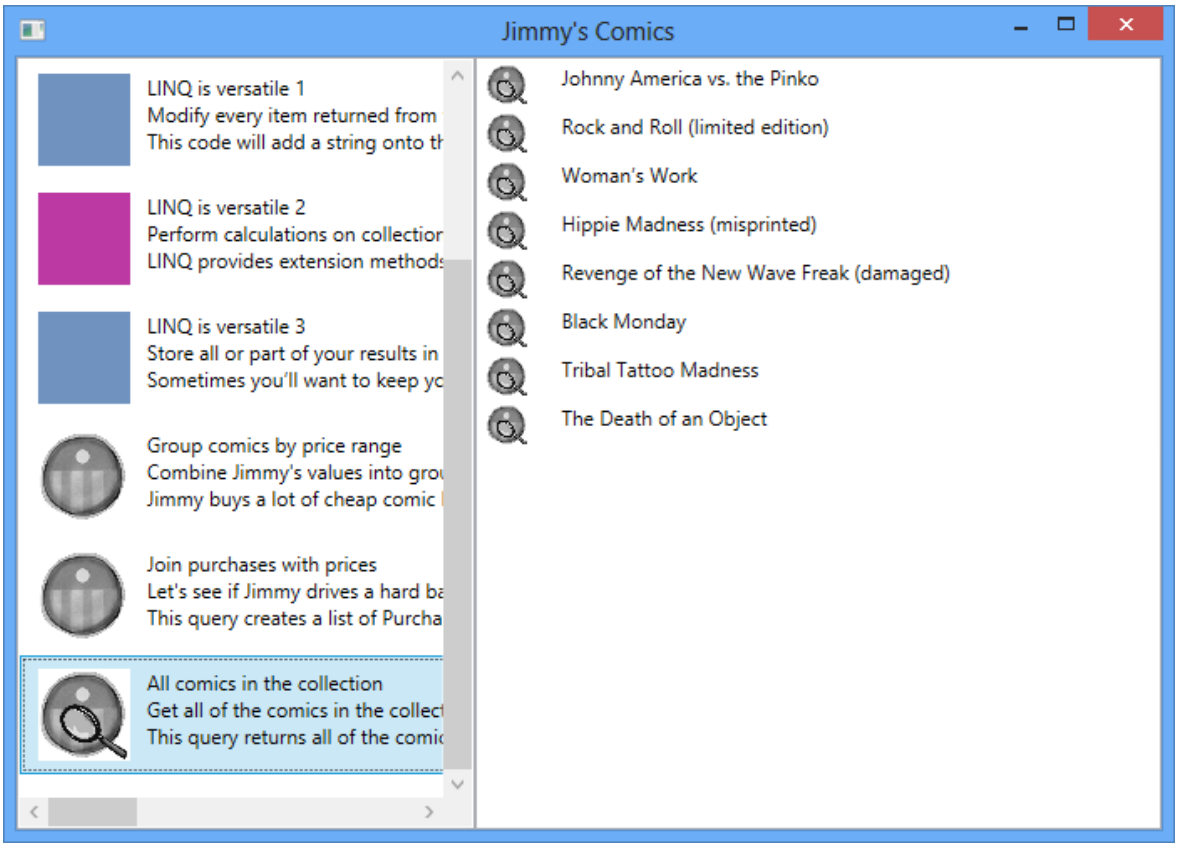
```
<Window x:Class="JimmysComics.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:JimmysComics"
        Title="Jimmy's Comics" Height="350" Width="525">

    <Window.Resources>
        <local:ComicQueryManager x:Key="comicQueryManager"/>
    </Window.Resources>

    <Grid DataContext="{StaticResource ResourceKey=comicQueryManager}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*"/>
            <ColumnDefinition Width="3*"/>
        </Grid.ColumnDefinitions>
        <ListView SelectionMode="Single" ItemsSource="{Binding AvailableQueries}"
                  SelectionChanged="ListView_SelectionChanged">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <Grid Height="55" Margin="6">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <Border Width="55" Height="55">
                            <Image Source="{Binding Image}" Stretch="UniformToFill"/>
                        </Border>
                        <StackPanel Grid.Column="1" VerticalAlignment="Top" Margin="10,0,0,0">
                            <TextBlock Text="{Binding Title}" TextWrapping="NoWrap"/>
                            <TextBlock Text="{Binding Subtitle}" TextWrapping="NoWrap"/>
                            <TextBlock Text="{Binding Description}" TextWrapping="NoWrap"/>
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

        <ListView Grid.Column="1" SelectionMode="Single"
                  ItemsSource="{Binding CurrentQueryResults}">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <StackPanel Orientation="Horizontal">
                        <Image Source="{Binding Image}" Margin="0,0,20,0"
                               Stretch="UniformToFill" Width="25" Height="25"
                               VerticalAlignment="Top" HorizontalAlignment="Right"/>
                        <StackPanel>
                            <TextBlock Text="{Binding Title}" />
                        </StackPanel>
                    </StackPanel>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>

    </Grid>
</Window>
```

This ListView's SelectionMode is set to Single so only one query can be selected at a time.

The ListView on the right has an item template that displays information about each query.

The ListView on the right has an item template that shows individual items in the query results.

When you run the app, the queries appear on the left, and the results of the selected query appear on the right.



QUERIES THAT RETURN COMIC BOOKS HAVE ADDITIONAL INFORMATION: PRICE, SYNOPSIS, EVEN A COVER IMAGE. CAN YOU FIGURE OUT HOW TO GET THE COMIC QUERIES TO DISPLAY ALL THE INFORMATION ABOUT EACH COMIC? YOU'LL NEED TO ADD THE COMIC BOOK COVER IMAGES TO THE PROJECT. YOU'LL FIND SOME HELPFUL XAML CODE IN THE CHAPTER ON PAGES 689 AND 690.

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 15

Events are useful for any app, but especially important for understanding XAML.

Events can be simple and straightforward, because you've been using them throughout the book. But there's a lot more depth to them than you might expect. This chapter helps you understand events in more detail.

Here's what we recommend for this chapter:

- Read the chapter in the book through page 711.
- Use the replacement pages in this appendix for the exercise on pages 712–713 and its solution on pages 714–715.
- Read pages 716–719 in the book.
- Pages 720–723 are specific to Windows Store apps, but **we recommend that you read them anyway**. They give you some insight not just into Windows Store apps, but also into some basic features of Windows 8.
- We provide replacement pages for pages 724-729 in this appendix.
- Read the rest of the chapter in the book. The only pages you should skip are the top of page 740, and pages 742–743.

It's time to put what you've learned so far into practice. Your job is to complete the `Ball` and `Pitcher` classes, add a `Fan` class, and make sure they all work together with a very basic version of your baseball simulator.

1 COMPLETE THE PITCHER CLASS.

Below is what we've got for `Pitcher`. Add the `CatchBall()` and `CoverFirstBase()` methods. Both should create a string saying that the catcher has either caught the ball or run to first base and add that string to a public `ObservableCollection<string>` called `PitcherSays`.

```
class Pitcher {
    public Pitcher(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        if (e is BallEventArgs){
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
}
```

You'll need to implement these two methods to add a string to the PitcherSays ObservableCollection.

2 WRITE A FAN CLASS.

Create another class called `Fan`. `Fan` should also subscribe to the `BallInPlay` event in its constructor. The fan's event handler should see if the distance is greater than 400 feet and the trajectory is greater than 30 (a home run), and grab for a glove to try to catch the ball if it is. If not, the fan should scream and yell. Everything that the fan screams and yells should be added to an `ObservableCollection<string>` called `FanSays`.

Look at the output on the facing page to see exactly what it should print.

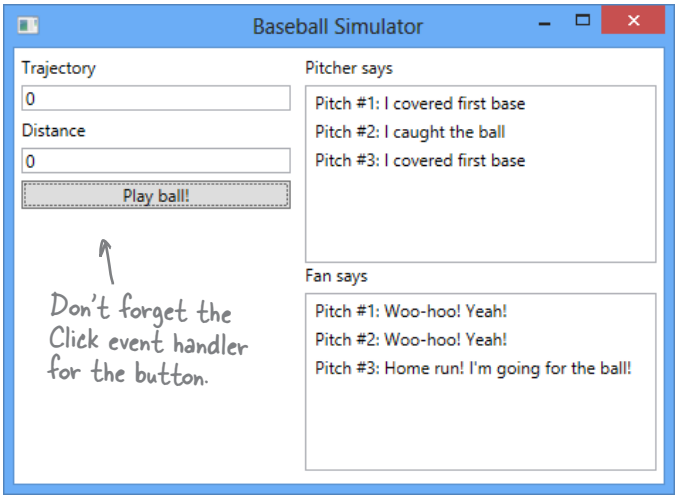
3 BUILD A VERY SIMPLE SIMULATOR.

If you didn't do it already, create a new WPF Application and add the following `BaseballSimulator` class. Then add it as a static resource to the page.

```
using System.Collections.ObjectModel;

class BaseballSimulator {
    private Ball ball = new Ball();
    private Pitcher pitcher;
    private Fan fan;

    public ObservableCollection<string> FanSays { get { return fan.FanSays; } }
    public ObservableCollection<string> PitcherSays { get { return pitcher.PitcherSays; } }

    public int Trajectory { get; set; }
    public int Distance { get; set; }

    public BaseballSimulator()     {
        pitcher = new Pitcher(ball);
        fan = new Fan(ball);
    }

    public void PlayBall() {
        BallEventArgs ballEventArgs = new BallEventArgs(Trajectory, Distance);
        ball.OnBallInPlay(ballEventArgs);
    }
}
```

4 BUILD THE MAIN WINDOW.

Can you come up with the XAML just from looking at the screenshot to the right? The two TextBox controls are bound to the `Trajectory` and `Distance` properties of the `BaseballSimulator` static resource, and the pitcher and fan chatter are ListView controls bound to the two `ObservableCollections`.



See if you can make your simulator generate the above fan and pitcher chatter with three successive balls put into play. Write down the values you used to get the result below:

Ball 1:

Trajectory: .....

Distance: .....

Ball 2:

Trajectory: .....

Distance: .....

Ball 3:

Trajectory: .....

Distance: .....

Here are the `Ball` and `BallEventArgs` from earlier, and the new `Fan` class that needed to be added:

```
class Ball {
    public event EventHandler BallInPlay;
    public void OnBallInPlay(BallEventArgs e) {
        EventHandler ballInPlay = BallInPlay;
        if (ballInPlay != null)
            ballInPlay(this, e);
    }
}
```

The OnBallInPlay() method just raises the BallInPlay event—but it has to check to make sure it's not null; otherwise, it'll throw an exception.

Read-only automatic properties work really well in event arguments because the event handlers read only the data passed to them.

```
class BallEventArgs : EventArgs {
    public int Trajectory { get; private set; }
    public int Distance { get; private set; }
    public BallEventArgs(int trajectory, int distance)
    {
        this.Trajectory = trajectory;
        this.Distance = distance;
    }
}

using System.Collections.ObjectModel;
class Fan {
    public ObservableCollection<string> FanSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Fan(Ball ball) {
        ball.BallInPlay += new EventHandler(ball_BallInPlay);
    }

    void ball_BallInPlay(object sender, EventArgs e)  {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if (ballEventArgs.Distance > 400 && ballEventArgs.Trajectory > 30)
                FanSays.Add("Pitch #" + pitchNumber
                            + ": Home run! I'm going for the ball!");
            else
                FanSays.Add("Pitch #" + pitchNumber + ": Woo-hoo! Yeah!");
        }
    }
}
```

The Fan object's constructor chains its event handler onto the BallInPlay event.

The fan's BallInPlay event handler looks for any ball that's high and long.

Here's the code-behind for the page:

```
public partial class MainWindow : Window {
    BaseballSimulator baseballSimulator;

    public MainWindow()     {
        InitializeComponent();

        baseballSimulator = FindResource("baseballSimulator") as BaseballSimulator;
    }

    private void Button_Click(object sender, RoutedEventArgs e)     {
        baseballSimulator.PlayBall();
    }
}
```

Here's the XAML for the page. It also needs: `<local:BaseballSimulator x:Key="baseballSimulator"/>`

```
<Window.Resources>
        <local:BaseballSimulator x:Key="baseballSimulator"/>
</Window.Resources>

<Grid Margin="5" DataContext="{StaticResource ResourceKey=baseballSimulator}">
  <Grid.ColumnDefinitions>
      <ColumnDefinition Width="200" />
      <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <StackPanel Margin="0,0,10,0">
    <TextBlock Text="Trajectory" Margin="0,0,0,5"/>
    <TextBox Text="{Binding Trajectory, Mode=TwoWay}"  Margin="0,0,0,5"/>
    <TextBlock Text="Distance" Margin="0,0,0,5"/>
    <TextBox Text="{Binding Distance, Mode=TwoWay}" Margin="0,0,0,5"/>
    <Button Content="Play ball!" Click="Button_Click"/>
  </StackPanel>
  <StackPanel Grid.Column="1">
    <TextBlock Text="Pitcher says" Margin="0,0,0,5"/>
    <ListView ItemsSource="{Binding PitcherSays}" Height="125"/>
    <TextBlock Text="Fan says" Margin="0,0,0,5"/>
    <ListView ItemsSource="{Binding FanSays}" Height="125"/>
  </StackPanel>
</Grid>
```

Make sure you also add the xmlns:local property to the <Window> tag.

And here's the `Pitcher` class (it needs `using System.Collections.ObjectModel;` at the top):

```
class Pitcher {
    public ObservableCollection<string> PitcherSays = new ObservableCollection<string>();
    private int pitchNumber = 0;

    public Pitcher(Ball ball)     {
        ball.BallInPlay += ball_BallInPlay;
    }

    void ball_BallInPlay(object sender, EventArgs e) {
        pitchNumber++;
        if (e is BallEventArgs) {
            BallEventArgs ballEventArgs = e as BallEventArgs;
            if ((ballEventArgs.Distance < 95) && (ballEventArgs.Trajectory < 60))
                CatchBall();
            else
                CoverFirstBase();
        }
    }
    private void CatchBall() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I caught the ball");
    }
    private void CoverFirstBase() {
        PitcherSays.Add("Pitch #" + pitchNumber + ": I covered first base");
    }
}
```

We gave you the pitcher's BallInPlay event handler. It looks for any low balls.

Here are the values we used to get the output. Yours might be a little different.

Ball 1:
Trajectory: 75
Distance: 105

Ball 2:
Trajectory: 48
Distance: 80

Ball 3:
Trajectory: 40
Distance: 435

XAML controls use routed events

Flip to page 722 in the main part of the book and have a closer look at the IntelliSense window that pops up when you type `override` into the IDE. Yes, it's for a Windows Store app, but the *same exact principle* applies to WPF. Two of the names of the event argument types are a little different from the others. The `DoubleTapped` event's second argument has the type `DoubleTappedRoutedEventArgs`, and the `GotFocus` event's is a `RoutedEventArgs`. The reason is that the `DoubleTapped` and `GotFocus` events are **routed events**. These are like normal events, except for one difference: when a control object responds to a routed event, first it fires off the event handler method as usual. Then it does something else: if the event hasn't been handled, it **sends the routed event up to its container**. The container fires the event, and then if it isn't handled, it sends the routed event up to its container. The event keeps **bubbling up** until it's either handled or it hits the **root**, or the container at the very top. Here's a typical routed event handler method signature.

```
private void EventHandler(object sender, RoutedEventArgs e)
```

The `RoutedEventArgs` object has a property called **`Handled`** that the event handler can use to indicate that it's handled the event. Setting this property to `true` **stops the event from bubbling up**.

In both routed and standard events, the `sender` parameter always contains a reference to the object that called the event handler. So if an event is bubbled up from a control to a container like a Grid, then when the Grid calls its event handler, `sender` will be a reference to the Grid control. But what if you want to find out which control fired the original event? No problem. The `RoutedEventArgs` object has a property called **`OriginalSource`** that contains a reference to the control that initially fired the event. If `OriginalSource` and `sender` point to the same object, then the control that called the event handler is the same control that originated the event and started it bubbling up.

IsHitTestVisible determines if an element is "visible" to the pointer or mouse

Typically, any element on the page can be "hit" by the pointer or mouse—as long as it meets certain criteria. It needs to be visible (which you can change with the `Visibility` property), it has to have a `Background` or `Fill` property that's not null (but can be `Transparent`), it must be enabled (with the `IsEnabled` property), and it has to have a `height` and `width` greater than zero. If all of these things are true, then the **`IsHitTestVisible` property will return `True`**, and that will cause it to respond to pointer or mouse events.

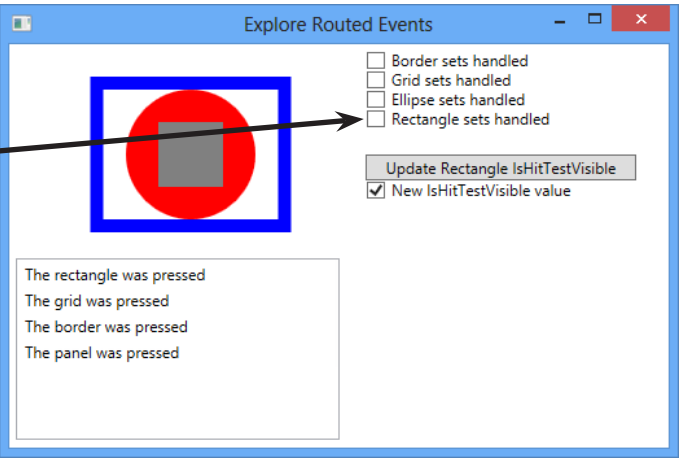
This property is especially useful if you want to make your events "invisible" to the mouse. If you set `IsHitTestVisible` to `False`, then any pointer taps or mouse clicks will **pass right through the control**. If there's another control below it, that control will get the event instead.

You can see a list of input events that are routed events here:
http://msdn.microsoft.com/en-us/library/windows/apps/Hh758286.aspx

The structure of controls that contain other controls that in turn contain yet more controls is called an object tree, and routed events bubble up the tree from child to parent until they hit the root element at the top.

Create an app to explore routed events

Here's a WPF application that you can use to experiment with routed events. It's got a StackPanel that contains a Border, which contains a Grid, and inside that grid are an Ellipse and a Rectangle. Have a look at the screenshot. See how the Rectangle is on top of the Ellipse? If you put two controls into the same cell, they'll stack on top of each other. But both of those controls have the same parent: the Grid, whose parent is the Border, and the Border's parent is the StackPanel. Routed events from the Rectangle or Ellipse bubble up through the parents to the root of the **object tree**.



You've already seen the **CheckBox** control, which you can use to toggle a value on and off. The `Content` property sets the label for the control. The `IsChecked` property is a **Nullable<bool>** because in addition to on and off, it can also have a third indeterminate state

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>
    <StackPanel x:Name="panel" MouseDown="StackPanel_MouseDown">
        <Border BorderThickness="10" BorderBrush="Blue" Width="155" x:Name="border"
            Margin="20" MouseDown="Border_MouseDown">
            <Grid x:Name="grid" MouseDown="Grid_MouseDown">
                <Ellipse Fill="Red"  Width="100" Height="100"
                    MouseDown="Ellipse_MouseDown"/>
                <Rectangle Fill="Gray" Width="50" Height="50"
                    MouseDown="Rectangle_MouseDown" x:Name="grayRectangle"/>
            </Grid>
        </Border>
        <ListBox BorderThickness="1" Width="250" Height="140" x:Name="output" Margin="0,0,20,0"/>
    </StackPanel>
    <StackPanel Grid.Column="1">
        <CheckBox Content="Border sets handled" x:Name="borderSetsHandled"/>
        <CheckBox Content="Grid sets handled" x:Name="gridSetsHandled" />
        <CheckBox Content="Ellipse sets handled" x:Name="ellipseSetsHandled"/>
        <CheckBox Content="Rectangle sets handled" x:Name="rectangleSetsHandled"/>
        <Button Content="Update Rectangle IsHitTestVisible"
            Click="UpdateHitTestButton" Margin="0,20,20,0"/>
        <CheckBox IsChecked="True" Content="New IsHitTestVisible value"
            x:Name="newHitTestVisibleValue" />
    </StackPanel>
</Grid>
```

Routed events bubble up the object tree.

IsChecked defaults to False. This CheckBox has it set to True because controls always have IsHitTestVisible set to true by default.

Flip the page to finish the app

YOU'LL NEED THIS ObservableCollection TO DISPLAY OUTPUT IN THE LISTBOX.

Make a field called `outputItems` and set the `ListBox.ItemsSource` property in the page constructor. And don't forget to add the `using System.Collections.ObjectModel;` statement for `ObservableCollection<T>`.

```
public partial class MainWindow : Window {
    ObservableCollection<string> outputItems = new ObservableCollection<string>();

    public MainWindow() {
        this.InitializeComponent();

        output.ItemsSource = outputItems;
    }
```

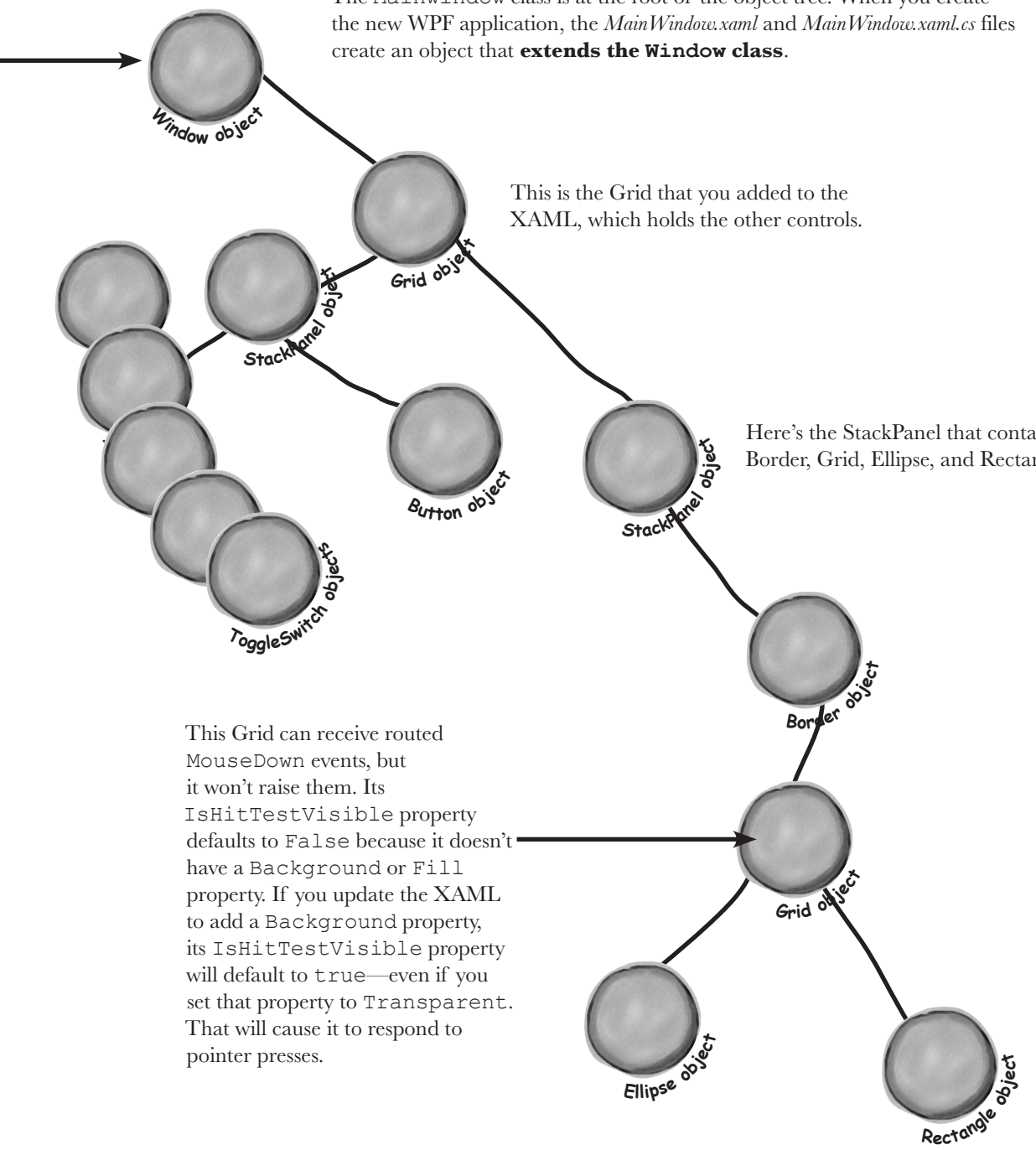
Here's the code-behind. Each control's `MouseDown` event handler clears the output if it's the original source, and then it adds a string to the output. If its "handled" toggle switch is on, it uses `e.Handled` to handle the event.

```
private void Ellipse_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The ellipse was pressed");
    if (ellipseSetsHandled.IsChecked == true) e.Handled = true;
}

private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;
}

private void Grid_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The grid was pressed");
    if (gridSetsHandled.IsChecked == true) e.Handled = true;
}

private void Border_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The border was pressed");
    if (borderSetsHandled.IsChecked == true) e.Handled = true;
}

private void StackPanel_MouseDown(object sender, MouseButtonEventArgs e) {
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The panel was pressed");
}

private void UpdateHitTestButton(object sender, RoutedEventArgs e) {
    grayRectangle.IsHitTestVisible = (bool)newHitTestVisibleValue.IsChecked;
}
```

The Click event handler for the button uses the IsOn property of the toggle switch to turn IsHitTestVisible on or off for the Rectangle control.

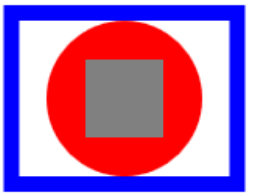
HERE'S THE OBJECT GRAPH FOR YOUR MAIN WINDOW.

The `Mainwindow` class is at the root of the object tree. When you create the new WPF application, the *MainWindow.xaml* and *MainWindow.xaml.cs* files create an object that **extends the `Window` class**.



Flip the page to use your new app to explore routed events →

RUN THE APP AND CLICK OR TAP THE GRAY RECTANGLE.



You should see the output in the screenshot to the right. ⟶

You can see exactly what's going on by putting a breakpoint on the first line of `Rectangle_MouseDown()`, the Rectangle control's `MouseDown` event handler:

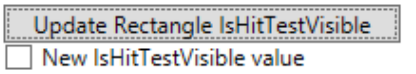
```
private void Rectangle_MouseDown(object sender, MouseButtonEventArgs e)
{
    if (sender == e.OriginalSource) outputItems.Clear();
    outputItems.Add("The rectangle was pressed");
    if (rectangleSetsHandled.IsChecked == true) e.Handled = true;
}
```

The rectangle was pressed
The grid was pressed
The border was pressed
The panel was pressed

Click the gray rectangle again—this time the breakpoint should fire. Use Step Over (F10) to **step through the code line by line**. First you'll see the `if` block execute to clear the `outputItems ObservableCollection` that's bound to the ListBox. This happens because `sender` and `e.OriginalSource` reference the same Rectangle control, which is true only inside the event handler method for the control that originated the event (in this case, the control that you clicked or tapped), so `sender == e.OriginalSource` is true.

When you get to the end of the method, **keep stepping through the program**. The event will bubble up through the object tree, first running the Rectangle's event handler, then the Grid's event handler, then the Border's, then the Panel's, and finally it runs an event handler method that's part of `LayoutAwarePage`—this is outside of your code and not part of the routed event, so it will always run. Since none of those controls are the original source for the event, none of their senders will be the same as `e.OriginalSource`, so none of them clear the output.

TURN *IsHitTestVisible* OFF, PRESS THE "UPDATE" BUTTON, AND THEN CLICK OR TAP THE RECTANGLE.



The ellipse was pressed
The grid was pressed
The border was pressed
The panel was pressed

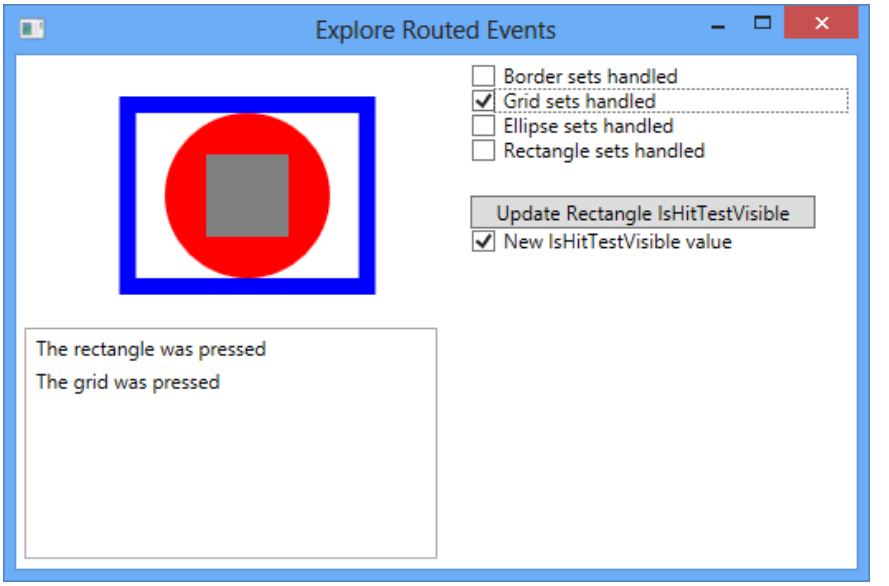
⟵ You should see this output.

Wait a minute! You pressed the Rectangle, but the Ellipse control's `MouseDown` event handler fired. What's going on?

When you pressed the button, its `Click` event handler updated the Rectangle control's `IsHitTestVisible` property to `false`, which made it "invisible" to pointer presses, clicks, and other pointer events. So when you tapped the Rectangle, your tap passed right through it to the topmost control underneath it on the page that has `IsHitTestVisible` set to `true` and has a `Background` property that's set to a color or `Transparent`. In this case, it finds the Ellipse control and fires its `MouseDown` event.

CHECK THE "GRID SETS HANDLED" BOX AND CLICK OR TAP THE GRAY RECTANGLE.

You should see this output. ⟶



So why did only two lines get added to the output ListBox? **Step through the code again** to see what's going on. This time, `gridSetsHandled.IsOn` was true because you toggled the `gridSetsHandled` to `On`, so the last line in **the Grid's event handler set `e.IsHandled` to `true`**. As soon as a routed event handler method does that, **the event stops bubbling up**. As soon as the Grid's event handler completes, the app sees that the event has been handled, so it doesn't call the Border or Panel's event handler method, and instead skips to the event handler method in `LayoutAwarePage` that's outside of the code you added.

USE THE APP TO EXPERIMENT WITH ROUTED EVENTS.

Here are a few things to try:

- Click on the gray Rectangle and the red Ellipse and watch the output to see how the events bubble up.
- Turn on each of the toggle switches, starting at the top, to cause the event handlers to set `e.Handled` to `true`. Watch the events stop bubbling when they're handled.
- Set breakpoints and debug through all of the event handler methods.
- Try setting a breakpoint in the Ellipse's event handler method, and then turn the gray Rectangle's `IsHitTestVisible` property on and off by toggling the bottom switch and pressing the button. Step through the code for the Rectangle when `IsHitTestVisible` is set to `false`.
- Stop the program and add a `Background` property to the Grid to make it visible to pointer hits.

A routed event first fires the event handler for the control that originated the event, and then bubbles up through the control hierarchy until it hits the top—or an event handler sets e.Handled to true.

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Chapter 16

WHEN YOU BUILD YOUR APPS USING THE **MODEL-VIEW-VIEWMODEL** PATTERN, YOUR CODE IS EASIER TO BUILD TODAY... AND TO MANAGE TOMORROW.

Great developers follow design patterns.

In this chapter, you'll learn about Model-View-ViewModel (MVVM), a design pattern for building effective WPF apps. Along the way, you'll learn what a design pattern is, and you'll learn how to use XAML controls to create great animations.

Here's how we recommend that you work through Chapter 16:

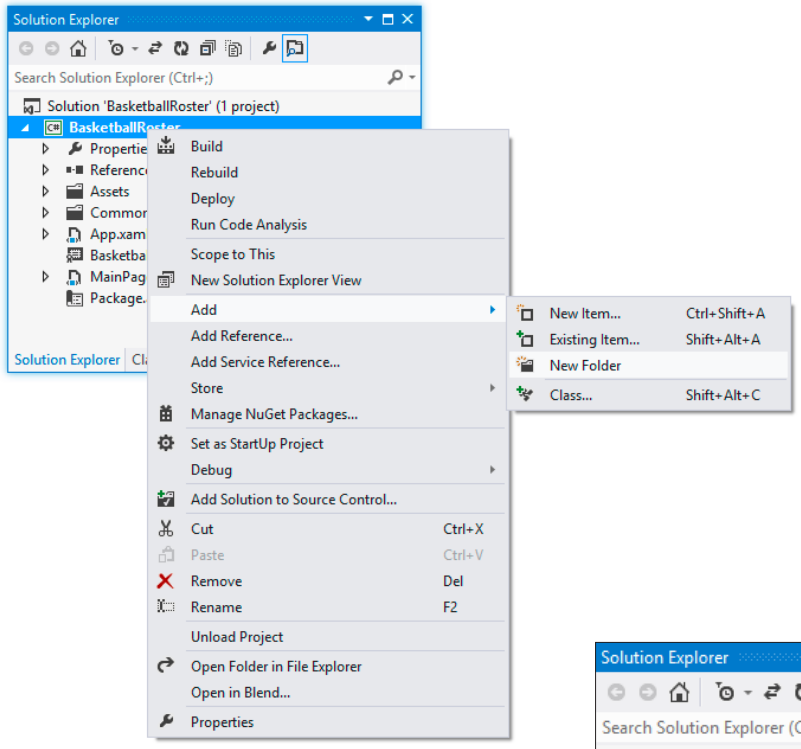
- Read through page 749.
- Follow our replacement pages for 750–757.
- Read pages 758–764.
- Start the Stopwatch project on page 762 in the book, and continue it using a combination of book pages and appendix replacement pages 765, 768, 770–773, and 781–787.
- Read page 788 in the book.
- The rest of Chapter 16 is replaced with pages 789–807 in this appendix.
- There's information on page 806 about how to do Lab #3.

Use the MVVM pattern to start building the basketball roster app

Create a new WPF application and **make sure it's called BasketballRoster** (because we'll be using the namespace `BasketballRoster` in the code, and this will make sure your code matches what's on the next few pages).

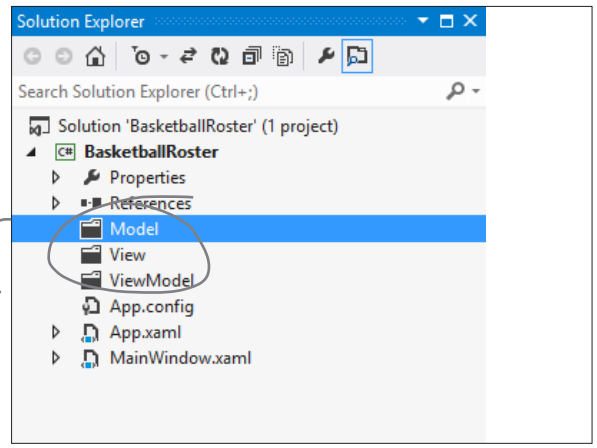
1 CREATE THE MODEL, VIEW, AND VIEWMODEL FOLDERS IN THE PROJECT.

Right-click on the project in the Solution Explorer and choose New Folder from the Add menu:

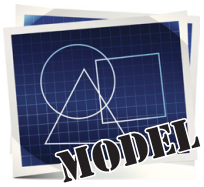


When you use the Solution Explorer to add a new folder to your project, the IDE creates a new namespace based on the folder name. This causes the Add→Class... menu option to create classes with that namespace. So if you add a class to the Model folder, the IDE will add `BasketballRoster.Model` to the `namespace` line at the top of the class file.

Add a *Model* folder. Then do it two more times to add the *View* and *ViewModel* folders, so your project looks like this:



These folders will hold the classes, controls, and windows for your app.



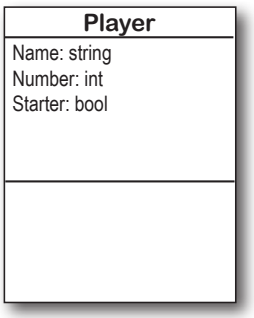
2 START BUILDING THE MODEL BY ADDING THE PLAYER CLASS.

Right-click on the *Model* folder and **add a class called `Player`**. When you add a class into a folder, the IDE updates the namespace to add the folder name to the end. Here's the `Player` class:

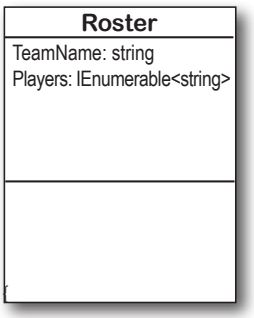
When you add a class file into a folder, the IDE adds the folder name to the namespace.

```
namespace BasketballRoster.Model {
    class Player {
        public string Name { get; private set; }
        public int Number { get;  private set; }
        public bool Starter { get; private set; }

        public Player(string name, int number, bool starter) {
            Name = name;
            Number = number;
            Starter = starter;
        }
    }
}
```

Different classes concerned with different things? This sounds familiar...



These classes are small because they're only concerned with keeping track of which players are in each roster. None of the classes in the Model are concerned with displaying the data, just managing it.



3 FINISH THE MODEL BY ADDING THE ROSTER CLASS

Next, **add the `Roster` class to the *Model* folder**. Here's the code for it.

```
namespace BasketballRoster.Model {
    class Roster {
        public string TeamName { get; private set; }

        private readonly List<Player> _players = new List<Player>();
        public IEnumerable<Player> Players {
            get { return new List<Player>(_players); }
        }

        public Roster(string teamName, IEnumerable<Player> players) {
            TeamName = teamName;
            _players.AddRange(players);
        }
    }
}
```

The _ tells you that this field is private.

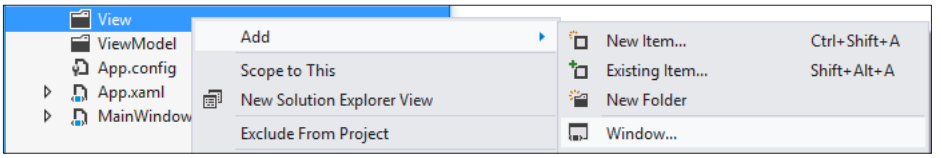
Your *Model* folder should now look like this:

We added an underscore to the beginning of the name of the `_players` field. Adding an underscore to the beginning of private fields is a very common naming convention. We're going to use it throughout this chapter so you can get used to seeing it.

We'll add the view on the next page

4 **ADD A NEW MAIN WINDOW TO THE VIEW FOLDER.**

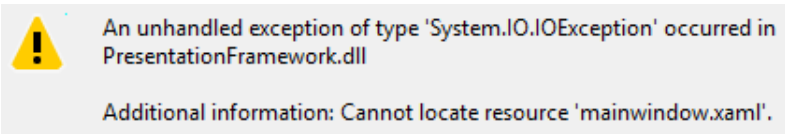
Right-click on the *View* folder and **add a new Window called** *LeagueWindow.xaml*.



Your project's View folder should now have a XAML window in it called LeagueWindow.xaml. This is just like the MainWindow.xaml window that you've been working with throughout the book. It's still a Window object with a graph that's defined with XAML. The only difference is that it's called LeagueWindow instead of MainWindow.

5 **DELETE THE MAIN WINDOW AND REPLACE IT WITH YOUR NEW WINDOW.**

Delete the *MainWindow.xaml* file from the project by **right-clicking on it and choosing Delete**. Now try building and running your project—you'll get an exception when the program starts:



Well, that makes sense, since you deleted MainWindow.xaml. When a WPF application starts up, it shows the **window specified in the StartupUri property in the <Application> tag App.xaml**:

```
<Application x:Class="BasketballRoster.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Open App.xaml and edit StartupUri so your program pops up the window you just added:

```
<Application x:Class="BasketballRoster.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="View/LeagueWindow.xaml">
```

Once you make that change, rebuild and rerun your program. Now it should start and show your newly added window.

User controls let you create your own controls

Take a look at the basketball roster program that you're building. Each team gets an identical set of controls: a TextBlock, another TextBlock, a ListView, another TextBlock, and another ListView, all wrapped up by a StackPanel inside a Border. Do we really need to add two identical sets of controls to the page? What if we want to add a third and fourth team—that's going to mean a whole lot of duplication. And that's where **user controls** come in. A user control is a class that you can use to create your own controls. You use XAML and code-behind to build a user control, just like you do when you build a page. Let's get started and add a user control to your BasketballRoster project.

UserControl is a base class that gives you a way to encapsulate controls that are related to each other, and lets you build logic that defines the behavior of the control.

1 Add a new user control to your View folder.

Right-click on the *View* folder and add a new item. Choose  from the dialog and call it *RosterControl.xaml*.

2 Look at the code-behind for the new user control.

Open up *RosterControl.xaml.cs*. Your new control extends the `UserControl` base class. Any code-behind that defines the user control's behavior goes here.

```
namespace BasketballRoster.View
{
    /// <summary>
    /// Interaction logic for RosterControl.xaml
    /// </summary>
    public partial class RosterControl : UserControl
    {
        public RosterControl()
        {
            InitializeComponent();
        }
    }
}
```

3 Look at the XAML for the new user control.

The IDE added a user control with an empty `<Grid>`. Your XAML will go here.

Before you flip the page, see if you can figure out what XAML should go into the new `RosterControl` by looking at the Windows Store app screenshot on page 746.

- It will have a `<StackPanel>` to stack up the controls that live inside a blue `<Border>`. Can you figure out which property gives a Border control rounded corners?
- It has two ListView controls that display data for players, so it also needs a `<UserControl.Resources>` section that contains a `DataTemplate`. We called it `PlayerItemTemplate`.
- Bind the ListView items to properties called `Starters` and `Bench`, and the top TextBlock to a property called `TeamName`.
- The Border control lives inside a `<Grid>` with a single row that has `Height="Auto"` to keep it from expanding past the bottom of the ListView controls to fill up the entire page.

"TEACH A MAN TO FISH..."

We're nearing the end of the book, so we want to challenge you with problems that are similar to ones you'll face in the real world. A good programmer takes a lot of educated guesses, so we're giving you barely enough information about how a `UserControl` works. You don't even have binding set up, so you won't see data in the designer! How much of the XAML can you build before you flip the page to see the code for `RosterControl`?

4 Finish the RosterControl XAML.

Here's the code for the `RosterControl` user control that you added to the *View* folder. Did you notice how we gave you properties for binding, but no data context? That should make sense. The two controls on the page show different data, so the page will set different data contexts for each of them.

```
<UserControl x:Class="BasketballRoster.View.RosterControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="300">

    <UserControl.Resources>
        <DataTemplate x:Key="PlayerItemTemplate">
            <TextBlock>
                <Run Text="{Binding Name, Mode=OneWay}"/>
                <Run Text=" #"/>
                <Run Text="{Binding Number, Mode=OneWay}"/>
            </TextBlock>
        </DataTemplate>
    </UserControl.Resources>

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

        <Border BorderThickness="2" BorderBrush="Blue" CornerRadius="6" Background="Black">
            <StackPanel Margin="20">
                <TextBlock Foreground="White" FontFamily="Segoe" FontSize="20px"
                           FontWeight="Bold" Text="{Binding TeamName}" />
                <TextBlock Foreground="White" FontFamily="Segoe" FontSize="16px"
                           Text="Starting Players" Margin="0,5,0,0"/>
                <ListView Background="Black" Foreground="White"  Margin="0,5,0,0"
                                      ItemTemplate="{StaticResource PlayerItemTemplate}"
                                      ItemsSource="{Binding Starters}" />
                <TextBlock Foreground="White"  FontFamily="Segoe" FontSize="16px"
                           Text="Bench Players" Margin="0,5,0,0"/>
                <ListView Background="Black" Foreground="White" ItemsSource="{Binding Bench}"
                          ItemTemplate="{StaticResource PlayerItemTemplate}" Margin="0,5,0,0"/>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

You already know that controls change size based on their Height and Width properties. You can change these numbers to alter how the control is displayed in the IDE's Designer window when you're modifying it.

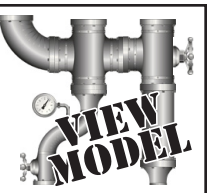
We put the data template for the ListView items in its own static resource. Then, instead of having a <ListView.ItemTemplate> section we used the static resource using the ItemTemplate property in the ListView tag:

```
ItemTemplate="{StaticResource PlayerItemTemplate}"
```

You can use the CornerRadius property to give a Border rounded corners.

Both ListView controls use the same template defined as a static resource.

Build the ViewModel for the BasketballRoster app by looking at the data in the Model and the bindings in the View, and figuring out what "plumbing" the app needs to connect them together.



1 ADD THE ROSTER CONTROLS TO LEAGUEWINDOW.XAML.

First add these `xmlns` properties to the page so it recognizes the new namespaces:

```
xmlns:view="clr-namespace:BasketballRoster.View"
xmlns:viewmodel="clr-namespace:BasketballRoster.ViewModel"
```

Then add an instance of `LeagueViewModel` as a static resource:

```
<Window.Resources>
    <viewmodel:LeagueViewModel x:Key="LeagueViewModel"/>
</Window.Resources>
```

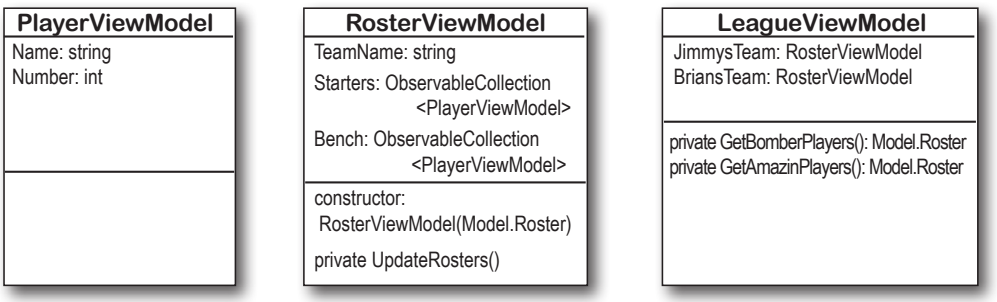
Now you can add a StackPanel with two `RosterControls` to the page:

```
<StackPanel Orientation="Horizontal" Margin="5"
            VerticalAlignment="Center" HorizontalAlignment="Center"
            DataContext="{StaticResource ResourceKey=LeagueViewModel}" >
    <view:RosterControl Width="200" DataContext="{Binding JimmysTeam}" Margin="0,0,20,0" />
    <view:RosterControl Width="200" DataContext="{Binding BriansTeam}" />
</StackPanel>
```

Make sure you created the classes and pages in the right folders; otherwise, the namespaces won't match the code in the solution.

2 CREATE THE VIEWMODEL CLASSES.

Create these three classes in the *ViewModel* folder.



3 MAKE THE VIEWMODEL CLASSES WORK.

- The `PlayerViewModel` class is a simple data object with two properties.
- The `LeagueViewModel` class has two private methods to create dummy data for the page. It creates `Model.Roster` objects for each team that get passed to the RosterViewModel constructor.
- The `RosterViewModel` class has a constructor that takes a `Model.Roster` object. It sets the `TeamName` property, and then it calls its private `UpdateRosters()` method, which uses LINQ queries to extract the starting and bench players and update the `Starters` and `Bench` properties. Add **`using Model;`** to the top of the classes so you can use objects in the `Model` namespace.

See page 748 for a hint about the LINQ query...

If the IDE gives you an error message in the XAML designer that `LeagueViewModel` does not exist in the `ViewModel` namespace, but you're 100% certain you added it correctly, try right-clicking on the BasketballRoster project and choosing Unload Project, and then right-click again and choose Reload Project to reload it. But make sure you don't have any errors in any of the C# code files.

The ViewModel for the BasketballRoster app has three classes: `LeagueViewModel`, `PlayerViewModel`, and `RosterViewModel`. They all live in the *ViewModel* folder.

If you left out the `using Model;` line then you'd have to use `Model.Roster` instead of `Roster` everywhere.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;

    class LeagueViewModel {
        public RosterViewModel BriansTeam { get; set; }
        public RosterViewModel JimmysTeam { get; set; }

        public LeagueViewModel() {
            Roster briansRoster = new Roster("The Bombers", GetBomberPlayers());
            BriansTeam = new RosterViewModel(briansRoster);

            Roster jimmysRoster = new Roster("The Amazins", GetAmazinPlayers());
            JimmysTeam = new RosterViewModel(jimmysRoster);
        }

        private IEnumerable<Player> GetBomberPlayers() {
            List<Player> bomberPlayers = new List<Player>() {
                new Player("Brian", 31, true),
                new Player("Lloyd", 23, true),
                new Player("Kathleen",6, true),
                new Player("Mike", 0, true),
                new Player("Joe", 42, true),
                new Player("Herb",32, false),
                new Player("Fingers",8, false),
            };
            return bomberPlayers;
        }

        private IEnumerable<Player> GetAmazinPlayers() {
            List<Player> amazinPlayers = new List<Player>() {
                new Player("Jimmy",42, true),
                new Player("Henry",11, true),
                new Player("Bob",4, true),
                new Player("Lucinda", 18, true),
                new Player("Kim", 16, true),
                new Player("Bertha", 23, false),
                new Player("Ed",21,  false),
            };
            return amazinPlayers;
        }
    }
}

namespace BasketballRoster.ViewModel {
    class PlayerViewModel {
        public string Name { get; set; }
        public int Number { get; set; }

        public PlayerViewModel(string name, int number) {
            Name = name;
            Number = number;
        }
    }
}
```

LeagueViewModel exposes RosterViewModel objects that a RosterControl can use as its data context. It creates the Roster model object for the RosterViewModel to use.

This private method generates dummy data for the Bombers by creating a new List of Player objects.

Dummy data typically goes in the ViewModel because the state of an MVVM application is managed using instances of the Model classes that are encapsulated inside the ViewModel objects.

You use classes from the View to store your data, which is why this method returns Player objects and not PlayerViewModel objects.

Here's the PlayerViewModel. It's just a simple data object with properties for the data template to bind to.

In a typical MVVM app, only classes in the ViewModel implement `INotifyPropertyChanged` because those are the only objects that XAML controls are bound to.

```
namespace BasketballRoster.ViewModel {
    using Model;
    using System.Collections.ObjectModel;
    using System.ComponentModel;

    class RosterViewModel {
        public ObservableCollection<PlayerViewModel> Starters { get; set; }
        public ObservableCollection<PlayerViewModel> Bench { get; set; }

        private Roster _roster;

        private string _teamName;
        public string TeamName {
            get { return _teamName; }
            set {
                _teamName = value;
            }
        }

        public RosterViewModel(Roster roster) {
            _roster = roster;

            Starters = new ObservableCollection<PlayerViewModel>();
            Bench = new ObservableCollection<PlayerViewModel>();

            TeamName = _roster.TeamName;

            UpdateRosters();
        }

        private void UpdateRosters() {
            var startingPlayers =
                from player in _roster.Players
                where player.Starter
                select player;

            foreach (Player player in startingPlayers)
                Starters.Add(new PlayerViewModel(player.Name, player.Number));

            var benchPlayers =
                from player in _roster.Players
                where player.Starter == false
                select player;

            foreach (Player player in benchPlayers)
                Bench.Add(new PlayerViewModel(player.Name, player.Number));
        }
    }
}
```

This is where the app stores its state—in Roster objects encapsulated inside the ViewModel. The rest of the class translates the Model data into properties that the View can bind to.

Whenever the TeamName property changes, the RosterViewModel fires off a PropertyChanged event so any object bound to it will get updated.

This LINQ query finds all the starting players and adds them to the Starters ObservableCollection property.

Here's a similar LINQ query to find the bench players.

In a typical MVVM app, only classes in the ViewModel implement `INotifyPropertyChanged`.

That's because the ViewModel contains the only objects that XAML controls are bound to. In this project, however, we didn't need to implement INotifyPropertyChanged because the bound properties are updated in the constructor. If you wanted to modify the project to let Brian and Jimmy change their team names, you'd need to fire a PropertyChanged event in the TeamName set accessor.

There is one change you'll need to make to get the ViewModel code on pages 766 and 767 in the book to work. On page 766 you're given three using statements, including this one:

```
using Windows.UI.Xaml;
```

You'll need to replace it with this using statement:

```
using System.Windows.Threading;
```

The Windows.UI.Xaml namespace is part of the .NET Framework for Windows Store, so you don't use it for WPF applications. But you need System.Windows.Threading because your ViewModel has a DispatcherTimer.

Other than that change, the code is identical. This is a good example of decoupled layers in the Model-View-ViewModel pattern: since you used identical C# code (except for that one using statement) for the ViewModel and Model, you could reuse those classes to port the stopwatch app to WPF.

Build the view for a simple stopwatch

Here's the XAML for a simple stopwatch control. **Add a WPF user control to the *View* folder called *BasicStopwatch.xaml*** and add this code. The control has TextBlock controls to display the elapsed and lap times, and buttons to start, stop, reset, and take the lap time.

```
<UserControl x:Class="Stopwatch.View.BasicStopwatch"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300"
             xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel">

    <UserControl.Resources>
        <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    </UserControl.Resources>

    <Grid DataContext="{StaticResource ResourceKey=viewModel}">
        <StackPanel>
            <TextBlock>
                <Run>Elapsed time: </Run>
                <Run Text="{Binding Hours, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding Minutes, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding Seconds, Mode=OneWay}"/>
            </TextBlock>
            <TextBlock>
                <Run>Lap time: </Run>
                <Run Text="{Binding LapHours, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding LapMinutes, Mode=OneWay}"/>
                <Run>:</Run>
                <Run Text="{Binding LapSeconds, Mode=OneWay}"/>
            </TextBlock>
            <StackPanel Orientation="Horizontal">
                <Button Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
                <Button Click="StopButton_Click" Margin="0,0,5,0">Stop</Button>
                <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
                <Button Click="LapButton_Click">Lap</Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

You'll need this xmlns property to add the namespace. We called our project Stopwatch, so the ViewModel namespace is Stopwatch.ViewModel.

This user control stores an instance of the ViewModel as a static resource and uses it as its data context. It doesn't need its container to set a data context. It keeps track of its own state.

The ViewModel has read-only properties for Hours, Minutes, Seconds, etc. WPF requires one-way binding for read-only properties.

This TextBlock is bound to properties in the ViewModel that return the elapsed time.

The ViewModel must be firing off PropertyChanged events to keep these values up to date.

This TextBlock is bound to properties that expose the lap time.

You'll need to add `Click` event handlers to the control and a `StopwatchViewModel` class to the `ViewModel` namespace for this to compile.

Here's a hint: use a DispatcherTimer to constantly check the Model and update the properties.

The code for the ViewModel is on pages 766 and 767 in the book. How much of the ViewModel code can you build just from the View and Model code before you flip the page? Add a BasicStopwatch control to the main window and see how far you can get.

But be really careful and don't assume the IDE is necessarily wrong. Sometimes an error in the XAML for one page (like a broken `xmlns` property) can cause all the designers to break.

Finish the stopwatch app

There are just a few more loose ends to tie together. Your BasicStopwatch user control doesn't have event handlers, so you need to add them. And then you just need to add the control to your main window.

1. First, **go back to *BasicStopwatch.xaml.cs*** and add these event handlers to the code-behind:

```
ViewModel.StopwatchViewModel viewModel;

public BasicStopwatch() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
}

private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}

private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}

private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}

private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

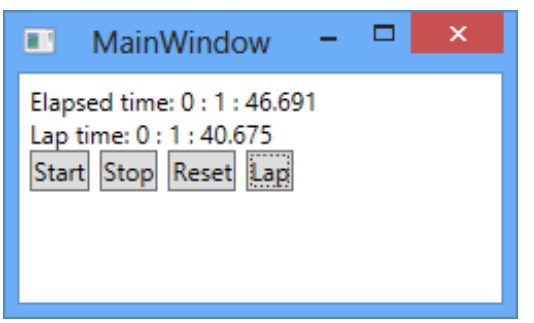
The buttons in the view just call methods in the ViewModel. This is a pretty typical pattern for the View.

2. Here's **all the XAML** for *MainWindow.xaml*:

```
<Window x:Class="Stopwatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="150" Width="250"
        xmlns:view="clr-namespace:Stopwatch.View">
    <Grid>
        <view:BasicStopwatch Margin="5"/>
    </Grid>
</Window>
```

All the behavior is in the user control, so there's no code-behind for the main window.

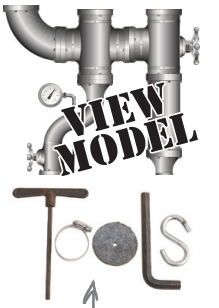
Your app should now run. Click the Start, Stop, Reset, and Lap buttons to see your stopwatch work.



We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Converters automatically convert values for binding

Anyone with a digital clock knows that it typically shows the minutes with a leading zero. Our stopwatch should also show the minutes with two digits. And it should show the seconds with two digits, and round to the nearest hundredth of a second. We *could* modify the ViewModel to expose string values that are formatted properly, but that would mean that we'd need to keep adding more and more properties each time we wanted to reformat the same data. That's where **value converters** come in very handy. A value converter is an object that the XAML binding uses to modify data before it's passed to the control. You can build a value converter by implementing the `IValueConverter` interface (which is in the `System.Windows.Data` namespace). Add a value converter to your stopwatch now.



Converters are useful tools for building your ViewModel.

1. **Add the TimeNumberFormatConverter class to the ViewModel folder.**
   Add **`using System.Windows.Data;`** to the top of the class, and then have it implement the **`IValueConverter`** interface. Use the IDE to automatically implement the interface. This will add two method stubs for the `Convert()` and `ConvertBack()` methods.

2. **Implement the Convert() method in the value converter.**
   The `Convert()` method takes several parameters—we'll use two of them. The **`value`** parameter is the raw value that's passed into the binding, and **`parameter`** lets you specify a parameter in XAML.

```
using System.Windows.Data;

class TimeNumberFormatConverter : IValueConverter {
    public object Convert(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture) {
        if (value is decimal)
            return ((decimal)value).ToString("00.00");
        else if (value is int) {
            if (parameter == null)
                return ((int)value).ToString("d1");
            else
                return ((int)value).ToString(parameter.ToString());
        }
        return value;
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

This converter knows how to convert decimal and int values. For int values, you can optionally pass in a parameter.

The ConvertBack() method is used for two-way binding. We're not using that in this project, so you can leave the method stub as is.

Is it a good idea to leave this `NotImplementedException` in your code? For this project, this is code that is never supposed to be run. If it does get run, is it better to fail silently, so the user never sees it? Or is it better to throw an exception so that you can track down the problem? Which of those gives you a more robust app? There's not necessarily one right answer.

3 **Add the converter to your stopwatch control as a static resource.**
It should go right below the `ViewModel` object:

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    <viewmodel:TimeNumberFormatConverter x:Key="timeNumberFormatConverter"/>
</UserControl.Resources>
```

The designer may make you rebuild the solution after you add this line. In rare cases, you might even need to unload and reload the project.

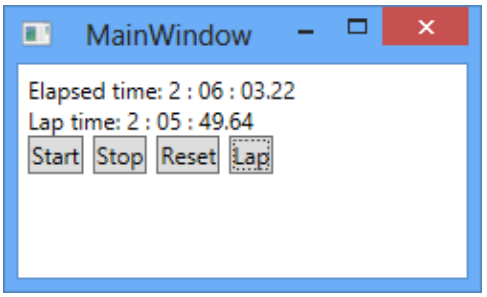
4 **Update the XAML code to use the value converter.**
Modify the `{Binding}` markup by adding the `Converter=` to it in each of the `<Run>` tags.

If there's no parameter specified, don't forget the extra closing bracket }}.

```
<TextBlock>
    <Run>Elapsed time: </Run>
    <Run Text="{Binding Hours, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding Minutes, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding Seconds, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
<TextBlock>
    <Run>Lap time: </Run>
    <Run Text="{Binding LapHours, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
    <Run>:</Run>
    <Run Text="{Binding LapMinutes, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}, ConverterParameter=d2}"/>
    <Run>:</Run>
    <Run Text="{Binding LapSeconds, Mode=OneWay,
        Converter={StaticResource timeNumberFormatConverter}}"/>
</TextBlock>
```

Use the ConverterParameter syntax to pass a parameter into the converter.

Now the stopwatch runs the values through the converter before passing them into the TextBlock controls, and the numbers are formatted correctly on the page.



Converters can work with many different types

TextBlock and TextBox controls work with text, so binding strings or numbers to the `Text` property makes sense. But there are many other properties, and you can bind to those as well. If your ViewModel has a Boolean property, it can be bound to any `true/false` property. You can even bind properties that use enums—the `IsVisible` property uses the `Visibility` enum, which means you can also write value converters for it. Let's add Boolean and `Visibility` binding and conversion to the stopwatch.

Here are two converters that will come in handy.

Sometimes you want to bind Boolean properties like `IsEnabled` so that a control is enabled if the bound property is `false`. We'll add a new converter called `BooleanNotConverter`, which uses the `!` operator to invert a Boolean target property.

```
IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
```

You'll often want to have controls show or hide themselves based on a Boolean property in the data context. You can only bind the Visibility property of a control to a target property that's of the type `Visibility` (meaning it returns values like `Visibility.Collapsed`). We'll add a converter called `BooleanVisibilityConverter` that will let us bind a control's `Visibility` property to a Boolean target property to make it visible or invisible.

```
Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"
```

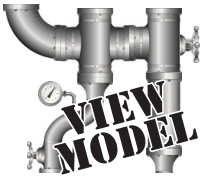
1 MODIFY THE VIEWMODEL'S TICK EVENT HANDLER.

Modify the `DispatcherTimer`'s `Tick` event handler to raise a `PropertyChanged` event if the value of the `Running` property has changed:

```
int _lastHours;
int _lastMinutes;
decimal _lastSeconds;
bool _lastRunning;
void TimerTick(object sender, object e) {
    if (_lastRunning != Running) {
        _lastRunning = Running;
        OnPropertyChanged("Running");
    }
    if (_lastHours != Hours) {
        _lastHours = Hours;
        OnPropertyChanged("Hours");
    }
    if (_lastMinutes != Minutes) {
        _lastMinutes = Minutes;
        OnPropertyChanged("Minutes");
    }
    if (_lastSeconds != Seconds) {
        _lastSeconds = Seconds;
        OnPropertyChanged("Seconds");
    }
}
```

We added the Running check to the timer. Would it make more sense to have the Model fire an event instead?

2 ADD A CONVERTER THAT INVERTS BOOLEAN VALUES.



Here's a value converter that converts `true` to `false` and vice versa. You can use it with Boolean properties on your controls like `IsEnabled`.

```
using System.Windows.Data;

class BooleanNotConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        if ((value is bool) && ((bool)value) == false)
            return true;
        else
            return false;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

3 ADD A CONVERTER THAT CONVERTS BOOLEANS TO VISIBILITY ENUMS.

You've already seen how you can make a control visible or invisible by setting its `Visibility` property to `Visible` or `Collapsed`. These values come from an enum in the *System.Windows* namespace called `Visibility`. Here's a converter that converts Boolean values to Visibility values:

```
using System.Windows;
using System.Windows.Data;

class BooleanVisibilityConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        if ((value is bool) && ((bool)value) == true)
            return Visibility.Visible;
        else
            return Visibility.Collapsed;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

4 MODIFY YOUR BASIC STOPWATCH CONTROL TO USE THE CONVERTERS.

Modify *BasicStopwatch.xaml* to add instances of these converters as static resources:

```
<viewmodel:BooleanVisibilityConverter x:Key="visibilityConverter"/>
<viewmodel:BooleanNotConverter x:Key="notConverter"/>
```

Now you can bind the controls' `IsEnabled` and `Visibility` properties to the ViewModel's `Running` property:

```
<StackPanel Orientation="Horizontal">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
            Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
    <Button IsEnabled="{Binding Running}" Click="StopButton_Click"
            Margin="0,0,5,0">Stop</Button>
    <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
<TextBlock Text="Stopwatch is running"
           Visibility="{Binding Running, Converter={StaticResource visibilityConverter}}"/>
```

This enables the Start button only if the stopwatch is not running.

This causes a TextBlock to become visible when the stopwatch is running.

We left this page blank so that you can read this appendix in two-page mode, so the exercise and its solution appear on different two-page spreads. If you're viewing this as a PDF in two-page mode, you may want to turn on the cover page so the even pages are on the right and the odd pages are on the left.

Build an analog stopwatch using the same ViewModel

Remember how you used the data classes you built for Jimmy's Comics in Chapter 14 and reused them to create a Split App without making any changes? This is the same idea.

The MVVM pattern **decouples** the View from the ViewModel, and the ViewModel from the Model. This is really useful if you need to make changes to one of the layers. Because of that decoupling, you can be very confident that the changes you make will not cause the "shotgun surgery" effect and ripple into the other layers. So did we do a good job decoupling the stopwatch program's View from its ViewModel? There's one way to be sure: let's build an entirely new View without changing the existing classes in the ViewModel. The only change you'll need in the C# code **is a new converter in the ViewModel** that converts minutes and seconds into angles.

Do this!

1 Add a converter to convert time to angles.

Add the `AngleConverter` class to the *ViewModel* folder. You'll use it for the hands on the face.

```
using System.Windows.Data;
class AngleConverter : IValueConverter {
    public object Convert(object value, Type targetType, object parameter,
                          System.Globalization.CultureInfo culture) {
        double parsedValue;
        if ((value != null)
            && double.TryParse(value.ToString(), out parsedValue)
            && (parameter != null))
            switch (parameter.ToString()) {
                case "Hours":
                    return parsedValue * 30;
                case "Minutes":
                case "Seconds":
                    return parsedValue * 6;
            }
        return 0;
    }
    public object ConvertBack(object value, Type targetType, object parameter,
                              System.Globalization.CultureInfo culture) {
        throw new NotImplementedException();
    }
}
```

An hour value ranges from 0 to 11, so to convert to an angle it's multiplied by 30.

Minutes and seconds range from 0 to 60, so the angle conversion means multiplying by 6.

2 Add the new UserControl.

Add a **new WPF user control called `AnalogStopwatch` to the *View* folder** and add the `ViewModel` namespace to the `<UserControl>` tag. Also, change the design width and height:

```
d:DesignHeight="300"
d:DesignWidth="400"
xmlns:viewmodel="clr-namespace:Stopwatch.ViewModel">
```

And add the ViewModel, two converters, and a style to the user control's static resources.

```
<UserControl.Resources>
    <viewmodel:StopwatchViewModel x:Key="viewModel"/>
    <viewmodel:BooleanNotConverter x:Key="notConverter"/>
    <viewmodel:AngleConverter x:Key="angleConverter"/>
</UserControl.Resources>
```

3 ADD THE FACE AND HANDS TO THE GRID.

Modify the <Grid> tag to add the stopwatch face, using four rectangles for hands.

```
<Grid x:Name="baseGrid" DataContext="{StaticResource ResourceKey=viewModel}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="400"/>
    </Grid.ColumnDefinitions>
    <Ellipse Width="300" Height="300" Stroke="Black" StrokeThickness="2">
        <Ellipse.Fill>
            <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                <GradientStop Color="#FFB03F3F"/>
                <GradientStop Color="#FFE4CECE" Offset="1"/>
            </LinearGradientBrush>
        </Ellipse.Fill>
    </Ellipse>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="150" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding Seconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="4" Height="100" Fill="Black">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding Minutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="1" Height="150" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-60"/>
                <RotateTransform Angle="{Binding LapSeconds,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Seconds}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle RenderTransformOrigin="0.5,0.5" Width="2" Height="100" Fill="Yellow">
        <Rectangle.RenderTransform>
            <TransformGroup>
                <TranslateTransform Y="-50"/>
                <RotateTransform Angle="{Binding LapMinutes,
                    Converter={StaticResource ResourceKey=angleConverter},
                    ConverterParameter=Minutes}"/>
            </TransformGroup>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Ellipse Width="10" Height="10" Fill="Black"/>
</Grid>
```

Setting the column width keeps it from expanding to fill whatever container it's in.

This is the face of the stopwatch. It has a black outline and a grayish gradient background.

Here's the second hand. It's a long, thin rectangle with a translate and rotate transform.

Here's the minute hand.

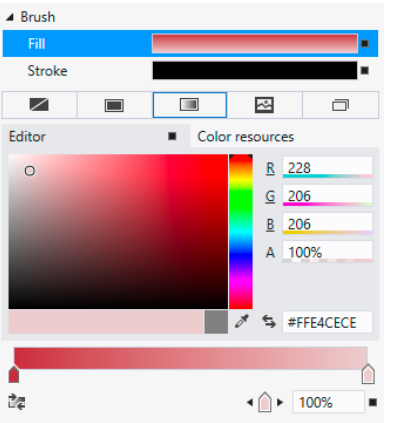
Every control can have one RenderTransform section.

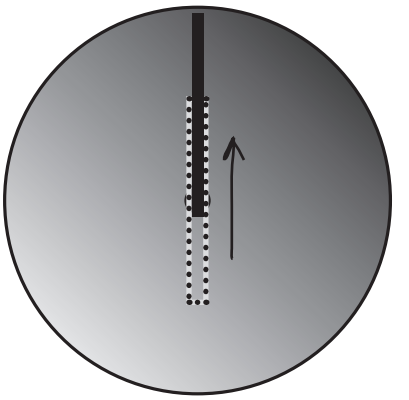
There are two yellow hands for the lap time.

The TransformGroup tag lets you apply multiple transforms to the same control.

This draws an extra circle in the middle to cover up where the hands overlap. Since it's at the bottom of the Grid, it's drawn last and ends up on top.

The stopwatch face is filled with a gradient brush, just like the background you used in *Save the Humans*.

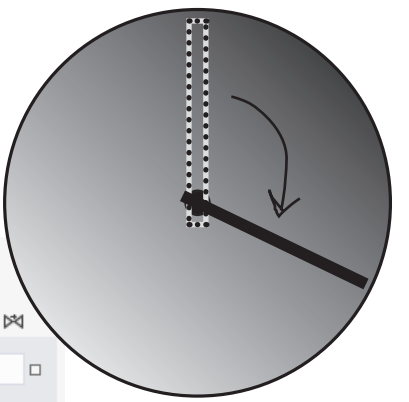




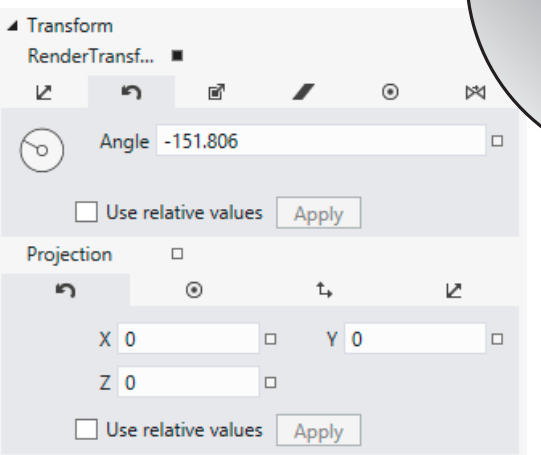
Each hand is transformed twice. It starts out centered in the face, so the first transform shifts it up so that it's in position to rotate.

```
<TranslateTransform Y="-60"/>

<RotateTransform Angle="{Binding Seconds,
    Converter={StaticResource ResourceKey=angleConverter},
    ConverterParameter=Seconds}"/>
```

The second transform rotates the hand to the correct angle. The `Angle` property of the rotation is bound to seconds or minutes in the ViewModel, and uses the angle converter to convert it to an angle.



Every control can have one `RenderTransform` element that changes how it's displayed. This can include rotating, moving to an offset, skewing, scaling its size up or down, and more.



You used transforms in *Save the Humans* to change the shape of the ellipses in the enemy to make it look like an alien.

Your stopwatch will start ticking as soon as you add the second hand, because it creates an instance of the ViewModel as a static resource to render the control in the designer. The designer may stop it updating, but you can restart it by switching away from the designer window and back again.

4 ADD THE BUTTONS TO THE STOPWATCH.

Since the ViewModel is the same, the buttons should work the same. Add the same buttons to *AnalogStopwatch.xaml* that you used for the basic stopwatch:

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Bottom">
    <Button IsEnabled="{Binding Running, Converter={StaticResource notConverter}}"
    Click="StartButton_Click" Margin="0,0,5,0">Start</Button>
    <Button IsEnabled="{Binding Running}"
                    Click="StopButton_Click" Margin="0,0,5,0">Stop</Button>
    <Button Click="ResetButton_Click" Margin="0,0,5,0">Reset</Button>
    <Button IsEnabled="{Binding Running}" Click="LapButton_Click">Lap</Button>
</StackPanel>
```

Here's the code-behind for *AnalogStopwatch.xaml.cs*:

```
ViewModel.StopwatchViewModel viewModel;

public AnalogStopwatch() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
}

private void StartButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Start();
}

private void StopButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Stop();
}

private void ResetButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Reset();
}

private void LapButton_Click(object sender, RoutedEventArgs e) {
    viewModel.Lap();
}
```

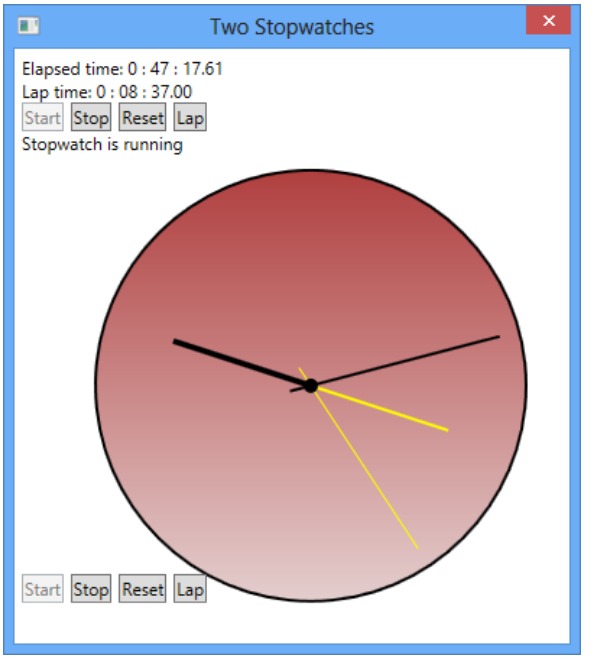
5

UPDATE THE MAIN WINDOW TO SHOW BOTH STOPWATCHES.

Now you just need to modify your *MainWindow.xaml* to add an `AnalogStopwatch` control:

```
<Window x:Class="Stopwatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Two Stopwatches" Height="450" Width="400" ResizeMode="NoResize"
        xmlns:view="clr-namespace:Stopwatch.View">
    <Grid>
        <StackPanel>
            <view:BasicStopwatch Margin="5"/>
            <view:AnalogStopwatch Margin="5"/>
        </StackPanel>
    </Grid>
</Window>
```

Run your app. Now you have two stopwatch controls on the page.



Each stopwatch keeps its own time, because each one has its own separate instance of the ViewModel as a static resource.

Try changing the ViewModel to make the _stopwatchModel field static. What does this change about how the stopwatch app behaves? Can you figure out why that happens?

UI controls can be instantiated with C# code, too

You already know that your XAML code instantiates classes in the `Windows.UI` namespace, and you even used the Watch window in the IDE back in Chapter 10 to explore them. But what if you want to create controls from inside your code? Well, controls are just objects, so you can create them and work with them just like you would with any other object. Go ahead and **modify the code-behind to add markings to the face of your analog stopwatch**.

```
public sealed partial class AnalogStopwatch : UserControl {
    public AnalogStopwatch() {
        InitializeComponent();

        viewModel = FindResource("viewModel") as ViewModel.StopwatchViewModel;
        AddMarkings();
    }

    private void AddMarkings() {
        for (int i = 0; i < 360; i += 3) {
            Rectangle rectangle = new Rectangle();
            rectangle.Width = (i % 30 == 0) ? 3 : 1;
            rectangle.Height = 15;
            rectangle.Fill = new SolidColorBrush(Colors.Black);
            rectangle.RenderTransformOrigin = new Point(0.5, 0.5);

            TransformGroup transforms = new TransformGroup();
            transforms.Children.Add(new TranslateTransform() { Y = -140 });
            transforms.Children.Add(new RotateTransform() { Angle = i });
            rectangle.RenderTransform = transforms;
            baseGrid.Children.Add(rectangle);
        }
    }
    // ... the button event handlers stay the same
```

Modify the constructor to call a method that adds the markings.

This statement uses the % modulo operator to make the marks for the hours thicker than the ones for the minutes. `i % 30` returns 0 only if `i` is divisible by 30.

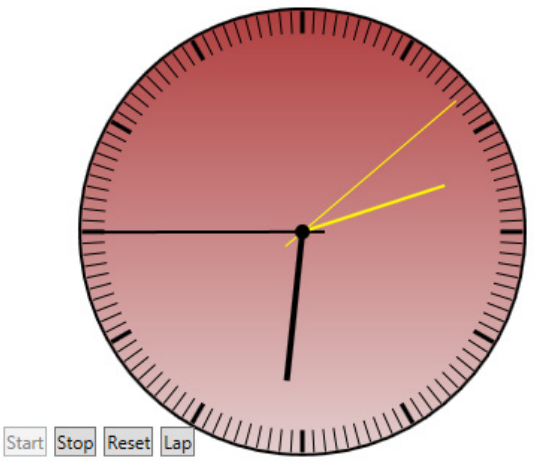
This creates instances of the same Rectangle object that you created with the <Rectangle> tag.

Flip back to the XAML for the hour and minute hands. This code sets up exactly the same transform, except instead of binding the Angle property it sets it to a value.

Controls like Grid, StackPanel, and Canvas have a `Children` collection with references to all the other controls contained inside them. You can add controls to the grid with its `Add()` method and remove all controls by calling its `Clear()` method. You add transforms to a `TransformGroup` the same way.

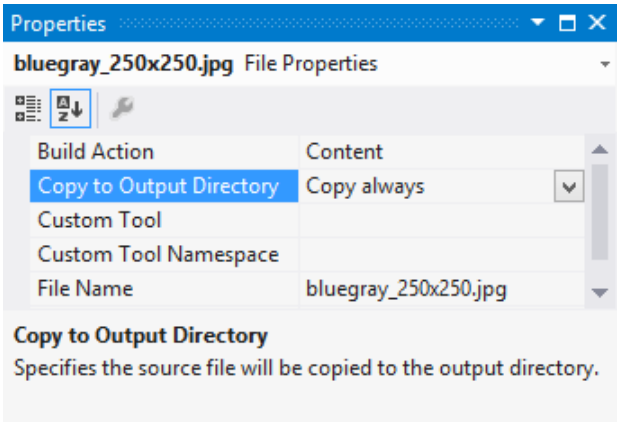
You used a `Binding` object to set up data binding in C# code back in Chapter 11. Can you figure out how to remove the XAML to create the Rectangle controls for the hour and minute hands and replace it with C# code to do the same thing?

Now that you added the markings to the stopwatch, the ref will make all the right calls.



Which team will dominate the conference and win the Objectville Trophy? Nobody's sure. All we know is that Joe, Bob, and Ed will be betting on it!

For the next few projects, you'll need to download the bee images from the Head First Labs website (http://www.headfirstlabs.com/hfcsharp). Make sure that you add the images to your project so they're in the top-level folder, just like you did with the Jimmy's Comics app. You'll also need to select each image file in the Solution Explorer and use the Properties window to set the "Build Action" to `Content` and "Copy to Output Directory" to `Copy always`. Here's what it looks like when you did it for the Jimmy's Comics app:



Make sure you do this for `Bee animation 1.png`, `Bee animation 2.png`, `Bee animation 3.png`, and `Bee animation 4.png`.

Create a user control to animate a picture

Let's encapsulate all the frame-by-frame animation code. **Add a WPF user control called *AnimatedImage* to your View folder**. It has very little XAML—all the intelligence is in the code-behind. Here's everything inside the `<UserControl>` tag in the XAML:

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

The work is done in the code-behind. Notice its overloaded constructor that calls the `StartAnimation()` method, which **creates storyboard and key frame animation objects** to animate the `Source` property of the Image control.

```
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;

public partial class AnimatedImage : UserControl {
    public AnimatedImage() {
        InitializeComponent();
    }

    public AnimatedImage(IEnumerable<string> imageNames, TimeSpan interval)
        : this() {
        StartAnimation(imageNames, interval);
    }

    public void StartAnimation(IEnumerable<string> imageNames, TimeSpan interval) {
        Storyboard storyboard = new Storyboard();
        ObjectAnimationUsingKeyFrames animation = new ObjectAnimationUsingKeyFrames();
        Storyboard.SetTarget(animation, image);
        Storyboard.SetTargetProperty(animation, new PropertyPath(Image.SourceProperty));

        TimeSpan currentInterval = TimeSpan.FromMilliseconds(0);
        foreach (string imageName in imageNames) {
            ObjectKeyFrame keyFrame = new DiscreteObjectKeyFrame();
            keyFrame.Value = CreateImageFromAssets(imageName);
            keyFrame.KeyTime = currentInterval;
            animation.KeyFrames.Add(keyFrame);
            currentInterval = currentInterval.Add(interval);
        }

        storyboard.RepeatBehavior = RepeatBehavior.Forever;
        storyboard.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
    private static BitmapImage CreateImageFromAssets(string imageFilename) {
        try {
            Uri uri = new Uri(imageFilename, UriKind.RelativeOrAbsolute);
            return new BitmapImage(uri);
        } catch (System.IO.IOException) {
            return new BitmapImage();
        }
    }
}
```

BitmapImage is in the Media.Imaging namespace. Storyboard and the other animation classes are in the Media.Animation namespace.

Every control must have a parameterless constructor if you want to create an instance of the control using XAML. You can still add overloaded constructors, but that's useful only if you're writing code to create the control.

The static `SetTarget()` and `SetTargetProperty()` methods from the `Storyboard` class set the target object being animated (`"image"`), and the property that will change (`Source`) using the PropertyPath() class.

Once the `Storyboard` object is set up and animations have been added to its `Children` collection, call its `Begin()` method to start the animation.

This is the same method you used in Chapter 14.

Make your bees fly around a page

Let's take your AnimatedImage control out for a test flight.

1 REPLACE THE MAIN WINDOW WITH A WINDOW IN THE VIEW FOLDER.

Add a **Window to your *View* folder** called *FlyingBees.xaml*. Delete *MainWindow.xaml* from the project. Then **modify the StartupUri property in the <Application> tag App.xaml**:

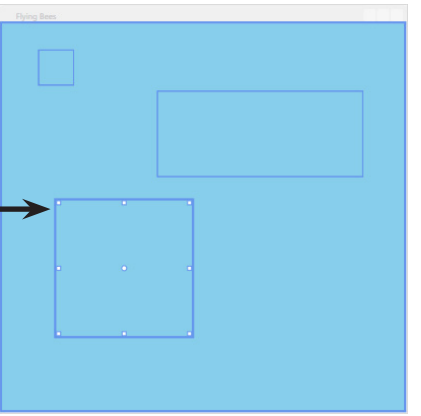
```
StartupUri="View\FlyingBees.xaml"
```

2 THE BEES WILL FLY AROUND A CANVAS CONTROL.

Here's the code for the window (you'll need to change the AnimatedBee namespace if you used a different project name). It uses a **Canvas control in *FlyingBees.xaml***. A Canvas control is a container, so it can contain other controls like a Grid or StackPanel. The difference is that a Canvas lets you set the coordinates of the controls using the `Canvas.Left` and `Canvas.Top` properties. You used a Canvas back in Chapter 1 to create the play area for *Save the Humans*. Here's the XAML for the *FlyingBees.xaml* window:

```
<Window x:Class="AnimatedBee.View.FlyingBees"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
               xmlns:view="clr-namespace:AnimatedBee.View"
        Title="Flying Bees" Height="600" Width="600">
    <Grid>
        <Canvas Background="SkyBlue">
            <view:AnimatedImage Canvas.Left="55" Canvas.Top="40"
                        x:Name="firstBee" Width="50" Height="50"/>
            <view:AnimatedImage Canvas.Left="80" Canvas.Top="260"
                        x:Name="secondBee" Width="200" Height="200"/>
            <view:AnimatedImage Canvas.Left="230" Canvas.Top="100"
                        x:Name="thirdBee" Width="300" Height="125"/>
        </Canvas>
    </Grid>
</Window>
```

The AnimatedImage control is invisible until its `CreateFrameImages()` method is called, so the controls in the Canvas will show up only as outlines. You can select them using the Document Outline. Try dragging the controls around the canvas to see the `Canvas.Left` and `Canvas.Top` properties change.



3 ADD THE CODE-BEHIND FOR THE PAGE.

You'll need this `using` statement for the namespace that contains `Storyboard` and `DoubleAnimation`:

```
using System.Windows.Media.Animation;
```

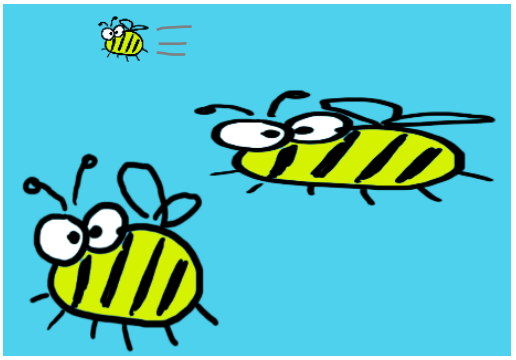
Now you can **modify the constructor in *FlyingBees.xaml.cs*** to start up the bee animation. Let's also create a `DoubleAnimation` to animate the `Canvas.Left` property. Compare the code for creating a storyboard and animation to the XAML code with `<DoubleAnimation>` earlier in the chapter.

```
public FlyingBees() {
    this.InitializeComponent();

    List<string> imageNames = new List<string>();
    imageNames.Add("Bee animation 1.png");
    imageNames.Add("Bee animation 2.png");
    imageNames.Add("Bee animation 3.png");
    imageNames.Add("Bee animation 4.png");

    firstBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(50));
    secondBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(10));
    thirdBee.StartAnimation(imageNames, TimeSpan.FromMilliseconds(100));

    Storyboard storyboard = new Storyboard();
    DoubleAnimation animation = new DoubleAnimation();
    Storyboard.SetTarget(animation, firstBee);
    Storyboard.SetTargetProperty(animation, new PropertyPath(Canvas.LeftProperty));
    animation.From = 50;
    animation.To = 450;
    animation.Duration = TimeSpan.FromSeconds(3);
    animation.RepeatBehavior = RepeatBehavior.Forever;
    animation.AutoReverse = true;
    storyboard.Children.Add(animation);
    storyboard.Begin();
}
```

The CreateFrameImages() method takes a sequence of asset names and a TimeSpan to set the rate that the frames are updated.

Instead of using a <Storyboard> tag and a <DoubleAnimation> tag like earlier in the chapter, you can create the Storyboard and DoubleAnimation objects and set their properties in code.

The `Storyboard` is garbage-collected after the animation completes. You can see this for yourself by using Make Object ID to watch it and clicking to refresh it after the animation ends.



Run your program. Now you can see three bees flapping their wings. You gave them different intervals, so they flap at different rates because their timers are waiting for different timespans before changing frames. The top bee has its Canvas.Left property animated from 50 to 450 and back, which causes it to move around the page. Take a close look at the properties that are set on the DoubleAnimation object and compare them with the XAML properties you used earlier in the chapter.

Something's not right about this project. Can you spot it?

Something's not right: there's nothing in your *Model* or *ViewModel* folder, and you're creating dummy data in the View. That's not MVVM!

If we wanted to add more bees, we'd have to create more controls in the View and then initialize them individually. What if we want different sizes or kinds of bees? Or other things to be animated? If we had a Model that was optimized for data, it would be a lot easier. How can we make this project follow the MVVM pattern?

That won't work. Data binding doesn't work with container controls' `Children` property—and for good reason.

Data binding is built to work with **attached properties**, which are the properties that show up in the XAML code. The `Canvas` object *does* have a public `Children` property, but if you try to set it using XAML (`Children="{Binding ...}"`) your code ***won't compile***.

However, you already know how to bind a collection of objects to a XAML control, because you did that with ListView and GridView controls using the `ItemsSource` property. We can take advantage of that data binding to add child controls to a Canvas.

Use ItemsPanelTemplate to bind controls to a Canvas

When you used the `ItemsSource` property to bind items to a ListView, GridView, or ListBox, it didn't matter which one you were binding to, because the `ItemsSource` property always worked the same way. If you were going to build three classes that had exactly the same behavior, you would put that behavior in a base class and have the three classes extend it, right? Well, the Microsoft team did exactly the same thing when they built the selector controls. The ListView, GridView, and ListBox all extend a class called `Selector`, which is a subclass of **the `ItemsControl` class that displays a collection of items**.

1. We're going to use its `ItemsPanel` property to **set up a template for the panel that controls the layout of the items**. Start by adding the `ViewModel` namespace to *FlyingBees.xaml*:

```
xmlns:viewmodel="clr-namespace:AnimatedBee.ViewModel"
```

If you used a different project name, change AnimatedBee to the correct namespace.

2. Next, **add an *empty* class called `BeeViewModel` to your *ViewModel* folder**, and then add an instance of that class as a static resource to *FlyingBees.xaml*:

```
<viewmodel:BeeViewModel x:Key="viewModel"/>
```

Edit *FlyingBees.xaml.cs* and **delete all the additional code that you added to the `FlyingBees()` constructor** in the `FlyingBees` control. Make sure that you ***don't delete*** the `InitializeComponent()` method!

3. Here's the XAML for the ItemsControl. Open *FlyingBees.xaml*, **delete the `<Canvas>` tag** you added, and **replace it with this ItemsControl**:

Use the static ViewModel resource as the data context, and bind the ItemsSource to a property called Sprites.

```
<ItemsControl
        DataContext="{StaticResource viewModel}"
        ItemsSource="{Binding Path=Sprites}" >
    <ItemsControl.ItemsPanel>
        <ItemsPanelTemplate>
            <Canvas Background="SkyBlue" />
        </ItemsPanelTemplate>
    </ItemsControl.ItemsPanel>
</ItemsControl>
```

You can set up the panel however you want. We'll use a Canvas with a sky-blue background.

Use the `ItemsPanel` property to set up an `ItemsPanelTemplate`. This contains a single Panel control, and both Grid and Canvas extend the `Panel` class. Any items bound to `ItemsSource` will be added to the Panel's `Children`.

When the ItemsControl is created, it creates a Panel to hold all of its items and uses the `ItemsPanelTemplate` as the control template.

The factory method pattern

MVVM is just one of many design patterns. One of the most common—and most useful—patterns is the factory method pattern, where you have a "factory" method that creates objects. The factory method is usually static, and the name often ends with "Factory" so it's obvious what's going on.

4 Create a **new class in the *View* folder called `BeeHelper`**. Make sure it's a static class, because it'll have only static methods to help your ViewModel manage its bees.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;

static class BeeHelper {
    public static AnimatedImage BeeFactory(
                double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetBeeLocation(AnimatedImage bee, double x, double y) {
        Canvas.SetLeft(bee, x);
        Canvas.SetTop(bee, y);
    }

    public static void MakeBeeMove(AnimatedImage bee,
                                   double fromX, double toX, double y) {
        Canvas.SetTop(bee, y);
        Storyboard storyboard = new Storyboard();
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, bee);
        Storyboard.SetTargetProperty(animation,
                                     new PropertyPath(Canvas.LeftProperty));
        animation.From = fromX;
        animation.To = toX;
        animation.Duration = TimeSpan.FromSeconds(3);
        animation.RepeatBehavior = RepeatBehavior.Forever;
        animation.AutoReverse = true;
        storyboard.Children.Add(animation);
        storyboard.Begin();
    }
}
```

This factory method creates bee controls. It makes sense to keep this in the View, because it's all UI-related code.

When you take a small block of code that's reused a lot and put it in its own (often static) method, it's sometimes called a helper method. Putting helper methods in a static class with a name that ends with "Helper" makes your code easier to read.

This is the same code that was in the page's constructor. Now it's in a static helper method.

This will come in handy in the last lab.

All XAML controls inherit from the `UIElement` base class in the `System.Windows` namespace. We explicitly used the namespace (`System.Windows.UIElement`) in the body of the class instead of adding a `using` statement to limit the amount of UI-related code we added to the ViewModel.

We used `UIElement` because it's the most abstract class that all the sprites extend. For some projects, a subclass like `FrameworkElement` may be more appropriate, because that's where many properties are defined, including `Width`, `Height`, `Opacity`, `HorizontalAlignment`, etc.

5 Here's the code for the empty `BeeViewModel` class that you added to the *ViewModel* folder. By moving the UI-specific code to the View, we can keep the code in the ViewModel simple and specific to managing bee-related logic.

When the AnimatedImage control is added to the _sprites ObservableCollection that's bound to the ItemsControl's ItemsSource property, the control is added to the item panel, which is created based on the ItemsPanelTemplate.

```
using View;
using System.Collections.ObjectModel;
using System.Collections.Specialized;

class BeeViewModel {
    private readonly ObservableCollection<System.Windows.UIElement>
                _sprites = new ObservableCollection<System.Windows.UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    public BeeViewModel() {
        AnimatedImage firstBee =
            BeeHelper.BeeFactory(50, 50,
                                 TimeSpan.FromMilliseconds(50));
        _sprites.Add(firstBee);

        AnimatedImage secondBee =
            BeeHelper.BeeFactory(200, 200, TimeSpan.FromMilliseconds(10));
        _sprites.Add(secondBee);

        AnimatedImage thirdBee =
            BeeHelper.BeeFactory(300, 125, TimeSpan.FromMilliseconds(100));
        _sprites.Add(thirdBee);

        BeeHelper.MakeBeeMove(firstBee, 50, 450, 40);
        BeeHelper.SetBeeLocation(secondBee, 80, 260);
        BeeHelper.SetBeeLocation(thirdBee, 230, 100);
    }
}
```

We're taking two steps to encapsulate the Sprites property. The backing field is marked readonly so it can't be overwritten later, and we expose it as an INotifyCollectionChanged property so other classes can only observe it but not modify it.

A sprite is the term for any 2D image or animation that gets incorporated into a larger game or animation.

You're changing properties and adding animations on the controls after they were added to the `ObservableCollection`. Why does that work?

6 Run your app. It should look exactly the same as before, but now the behavior is split across the layers, with UI-specific code in the View and code that deals with bees and moving in the ViewModel.

The readonly keyword

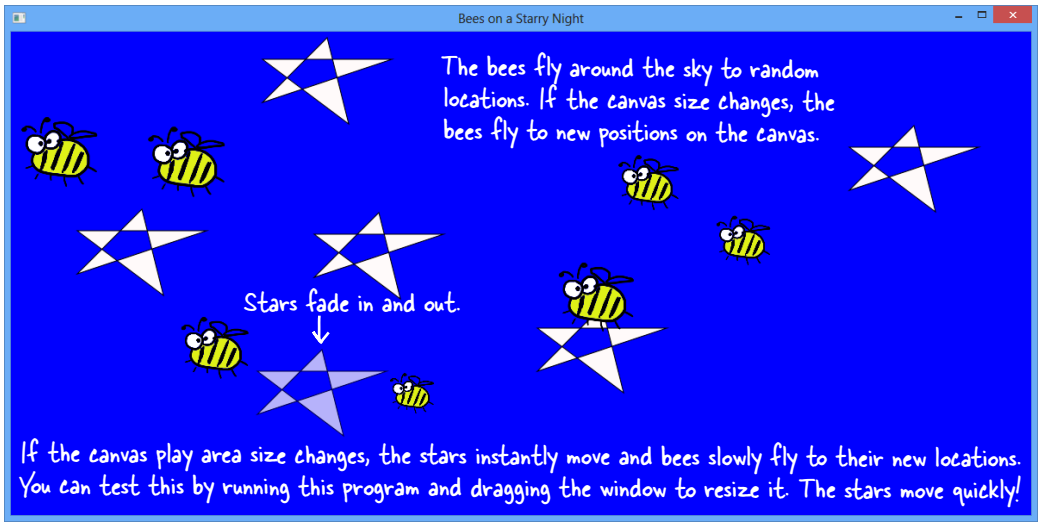
An important reason that we use encapsulation is to prevent one class from accidentally overwriting another class's data. But what's preventing a class from overwriting its own data? The readonly keyword can help with that. Any field that you mark readonly can be modified only in its declaration or in the constructor.

Long Exercise

This is the last exercise in the book. Your job is to build a program that animates bees and stars. There's a lot of code to write, but you're up to the task...and once you have this working, you'll have all the tools you need to build a complete video game. *(Can you guess what's in Lab #3?)*

1 Here's the app you'll create.

Bees with flapping wings fly around a dark blue canvas, while behind them, stars fade in and out. You'll build a View that contains the bees, stars, and page to display them; a Model that keeps track of where they are and fires off events when bees move or stars change; and a ViewModel to connect the two together.



2 Create a new WPF application project.

Create a new project called StarryNight. Next, **add the *Model*, *View*, and *ViewModel* folders**. Once that's done, you'll need to add **an empty class called `BeeStarViewModel`** to the *ViewModel* folder.

3 Create a new window in the View folder.

Delete *MainWindow.xaml*. Then **add a window in the *View* folder** called *BeesOnAStarryNight.xaml*. Add the namespace to the top-level tag in the *BeesOnAStarryNight.xaml* (it should match your project's name, `StarryNight`):

```
xmlns:viewmodel="clr-namespace:StarryNight.ViewModel"
```

Add the ViewModel as a static resource and change the page name:

```
<Window.Resources>
    <viewmodel:BeeStarViewModel x:Key="viewModel"/>
</Window.Resources>
```

The XAML for the page is **exactly the same** as *FlyingBees.xaml* in the last project, except the Canvas control's background is **`Blue`** and it has a `SizeChanged` event handler:

```
<Canvas Background="Blue" SizeChanged="SizeChangedHandler" />
```

The `SizeChanged` event is fired when a control changes size, with `EventArgs` properties for the new size.

Then modify the `<Application>` tag in *App.xaml* so the application starts with the new window:

```
StartupUri="View\BeesOnAStarryNight.xaml"
```

Visual Studio comes with a fantastic tool to help you experiment with shapes! Fire up Blend for Visual Studio 2013 and use the pen, pencil, and toolbox to create XAML shapes that you can copy and paste into your C# projects.

The code in step 4 won't compile until you add the PlayAreaSize property to the ViewModel in step 9. You can use the IDE to generate a property stub for it for now.

4 ADD CODE-BEHIND FOR THE PAGE AND THE APP.

Add the `SizeChanged` event handler to *BeesOnAStarryNight.xaml.cs* in the *View* folder:


```
ViewModel.BeeStarViewModel viewModel;

public BeesOnAStarryNight() {
    InitializeComponent();

    viewModel = FindResource("viewModel") as ViewModel.BeeStarViewModel;
}

private void SizeChangedHandler(object sender, SizeChangedEventArgs e) {
    viewModel.PlayAreaSize = new Size(e.NewSize.Width, e.NewSize.Height);
}
```

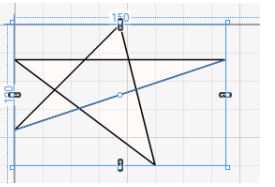
5 ADD THE ANIMATEDIMAGE CONTROL TO THE VIEW FOLDER.

Go back to the *View* folder and add the AnimatedImage control. This is exactly the same control from earlier in the chapter. Make sure you **add the image files** for the animation frames to the project and update each file's Build Action to `Content` and its Copy to Output Directory to `Copy always`.

6 ADD A USER CONTROL CALLED STARCONTROL TO THE VIEW FOLDER.

This control draws a star. It also has two storyboards, one to fade in and one to fade out. **Add methods called `FadeIn()` and `FadeOut()`** to the code-behind to trigger the storyboards.

A Polygon control uses a set of points to draw a polygon. This UserControl uses it to draw a star.



```
<UserControl
    // The usual XAML code that the IDE generates is fine,
    // no extra namespaces are needed for this User Control.
    >

    <UserControl.Resources>
        <Storyboard x:Key="fadeInStoryboard">
            <DoubleAnimation From="0" To="1" Storyboard.TargetName="starPolygon"
                    Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
        <Storyboard x:Key="fadeOutStoryboard">
           <DoubleAnimation From="1" To="0" Storyboard.TargetName="starPolygon"
                    Storyboard.TargetProperty="Opacity" Duration="0:0:1.5" />
        </Storyboard>
    </UserControl.Resources>

    <Grid>
        <Polygon Points="0,75 75,0 100,100 0,25 150,25" Fill="Snow"
                 Stroke="Black" x:Name="starPolygon"/>
    </Grid>
</UserControl>
```

You'll need to add public FadeIn() and FadeOut() methods to the code-behind that starts these storyboards. That's how the stars will fade in and out.

This polygon draws the star. You can replace it with other shapes to experiment with how they work.

There are even more shapes beyond ellipses, rectangles, and polygons: http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh465055.aspx

Long Exercise (continued)

7 **ADD THE BeeStarHelper CLASS TO THE VIEW.**

Here's a useful helper class. It's got some familiar tools and a couple of new ones. Put it in the *View* folder.

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

static class BeeStarHelper {
    public static AnimatedImage BeeFactory(double width, double height, TimeSpan flapInterval) {
        List<string> imageNames = new List<string>();
        imageNames.Add("Bee animation 1.png");
        imageNames.Add("Bee animation 2.png");
        imageNames.Add("Bee animation 3.png");
        imageNames.Add("Bee animation 4.png");

        AnimatedImage bee = new AnimatedImage(imageNames, flapInterval);
        bee.Width = width;
        bee.Height = height;
        return bee;
    }

    public static void SetCanvasLocation(UIElement control, double x, double y) {
        Canvas.SetLeft(control, x);
        Canvas.SetTop(control, y);
    }

    public static void MoveElementOnCanvas(UIElement uiElement, double toX, double toY) {
        double fromX = Canvas.GetLeft(uiElement);
        double fromY = Canvas.GetTop(uiElement);

        Storyboard storyboard = new Storyboard();
        DoubleAnimation animationX = CreateDoubleAnimation(uiElement,
                        fromX, toX, new PropertyPath(Canvas.LeftProperty));
        DoubleAnimation animationY = CreateDoubleAnimation(uiElement,
                        fromY, toY, new PropertyPath(Canvas.TopProperty));
        storyboard.Children.Add(animationX);
        storyboard.Children.Add(animationY);
        storyboard.Begin();
    }

    public static DoubleAnimation CreateDoubleAnimation(UIElement uiElement,
                                        double from, double to, PropertyPath propertyToAnimate) {
        DoubleAnimation animation = new DoubleAnimation();
        Storyboard.SetTarget(animation, uiElement);
        Storyboard.SetTargetProperty(animation, propertyToAnimate);
        animation.From = from;
        animation.To = to;
        animation.Duration = TimeSpan.FromSeconds(3);
        return animation;
    }

    public static void SendToBack(StarControl newStar)
    {
        Canvas.SetZIndex(newStar, -1000);
    }
}
```

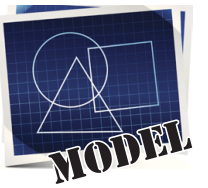
Canvas has `SetLeft()` and `GetLeft()` methods to set and get the X position of a control. The `SetTop()` and `GetTop()` methods set and get the Y position. They work even after a control is added to the Canvas.

We added a helper called CreateDoubleAnimation() that creates a three-second DoubleAnimation. This method uses it to move a UIElement from its current location to a new point by animating its Canvas.Left and Canvas.Top properties.

"Z Index" means the order the controls are layered on a panel. A control with a higher Z index is drawn on top of one with a lower Z index.

8 ADD THE BEE, STAR, AND EVENTARGS CLASSES TO THE MODEL.

Your model needs to keep track of the bees' positions and sizes, and the stars' positions, and it will fire off events so the ViewModel knows whenever there's a change to a bee or a star.




```
using System.Windows;
class Bee {
    public Point Location { get; set; }
    public Size Size { get; set; }
    public Rect Position { get { return new Rect(Location, Size); } }
    public double Width { get { return Position.Width; } }
    public double Height { get { return Position.Height; } }

    public Bee(Point location, Size size) {
        Location = location;
        Size = size;
    }
}
```

```
using System.Windows;
class Star {
    public Point Location {
        get; set;
    }

    public Star(Point location) {
        Location = location;
    }
}
```

Once you get your program working, try adding a Boolean Rotating property to the Star class and use it to make some of your stars slowly spin around.

```
using System.Windows;
class BeeMovedEventArgs : EventArgs {
    public Bee BeeThatMoved { get; private set; }
    public double X { get; private set; }
    public double Y { get; private set; }

    public BeeMovedEventArgs(Bee beeThatMoved, double x, double y) {
        BeeThatMoved = beeThatMoved;
        X = x;
        Y = y;
    }
}
```

The model will fire events that use these EventArgs to tell the ViewModel when changes happen.

```
using System.Windows;
class StarChangedEventArgs : EventArgs {
    public Star StarThatChanged { get; private set; }
    public bool Removed { get; private set; }

    public StarChangedEventArgs(Star starThatChanged, bool removed) {
        StarThatChanged = starThatChanged;
        Removed = removed;
    }
}
```

The Points property on the Polygon control is a collection of Point structs.

The Rect struct has several overloaded constructors, and methods that let you extract its width, height, size, and location (either as a Point or individual X and Y double coordinates).

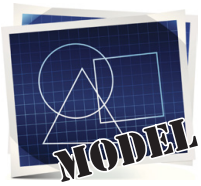
The Point, Size, and Rect structs

The System.Windows namespace has several very useful structs. Point uses X and Y double properties to store a set of coordinates. Size has two double properties too, Width and Height, and also a special Empty value. Rect stores two coordinates for the top-left and bottom-right corner of a rectangle. It has a lot of useful methods to find its width, height, intersection with other Rects, and more.

Long Exercise (continued)

9 **ADD THE BEESTARMODEL CLASS TO THE MODEL.**

We've filled in the private fields and a couple of useful methods. Your job is to finish building the `BeeStarModel` class.




```
using System.Windows;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    public Size PlayAreaSize {
        // Add a backing field, and have the set accessor call CreateBees() and CreateStars()
    }
    private void CreateBees() {
        // If the play area is empty, return. If there are already bees, move each of them.
        // Otherwise, create between 5 and 15 randomly sized bees (40 to 150 pixels), add
        // it to the _bees collection, and fire the BeeMoved event.
    }
    private void CreateStars() {
        // If the play area is empty, return. If there are already stars,
        // set each star's location to a new point and fire the StarChanged
        // event, otherwise call CreateAStar() between 5 and 10 times.
    }
    private void CreateAStar() {
        // Find a new non-overlapping point, add a new Star object to the
        // _stars collection, and fire the StarChanged event.
    }
    private Point FindNonOverlappingPoint(Size size) {
        // Find the upper-left corner of a rectangle that doesn't overlap any bees or stars.
        // You'll need to try random Rects, then use LINQ queries to find any bees or stars
        // that overlap (the RectsOverlap() method will be useful).
    }
    private void MoveOneBee(Bee bee = null) {
        // If there are no bees, return. If the bee parameter is null, choose a random bee,
        // otherwise use the bee argument. Then find a new non-overlapping point, update the bee's
        // location, update the _bees collection, and then fire the OnBeeMoved event.
    }
    private void AddOrRemoveAStar() {
        // Flip a coin (_random.Next(2) == 0) and either create a star using CreateAStar() or
        // remove a star and fire OnStarChanged. Always create a star if there are <= 5, remove
        // one if >= 20. _stars.Keys.ToList()[_random.Next(_stars.Count)] will find a random star.
    }
    // You'll need to add the BeeMoved and StarChanged events and methods to call them.
    // They use the BeeMovedEventArgs and StarChangedEventArgs classes.
}
```

You can use readonly to create a constant struct value.

Size.Empty is a value of Size that's reserved for an empty size. You'll use it only to create bees and stars when the play area is resized.

The ViewModel will use a timer to call this Update() method periodically.

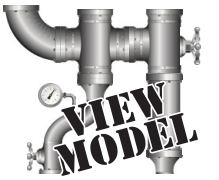
This method checks two Rect structs and returns true if they overlap each other using the Rect.Intersect() method.

PlayAreaSize is a property.

If the method's tried 1,000 random locations and hasn't found one that doesn't overlap, the play area has probably run out of space, so just return any point.

You can debug your app with the simulator to make sure it works with different screen sizes and orientations.

10 ADD THE BEESTARVIEWMODEL CLASS TO THE VIEWMODEL.



Fill in the commented methods. You'll need to look closely at how the Model works and what the View expects. The helper methods will also come in very handy.

We wanted to make sure that `DispatcherTimer` and `UIElement` are the only classes from the `Windows.UI.Xaml` namespace that we used in the ViewModel. The `using` keyword lets you use = to declare a single member in another namespace.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Windows;
using DispatcherTimer = Windows.UI.Xaml.DispatcherTimer;
using UIElement = Windows.UI.Xaml.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();

    public Size PlayAreaSize { /* get and set accessors return and set _model.PlayAreaSize */ }

    public BeeStarViewModel() {
        // Hook up the event handlers to the BeeStarModel's BeeMoved and StarChanged events,
        // and start the timer ticking every two seconds.
    }
    void timer_Tick(object sender, object e) {
        // Every time the timer ticks, find all StarControl references in the _fadedStars
        // collection and remove each of them from _sprites, then call the BeeViewModel's
        // Update() method to tell it to update itself.
    }
    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        // The _bees dictionary maps Bee objects in the Model to AnimatedImage controls
        // in the view. When a bee is moved, the BeeViewModel fires its BeeMoved event to
        // tell anyone listening which bee moved and its new location. If the _bees
        // dictionary doesn't already contain an AnimatedImage control for the bee, it needs
        // to create a new one, set its canvas location, and update both _bees and _sprites.
        // If the _bees dictionary already has it, then we just need to look up the corresponding
        // AnimatedImage control and move it on the canvas to its new location with an animation.
    }
    void StarChangedHandler(object sender, StarChangedEventArgs e) {
       // The _stars dictionary works just like the _bees one, except that it maps Star objects
       // to their corresponding StarControl controls. The EventArgs contains references to
       // the Star object (which has a Location property) and a Boolean to tell you if the star
       // was removed. If it is then we want it to fade out, so remove it from _stars, add it
       // to _fadedStars, and call its FadeOut() method (it'll be removed from _sprites the next
       // time the Update() method is called, which is why we set the timer's tick interval to
       // be greater than the StarControl's fade out animation).
       //
       // If the star is not being removed, then check to see if _stars contains it - if so, get
       // the StarControl reference; if not, you'll need to create a new StarControl, fade it in,
       // add it to _sprites, and send it to back so the bees can fly in front of it. Then set
       // the canvas location for the StarControl.
    }
}
```

When you set the new Canvas location, the control is updated—even if it's already on the Canvas. This is how the stars move themselves around when the play area is resized.

Long Exercise SOLUTION

Here are the filled-in methods in the `BeeStarModel` class.

```
using System.Windows;

class BeeStarModel {
    public static readonly Size StarSize = new Size(150, 100);

    private readonly Dictionary<Bee, Point> _bees = new Dictionary<Bee, Point>();
    private readonly Dictionary<Star, Point> _stars = new Dictionary<Star, Point>();
    private Random _random = new Random();

    public BeeStarModel() {
        _playAreaSize = Size.Empty;
    }

    public void Update() {
        MoveOneBee();
        AddOrRemoveAStar();
    }

    private static bool RectsOverlap(Rect r1, Rect r2) {
        r1.Intersect(r2);
        if (r1.Width > 0 || r1.Height > 0)
            return true;
        return false;
    }

    private Size _playAreaSize;
    public Size PlayAreaSize {
        get { return _playAreaSize; }
        set
        {
            _playAreaSize = value;
            CreateBees();
            CreateStars();
        }
    }

    private void CreateBees() {
        if (PlayAreaSize == Size.Empty) return;

        if (_bees.Count() > 0) {
            List<Bee> allBees = _bees.Keys.ToList();
            foreach (Bee bee in allBees)
                MoveOneBee(bee);
        } else {
            int beeCount = _random.Next(5, 10);
            for (int i = 0; i < beeCount; i++) {
                int s = _random.Next(50, 100);
                Size beeSize = new Size(s, s);
                Point newLocation = FindNonOverlappingPoint(beeSize);
                Bee newBee = new Bee(newLocation, beeSize);
                _bees[newBee] = new Point(newLocation.X, newLocation.Y);
                OnBeeMoved(newBee, newLocation.X, newLocation.Y);
            }
        }
    }
```

We gave these to you.

Whenever the PlayAreaSize property changes, the Model updates the _playAreaSize backing field and then calls CreateBees() and CreateStars(). This lets the ViewModel tell the Model to adjust itself whenever the size changes—which will happen if you run the program on a tablet and change the orientation.

If there are already bees, move each of them. MoveOneBee() will find a new nonoverlapping location for each bee and fire a BeeMoved event.

If there aren't any bees in the model yet, this creates new Bee objects and sets their locations. Any time a bee is added or changes, we need to fire a BeeMoved event.

```
private void CreateStars() {
    if (PlayAreaSize == Size.Empty) return;

    if (_stars.Count > 0) {
        foreach (Star star in _stars.Keys) {
            star.Location = FindNonOverlappingPoint(StarSize);
            OnStarChanged(star, false);
        }
    } else {
        int starCount = _random.Next(5, 10);
        for (int i = 0; i < starCount; i++)
            CreateAStar();
    }
}

private void CreateAStar() {
    Point newLocation = FindNonOverlappingPoint(StarSize);
    Star newStar = new Star(newLocation);
    _stars[newStar] = new Point(newLocation.X, newLocation.Y);
    OnStarChanged(newStar, false);
}

private Point FindNonOverlappingPoint(Size size) {
    Rect newRect = new Rect();
    bool noOverlap = false;
    int count = 0;
    while (!noOverlap) {
        newRect = new Rect(_random.Next((int)PlayAreaSize.Width - 150),
            _random.Next((int)PlayAreaSize.Height - 150),
            size.Width, size.Height);

        var overlappingBees =
            from bee in _bees.Keys
            where RectsOverlap(bee.Position, newRect)
            select bee;

        var overlappingStars =
            from star in _stars.Keys
            where RectsOverlap(
                new Rect(star.Location.X, star.Location.Y, StarSize.Width, StarSize.Height),
                newRect)
            select star;

        if ((overlappingBees.Count() + overlappingStars.Count() == 0) || (count++ > 1000))
            noOverlap = true;
    }
    return new Point(newRect.X, newRect.Y);
}

private void MoveOneBee(Bee bee = null) {
    if (_bees.Keys.Count() == 0) return;
    if (bee == null) {
        int beeCount = _stars.Count;
        List<Bee> bees = _bees.Keys.ToList();
        bee = bees[_random.Next(bees.Count)];
    }
    bee.Location = FindNonOverlappingPoint(bee.Size);
    _bees[bee] = bee.Location;
    OnBeeMoved(bee, bee.Location.X, bee.Location.Y);
}
```

If there are already stars, we just set each existing star's location to a new point on the PlayArea and fire the StarChanged event. It's up to the ViewModel to handle that event and move the corresponding control.

This creates a random Rect and then checks if it overlaps. We gave it a 250-pixel gap on the right and a 150-pixel gap on the bottom so the stars and bees don't leave the play area.

These LINQ queries call RectsOverlap() to find any bees or stars that overlap the new Rect. If either return value has a count, the new Rect overlaps something.

If this iterated 1,000 times, it means we're probably out of nonoverlapping spots in the play area and need to break out of an infinite loop.

Long Exercise SOLUTION

The last few members of the `BeeStarModel` class.

Flip a coin by choosing either 0 or 1 at random, but always create a star if there are under 5 and remove if 20 or more.

```
    private void AddOrRemoveAStar() {
        if (((_random.Next(2) == 0) || (_stars.Count <= 5)) && (_stars.Count < 20 ))
            CreateAStar();
        else {
            Star starToRemove = _stars.Keys.ToList()[_random.Next(_stars.Count)];
            _stars.Remove(starToRemove);
            OnStarChanged(starToRemove, true);
        }
    }

    public event EventHandler<BeeMovedEventArgs> BeeMoved;

    private void OnBeeMoved(Bee beeThatMoved, double x, double y)
    {
        EventHandler<BeeMovedEventArgs> beeMoved = BeeMoved;
        if (beeMoved != null)
        {
            beeMoved(this, new BeeMovedEventArgs(beeThatMoved, x, y));
        }
    }

    public event EventHandler<StarChangedEventArgs> StarChanged;

    private void OnStarChanged(Star starThatChanged, bool removed)
    {
        EventHandler<StarChangedEventArgs> starChanged = StarChanged;
        if (starChanged != null)
        {
            starChanged(this, new StarChangedEventArgs(starThatChanged, removed));
        }
    }
}
```

Every time the Update() method is called, we want to either add or remove a star. The CreateAStar() method already creates stars. If we're removing a star, we just remove it from _stars and fire a StarChanged event.

These are typical event handlers and methods to fire them.

Here are the filled-in methods of the `BeeStarViewModel` class.

```
using View;
using Model;
using System.Collections.ObjectModel;
using System.Collections.Specialized;
using System.Windows;
using DispatcherTimer = System.Windows.Threading.DispatcherTimer;
using UIElement = System.Windows.UIElement;

class BeeStarViewModel {
    private readonly ObservableCollection<UIElement>
                _sprites = new ObservableCollection<UIElement>();
    public INotifyCollectionChanged Sprites { get { return _sprites; } }

    private readonly Dictionary<Star, StarControl> _stars = new Dictionary<Star, StarControl>();
    private readonly List<StarControl> _fadedStars = new List<StarControl>();

    private BeeStarModel _model = new BeeStarModel();

    private readonly Dictionary<Bee, AnimatedImage> _bees
                                        = new Dictionary<Bee, AnimatedImage>();

    private DispatcherTimer _timer = new DispatcherTimer();
```

We gave these to you.

```
    public Size PlayAreaSize {
        get { return _model.PlayAreaSize; }
        set { _model.PlayAreaSize = value; }
    }

    public BeeStarViewModel() {
        _model.BeeMoved += BeeMovedHandler;
        _model.StarChanged += StarChangedHandler;

        _timer.Interval = TimeSpan.FromSeconds(2);
        _timer.Tick += timer_Tick;
        _timer.Start();
    }

    void timer_Tick(object sender, object e) {
        foreach (StarControl starControl in _fadedStars)
            _sprites.Remove(starControl);

        _model.Update();
    }

    void BeeMovedHandler(object sender, BeeMovedEventArgs e) {
        if (!_bees.ContainsKey(e.BeeThatMoved)) {
            AnimatedImage beeControl = BeeStarHelper.BeeFactory(
                        e.BeeThatMoved.Width, e.BeeThatMoved.Height, TimeSpan.FromMilliseconds(20));
            BeeStarHelper.SetCanvasLocation(beeControl, e.X, e.Y);
            _bees[e.BeeThatMoved] = beeControl;
            _sprites.Add(beeControl);
        } else {
            AnimatedImage beeControl = _bees[e.BeeThatMoved];
            BeeStarHelper.MoveElementOnCanvas(beeControl, e.X, e.Y);
        }
    }

    void StarChangedHandler(object sender, StarChangedEventArgs e) {
        if (e.Removed) {
            StarControl starControl = _stars[e.StarThatChanged];
            _stars.Remove(e.StarThatChanged);
            _fadedStars.Add(starControl);
            starControl.FadeOut();
        } else {
            StarControl newStar;
            if (_stars.ContainsKey(e.StarThatChanged))
                newStar = _stars[e.StarThatChanged];
            else {
                newStar = new StarControl();
                _stars[e.StarThatChanged] = newStar;
                newStar.FadeIn();
                BeeStarHelper.SendToBack(newStar);
                _sprites.Add(newStar);
            }
            BeeStarHelper.SetCanvasLocation(
                        newStar, e.StarThatChanged.Location.X, e.StarThatChanged.Location.Y);
        }
    }
}
```

The _fadedStars collection contains the controls that are currently fading and will be removed the next time the ViewModel's Update() method is called.

If a star is being added, it needs to have its FadeIn() method called. If it's already there, it's just being moved because the play area size changed. Either way, we want to move it to its new location on the Canvas.

Long Exercise Solution

Here are the methods for the StarControl code-behind:

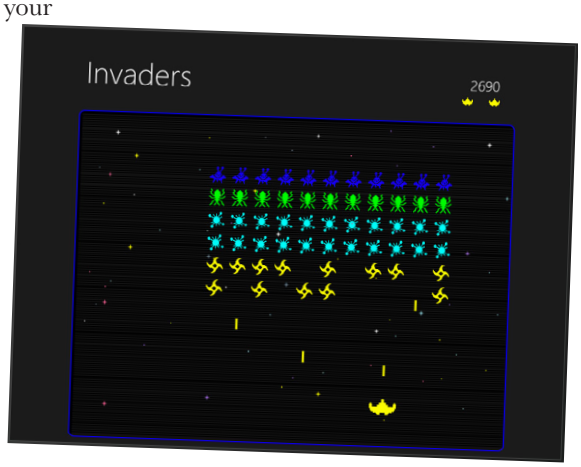
```
using System.Windows.Media.Animation;

public partial class StarControl : UserControl {
    public StarControl()
    {
        InitializeComponent();
    }

    public void FadeIn() {
        Storyboard fadeInStoryboard = FindResource("fadeInStoryboard") as Storyboard;
        fadeInStoryboard.Begin();
    }

    public void FadeOut() {
        Storyboard fadeOutStoryboard = FindResource("fadeOutStoryboard") as Storyboard;
        fadeOutStoryboard.Begin();
    }
}
```

If you've done a good job with separation of concerns, your designs often tend to naturally end up being loosely coupled.

The ViewModel's `PlayAreaSize` property just passes through to the property on the Model—but the Model's `PlayAreaSize` set accessor calls methods that fire `BeeMoved` and `StarChanged` events. So when the screen resolution changes: 1) the Canvas fires its `SizeChanged` event, which 2) updates the ViewModel's `PlayAreaSize` property, which 3) updates the Model's property, which 4) calls methods to update bees and stars, which 5) fire `BeeMoved` and `StarChanged` events, which 6) trigger the ViewModel's event handlers, which 7) update the `Sprites` collection, which 8) update the controls on the Canvas. This is an example of loose coupling, where there's no single, central object to coordinate things. This is a very stable way to build software because each object doesn't need to have explicit knowledge of how the other objects work. It just needs to know one small job: handle an event, fire an event, call a method, set a property, etc.

You've got all the tools to do Lab #3 and build Invaders!

We saved the best for last. In the last lab in the book, you'll build your own version of Space Invaders, the grandfather of video games. And while the lab is aimed at Windows Store apps, if you finished the *Bees on a Starry Night* project—and you understood it all—then you have the knowledge and know-how to build a WPF version of the Invaders game. Almost everything in the lab applies to WPF. The only thing that's different is how the user controls the ship. Windows Store apps have advanced gesture events that process touch and mouse input, but WPF windows don't support those events. You'll need to use the WPF Window object's KeyUp and KeyDown events. Luckily, you've already got a good example. Flip back to the Key Game in Chapter 4—your Invaders game can handle keystrokes in exactly the same way.



Congratulations! (But you're not done yet...)

Did you finish that last exercise? Did you understand everything that was going on? If so, then **congratulations**—you've learned a whole lot of C#, and probably in less time than you'd expected! The world of programming awaits you.

Still, there are a few things that you should do before you move on to the last lab, if you really want to make sure all the information you put in your brain stays there.

Take one last look through Save the Humans.

If you did everything we asked you to do, you've built *Save the Humans* twice, once at the beginning of the book and again before you started Chapter 10. Even the second time around, there were parts of it that seemed like magic. But when it comes to programming, **there is no magic**. So take one last pass through the code you built. You'll be surprised at how much you understand! There's almost nothing that seals a lesson into your brain like positive reinforcement.

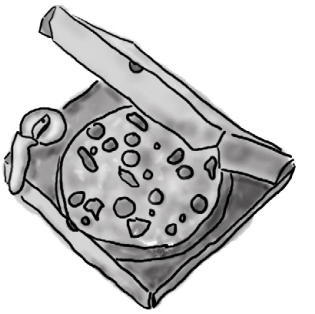
When it comes to programming, there is no magic. Every program works because it was built to work, and all code can be understood.

Talk about it with your friends.

Humans are social animals, and when you talk through things you've learned with your social circle you do a better job of retaining them. And these days, "talking" means social networking, too! Plus, you've really accomplished something here. Go ahead and **claim your bragging rights**!

Take a break. Even better, take a nap.

Your brain has absorbed a lot of information, and sometimes the best thing you can do to "lock in" all that new knowledge is to sleep on it. There's a lot of neuroscience research that shows that information absorption is significantly improved **after a good night's sleep**. So give your brain a well-deserved rest!



...but it's a lot easier to understand code if the programmer used good design patterns and object-oriented programming principles.

The humans forgot about us! Time to attack while they've lowered their guard!